WDA-3 2008

3rd Workshop on Dependable Architectures

(extends previously held Workshop on Architectural Reliability - WAR)

In conjunction with the

41st International Symposium on Microarchitecture (MICRO-41)

Saturday, Nov. 8, 2008
Lake Como, Italy

Welcome to the 3rd Workshop on Dependable Architectures!

Current computer technology trends present to the hardware and software designer novel opportunities to improve performance and at the same time many challenges to overcome. One of the formidable challenges is to provide dependable operation - in terms of reliability and availability - for a system made of unreliable components.

The combination of various developments brought dependability to prominence: soft-error rate is projected to increase with scaling; variability due to non-deterministic placement of dopant atoms and channel length is increasing design margins; better than worst-case design techniques for power/performance require error detection/correction; aggressive application of power-saving mechanisms such as clock- and Vdd-gating are increasing voltage droops; the verification manpower budget is becoming a significant part of the design effort; oxide breakdown and electromigration are decreasing processor lifetimes.

New research frontiers are therefore open for exploration that will lead to the discovery and development of dependable architectures, this includes research at all design levels: circuit, architecture, compiler, OS and network. This workshop aims to become a forum for academia and industry to discuss and present ideas and recent developments in the design and evaluation of dependable architectures both software and hardware.

We like to thank the authors for submitting their work at WDA-3 and the program committee members for providing on-time detail reviews. Finally, we like to recognize Costas Kourougiannis for handling the workshop's web-page.

November 8, 2008

Co-Organizers

Yiannakis Sazeides, University of Cyprus
Osman Unsal, Barcelona Supercomputing Center
Oguz Ergin, TOBB University of Economic and Technology

Program Committee

Todd Austin, University of Michigan
David Brooks, Harvard University
Veerle Desmet, Ghent University
Oguz Ergin, TOBB University of Economic and Technology
Babak Falsafi, EPFL
C. Mani Krishna, UMass, Amherst
Shubu Mukherjee, Intel
Onur Mutlu, Microsoft
Jude Rivers, IBM
Yiannakis Sazeides, University of Cyprus
Osman Unsal, Barcelona Supercomputing Center
Xavi Vera, Intel
David Kaeli, Northeastern University

WDA-3 Program

8:30-9:30 Opening Session

- Welcome and Outline
- Keynote, Babak Falsafi, EPFL

9:30-10:00 Session I

- *Reducing Fault Detection Latencies in Virtually-Lockstepped Systems*
  Casey Jeffery and Renato J. O. Figueiredo (University of Florida)

10.00-10.30 Break

10.30-12.00 Session II

- *Automatic Adjustment of System Performance to Mitigate Device Aging via a Co-designed Virtual Machine*
  Omer Khan and Sandip Kundu (University of Massachusetts Amherst)
- *Exploiting Value Prediction for Fault Tolerance*
  Xuanhua Li and Donald Yeung (University of Maryland)
- *Multicore Power Management: Ensuring Robustness via Early-Stage Formal Verification*
  Anita Lungu (Duke), Pradip Bose (IBM), Dan Sorin (Duke), Steven German (IBM), and Geert Janssen (IBM)
- Concluding Remarks

KEYNOTE

What to do with 100 Billion potentially misbehaving transistors on a chip

Babak Falsafi
Professor of CS, EPFL
Adjunct Professor of ECE & CS, Carnegie Mellon

The demand for computer system performance continues to grow to keep pace with our daily needs and to enable solutions to previously infeasible computing problems. Advances in semiconductor fabrication along with architectural and circuit innovation have helped computer system designers to accommodate this increase in performance demand since the emergence of microprocessors in the 70's. As a result, microprocessor vendors today market high-end products with roughly two billion transistors per chip offering unprecedented computational performance and capabilities. Unfortunately, while technology roadmap projections forecast the continued increase in the number of transistors per chip well into the next decade, there are fundamental sources of hardware and software bottleneck in sight that may impede the way to design and performance scalability of computer systems.

In this talk, I will present a few of these fundamental challenges and potential research directions in computer system designs to harness performance from future hundred-billion transistor chips and beyond.

Bio:

Babak Falsafi is a Professor in the School of Computer and Communication Sciences at EPFL, and an Adjunct Professor of Electrical and Computer Engineering and Computer Science at Carnegie Mellon. He is the Microarchitecture thrust leader for the FCRP Center for Circuit and System Solutions and directs the Parallel Systems Architecture Laboratory (PARSA) at EPFL. His research targets architectural support for parallel programming, resilient systems, architectures to break the memory wall, and analytic and simulation tools for computer system performance evaluation. In 1999, he showed in collaboration with T. N. Vijaykumar for the first time that multiprocessors need not support relaxed memory consistency models to achieve high performance. He is a recipient of an NSF CAREER award in 2000, IBM Faculty Partnership Awards between 2001 and 2004, and an Alfred P. Sloan Research Fellowship in 2004. He is a senior member of IEEE and ACM.

Reducing Fault Detection Latencies in Virtually-Lockstepped Systems

Casey Jeffery and Renato J. O. Figueiredo
Advanced Computing and Information System Lab
University of Florida, Gainesville, FL, USA
cjeffery@ufl.edu – renato@acis.ufl.edu

ABSTRACT

The relentless pace of transistor scaling has brought with it an increasing need for fault tolerance capabilities in logic devices. A common technique for providing this is processor replication in a fully-lockstepped fashion. This paper presents a hypervisor-based replication implementation, which can be applied to commodity hardware to allow for virtually-lockstepped system operation. It offers the benefits of full replication ranging from error detection through simple duplex execution to error correction through triplex execution, and can be extended to support Byzantine fault tolerance (BFT).

Virtualization hardware support is used to minimize replication overhead and processor state fingerprinting is employed to reduce the fault detection latency. The fingerprinting facilitates the detection of errors before they are recorded to a checkpointed state, which allows for recovery to a known-good state prior to a crash. The benchmarks considered indicate a performance overhead in the range of 2% to 5% with a non-optimized implementation, and fault injection trials show that fault detection latency can be reduced between 43% and 98% for the prototype considered.

1. INTRODUCTION

A major challenge that has emerged in the pursuit to fabricate ever smaller and faster transistors into increasingly complex chip designs is the ability to maintain a very high level of processor reliability. It is a concern in all modern semi-conductor process technologies and continues to become more so as Moore's Law leads scaling of devices down to only tens of nanometers and allows designers to incorporate many billions of transistors into a single chip.

There are a number of factors that contribute to the challenge. First, the devices are becoming increasingly susceptible to transient faults, which are caused by radiation events and electromagnetic interference. The soft error rates of combinational logic are fast approaching the levels at which protection was necessary in memory devices [24] and are expected to induce a higher failure rate than all other means of failure combined if not countered by fault-tolerance techniques [2].

In addition to soft errors, there continues to be an increase in the degree of static and dynamic transistor variability and much higher rates of transistor performance degradation and wear-out [4, 26]. The increases to chip complexity are also expected to limit the validation possible during post-fabrication testing [13]. This will result in marginal hardware being produced that must be maintained in the field by fault tolerance mechanisms.

Unlike memory devices, which can typically be protected from faults in a straightforward fashion by incorporating redundant information in the form of parity or error correction coding, logic devices have proven much more difficult to protect. The most common approach taken is to replicate the entire device, as doing so allows for comparisons to be made between replicas. The replication can be done at a micro-architectural level, such as the pipeline of the processor [11, 18, 19], at the system level through full machine duplication [3, 29], or somewhere in between [17]. It may be done in a software-transparent fashion with specialized hardware, through software-only approaches, or by incorporating a combination of both hardware and software support [1, 11, 19].

The goal of this paper is to explore a low-overhead, hypervisor-based replication that reduces fault detection latency by comparing a hashed "fingerprint" of the virtual CPU state at regular intervals. Specifically, the latency being considered is the period of time from when a fault is introduced in the system until it is discovered, if ever. If the fault is manifested as an erroneous value in a register, it is often discovered much later when the system crashes or hangs, but it may also be masked or lead to silent data corruption (SDC). The benefit of the early detection afforded by the fingerprint comparisons is that a rollback can be done to a checkpoint of a known good state prior to the error.

The prototype is based on the KVM virtual machine monitor [15], which takes advantage of commonly-available hardware support to improve

system performance and fault detection capabilities. The benefit of this approach over previous work is that it is inexpensive to deploy and maintains high performance and simplicity by exploiting hardware support present in practically all modern processors ranging from low-power netbooks to enterprise-level servers. The benchmarks considered in this paper indicate that this is a viable option with an overhead on the order of 5%, while the fault injection experiments indicate that processor state fingerprinting can significantly reduce fault detection latency.

2. REPLICATION & VIRTUAL LOCKSTEP

There is a long history in the enterprise server space of providing fault tolerance through lockstep-based replication. Lockstepped execution ensures that all replicas begin in the same state, receive the same, deterministic inputs, and progress through the same state transitions. Any divergences can be detected by differences in processor state or in the output from the chip.

A varying degree of lockstep is possible, ranging from cycle-level lockstep in which all cores execute exactly the same stream of execution to systems in which a single core actively executes the code and sends updates to one or more passive replicas that can take over if a failure is detected. These enterprise-class systems are highly specialized and very costly to deploy. They require customized hardware and possibly a layer of middleware for managing the replication [3, 29].

Virtual lockstep is a term given to systems that do not necessarily execute in full lockstep directly on the hardware. Instead, a virtualization layer in interposed to act as a middleware for coordination of the lockstepped operation without the need for specialized underlying hardware.

2.1. Virtualization Technology

A hypervisor, also referred to as a virtual machine monitor (VMM), sits logically between the hardware and the operating system. It facilitates system virtualization by allowing a software-only implementation of a machine to be seen as real hardware by the operating system running on it. By offering this additional layer of abstraction, the virtual hardware interface offers a simplified view of hardware that is amenable to deterministic execution, and therefore to virtual lockstep operation.

Although the concept of virtualization was originally developed over forty years ago, the wide-spread use of virtual machines only started to take hold within the last decade. In that time, the capabilities of system virtualization have expanded significantly, including the introduction of hardware support by all major processor manufacturers. Practically all modern computing platforms support virtualization to some degree and most offer a high level of hardware support.

The hardware support specifically considered in this paper is the Intel Virtualization Technology (VT-x) [27]. It provides for a new mode of operation termed VMX root mode. This new mode was designed to overcome challenges in software-only virtualization such as ring aliasing, address-space compression, and non-faulting access to privileged processor state. It does this by running (*resuming*) an operating system (the *guest*) in VMX non-root mode and transitioning control (*exiting*) back to the VMM (the *host*) in VMX root mode whenever necessary. The guest can then use all four ring levels, is guaranteed to exit to the host on all privileged instructions, and the processor changes the linear address space whenever control is transferred between the guest and the host.

A portion of the state of the host and the guest is maintained in a page of memory called a Virtual Machine Control Structure (VMCS), which is set up with a physical address mapping known to the VMM and passed to the processor whenever a guest is resumed. This structure stores the segment, control, instruction pointer, stack pointer, and flag registers. Control fields are also present to define the types of operations for which the VMM has requested control. There is some virtual processor state, such as the remaining general purpose registers, that are not contained in the VMCS and must be saved and restored manually by the VMM software.

2.2. Related Work in Virtual Lockstep

The first work in the area of virtual lockstep operation made use of a custom hypervisor designed to run on the HP PA-RISC Unix system [5]. This implementation made use of a software-only VMM, which ran single primary and backup nodes in a leader/follower configuration synchronized on epoch boundaries.

An epoch, defined by the instruction retirement counters, provides a means of injecting interrupt vectors at the same deterministic position in both the primary and backup. The primary executes one epoch ahead of the backup and has the task of choosing values for all nondeterministic events, such as the reading of the timestamp counter or input

from an I/O operation, as well as buffering of external interrupts for delivery at the end of the epochs.

In the event that the primary crashes, the backup will detect the missing heartbeat signal, which is triggered by the expiration of a timer, and take over beginning at the start of the epoch in which the primary crashed. It may repeat disk or network operations that the primary completed before crashing, but it is assumed that the network or storage drivers are capable of handling these repeat requests.

One key limitation of this model is the necessity of fail-stop behavior in the primary. In other words, the only failure of the primary for which the backup is capable of detecting and recovering from is a system crash or similar event which causes the heartbeat signal to not be received. It is possible that the error was caused by a latent fault that occurred many epochs prior and did not result in an immediate crash. The fault may have also been in state that was transferred to the backup, in which case the backup will proceed to reproduce the crash.

A more recent proposal for a virtually lockstepped system is based on the Xen hypervisor [20]. In this model, a new network/voting (NV) domain is defined, which has the logic for the replication and communication of the dom0 (host) and domU (guest) domains. This model is based on replicating at a much higher level, however. The replicas provide network-based services and must be consistent only from the point of view of a network client, which gives flexibility in the underlying hardware and software at the cost of limited scope.

Another approach to providing benefits similar to virtual lockstep is the use of rapid checkpointing. In contrast to lockstepped execution, a backup replica in a checkpoint-based system does not actively re-execute code. Instead, the primary records a full snapshot of the current state of the processor and memory and sends it to the backup, which allows it to pick up where a primary left off in case of failure. An example of this is given in [8], where the Xen hypervisor is again used. In this model, the backup receives system checkpoints from the primary at a rate as high as once every 25ms. The primary is able to buffer I/O until the end of an epoch, at which time it is committed to both the primary and backup.

The rapid checkpoint model has a simpler implementation than a virtually lockstepped system, and is also much easier to apply to multiprocessor guests. These benefits come at the cost of the actual execution not being replicated. That is, if a fault is present in the primary that causes an incorrect value to be computed, the value will be transferred to the backup and not regenerated. The checkpointing of faulty state was found to happen with high probability in [7]. This limits the fault model to immediate fail-stop behavior of the primary.

3. FAULT DETECTION

As indicated in the previous section, it is preferable to not just detect an error when a system crashes, but detect when the fault that caused it occurs. This gives the system an opportunity to take the steps necessary to avoid a crash. To do this, it is necessary to track the state of the processor and not just monitor the signals that leave the chip. This requirement is due to the latency in the manifestation of errors at the outputs of the processor. For example, a register that is struck by a particle and has a bit flipped will likely not be immediately accessed. It may even be written back to memory and read in again much later. By the time the error becomes software-visible, it is unlikely that a recovery is possible.

3.1. Processor State Fingerprinting

One way to ensure that an error is detected as soon as possible is to execute all replicas in virtual lockstep and compare the full state of the processor on a regular basis. It is very expensive in terms of bandwidth and power consumption to make such extensive comparisons, however. An optimization is to hash the processor state into a unique fingerprint and make a comparison based on that single value. This is much faster, and if an appropriate hash is used, collisions can be kept to a minimum and the accuracy of the fault detection can be maintained.

There is a wide range of hashing algorithms that can be applied to the data representing the processor state varying in complexity from simply adding the registers together to applying a complex cryptographic hash like SHA-2.

It is not possible to tell from the fingerprint exactly which parts of the processor state have diverged since that information is lost in the hashing process. It is sufficient, if intermittent checkpoints are taken, to trigger a roll back to the last known good checkpoint and resume execution.

3.2. Related Work in Fingerprinting

Fingerprints have been used to reduce fault detection latency on enterprise-class server systems [10, 25]. In this work, a dual-modular redundant server with cores lockstepped at the hardware level

is enhanced to maintain a hash representative of the history of execution. Information about instruction commits are hashed using a cyclic redundancy code (CRC). It is observed that error detection latency can be improved considerably at a small cost in terms of compute and bandwidth resources, although the cost of developing such specialized capabilities in the processor pipeline is quite significant.

4. FAULT-TOLERANT SYSTEM MODEL

The model presented in this paper is a virtually lockstepped system described in detail in [14]. The goal of the model is to be directly applicable to a variety of systems, ranging from simple dual-core platforms to future networks on chip (NoCs) with hundreds of cores on a single die. This is accomplished by allowing a primary instance of a hypervisor to be coupled with an arbitrary number of backup replicas, each instantiated on a separate machine or logical partition.

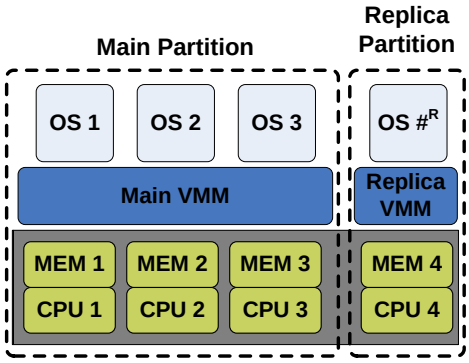


Figure 1. System with many-to-one protection level.

An example is shown in Figure 1 where a single backup can be dynamically tied to one of the guests running on the primary VMM when the underlying hardware for the guest is detected to be faulty. The other extreme is shown in Figure 2, where there are three backup replicas for a single primary instance, which is necessary to support BFT [6].

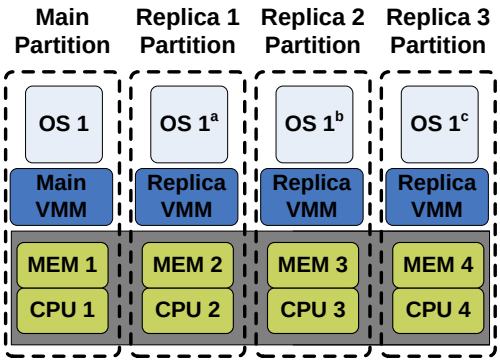


Figure 2. System with one-to-many protection level.

4.1. Replication Coverage

The main goal of the model is to protect the processor from single-event upsets. It is assumed that the main memory, storage, and network devices can be replicated through other means, such as memory sparing, RAID, or network adapter teaming, respectively. For this reason, these devices are left outside the sphere of replication.

4.2. Fault Injection Model

The behavior of the system in response to faults is simulated by altering the state of the virtual processor to model bit flips. This can be done easily since the VMM has complete control over the state of the guest. The injection is done in a manner similar to that presented in [16], where the Xen hypervisor is used as a fault injection vehicle for a Linux guest. That work looked at only four registers, but defined a method of categorizing the types of errors seen and recorded the latency of the error in terms of machine cycles.

4.3. Error Detection

Errors are detected by comparing a hashed fingerprint of the virtual processor state at regular intervals. This is different from the fingerprinting approach taken in [10, 25]. Rather than hashing information about retired instructions, the virtualization hardware is utilized by hashing the information stored in the VMCS.

The main reason for this choice is that it takes advantage of a well-defined, hardware-accessible structure, which allows for the hardware to be trivially optimized, reducing the hashing overhead. For example, the microcode could be updated to do the hashing, or a specialized or idle processing core could be used to hash the memory region.

State comparisons are made on execution boundaries determined by exits from the guest to the VMM. The rate of comparison can be adjusted to trade off performance for reduced detection latency and higher detection accuracy.

5. PROTOTYPE DETAILS

The prototype that has been developed is based on the KVM hypervisor. For the purposes of this paper, it has been implemented only on the Intel x86 architecture with support for uniprocessor guests, since it is significantly more complex to support multi-processor deterministic execution and the performance overhead is not yet practical [9]. All replication logic is incorporated into the userspace portion of the hypervisor, along with a small amount of support code in the kernel module.

5.1. Hypervisor Overview

The KVM hypervisor has been integrated into the mainline Linux kernel since version 2.6.20 and has recently been ported to support most major architectures, including x86, IA64, and PowerPC. It uses the QEMU emulator for virtual device models, and so runs in the context of a Linux process, which makes it trivial to start multiple instances and to tie them to specific processing cores. The hard disk image provided to each instance can be backed by a file-based disk image (qcow2), which is easily replicated and supports checkpoint snapshots. Inter-replica communication in the prototype is done through a buffer allocated in shared memory, which is sufficient for duplex and triplex configurations. An extension to BFT is possible if a decentralized, group coordination and communication protocol is applied.

5.2. Virtual Lockstep Details

In order to run replicas in lockstep, it is necessary to remove all nondeterminism from the system [22]. This includes synchronous sources such as instructions that access the timestamp counter or read in data from an I/O port, as well as asynchronous sources, such as external interrupts. Direct memory access (DMA) replication has not yet been implemented, but it must be dealt with as a combination of both cases (i.e., an I/O instruction that occurs asynchronously). This is an optimization left as future work.

The replicas are synchronized based on the number of deterministic exits that occur to VM-root mode. That is, each time a VM exit occurs at a point in the execution that is guaranteed to be deterministic, a counter is incremented and the hypervisor is given the opportunity to inject asynchronous events, such as virtual interrupts. This ensures that the asynchronous events occur deterministically and at the same point in all replicas. In the prototype, lightweight exits (those handled entirely in the kernel) are not counted.

The data from the synchronous, non-deterministic instructions, which include those that do string or value I/O operations, memory-mapped I/O, or read the timestamp counter, are copied into a structure and stored into a circular broadcast buffer with a flag indicating the type of operation. The buffer is shared among all hypervisors in a typical producer/consumer fashion. The primary stores the items and signals the backup(s) when an item is available. When a backup gets to the same synchronous instruction, it retrieves the information stored in the buffer and verifies the flag matches the type of operation it expects. It then either overrides the input it received or verifies the output it produced, depending on the direction of the event. It is possible in this way to detect errors at the I/O level, but as described earlier, it is desirable to detect them even sooner.

The asynchronous, nondeterministic events (i.e., external interrupts) are captured by the primary and placed into a second shared buffer. Their delivery into the primary is delayed until the first VM entry following a deterministic exit for which the guest is ready to accept the interrupt. If both of the requirements are met, the event is injected and its details are recorded in the shared buffer. The backup replicas peek at this buffer to determine the point at which they must inject the next event, which is defined by the deterministic exit count. When a replica arrives at the target position, the event is removed from the buffer and injected. It is assured that the event will be deliverable, unless a fault has occurred, since the state of the system is identical to the primary.

One of the sources of nondeterminism seen in the KVM hypervisor is in memory paging. It appears to be due to different paging behavior of the file-backed guest hard disk images. To skirt this problem, the replicas are run in a ramdisk, which means that the virtual hard disks are placed entirely in memory so that access to a physical hard disk, and the subsequent page faults, are not required. This limitation does not affect the main goals of assessing the benefit of early fault detection, and resolving it is left as a future optimization.

A second potential issue is that by synchronizing only on deterministic exits from the guest, it is possible for the guest to never exit deterministically and consequently make no forward progress. For the purposes of this paper, the benchmarks executed have a steady rate of deterministic exits and avoid the problem. This limitation can be averted by ensuring a minimum rate of deterministic exits by generating interrupts with the performance counters, for example.

5.3. Fingerprinting Details

The error detection capabilities of the system are enhanced by verifying the state of the virtual processor at the deterministic execution boundaries. This means that ideally, faults are detected at the first deterministic exit after they are introduced, as long as they affect one or more of the fields in the VMCS.

The fingerprints are generated using a simple multiplicative hashing algorithm defined in [23] and added by the primary to each item placed in the shared buffers. This allows the backup to easily compare its state to that of the primary while doing the standard checks against the buffer entry.

It is possible to enhance the error detection capabilities further by including additional state into the fingerprint calculation. For example, it would be desirable to have the general purpose registers included, even though they are not part of the VMCS. This is certainly possible, but it precludes the optimization of directly using a hardware-only approach to generating the fingerprints (at least without a microcode or hardware extension).

6. EVALUATION AND BENCHMARKS

The prototype of the proposed model is evaluated along multiple vectors. First, the performance overhead of the virtual lockstep implementation is assessed. The fault injection capabilities are then considered, and finally the fault detection latency is evaluated.

6.1. Test Platform

The test platform includes an Intel Xeon X3360, which is a 2.93GHz quad-core processor with support for the latest VT-x hardware extensions. The system has 4GB of main memory with 2GB reserved for use in hosting a single primary and backup. The hypervisor is a modified version of KVM-33 that is run on a 32-bit Ubuntu 7.04 installation. The guest image is a 32-bit Slackware 10.2 installation with default kernel settings and 128MB of main memory.

6.2. Benchmarks

The benchmarks considered for this paper is are Linux kernel compilations. These were chosen because they offer high levels of both processor and I/O activity. For the purposes of overhead estimation, both a relatively small kernel (2.4.31) and a larger kernel (2.6.20) are considered. The 2.4 kernel is the default for the Slackware 10.2 guest and the 2.6.20 kernel is from the public Linux kernel servers. They are compiled in the guest with gcc 3.3.6 and default configuration options.

6.3. Virtual Lockstep Overhead

The overhead of virtual lockstepped execution comes from a number of sources. There is the cost of the primary recording the values for all nondeterministic events and the backup then retrieving them and making the necessary comparisons. There is also the cost of delaying interrupt delivery to occur at deterministic boundaries, and finally, there is the cost of the primary stalling when it runs too far ahead and fills the buffer or similarly when the buffer is empty and the backup must stall. Because the hypervisors are pinned to processing cores, there is a relatively small slack that accumulates between them so the final issue can be handled using reasonably sized buffers.

The overhead attributed to the replication is estimated by comparing the performance of a virtually lockstepped execution to an identical instance that is virtualized but not replicated. The results are shown in Table 1 and indicate a very reasonable overhead of approximately 2%-5%. The times are an average of ten trials and were tracked using VMCALL instructions, which allow the guest to call back into host. The guest executes the VMCALL immediately before and after the compilation and the host reads the platform timestamp counter and calculates the difference.

These numbers will vary significantly depending on the platform on which it is run, and it is expected that running the guests on a hard disk will add to the overhead. There are also workloads that will exhibit a larger performance hit, but this initial analysis indicates that it will likely be a tolerable hit and that virtual lockstep can be made practical given the benefits it provides.

Table 1: Overhead of Linux kernel compile for virtual lockstep compared to virtualization only

Primary	Virtualized	Lockstepped	Overhead
Linux 2.4.31	128.5s	135.1s	5.1%
Linux 2.6.20	255.3s	258.8s	1.4%
Backup			
Linux 2.4.31	128.5s	135.6s	5.5%
Linux 2.6.20	255.3s	259.2s	1.5%

6.4. Fault Injection

As mentioned previously, there are benefits to using a virtual machine as a platform for fault injection experiments. First, it has direct access to the system registers, as well as the guest stack, interrupt descriptor table, and memory. This makes fault injection as simple as altering bits of state and resuming the guest execution.

From the very large space of possible fault targets, a few key registers have been chosen and are listed in Table 2. They were selected to align well with similar work [12, 16, 28], as well as to give reasonable coverage of both registers that are stored in the VMCS and those that are not.

A fault is modeled by flipping a bit, which is done by xor-ing a 1 to the target bit. All faults are injected into the backup replica of a duplex system. This allows for direct comparison of the processor state and output to the primary to detect the effects of the fault. The two bit positions targeted were chosen somewhat arbitrarily as bit 4 and bit 16. The main reasoning was to flip one near the lower portion of the register so that the affected value will move only a small amount (e.g., two to 16 instructions in the case of RIP) and to flip a higher order bit so as to cause a more significant change for cases when the value is treated as a number.

Table 2: Registers considered for fault injection

Register	In VMCS?	Description
RIP	Y	Instruction pointer
RSP	Y	Stack pointer
RAX	N	Accumulator
RCX	N	Counter
RBP	N	Base pointer
RSI	N	Data (Source)
CS_B	Y	Code segment (Base)

6.5. Fault Detection

Faults are injected during a compilation of the Linux 2.4.31 kernel, and the time of the injection is varied randomly. The compilation is run for 10,000 deterministic VM exits plus a random number of additional deterministic VM exits from 0 to 2^{16}, which is generated by /dev/random in the host Linux kernel. After injection, the guest is run for at least 50,000 additional deterministic VM exits, which is on the order of the runtime considered in [16, 21].

The first set of data considered are whether the guest fails or continues to run to completion. This is broken down by failure mode in Figure 3. A crash means that a guest fails, dumps failure information, and stops executing, while a hang means that the guest ends up in a state in which no forward progress is made but it doesn't stop.

It is notable that faults in RAX and RCX rarely cause the system to crash. Only a fault in the high bit of RAX causes a significant failure rate. This also holds true for RBP and RSI, which are the other registers not saved in the VMCS. In general, it is observed that faults in registers that are not part of the VMCS caused a much lower failure rate than those that are. This isn't surprising since the point of the VMCS is to automatically store the state of the most critical registers in the CPU.

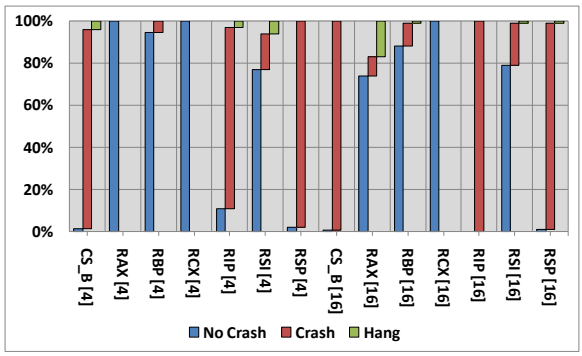


Figure 3. % of failures by register [faulty bit position].

It may be that data are silently corrupted (SDC) in the cases where no crash occurs and work is in progress to verify this. It is straightforward to detect some forms of SDC in the kernel compile benchmark considered by retrieving the generated binaries and comparing them to known-good copies. It is more difficult, however, to determine if latent errors have been introduced into the running kernel of the guest machine.

To detect the errors resulting from the fault injections, guest state fingerprints are generated and compared on every deterministic exit. The fingerprints are derived by hashing most of the fields in the VMCS. The CR3 and TSC Offset are excluded from the hash since they are expected to differ. Registers for unused features such as SMM are also excluded. The Interrupt Error Code is not included since it is updated only on exits for interrupts that would deliver an error code to the stack and may be stale otherwise. The final two fields not part of the hash are Access Rights for FS and GS registers. The descriptor privilege level of these fields is not consistent on all exits.

It is extremely unlikely that a fault introduced into a system register will cause an immediate crash or hang. There is generally a period of time from when a fault is injected until the system fails, and for the purposes of this paper, this is considered the base fault detection latency. The improved fault detection latency is the time from when the same fault is injected until it is detected in a fingerprint comparison. The benefit of the improved detection latency is that it is typically much shorter and improves the probability of successful rollback and recovery.

The data presented in Figure 4 demonstrate that the fingerprint-based model is capable of detecting errors within only a few exits, whereas there are often dozens or hundreds of exits before the system finally crashes. The average reduction across all

registers except RSP is 97% and is as high as 98% for RIP. It is notable that a fault in the high bit of RSP does cause the system to crash much earlier than the other fault targets considered, but the faults are still detected in the fingerprint comparisons 43% earlier, on average.

6.6. Fingerprinting Optimizations

The final breakdown of the data is focused on finding ways of optimizing the performance of the fingerprinting approach. Specifically, the size of all fields of the VMCS that have been considered in the hash is only 396 bytes, which is quite small, but hashing the data does have a performance cost that should be minimized. The most obvious optimization is to exclude VMCS fields that are unlikely to play a part in detecting a fault in the system.

Figures 5 and 6 break down the fields of the VMCS in which differences were detected for the fault targets considered. These are the fields affected only on the first exit after which a difference is detected, and there are a surprising few. Additional fields often become corrupted on subsequent exits before the guest crashes, but are not included.

The small subset of VMCS fields consists of 44 or 60 bytes on 32- or 64-bit host systems, respectively, and represents the minimal set of fields that need to be included in the fingerprint to provide equivalent detection coverage to including all fields for the fault model considered in this paper. We believe that this subset will expand very little as data are gathered for additional workloads. It is possible to optimize even more since there are a number of fields that always appear to occur together or in addition to other fields. For example, every time a fault is detected in FS Selector, GS Selector is also faulty.

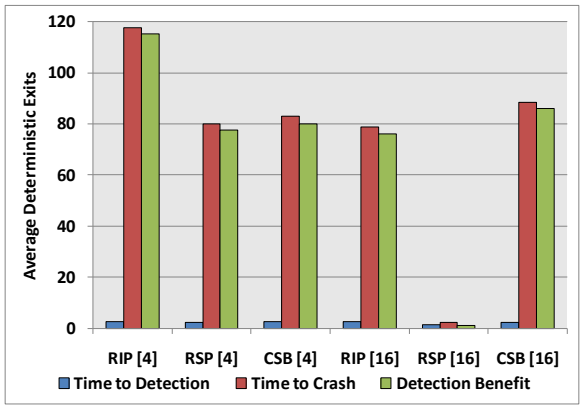


Figure 4. Average time to error detection using fingerprints versus time to guest crash measured in terms of deterministic exits from fault injection.

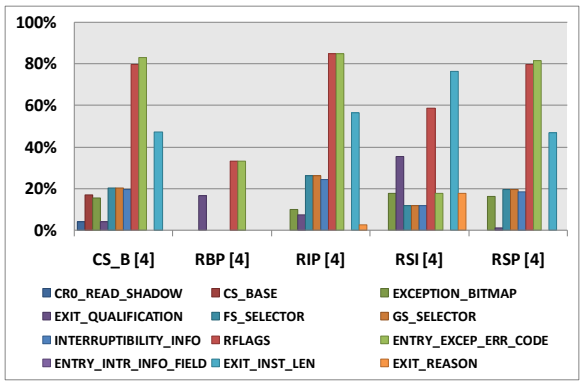


Figure 5. VMCS fields first affected by fault in bit 4 of registers.

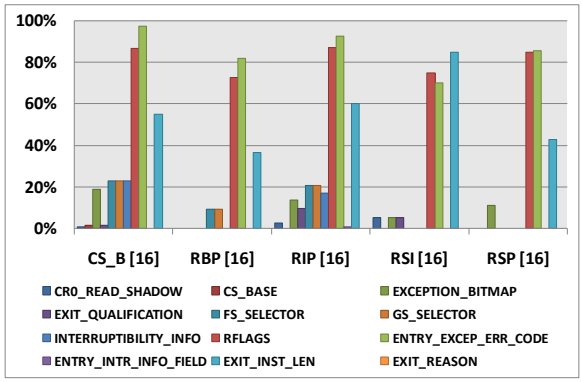


Figure 6. VMCS fields first affected by fault in bit 16 of registers.

7. CONCLUSION AND FUTURE WORK

In the near future, it will be essential to apply new techniques to computing systems to ensure reliable operation. The goal of this paper is to present a virtual lockstep implementation that is software based, yet capable of using hardware features for enhanced performance and fault detection capabilities. The result is a system that has a low performance overhead and significantly reduces the time to detection of faults that occur in the processor.

This work also indicates that it may be beneficial to extend the virtualization hardware capabilities to support fingerprinting and state comparison. Our current implementation uses a hash of a subset of the processor state as a basis for fault detection; this provides limited detection coverage as faults that occur between VM exits may not manifest in changes to the VMCS state. Nonetheless, our approach can accommodate additional fault detection coverage if provided by hardware, without significant changes to the framework.

For instance, detection can take into account the history of instructions between VM exits if

hardware is enhanced in a manner similar to what is described in [10, 25], as well as by including more registers than are currently in the VMCS and performing hashing using optimized hardware. By making these capabilities available to the hypervisor, fault detection and checkpoint rollback at VM exit boundaries can be done reliably.

References

[1] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *Proc. of 32nd Annu. Int. Symp. on Microarchitecture*, pp. 196-207, Nov. 1999.

[2] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. on Device and Materials Reliability*, vol. 5, no. 3, pp. 305-316, Sep. 2005.

[3] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "NonStop® Advanced Architecture," in *Proc. of the Int. Conf. on Dependable Systems and Networks*, Jun. 2005.

[4] S. Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10-16, Dec. 2005.

[5] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault-tolerance," *ACM Trans. on Computer Systems*, vol. 14, no. 1, pp. 80-107, Feb. 1996.

[6] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. of the 3rd Symp. on Operating System Design and Implementation*, Feb. 1999.

[7] S. Chandra and P. M. Chen, "The Impact of Recovery Mechanisms on the Likelihood of Saving Corrupted State," in *Proc. of the 13th Int. Symp. on Software Reliability Engineering*, Nov. 2002.

[8] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High Availability via Asynchronous Virtual Machine Replication," in *Proc. of the 5th USENIX Symp. On Networked Systems Design and Implementation*, Apr. 2008.

[9] G. W. Dunlap, D. G. Lucchetti, P. M. Chen, and M. A. Fetterman, "Execution Replay for Multiprocessor Virtual Machines," in *Proc. of the Int. Conf. on Virtual Execution Environments*, Mar. 2008.

[10] B. T. Gold, J. Kim, J. C. Smolens, E. S. Chung, V. Liaskovitis, E. Nurvitadhi, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "TRUSS: a reliable, scalable server architecture," *IEEE Micro*, vol. 25, no. 6, pp. 51-58, Dec. 2005.

[11] M. A. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," *IEEE Micro*, vol. 23, no. 6, pp. 76-83, Nov. 2003.

[12] W. Gu, Z. Kalbarczyk, and R. K. Iyer, "Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors," in *Proc. of the Int. Conf. on Dependable Systems and Networks*, July 2004.

[13] International Technology Roadmap for Semiconductors, 2007 ed. Austin, TX: Semiconductor Industry Association, International SEMATECH, 2007.

[14] C. M. Jeffery and R. J. O. Figueiredo, "Towards Byzantine Fault Tolerance in Many-core Computing Platforms," in *Proc. of 13th Pacific Rim Int. Symp. On Dependable Computing*, Dec. 2007.

[15] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux Virtual Machine Monitor," in *Proc. of the 9th Ottawa Linux Symp.*, Jun. 2007.

[16] M. Le, A. Gallagher, and Y. Tamir, "Challenges and Opportunities with Fault Injection in Virtualized Systems," in *Proc. of the 1st Int. Workshop on Virtualization Performance: Analysis, Characterization, and Tools*, Apr. 2008.

[17] M. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *Proc. of the 13th Int. Conf. on Arch. Support for Programming Languages and Operating Systems*, Mar. 2008.

[18] C. McNairy and R. Bhatia, "Montecito: A Dual-Core, Dual-Threaded Itanium Processor," *IEEE Micro*, vol. 25, no. 2, pp. 10-20, Apr. 2005.

[19] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," in *Proc. of the 29th Int. Symp. On Computer Architecture*, May 2002.

[20] H. P. Reiser and R. Kapitza, "Hypervisor-based Efficient Proactive Recovery," in *Proc. of the 26th IEEE Symp. On Reliable Distributed Systems*, Oct. 2007.

[21] G. P. Saggese, A. Vetteth, Z. Kalbarczyk, and R. Iyer, "Microprocessor Sensitivity to Failures: Control vs. Execution and Combinational vs. Sequential Logic," in *Proc. of the Int. Conf. on Dependable Systems and Networks*, Jun. 2005.

[22] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: a tutorial,” *ACM Computing Surveys*, vol. 22, no. 4, pp. 299-319, Dec. 1990.

[23] R. Sedgewick, *Algorithms in C*. Boston, MA: Addison-Wesley, 1997.

[24] 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 the Int. Conf. on Dependable Systems and Networks*, May 2002.

[25] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, “Fingerprinting: Bounding Soft-Error-Detection Latency and Bandwidth,” *IEEE Micro*, vol. 24, no. 6, pp. 22-29, Nov. 2004.

[26] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, “The Impact of Technology Scaling on Lifetime Reliability,” in *Proc. of the Int. Conf. on Dependable Systems and Networks*, Jul. 2004.

[27] R. Uhlig et al., “Intel Virtualization Technology,” *IEEE Computer*, vol. 38, no. 5, pp. 48-56, May 2005.

[28] N. J. Wang and S. J. Patel, “ReStore: Symptom-based Soft Error Detection in Microprocessors,” *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 188-201, Sep. 2006.

[29] S. Webber and J. Beirne, “The Stratus Architecture,” in *Proc. of the 21st Int. Symp. on Fault-Tolerant Computing*, Jun. 1991.

Automatic Adjustment of System Performance to Mitigate Device Aging via a Co-designed Virtual Machine

Omer Khan and Sandip Kundu

Department of Electrical and Computer Engineering
University of Massachusetts Amherst
Amherst, MA 01002
{okhan, kundu}@ecs.umass.edu

Abstract

As semiconductor manufacturing enters advanced nanometer design paradigm, aging and device wear-out related degradation is becoming a major concern. Negative Bias Temperature Instability (NBTI) is one of the main sources of device lifetime degradation. The severity of such degradation depends on the operation history of a chip in the field, including such characteristics as temperature and workloads. In this paper, we propose a system level reliability management scheme where a chip dynamically adjusts its own operating frequency and supply voltage over time as the device ages. Major benefits of the proposed approach are (i) increased performance due to reduced frequency guard banding in the factory and (ii) continuous field adjustments that take environmental operating conditions such as actual room temperature and the power supply tolerance into account. The greatest challenge in implementing such a scheme is to perform calibration without a tester. Much of this work is performed by a hypervisor like software with very little hardware assistance. This keeps both the hardware overhead and the system complexity low. This paper describes the entire system architecture including hardware and software components. Our simulation data indicates that under aggressive wear-out conditions, scheduling interval of days or weeks is sufficient to reconfigure and keep the system operational, thus the run time overhead for such adjustments is of no consequence at all.

1. Introduction

The likelihood of device wear-out is a growing problem for advanced nanometer technology. International Technology Roadmap for Semiconductors (ITRS) states that "the development of semiconductor technology in the next 7 years will bring a broad set of reliability challenges at a pace that has not been seen in the last 30 years" [1]. The relentless pursuit of smaller geometries is approaching a point where technology limitations are pushing designs toward tighter constraints and expensive margins, elevating concerns about device availability and reliability [2]. The potential for these failures[1] decreases the expected lifetime of the processor, creating a lifetime reliability problem.

Processor lifetimes are traditionally managed through a combination of quality control in manufacturing and conservative design parameters that reduce stress on a processor (e.g., running at a lower clock frequency and voltage to avoid high temperatures). Processors are typically designed with a mean-time-to-failure of 30 years, which assures few if any units will fail during 11 years of "expected consumer use" assumed by manufacturers [3]. Scaling trends make quality control to meet this reliability goal more expensive while conservative designs negatively impact performance.

Device aging has had a significant impact on transistor performance. Increased current density and temperature leads to faster degradation of transistors over time due to oxide wear out and hot-carrier degradation effects. Until 90nm technology, the degradation was small enough to be concealed by an upfront design margin in the product specification. But as the technology approaches 45nm and below, the *worst case* degradation is expected to become too large to be taken as an upfront design margin [2].

Product life acceleration with burn-in test is becoming less meaningful as well. To quote ITRS [1], "Two trends are forcing a dramatic change in the approach and methods for assuring product reliability. First, the gap between normal operating and accelerated test conditions is continuing to narrow, reducing the acceleration factors. Second, increased device complexity is making it impossible or prohibitively expensive to exercise or stimulate the product to obtain sufficient fault coverage in accelerated life tests. As a result, the efficiency and even

[1] Wear-out related failures, or intrinsic hard faults are distinct from extrinsic hard faults, which are permanent faults that result from manufacturing defects and are already present when a processor is tested in the factory. Thus, extrinsic hard faults are weeded out by testing. In contrast to extrinsic hard faults, the probability of intrinsic hard faults increases with long-term processor utilization. This paper addresses intrinsic hard faults.

the ability to meaningfully test reliability at the product level are rapidly diminishing."

Negative Bias Temperature Instability (NBTI) is a major source of device lifetime degradation [4]. NBTI affects PMOS transistors when the voltage at the gate is negative, causing the threshold voltage to increase. As a result both F_{MAX} and V_{MIN} of the design are impacted. The F_{MAX} is degraded because the circuits become slower over time, while memory structures experience an increase of their minimum voltage (V_{MIN}) to keep their contents.

Current practice is to use conservative frequency guard-bands of 10-20% to account for performance loss due to device aging [5][6]. For example, a device that clocks at 3GHz/1.1V during testing may be sold as a 2.7GHz/1.0V part to account for expected performance loss over product life time. This, in turn, requires designers to target for higher frequency of operation, thus significantly increasing power consumption [5][7].

The solution we propose avoids a large guard-band upfront, continually adjusting frequency and voltage over product lifetime. The main idea behind this scheme is to enable the system to adaptively adjust the operating frequency/voltage with minimal guard-bands to allow the system to operate at its peak performance throughout its life. The adjustments are transparent to the operating system and application's software. This fine-grain management of device aging provides additional benefits of workload adaptation, runtime field testing, and non-stop system operation, which is not permissible in the conventional F_{MAX} or V_{MIN} testing that requires a tester.

The rest of the paper is organized as follows. The remainder of this section is devoted to providing some background and related work on NBTI and its impact on device reliability, followed by motivation for the proposed scheme. In section 2, discuss lifetime reliability models for processors. In section 3, we describe our proposed reliability management architecture. Section 4 and 5 provide our experimental methodology and data analysis. We conclude in section 6.

1.1 NBTI & Related Work

The severity of threshold voltage degradation due to NBTI depends on the operation history of a chip in the field: circuit parameters like operating frequency, supply voltage and temperature variance play a role, as well as data patterns due to variation in the workload characteristics. The workload determines the length of time a PMOS transistor may spend in ON state, when most of the performance degradation happens.

We have already mentioned why burn-in is losing effectiveness against NBTI problems [8]. Researchers have proposed solutions to mitigate NBTI by: reducing the amount of time the PMOS transistors observe a "0" at their gates [9]; resorting to classical redundancy techniques [10]; using software logging to handle crash detection and recovery [11]; using circuit and logic techniques to catch dynamic errors using special sequential circuits [11][12]; using runtime adaptation of the processor to changing application behavior, termed as Dynamic Reliability Management (DRM) [13][14].

Although these techniques address the shortcomings of burn-in and guard-bands, they are either applied at a coarse-grain granularity or they require significant design cost overhead. For example, Razor DVS [15] proposes a technique to eliminate safety margins by running below critical voltage and subsequently tuning the processor voltage based on error rate. One of the main drawbacks of this work is the upfront additional circuitry required for Razor flip flops (RFF) and their associated power overhead. As RFFs are used on critical paths, meeting the chip's timing requirements and recovering pipeline state are challenging tasks that incur design overheads. On the other hand, DRM's uniform allocation provides high performance only for some applications, those that have high reliability slack, whereas our technique provides higher performance for all applications during the initial years and gracefully degrade performance as the device ages.

T. Austin et al., [16] propose a new software-based defect detection and diagnosis technique, which is based on using special firmware to insert tests for diagnosis and if needed repair through resource reconfiguration. Smolens et al., [17] present an in-field early wear-out fault detection scheme that relies on the Operating System to switch between functional and scan mode to test the chip in near-marginal conditions. Our technique uses similar software/hardware framework to address transistor aging, where the chip not only tests itself but also adapts to the changing conditions.

1.2 Motivation & Vision

The main drawback of burn-in and manufacturing time guard-bands is that they are *static* and *expensive*. Static guard-band may not be adequate for all parts; if the guard-band is increased, it may be excessive for other parts. This points to a need for flexible and scalable approaches that allow for continuous adjustments to combat degradation. The workloads running on a hardware platform are not static, but variable. The number of applications, their performance and power requirements, and the usage models vary based on the user demands and environmental conditions. Therefore, continuous adjustment of frequency/voltage seems natural.

We propose a system level architecture that is based on virtualization of device aging management. Virtualization, in this context, is a software process with some hardware collateral that helps finding the optimal frequency. The proposed virtual framework provides architects with a layer of software that resides in memory concealed from all conventional software, thus isolating the functions of the implementation-specific device aging

management features from the user and the operating system. The main idea is to expose the details of lower level hardware specific components to special software. This software provides flexible management capabilities of sensing, testing, and adapting the system over its lifetime. In an effort to address the drawbacks of conventional approaches discussed earlier, the proposed scheme has the following objectives:

Flexibility and Scalability: The layers of abstraction that exist between the hardware and software should hide intricate details that are necessary to manage frequency/voltage of the system efficiently and insulate OS. This will allow hardware to evolve freely.

Low Cost: Frequency calibration without a tester will require some hardware collateral. This should be kept at bare minimum and should not impact power and performance of a processor.

Maximized Performance with non-stop management: Benefits from frequency adjustments will be greatest when the frequency decrements are small and adjustment is continuous.

Self and Field Testing: Proposed scheme allows the hardware to be its own instrument and enables self test during field operation. The flexibility of software allows the system to adapt to the changing environment and invoke the device aging management at variable intervals. Thus, if the device was controlling a Mars Rover, it will continue to adjust its operating frequency and voltage without requiring a tester attached to it.

Crash Recovery and Workload Adaptation: The proposed management software provides checkpoint capabilities to enable system recovery while the system tests itself. Additionally, the real-time environment and varying workload demands are used to optimize their effects on the lifetime reliability.

In summary the vision of the proposed virtual framework for device aging management is to adjust the system as performance degrades over its lifetime, and provide a cost effective and flexible solution that scales for future technologies.

2. Modeling Lifetime Reliability

In this section we discuss models for lifetime reliability. We provide a brief background on lifetime reliability concepts. Then we discuss the failure mechanisms and models proposed in [3]. The NBTI model and reliability concepts form the basis of our work.

2.1 Lifetime Reliability Background

Processor lifetime can be expressed in mean-time-to-failure (MTTF). Typical designs target a MTTF of 30 years [3]. While this value may seem long for processors, which are typically replaced every few years, it is important to distinguish between the expected years of consumer use and the MTTF. The expected consumer use for a processor is 11 years [3]; the much longer MTTF ensures that the probability of failure during the expected use is small and in lies the tail end of the failure distribution. An alternative lifetime metric is failures-in-time (FIT), or the number of failures expected per billion hours. FIT relates to MTTF as:

$$FIT = 10^9 / MTTF$$

FIT is a convenient expression compared to MTTF because FIT values can be summed while MTTF cannot. A MTTF of 30 years can be expressed as about 4000 FIT.

Failures can occur in several components due to several mechanisms, as is discussed in the next subsection. These component failures are typically related to processor failure using the sum-of-failure-rates (SOFR) model, which assumes the first failure of any component under any mechanism causes the entire system to fail, that each failure mechanism is independent, and that each mechanism's failure rate is constant (i.e., not a function of time or the age of the processor). Using this model, the FIT of the processor can be computed by summing the FIT rates of each failure mechanism for each component.

Of course, actual failure rates are not constant, they increase with processor age. However, time-invariant failure models are commonly used due to their availability and simplicity.

2.2 RAMP: Failure Mechanisms and Model

Lifetime reliability is affected by five primary wear-out mechanisms expressed in the RAMP (Reliability Aware Microprocessor) model proposed in [3][30]: Electro-migration, Stress migration, Time-dependent dielectric breakdown, Thermal cycling, and NBTI. Electro-migration is the accumulation or depletion of interconnect material due to long-term current flow. Stress migration is the migration of interconnect material due to mechanical stress caused by differing thermal expansion rates of materials. Time-dependent dielectric breakdown is the formation of a conductive path in the nominally insulating gate-oxide of transistors. Thermal cycling is damage, particularly in the processor package, from repeated changes in temperature. Reference [30] provides a more detailed description of the above failure mechanisms.

For each failure mechanism, RAMP provides expressions proportional to the MTTF for each individual component. The MTTF can be expressed as a function of temperature – higher temperature and wider temperature swings generally cause more failures than other parameters such as voltage, frequency, and activity factor. The relevant equations all take this simplified form [30]:

$$MTTF = K * f(Temperature)$$

The proportionality constants (K) in these equations relate to the cost of "qualifying" the processor to achieve the

desired MTTF. For a system with the same target MTTF, a design with higher proportionality constants (K) survives more wear and incurs more expense for materials, testing, reliability analysis, and so on.

To relate easily-understandable architectural parameters to reliability cost, [3] uses a "qualification temperature," T_{qual}, as a proxy for cost and these proportionality constants. T_{qual} is a fixed, design-time parameter for a processor. A design with higher T_{qual} implies higher proportionality constants (i.e., K from above) and higher reliability cost. As in [3], for a given target MTTF, the proportionality constants (K) for a specific T_{qual} for each failure mechanism are computed by assuming a constant temperature of T_{qual} (using the technology's voltage and frequency values and worst-case activity factors for the functions that take those parameters). RAMP uses the proportionality constants computed for a T_{qual} to determine the observed MTTF based on observed processor temperature, voltage, frequency, and activity factor.

In this paper we primarily focus on NBTI as it has received a lot of recent attention. However, we note that the proposed scheme will work equally well for several other failure models.

3. System Reliability Manager

In this section we present the idea of a system reliability manager in the context of protection against device performance degradation caused by NBTI or similar physical causes. The core requirement for this manager is to sense the impact of power delivery, temperature and the workload on the hardware platform, and subsequently respond by reconfiguring the platform. The reconfiguration is primarily confined to the adaptation of supply voltage and/or operating frequency.

Pure hardware implementation of a reliability manager is costly and requires a priori information about the usage of a chip. On the other hand, pure software based approach needs instrumentation capabilities to address the issue of low level communication with the hardware. Additionally, operating system based implementation lacks flexibility due to strict interface abstractions to the hardware platform. These constraints drive us towards virtual management where the processor tests itself and finds its own frequency and voltage. An integral part of this system is crash recovery management that is built into the virtual layer.

The viability of a system reliability manager revolves around a cost-effective solution that can deliver self-testing and self-recovery capabilities in a flexible and scalable manner. In this section we describe this in detail. Our scheme has both hardware and software components. The hardware components are the knobs and their control mechanisms to adapt supply voltage and/or frequency to the changing reliability requirements [18]. The hardware platform also provides support for processor virtualization features like expanded isolation capabilities, and mechanisms for smooth and quick thread context switching capabilities [19].

The software component of our scheme is the device aging management software than runs natively as a guest privileged process on the hardware platform. We assume a thin Virtual Machine Monitor (VMM) running underneath the OS software stack, which is primarily used to enter and exit the System Reliability Manager (SRM) [19]. SRM software is concealed from all conventional software including the Operating System and may share the caching hierarchy of the platform for performance reasons. SRM software maintains a software timer for invocation control and crash recovery. SRM software also provides system checkpoint capabilities to enable self-testing capabilities without taking the system offline. Finally, the SRM software enables carefully crafted functional stress tests or built-in self-test control to identify degradation at a component granularity, and provides adjustments for sustained performance levels at target reliability. SRM software is akin to hypervisor that is commercially available [20].

3.1 SRM Architecture Framework

A high level system's view of the SRM architecture is shown in Figure 1. The SRM maintains a timer that is setup at chip initialization and then on every subsequent SRM exit. This timer is adjusted by the SRM to adjust its sampling to optimize the reliability requirements. When SRM is active, it has the highest privileged access to the hardware platform and the knobs to control supply voltage and operating frequency. The interface between SRM and the hardware platform is shown in Figure 1.

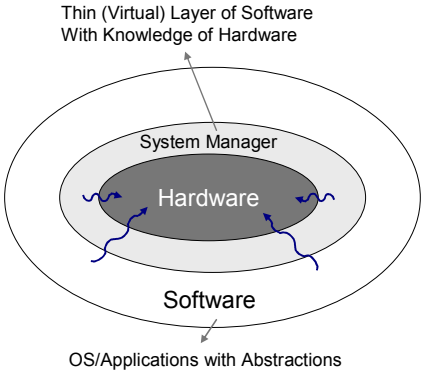


Figure 1. System Reliability Manager's System View

The Voltage Control Register (VCR) and the Frequency Control Register (FCR) are adjusted to control the hardware platform configuration. Once SRM software completes its work to determine the actions regarding device aging management, it exits via the VMM and passes control back to the Operating System. As a result, our approach delivers a hardware-software co-designed

solution that assists the hardware to dynamically adjust to tackle the reliability concerns over the chip lifetime.

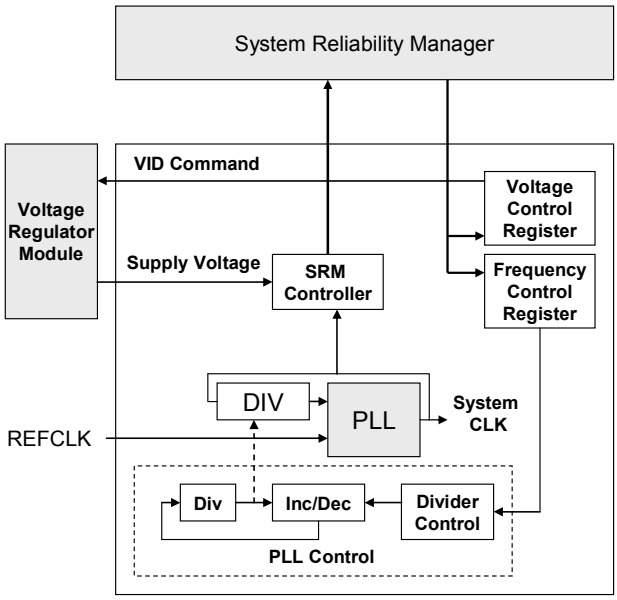


Figure 2. SRM Interface & Hardware View

3.2 SRM Software Flow

Figure 3 shows the flow diagram for the SRM software. Instead of using a worst-case guard-band over the entire lifetime of a design, the system starts off with the best-case frequency and voltage setting at first boot-up by invoking SRM. First invocation of SRM is specifically useful to calibrate a system to its power supply and cooling environment.

The steps for F_{MAX} testing are as follows:

i) Upon entry to SRM, all states are check pointed to ensure recovery from catastrophic system failure during testing. This includes the known operating F_{MAX}/V_{MIN} for system

ii) FCR is initialized to a low frequency value to set the frequency of the system. SRM timer is setup to enable self-recovery, and then test sequences are initiated

iii) If the test passes, FCR value is adjusted for a higher frequency, the timer is reset and test is rerun (back to step ii)

iv) If the test fails, upper limit on frequency is found

v) If the system hangs, the timer interrupts. This interrupt automatically updates FCR to the last good value and passes control back to SRM for system recovery

Once the F_{MAX} is found for a given V_{DD}, the SRM adds a small guard-band to last until the next invocation of SRM. It also schedules the timer for next invocation of SRM and exits by giving control back to the OS. SRM can be invoked during subsequent boot-ups or by request from system administrator. This is especially helpful when user/OS knowledge of system's usage and load can be used to invoke re-evaluation of the chip. Additionally, SRM timer can be setup based on product specification or some on-chip degradation sensing mechanism. For example, NBTI which is shown to have a large dependence on temperature can be analytically modeled in the SRM software, which can use the chip's thermal sensors to approximate the scheduling interval for re-evaluation. Additionally, if the system is expected to degrade 10MHz every month, the SRM timer can also be statically setup to re-evaluate monthly.

Similar set of steps can also be used to find V_{MIN} for a given frequency. The information about V_{MIN} is critical for correct operation of Dynamic Voltage and Frequency Scaling (DVFS) for thermal management [21].

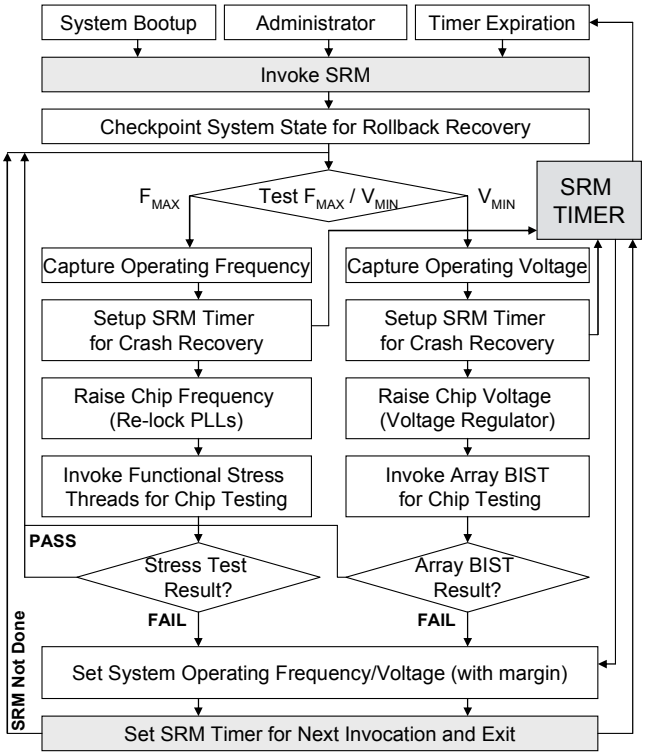


Figure 3. SRM Software Flow

3.3 Self-Testing Mechanisms

A key requirement for successful reconfiguration is complete knowledge about locations of failures and the nature of such failures. Our architecture framework offers low cost testing similar to the work presented in earlier research [16][17][22][23]. Instead of relying on costly and time consuming built-in structures our software based scheme offers comprehensive functional testing

framework. Based on our data analysis, presented in section 5, SRM is invoked at the granularity of weeks or days, so our methodology can use tests that run for longer durations (10s of ms). Figure 4 shows a flow diagram of the major components and their interactions for F_{MAX} testing. First phase involves carefully crafting software threads for the target system. In the second phase, these tests are compiled into SRM software, where code, data and exception handlers are setup along with routines for final result checking. During runtime, these test sequences are applied to the hardware platform as shown in Figure 3. Since these tests are run in the system environment unlike [23], the tests can make explicit external memory references.

Most of the modern designs come with lots of SRAM arrays. Due to a standardized structure of these arrays, built-in self-test (BIST) is commonly available on most designs with a diverse set of test vectors. Our framework provides a simple interface through SRM software to invoke these BIST engines and then check their results to determine a pass/fail for V_{MIN} testing.

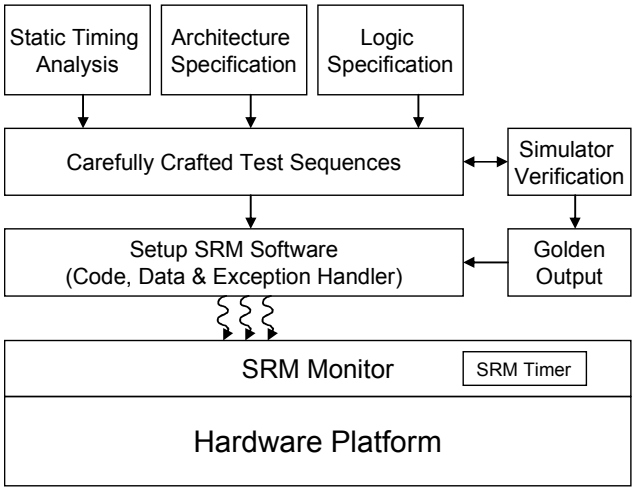


Figure 4. Functional F_{MAX} Testing Framework

3.4 Checkpoint and Crash Recovery

The main idea presented in this paper is to push the operating frequency and voltage to its limit, while the chip degrades during its lifetime. A major hurdle in such an architecture framework is that the system may crash during testing under such extreme operating conditions. The result of such a crash may range from incorrect results to a total system failure where a reset may be necessary. Our framework provides a cost-effective software-only mechanism to revert the system back to its pre-crash checkpoint of the system similar to SafetyNet [24] and ReVive [25].

Whenever SRM is invoked to find the optimal operating frequency and voltage, a system-wide checkpoint is initiated. The checkpoint includes the state of the core registers, memory values and coherence/communication messages. The core registers are explicitly check-pointed, while the memory/coherence state is logged whenever an action (store or a transfer of ownership) might have to be undone. Additionally, all components in the chip are coordinated such that a consistent checkpoint is taken and stored in the non-volatile memory. Now the SRM can start its path finding process as shown in Figure 3.

In case the SRM is invoked due to a crash, the system rollback process is initiated. The cores restore their register checkpoints and the caches/memories unroll their local logs to recover the system to the consistent global state at the pre-crash recovery point. After the recovery, the system resumes execution. As the SRM invocation is done infrequently, the cost of taking a checkpoint and rollback is negligible considering that it's a one time cost for each SRM invocation.

3.5 Self-Recovery Knobs

The knobs needed to adjust F_{MAX} and V_{MIN} at runtime are shown in Figure 1. For operating frequency adjustment the new frequency setting can be adjusted by re-locking the PLL to the required setting. Additionally, the operating voltage is adjusted by sending a command to the voltage regulator module (VRM) to adjust the chip voltage. The VRM subsequently returns a new supply voltage. The SRM provides a simple interface to the hardware platform to request changes to the operating frequency and voltage.

4. Experimental Methodology

In this section we discuss our simulation environment. We use SESC cycle-level MIPS simulator for developing the SRM framework [26]. We have extended SESC to invoke Wattch [27] and Cacti [28] power estimation tools, and HotSpot temperature modeling tool [29]. For evaluating processor lifetime reliability at runtime, we integrated the RAMP model [30] in our simulator. Although RAMP provides analytical models for five intrinsic failure mechanisms, we only use NBTI in this study. We model a single superscalar processor with a floorplan containing twenty two structures. System parameters used are shown in TABLE I .

The NBTI model used in RAMP is based on recent work by Zafar et al. at IBM [4]. This model shows that NBTI has a strong dependence on temperature in addition to electric field. The temperature and average MTTF is tracked for each structure in the processor over the entire simulation run. Our framework assumes that the first instance of any structure failing causes the entire processor to fail.

Processor Parameters	
Fetch, Issue, Retire Width	6, 4, 4 (out-of-order)
L1	64KB 4-way I & D, 2 cycles
L2	2M 8-way shared, 10 cycles
ROB Size, LSQ	152, 64
Off-chip memory latency	200 cycles

Hotspot Parameters	
Ambient Temperature	45°C
Package Thermal Resistance	0.8 K/W
Die Thickness	0.5 mm
Maximum Temperature	85°C
Temperature Sampling Interval	10,000 cycles

RAMP Parameters	
Qualification Temperature per Structure	82°C
RAMP Sampling Interval	10,000 cycles

TABLE I . System Parameters

For our analysis, we chose SPEC2000 benchmarks. The choice of benchmark phases is primarily based on their thermal behavior with mcf being *cold*, gcc, gzip, ammp being *moderate*, and vortex, equake, art, bzip2 being *hot*. Each benchmark is fast forwarded 2 billion instructions, followed by HotSpot and RAMP initialization for each structure. This ensures that the processor as well as HotSpot and RAMP model get sufficient warm up.

We assume 65nm technology with chip wide maximum V_{DD} of 1.1V, and frequency of 2.0 GHz. For SRM evaluation, we vary Vdd and frequency by 5% downward steps up to a minimum of 0.88V and 1.6 GHz. For each benchmark, twenty five simulations are conducted with a pre-determined frequency/voltage setting. Each simulation is run for 1 billion instructions and performance evaluated based on throughput. At the end of each simulation, average MTTF per structure is sorted for each structure and the worst case MTTF is reported. For analysis purposes, we use an MTTF of 1 year, while we realize that expected consumer use for a processor is 11 years. Our results should hold for an 11 year MTTF as well.

5. Data Analysis

Figure 5 shows the impact of varying frequency at a given voltage setting for the hottest structure in the bzip2 benchmark. As the frequency is scaled down, the benchmark's performance degrades, while the temperature falls. On the other hand, Figure 6 shows that impact of voltage on temperature, assuming the chip remains functional. We assume that initially the chip is fully functional at 2.0GHz and 1.1V, which implies that for this voltage, frequencies below 2.0GHz are allowed. If the voltage is lowered, the maximum operating frequency will degrade. SRM on its invocation iteratively evaluates for the maximum possible operating frequency under a specified operational voltage. The key question is: What is the scheduling interval for SRM?

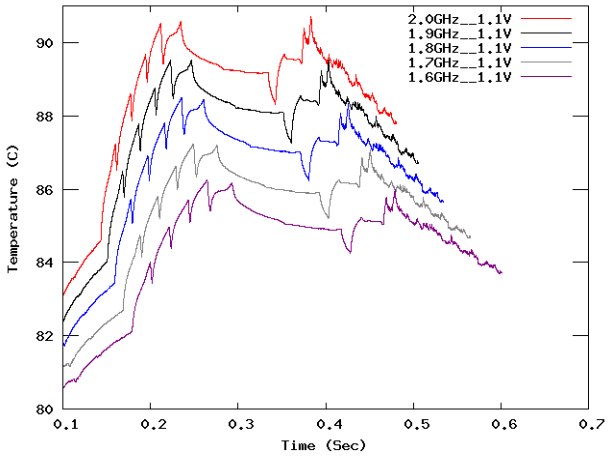


Figure 5. Scaling Freq. for bzip2: Thermal profile for IntReg

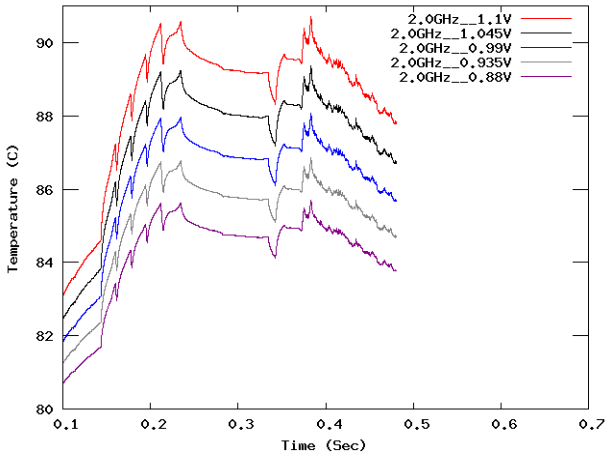


Figure 6. Scaling voltage for bzip2: Thermal profile for IntReg

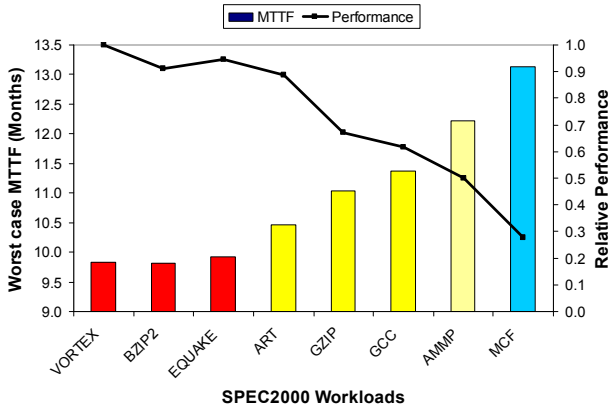


Figure 7. Lifetime Reliability Tradeoffs for SRM

Figure 7 plots the performance and the worst case MTTF for all benchmarks when simulation is run at 2GHz and 1.1V. It can be concluded from our simulations that the

performance (IPC or throughput) of a benchmark directly impacts the worst case MTTF for the chip. How quickly one can expect a failure to occur is dependent on the workload. So, a mechanism that keeps track of the performance of each live thread in the system is desirable for tuning the scheduling algorithm for the SRM.

Figure 8 and Figure 9 show the impact of chip's lifetime degradation when voltage or frequency is scaled down. The rate of change in MTTF is linear and its slope is dependent on the type of benchmark. This data shows that workload's thermal and performance behavior can be used as a metric to track the rate of change in the MTTF.

We also observe from this data that even though we designed our system to sustain MTTF of 1 year with a qualification temperature of 82°C for each structure, the worst case MTTF can be better than expected. For example, mcf benchmark has the worst case temperature of ~80°C, which results in no expected degradation for the 1 year period. Hence, if the system only runs under a similar workload conditions, the SRM scheduling is not needed for the 1 year period.

On the other hand, if vortex or similar thermally hot workloads are being run on the system for the prescribed period, initially a monthly re-evaluation will suffice and once the system starts degrading and the operational settings have to be changed, re-evaluation can be scheduled for twice a week.

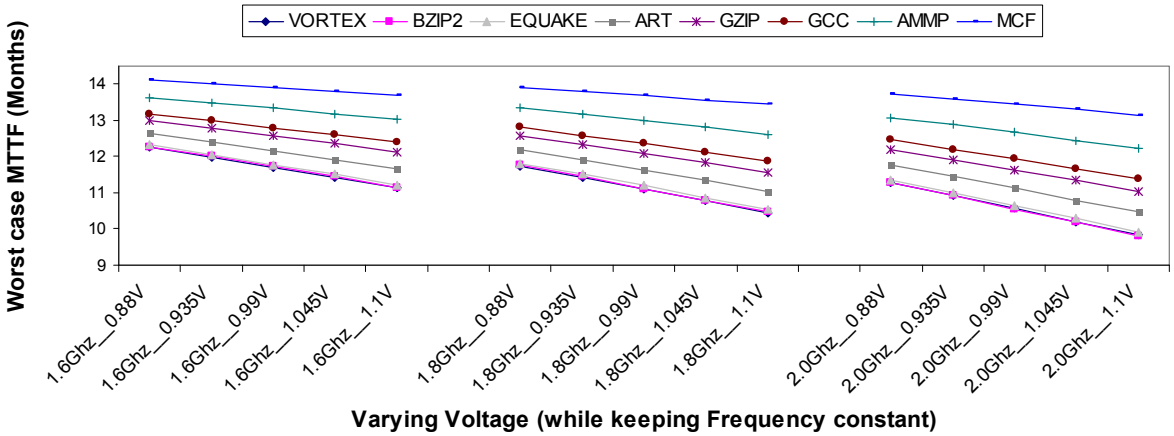


Figure 8. Impact of varying Voltage across benchmarks

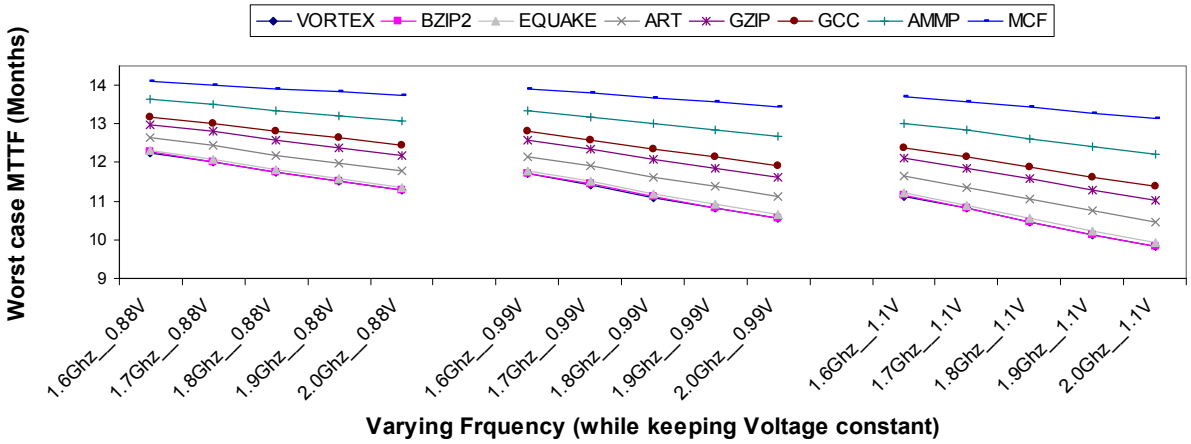


Figure 9. Impact of varying Frequency across benchmarks

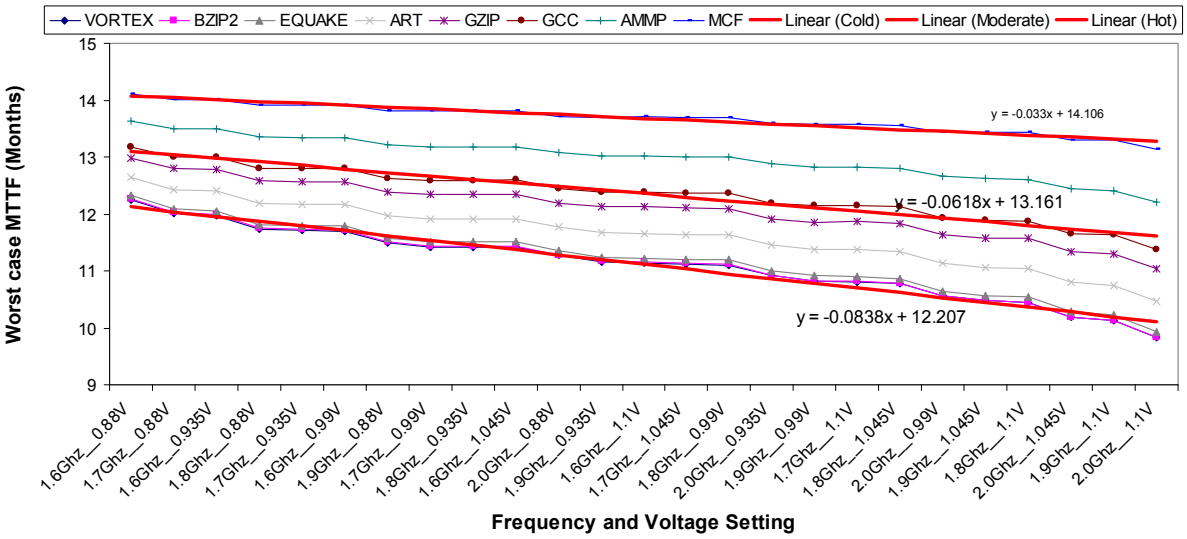


Figure 10. Rate of change for worst case MTTF as a function of benchmarks and operating conditions

Figure 10 shows the worst case MTTF degradation for each benchmark under all operating conditions (sorted by MTTF) considered in this study. The x-axis shows that all benchmarks can achieve an MTTF of 1 year if the system constantly operates at 1.6GHz and 0.88V, but this comes at the cost of performance. On the other hand, if the operating conditions are initially set to 2GHz and 1.1V, the system can be operational for most of its lifetime. SRM can be invoked at regular intervals to adjust the frequency downwards and keep the system operational. Additionally, as the system starts deteriorating, the history can be recorded to guide the fine-grain scheduling interval for SRM. This is not described here for the sake of brevity of this paper.

6. Conclusions

We have presented a novel device aging management scheme for continuous adjustment of frequency and minimum supply voltage based on a co-designed virtual machine. The scheme requires no tester for determining F_{MAX} and V_{MIN}. Hardware collateral to implement this scheme is minimal that includes instructions for updating frequency and voltage control registers, which are already found in modern processor systems. The proposed solution allows the hardware to be its own instrument and enables self test during field operation by guiding the system to crash and recover during adjustment of its operating conditions. By insulating the device aging management from conventional software, the proposed framework shields the system and application software from managing low level details. The flexibility of software allows the system to adapt to the changing environment and invokes device aging management at appropriate intervals. The greatest benefits of this approach are (i) device operation near peak frequency throughout product life and (ii) protection against failure due to insufficient lifetime guard-band, (iii) no system downtime or change from a user perspective.

Acknowledgments

This material is based upon work supported by the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

We thank Michael Powell and Krishna Rangan of Intel Corporation for their insightful discussions and feedback. We would also like to thank David Albonesi and Paula Petrica of Cornell University and Pradip Bose of IBM Corporation for supporting the integration of RAMP reliability model in our simulation framework.

References

[1] International Technology Roadmap for Semiconductors (ITRS). Document available at *http://public/itrs.net/*

[2] S.Y. Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation", In IEEE Micro, Vol. 25, Issue 6, Nov.–Dec. 2005

[3] J. Srinivasan et al., "The case for lifetime reliability-aware microprocessors", In Int'l Symposium on Computer Architecture, June 2004

[4] S. Zafar et al., " A Model for Negative Bias Temperature Instability (NBTI) in Oxide and High K PFETs", In Symposia VLSI Technology and Circuits, 2004

[5] W. Abadeer, W. Ellis, "Behavior of NBTI under AC Dynamic Circuit Conditions", In Int'l Reliability Physics Symposium, 2003

[6] M. Agostinelli et al., "Erratic Fluctuations of SRAM Cache Vmin at the 90nm Process Technology Node", In Electron Devices Meeting (IEDM), 2005
[7] V. Reddy et al., "Impact of Negative Bias Temperature Instability on Digital Circuit Reliability", In Intl. Reliability Physics Symposium, 2002
[8] S. Kundu et al., "Trends in manufacturing test methods and their implications", In Intl. Test Conference, pp. 679-687, 2004
[9] J. Abella et al., "Penelope: The NBTI-Aware Processor", In Int'l Symposium on Microarchitecture, 2007
[10] S. Mitra, E. J. McClusky, "WORD VOTER: A New Voter Design for Triple Modular Redundancy Systems", In Symposium VLSI Test., 2000
[11] T. Lin et al., "Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis", In IEEE Transactions on Reliability, Oct. 1990
[12] A. Tiwari et al., "ReCycle: pipeline adaptation to tolerate process variation", In Int'l Symposium on Computer Architecture, 2007
[13] J. Srinivasan et al., "Lifetime Reliability: Toward an Architectural Solution", In Int'l Symposium on Computer Architecture, May 2005
[14] J. Srinivasan et al., "The Impact of Technology Scaling on Lifetime Reliability", In Intl. Conference on Dependable Systems and Networks, 2004
[15] S. Das et al., "Razor: A self-tuning DVS processor using delay-error detection and correction", In Symposium on VLSI Circuits, 2005
[16] K. Constantinides, O. Mutlu, T. Austin, V. Bertacco, "Software-Based Online Detection of Hardware Defects: Mechanisms, Architectural Support and Evaluation", In Int'l Symposium on Microarchitecture, 2007
[17] J. Smolens et al., "Detecting Emerging Wearout Faults", In IEEE Workshop on Silicon Errors in Logic – System Effects, 2007
[18] J. Tschanz et al., "Adaptive frequency and biasing techniques for tolerance to dynamic temperature-voltage variations and aging", In Int'l Solid State Circuits Conference, 2007
[19] J. Smith, R. Nair, "Virtual Machines: Versatile Platforms for Systems and Processes", Morgan Kaufmann Pub, 2005
[20] "IBM Systems Virtualization", IBM Corp., Ver.2 Rel.1, 2005
[21] S. Naffziger et al., "Power and Temperature Control on a 90nm Itanium®-Family Processor", In Int'l Solid State Circuits Conference, 2005
[22] Y. Li, S. Makar, S. Mitra, "CASP: Concurrent Autonomous Chip Self-Test Using Stored Test Patterns", In Design, Automation and Test in Europe, 2008
[23] P. Parvathala et al., "FRITS – A Microprocessor Functional BIST Method", In Int'l Test Conference, 2002
[24] D. J. Sorin, et al., "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery", In Int'l Symposium on Computer Architecture, 2002
[25] M. Prvulovic et al., "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors", In Int'l Symposium on Computer Architecture, May, 2002
[26] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC Simulator", 2005*; http://sesc.sourceforge.net*
[27] D. Brooks et al., "Wattch: A framework for architectural-level power analysis and optimizations", In Int'l Symposium on Computer Architecture, 2000
[28] P. Shivakumar and N. P. Jouppi, "CACTI 3.0: An integrated cache timing, power, and area model", WRL Technical Report, Compaq, 2001
[29] K. Skadron et al., "HotSpot: Techniques for Modeling Thermal Effects at the Processor-Architecture Level", In THERMINIC, 2002
[30] J. Srinivasan et al., "RAMP: A Model for Reliability Aware Microprocessor Design", IBM Research Report, Dec. 2003

Exploiting Value Prediction for Fault Tolerance

Xuanhua Li and Donald Yeung
Department of Electrical and Computer Engineering
Institute for Advanced Computer Studies
University of Maryland at College Park

Abstract

Technology scaling has led to growing concerns about reliability in microprocessors. Currently, fault tolerance techniques rely on explicit redundant execution for fault detection or recovery which incurs significant performance, power, or hardware overhead. This paper makes the observation that value predictability is a low-cost (albeit imperfect) form of program redundancy that can be exploited for fault tolerance. We propose to use the output of a value predictor to check the correctness of predicted instructions, and to treat any mismatch as an indicator that a fault has potentially occurred. On a mismatch, we trigger recovery using the same hardware mechanisms provided for mispeculation recovery. To reduce false positives that occur due to value mispredictions, we limit the number of instructions that are checked in two ways. First, we characterize fault vulnerability at the instruction level, and only apply value prediction to instructions that are highly susceptible to faults. Second, we use confidence estimation to quantify the predictability of instruction results, and apply value prediction accordingly. In particular, results from instructions with higher fault vulnerability are predicted even if they exhibit lower confidence, while results from instructions with lower fault vulnerability are predicted only if they exhibit higher confidence. Our experimental results show such selective prediction significantly improves reliability without incurring large performance degradation.

1 Introduction

Soft errors are intermittent faults caused by cosmic particle strikes and radiation from packaging materials. They do not cause permanent damage, but still corrupt normal program execution. Technology scaling combined with lower supply voltages make systems more vulnerable to soft errors. Hence, soft errors have become an increasingly important design consideration with each successive generation of CPUs.

To enhance system reliability, existing techniques typically introduce redundant execution–by taking either a hardware or software approach–to detect or recover from faults. On the hardware side, error-detection circuitry (ECC or parity bits) can be added to storage structures. Other hardware techniques utilize additional structures such as extra processor cores, hardware contexts, or functional units [1, 2, 3, 4, 5] to execute redundantly in order to compare results and detect faults. In contrast to hardware techniques, software-based techniques rely on the compiler to duplicate program code [6, 7, 8]. This software redundancy also permits comparison of results at runtime, but without any additional hardware cost. While differing in implementation, both hardware and software approaches create explicit redundancy to provide fault tolerance which incurs significant performance, power, or hardware overhead.

Prior studies have shown that program execution itself contains a high degree of redundancy–*i.e.*, instruction and data streams exhibit repeatability. One example of exploiting such inherent redundancy is value prediction which predicts instruction results through observation of past values. By predicting values before they are executed, data dependency chains can be broken, permitting higher performance. Unfortunately, value prediction has had limited success in commercial CPUs due to its relatively low prediction accuracy and high misprediction penalty, the latter becoming increasingly severe with deeper processor pipelines.

In this work, we employ value prediction to improve system reliability. Compared to explicit redundant execution techniques, the advantage of value prediction is it exploits programs' inherent redundancy, thus avoiding the cost of explicitly duplicating hardware or program code as well as the associated area, power, and performance overheads. Although a value predictor itself incurs some addi-

tional hardware, we find a relatively small predictor can effectively detect faults; hence, our approach incurs less hardware than traditional explicit duplication techniques.

In addition to exploiting inherent program redundancy, another advantage of our approach is it is less sensitive to the negative effects of misprediction. Mispredictions are always undesirable from the standpoint of performance since they require flushing the pipeline. Because traditional uses of value prediction are focused on improving performance, such flushes undermine their bottom line. However, in the context of fault detection/recovery, flushes can be desirable because they reduce the time that program instructions (particularly those that are stalled) spend in the pipeline, thus improving architectural vulnerability. Rather than always being undesirable, for our technique, mispredictions represent a tradeoff between performance and reliability. Lastly, compared to traditional uses of value prediction, our technique does not require as fast value predictors. For performance-driven techniques, value predictions are needed early in the pipeline. In contrast, for fault detection/recovery, value predictions can be delayed until the writeback stage, where value checking occurs.

To maximize the efficacy of our technique, we focus value prediction only on those instructions that receive the greatest benefit. In particular, we characterize fault vulnerability at the instruction level, and apply value prediction only to those instructions that are most susceptible to faults.[1] An instruction's fault vulnerability in a specific hardware structure is quantified by measuring the fraction of the structure's total AVF (Architectural Vulnerability Factor) that the instruction accounts for. Our results show a small portion of instructions account for a large fraction of system vulnerability. For example, for the fetch buffer in our processor model, about 3.5% of all instructions are responsible for 53.9% of the fetch buffer's total AVF in the TWOLF benchmark. This suggests that selectively protecting a small number of instructions can greatly enhance the overall reliability. Because we apply value prediction only on a small number of instructions, the potential performance loss due to mispredictions is also quite small.

To further reduce the impact of mispredictions, we use an adaptive confidence estimation technique to assess the predictability of instructions, and apply prediction accordingly. Our approach is adaptive because it applies prediction more or less aggressively depending on each instruction's fault vulnerability (which can be quantified through its latency). Instructions with high fault vulnerability are predicted even if they exhibit low confidence, while instructions with low fault vulnerability are predicted only if they exhibit high confidence. Our results show that this technique achieves significant improvements in reliability without sacrificing much on performance.

The rest of the paper is organized as follows. Section 2 introduces how we apply value prediction for fault detection. We mainly discuss our study on characterizing instructions' vulnerability to faults, as well as our methods for selecting instructions for fault protection. Then, Section 3 describes our experimental methodology, and reports on the reliability and performance results we achieve. Finally, Section 4 presents related work, and Section 5 concludes the paper.

2 Reducing Error Rate with Value Prediction

This section describes how value prediction can be used to reduce error rate. First, Section 2.1 discusses how we use value predictors to check instruction results. Then, Section 2.2 briefly describes fault recovery. Finally, Section 2.3 quantifies instructions' vulnerability to faults, and proposes selectively predicting instructions to mitigate performance loss.

2.1 Predictor-Based Fault Detection

To identify potential faults, we use a value predictor to predict instruction outputs. We employ a hybrid predictor composed of one stride predictor and one context predictor [9]. Prediction from the context predictor is attempted first. If the context predictor cannot make a prediction (see Section 3.1), then the stride predictor is used instead to produce a result. After a prediction is made, the result is compared with the actual computation result. The comparison is performed during the instruction's writeback stage, so the predictor's output is not needed until late in the pipeline. Since prediction can be initiated as soon as the instruction is fetched, there is significant time for the predictor to make its prediction, as mentioned in Section 1.

[1] Identifying the most vulnerable instructions occurs late in the pipeline. Thus for implementation with more advanced but slower value predictor, all result-producing instructions are eligible for prediction once they enter the pipeline, but only those that are later identified as the most susceptible to faults will have their results checked and update the predictor.

During each predictor comparison, the prediction and actual instruction result will either match or differ. If they match, two interpretations are possible. First, the predictor predicted the correct value. In this case, no fault occurred since the instruction also produced the same correct value. Second, the predictor predicted the wrong value, but a fault occurred such that the instruction produced the same wrong value. This case is highly unlikely, and for all practical purposes, will never happen. Hence, on a match, we assume no fault has occurred, and thus, no additional action is required.

Another possibility is the prediction and actual instruction result differ. Again, two interpretations are possible. First, the predictor predicted the correct value. In this case, a fault has occurred since the instruction produced a different value. Second, the predictor predicted the wrong value, and the instruction either produced a correct or wrong value (again, we assume a misprediction and incorrect result will never match). Unfortunately, there is no way to tell which of these has occurred, so at best on a mismatch, we can only assume that there is the *potential* for a fault. We always assume conservatively that a fault has occurred, and initiate recovery by squashing the pipeline and re-executing the squashed instructions in the hopes of correcting the fault. During re-execution, if the instruction produces the same result, then with high probability the original instruction did not incur a fault.[2] If no fault occurred (the most likely case), the pipeline flush was unnecessary, and performance is degraded. (However, as we will see in Section 2.2, such "unnecessary" flushes can actually improve reliability in many cases).

To mitigate the performance degradation caused by false positives, we use confidence estimation. In particular, we employ the confidence estimator described in [10]. We associate a saturating counter with each entry in the value predictor table. A prediction is made only when the corresponding saturating counter is equal to or above a certain threshold. If the prediction turns out to be correct (the match case), the saturating counter is incremented by some value. If the prediction turns out to be incorrect (the mismatch case in which the original and re-executed results are the same), the saturating counter is decremented by some value. Given confidence estimation, we can tradeoff the number of false positives with the number of predicted instructions (and hence, the fault coverage) by varying the confidence threshold. Section 3 will discuss how we select confidence thresholds.

[2]The comparison of a re-executed result with the originally executed result is not necessary on-line for our technique to work properly. In fact, our technique never knows whether a mismatch was caused by a misprediction or an actual fault. The main issue with re-execution is predictor updates, which is discussed in Section 3.1.

2.2 Fault Recovery

When an instruction's prediction differs from its computed value, it is possible a fault occurred before or during the instruction's execution. To recover from the fault, it is necessary to roll back the computation prior to the fault, and re-execute. In our work, we perform roll back simply by flushing from the pipeline the instruction with the mismatch as well as all subsequent instructions. Then, we refetch and re-execute from the flush point. (A similar mechanism for branch misprediction recovery can be used for our technique).

Notice, our simple approach can only recover faults that attack predicted instructions, or instructions that are downstream from a mispredicted instruction (which would flush not only the mispredicted instruction, but also all subsequent instructions). If a fault attacks a non-predicted instruction that is not flushed by an earlier mispredicted instruction, then even if the fault propagates to a predicted instruction later on, recovery would not roll back the computation early enough to re-execute the faulty instruction. However, even with this limitation, we find our technique is still quite effective.

Because soft errors are rare, most recoveries are triggered by the mispredictions of the value predictor. As mentioned in Section 2.1, such false positives can degrade performance. However, they can also improve reliability. Often times, re-executed instructions run faster than the original instructions that were flushed (the flushed instructions can prefetch data from memory or train the branch predictor on behalf of the re-executed instructions). As a result, the re-executed instructions occupy the instruction queues for a shorter amount of time, reducing their vulnerability to soft errors compared to the original instructions. This effect is particularly pronounced for instructions that stall for long periods of time due to cache misses. Hence, while false positives due to mispredictions can degrade performance, this degradation often provides a reliability benefit in return. The next section describes how we can best exploit this tradeoff.

2.3 Instruction Vulnerability

In order to reduce the chance of mispredictions and unnecessary squashes, we not only apply confi-

dence estimation (as described in Section 2.1), but we also limit value prediction to those instructions that contribute the *most* to overall program reliability. This section describes how we assess the reliability impact of different instructions.

Recently, many computer architects have used Architectural Vulnerability Factor (AVF) to reason about hardware reliability [11]. AVF captures the probability that a transient fault in a processor structure will result in a visible error at a program's final outputs. It provides a quantitative way to estimate the architectural effect of fault derating. To compute AVF, bits in a hardware structure are classified as critical for architecturally correct execution (ACE bits), or not critical for architecturally correct execution (un-ACE bits). Only errors in ACE bits can result in erroneous outputs. A hardware structure's AVF is the percentage of ACE bits that occupy the hardware structure on average.

To identify ACE bits, instructions themselves must first be distinguished as ACE or un-ACE. We make the key observation that not all ACE instructions contribute equally to system reliability. Instead, each ACE instruction's occupancy in hardware structures determines its reliability contribution. As observed by Weaver *et al.* [12], the longer instructions spend in the pipeline, the more they are exposed to particle strikes, and hence, the more susceptible they become to soft errors. Weaver *et al.* proposed squashing instructions that incur long delays (*e.g.*, L2 cache misses) to minimize the occupancy of ACE instructions. We extend this idea by quantifying fault vulnerability at the instruction level, and selectively protecting the instructions that are most susceptible to faults.

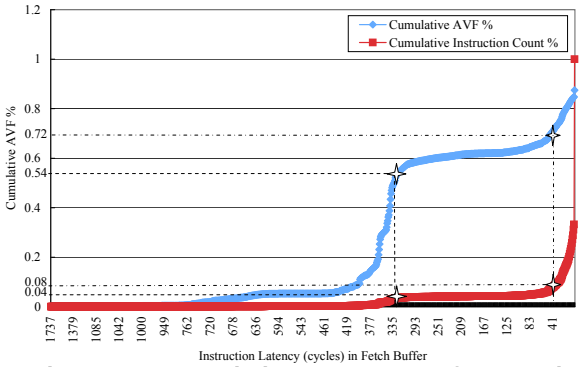


Figure 1. Accumulative Percentage of AVF and Instruction Count in Fetch Buffer on TWOLF.

Our approach is particularly effective because we find a very small number of instructions account for a majority of the AVF in hardware structures. Figure 1 illustrates this for the processor's fetch buffer when executing TWOLF, a SPEC2000 benchmark. In Figure 1, the top curve plots the cumulative fraction of overall AVF (y-axis) incurred by instructions that occupy the fetch buffer for different latencies (x-axis) sorted from highest latency to lowest latency. The bottom curve plots the cumulative fraction of dynamic instructions (y-axis) that experience the given latencies. In total, there are 1,944 static instructions that have been simulated. As Figure 1 shows (the two datapoints marked on the left of the graph), 53.9% of the fetch buffer's AVF is incurred in 3.5% of all dynamic instructions. These instructions have large latencies–300 cycles or more. As indicated by the other two datapoints marked on the right side of the graph, the majority of instructions (about 91.8%) exhibit a latency smaller than 40 cycles, and account for a relatively small portion of the overall AVF (about 28.4%). We find similar behavior occurs for the other benchmarks as well as for the other hardware structures. Such results show that using our value predictor to target a small number of instructions–those with very large latencies–is sufficient to provide the majority of fault protection. This is good news since it will minimize the performance impact of mispredictions.

In our study, we find that even though an instruction may stall for a long time in one hardware structure, it may not stall for very long in other structures. In other words, a single instruction can contribute differently to different structures' vulnerability. Thus, an important question is how can we select the smallest group of instructions that will provide the largest benefit to reliability? In our work, we measure the latency an instruction incurs from the fetch stage to the issue stage, and use this to determine each instructions' contribution to reliability, applying value prediction only to those instructions that meet some minimum latency threshold. Because our approach accounts for "front-end" pipeline latency, we directly quantify the occupancy of instructions in the fetch and issue queues, and hence, are able to identify the instructions that contribute the most to reliability in these 2 hardware structures. This is appropriate for our work since later on (in Section 3) we study our technique's impact on both fetch and issue queue reliability (we also study the impact on the physical register file's reliability, though our latency metric does not directly quantify result occupancy in this structure). If improving reliability in other hardware structures is desired, it may be necessary to use a different latency metric. This is an important direction for future work.

Processor Parameters	
Bandwidth	8-Fetch, 8-Issue, 8-Commit
Queue size	64-IFQ, 40-Int IQ, 30-FP IQ, 128-LSQ
Rename reg/ROB	128-Int, 128-FP / 256 entry
Functional unit	8-Int Add, 4-Int Mul/Div, 4-Mem Port 4-FP Add, 2-FP Mul/Div
Branch Predictor Parameters	
Branch predictor	Hybrid 8192-entry gshare/2048-entry Bimod
Meta table	8192 entries
BTB/RAS	2048 4-way / 64
Memory Parameters	
IL1 config	64kbyte, 64byte block, 2 way, 1 cycle lat
DL1 config	64kbyte, 64byte block, 2 way, 1 cycle lat
UL2 config	1Mbyte, 64byte block, 4 way, 20 cycle lat
Mem config	300 cycle first chunk, 6 cycle inter chunk
Hybrid Value Predictor Parameters	
VHT size	1024
value history depth	4
PHT size	1024
PHT counter thresh	3

Table 1. Parameter settings for the detailed architectural model used in our experiments.

3 Experimental Evaluation

In Section 2, we showed a small number of instructions account for a large portion of hardware vulnerability. We also qualitatively analyzed the impact of pipeline flushes: flushing degrades performance, but in some cases may improve program reliability. We consider both findings in our design, and use insights from both to drive confidence estimation (which ultimately determines which instructions will be predicted).

This section studies these issues in detail. First, we present the simulator and benchmarks used throughout our experiments (Section 3.1). Then, we present our experiments on applying value prediction without confidence estimation. (Section 3.2). The goal of these experiments is to show that we can limit performance degradation by focusing the value predictor on the portion of instructions that impact system reliability the most. Finally, we add confidence estimation, and show the improvements this can provide (Section 3.3).

3.1 Simulator and Benchmarks

Throughout our experiments, we use a modified version of the out-of-order processor model from Simplescalar 3.0 for the PISA instruction set [13], configured with the simulator settings listed in Table 1. Our simulator models an out-of-order pipeline consisting of fetch, dispatch, issue, execute, writeback, and commit pipeline stages. Compared to the original, our modified simulator models rename registers and issue queues separately from the Register Update Unit (RUU). We also model a hybrid value predictor that includes a single stride predictor and a single context predictor, as described in [9]. The value predictor configuration is shown in Table 1.

Our stride predictor contains a Value History Table (VHT). For each executed instruction, the VHT maintains a last-value field (which stores the instruction's last produced value) and a stride field. When a new instance of the instruction is executed, the difference between the new value and the last-value field is written into the stride field, and the new value itself is written into the last-value field. If the same stride value is computed twice in a row, the predictor predicts the instruction's next value as the sum of the last-value and stride fields. When a computed stride differs from the previously computed stride, the predictor stops making predictions until the stride repeats again.

Our context predictor is a 2-level value predictor consisting of a VHT and a Pattern History Table (PHT). For each executed instruction, the VHT maintains the last history-depth number of unique outcomes produced by the instruction (we employ a history depth = 4). In addition, the VHT also maintains a bit field that encodes the pattern in which these outcomes occurred during the last pattern-length dynamic instances of the instruction (we employ a pattern length = 4). During prediction, the instruction's bit field is used to index the PHT. Each PHT entry contains several frequency counters, one for each instruction outcome in the VHT. The counter with the highest count indicates the most frequent successor value given the instruction's current value pattern. If this maximum count is above some threshold (we employ a threshold = 3), then the corresponding outcome is predicted for the instruction; otherwise, no prediction is made. After an instruction executes and its actual outcome is known, the corresponding PHT entry counter is incremented by 3 while the other counters from the same PHT entry are decremented by 1. Lastly, the corresponding bit field in the VHT is updated to reflect the instruction's new outcome pattern.

For some of our experiments (*e.g.*, Section 3.3), we employ confidence estimation along with value prediction. As discussed in Section 2.1, we use the confidence estimator described in [10] which associates a 4-bit saturating counter with each PHT entry. Update to all predictor structures (stride, context, and confidence estimator) only occurs on predicted instructions. In our technique, many instructions are not predicted because their latencies are short, making them less important to overall reliability. These non-predicted instructions do not up-

Benchmark	Input	Instr Count	IPC
300.twolf	ref	109546670	0.79
176.gcc	166.i	240000000	1.42
254.gap	train.in	411061781	1.65
164.gzip	input.compressed	192015257	2.06
256.bzip2	input.compressed	2346534735	3.20
253.perlbmk	diffmail.pl	1000000000	1.57
197.parser	ref.in	1404572471	1.32
181.mcf	inp.in	500000000	0.13
175.vpr	test	1512992144	1.87

Table 2. Benchmarks and input datasets used in our experiments. The last two columns report instructions executed and baseline IPC for each benchmark.

date the predictor structures. During re-execution after a misprediction, the CPU will likely re-execute the mispredicted instruction, and the predictor may predict again.[3] In this case, the predictor is very likely to generate a correct prediction due to training from the misprediction. In any case, we still update the predictor after the prediction (*i.e.*, predictor updates do not distinguish between the first execution of some instruction and its re-execution after a misprediction).

In terms of timing, our simulator assumes the stride and context predictors can always produce a prediction by each instruction's writeback stage. We believe this is reasonable given the small size of our predictor structures in Table 1. In particular, our predictors are either smaller than or equal to the value predictors found in the existing literature for performance enhancement [10, 14, 9]. Since our technique is not as timing critical (conventional value predictors must make predictions by the issue stage), we believe there will not be any timing-related problems–both in terms of latency and bandwidth–when integrating our predictors into existing CPU pipelines. On a misprediction, our simulator faithfully models the timing of the subsequent pipeline flush as well as the cycles needed to re-fetch and re-execute the flushed instructions. Our simulator also assumes a 3-cycle penalty from when a misprediction is detected until the first re-fetched instruction can enter the pipeline.

Table 2 lists all the benchmarks used in our experiments. In total, we employ 9 programs from the SPEC2000 benchmark suite. All of our benchmarks are from the integer portion of the suite; we did not study floating-point benchmarks since our value predictors only predict integer outcomes. In Table 2, the column labeled "Input" specifies the input dataset used for each benchmark, and the column labeled "Instr Count" reports the number of instructions executed by each benchmark. The last column, labeled "IPC," reports each benchmark's average IPC without value prediction. The latter represents baseline performance from which the IPC impact of our technique is computed.

Finally, throughout our experiments, we report both performance and reliability to investigate their tradeoff. In particular, we measure IPC for performance and AVF for reliability. We analyze reliability for three hardware structures only–the fetch queue, issue queue, and physical register file. Since we use value prediction to perform fault checking on architectural state at writeback, we can detect faults that attack most hardware structures in the CPU, including functional units, the reorder buffer, etc. But our results do not quantify the added protection afforded to structures outside of the three we analyze. Furthermore, we do not analyze reliability for the value predictors themselves. Predictors do not contain ACE bits; however, soft errors that attack the value predictors could cause additional mispredictions and flushes that can impact both performance and reliability. Again, our results do not quantify these effects. Lastly, our technique incurs additional power consumption in the value predictor tables. Since we do not model power, our results do not quantify these effects. However, we believe the power impact will be small given the small size of our predictors. Furthermore, given their relaxed timing requirements, there is room for voltage scaling optimizations to minimize the power impact.

3.2 Value Prediction Experiments

We first present our experiments on applying value prediction without confidence estimation. We evaluate the impact on both reliability and performance when predicting all or a portion of the result-producing instructions. We call these full and selective prediction, respectively. For selective prediction, we predict instructions based on their latency measured from the fetch stage to the issue stage. Since we do not know if an instruction should be predicted when we fetch it, we initiate prediction for all result-producing instructions upon fetch, but only perform fault checking and predictor updates for those instructions that meet the latency thresh-

[3] With confidence estimation, this will not happen because the original misprediction would lower the confidence value for the re-executed instruction enough to suppress prediction the second time around. But without confidence estimation, prediction during re-execution can happen.

	twolf	gcc	gap	gzip	bzip2	perl	parser	mcf	vpr
1.	12283990	32105002	43971177	6637540	70107090	69058496	130089349	21126931	240467394
2.	4323690	18770890	20481573	1973416	12948479	24132481	53091620	8499490	98006932
3.	717996	430799	4417390	100555	21505023	9639124	1611219	353809	12920875
4.	986816	96830	996890	382	3960326	2124938	2446684	7411123	74132

Table 3. Number of mispredictions for the 1. "Total," 2. "lat<5," 3. "15<=lat<20," and 4. "lat>=100" datapoints from Figure 2 for all our benchmarks.

olds when they arrive at the writeback stage.

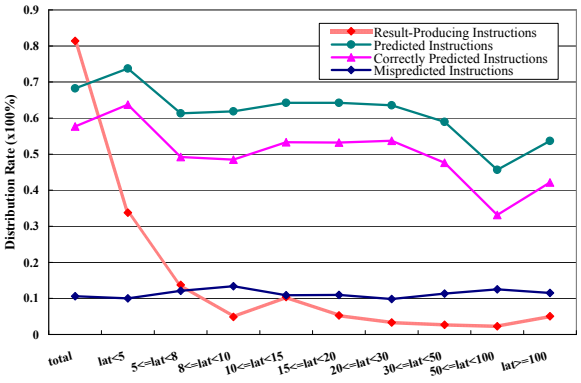


Figure 2. Fraction of instructions that are result-producing as well as the fraction of predicted, correctly predicted, and mispredicted instructions across different latency ranges over all 9 spec2000 integer benchmarks.

Figure 2 reports the number of result-producing instructions, first across all instructions (labeled "total") and then for different latency ranges, as a fraction of all executed instructions. The figure also reports the fraction of instructions from each category ("total" and the different latency ranges) that are predicted, predicted correctly, and mispredicted. (Note, the fraction of predicted instructions–and hence, the sum of the fraction of correctly predicted and mispredicted instructions–is not 1.0 because the predictor is unable to make predictions for some instructions, as described in Section 3.1). Every datapoint in Figure 2 represents an average across all the benchmarks listed in Table 2. Figure 2 shows result-producing instructions account for 81.4% of all instructions. In particular, instructions with a latency less than 5 cycles (from fetch to issue) account for 32.1% of all instructions, or 41.4% of result-producing instructions. Moreover, these short-latency instructions exhibit relatively good prediction rates–63.7% on average. In contrast, instructions with greater than 5-cycle latency have slightly lower prediction rates–around 40% to 50%. However, given that long-latency instructions contribute the most to fault vulnerability, it is still worthwhile to check their values via prediction.

As our results will show, the mispredictions in Figure 2 (which represent false positives in our technique) lead to performance degradation because they initiate pipeline squashes. Table 3 reports the number of such performance-degrading mispredictions for all our benchmarks. In particular, the rows in Table 3 numbered 1 through 4 report mispredictions for the "Total," "lat<5," "15<=lat<20," and "lat>=100" datapoints, respectively, from Figure 2. As Table 3 shows, the number of mispredictions, and hence the amount of performance degradation, generally reduces for selective prediction of longer latency instructions.

Next, we study the impact of value prediction on actual program reliability and performance. Figure 3 reports the percent AVF reduction (*i.e.*, reliability improvement) with value prediction in three hardware structures compared to no value prediction averaged across our 9 SPEC2000 integer benchmarks. In particular, the curves labeled "issue queue," "fetch buffer," and "physical register file" report the AVF reductions for the issue queue, fetch buffer, and physical register file, respectively. Also, the datapoints labeled "pred all" report the AVF reduction assuming full prediction, while the remaining datapoints report the AVF reduction with selective prediction based on instruction latency (*e.g.*, "pred lat $\geq$ 15" performs prediction only for instructions with at least 15-cycle latency between the fetch and issue stages).

Figure 3 shows prediction-based fault protection can be very effective at improving reliability (*i.e.*, reducing AVF). The AVF for the fetch queue, issue queue, and register file is reduced by as much as 96.0%, 89.8%, and 59.0%, respectively (under full prediction) compared to no prediction. This is due to both correct and incorrect predictions. On a correct prediction, the value of the predicted instruction is checked, so the instruction is no longer vulnerable, and hence, does not contribute to the AVF of the structures it occupies. On a misprediction, the pipeline is flushed. As discussed in

Section 2.2, re-execution after flushing is typically faster than the original execution, thus reducing the occupancy of ACE instructions in the hardware structures. Both combine to provide the AVF improvements shown in Figure 3.

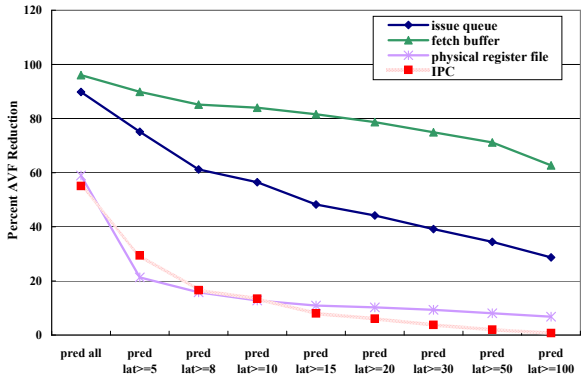


Figure 3. Percent AVF reduction in 3 hardware structures averaged across 9 SPEC2000 integer benchmarks by applying value prediction to instructions with varying latencies. The curve labeled "IPC" reports the percent IPC reduction for the same. All reductions are computed relative to no value prediction.

Unfortunately, these reliability improvements come at the expense of performance. In Figure 3, the curve labeled "IPC" reports the percent IPC reduction (*i.e.*, performance degradation) for the same experiments. This curve shows IPC can degrade significantly due to the penalty incurred by mispredictions, particularly when a large number of instructions are predicted. Under full prediction, IPC reduces by 55.1% compared to no prediction. But the performance impact lessens as fewer instructions are predicted (moving towards the right side of Figure 3). For example, when only predicting instructions with latency greater than or equal to 30 cycles, the performance impact is less than 3.8%. Of course, reliability improvement is not as great when predicting fewer instructions. But it can still be significant–we achieve a 74.9%, 39.2%, and 9.3% reduction in AVF for the fetch queue, issue queue, and register file, respectively at $\geq$ 30-cycle latency.

In general, Figure 3 shows there exists a trade-off between reliability and performance. The more instructions we predict, the larger the improvement in reliability, but also the larger the degradation in performance. We find a good strategy is to focus the value predictor on long-latency instructions (*e.g.*, instructions with $\geq$ 30-cycle latency). This is because the longer the instruction latency, the smaller the impact mispredictions will have on performance. Furthermore, the longer the instruction latency, the more critical the instructions are from a reliability standpoint.

3.3 Confidence Estimation

Confidence estimation can be used to reduce the number of performance-degrading mispredictions. To investigate the potential benefits of this approach, we added a confidence estimator to our value predictor. Figure 4 reports the fraction of predicted, correctly predicted, and mispredicted instructions for all instructions, labeled "total," and for instructions with different latency ranges. (The format for Figure 4 is almost identical to Figure 2, except there is no "Result-Producing Instructions" curve since it would be the same). Compared to no confidence estimation, our value predictor achieves fewer correct predictions with confidence estimation. The reduction ranges between 10% and 15%. This is because the confidence estimator prevents predicting the less predictable instructions. As a result, the fraction of mispredicted instructions goes down to almost 0 across all latency ranges. As Figure 4 shows, our confidence estimator is quite effective at reducing mispredictions with only a modest dip in the number of correct predictions.

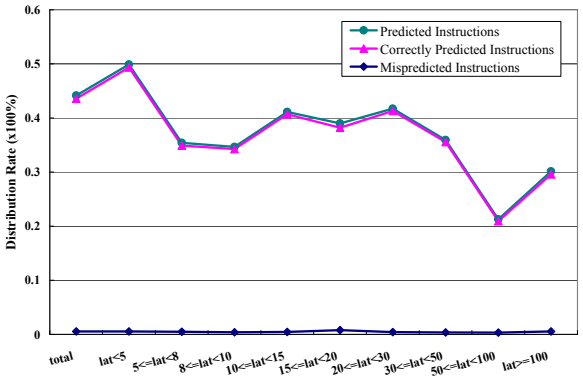


Figure 4. Fraction of predicted, correctly predicted, and mispredicted instructions–with confidence estimation–across different latency ranges over all 9 spec2000 integer benchmarks.

Table 4 reports the actual number of mispredictions with confidence estimation for all our benchmarks. (The format for Table 4 is identical to the format used in Table 3). Compared to Table 3, Table 4 shows the number of mispredictions with confidence estimation is indeed dramatically reduced relative to no confidence estimation.

	twolf	gcc	gap	gzip	bzip2	perl	parser	mcf	vpr
1.	861000	840153	2206916	328586	12479900	4512412	7806876	2712208	800670
2.	219590	545557	1036604	176886	2786259	1498557	4007555	691834	637547
3.	145622	10486	78163	4943	6583023	720734	172598	18595	14652
4.	82034	2158	10958	5	46766	388546	129277	1620071	263

Table 4. Number of mispredictions for the 1. "Total," 2. "lat<5," 3. "15<=lat<20," and 4. "lat>=100" datapoints from Figure 4 for all our benchmarks.

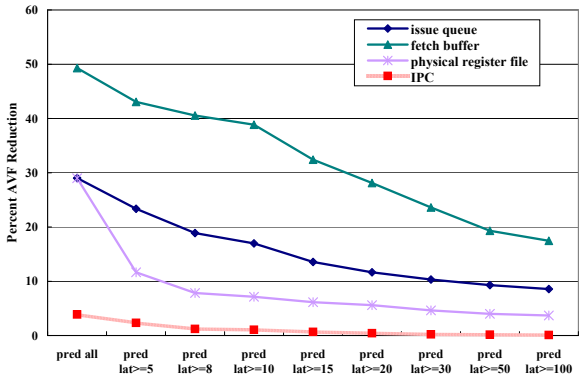


Figure 5. Percent AVF reduction in 3 hardware structures averaged across 9 SPEC2000 integer benchmarks by applying value prediction and confidence estimation to instructions with varying latencies. The curve labeled "IPC" reports the percent IPC reduction for the same. All reductions are computed relative to no value prediction.

Figure 5 shows the impact of confidence estimation on the AVF of our three hardware structures, as well as on IPC. (This figure uses the exact same format as Figure 3). In Figure 5, we see IPC never degrades more than 4% compared to no prediction, even when performing full prediction. These results show confidence estimation is indeed effective at mitigating performance degradation. Unfortunately, with confidence estimation, the reliability improvement is not as significant as before. In particular, under full prediction, the AVF for the fetch queue, issue queue, and register file is reduced by at most 49.3%, 29.0%, and 29.0%, respectively; under selective prediction with baseline latency of 30 cycles, the AVF for the fetch queue, issue queue, and register file is reduced by about 23.6%, 10.3%, and 4.6%, respectively. The lower reliability improvements compared to Figure 3 are due to the fact that confidence estimation suppresses prediction of many instructions, reducing the coverage achieved by the value predictor.

Thus far, we have applied confidence estimation uniformly across all instructions–*i.e.*, we use a single confidence threshold to determine whether any particular instruction should be predicted or not. However, predicting all instructions using a uniform confidence level may not be the best policy since instructions do not contribute equally to reliability nor to performance impact. In particular, for longer latency instructions which contribute more to overall reliability and incur less performance degradation during mispredictions, it may be better to perform value prediction more aggressively. Conversely, for shorter latency instructions which contribute less to overall reliability and incur more performance degradation during mispredictions, it may be better to perform value prediction less aggressively. This suggests an adaptive confidence estimation technique has the potential to more effectively tradeoff reliability and performance.

We modify our confidence estimation scheme to adapt the confidence threshold based on each instruction's latency. In particular, we employ three different threshold levels, similar to what is proposed in [10]. (The thresholds for low, medium, and high confidence are 3, 7, and 15, respectively for a saturating value of 15). We use the lowest confidence threshold for instructions that incur a latency equal to or larger than 4 times the baseline latency; we use the medium confidence threshold for instructions that incur a latency equal to or larger than 2 times the baseline latency but smaller than 4 times the baseline latency; and we use the highest confidence threshold for instructions that incur a latency equal to or larger than the baseline latency but smaller than 2 times the baseline latency. Here, the baseline latency is the minimum instruction latency that is considered for prediction as given by latency-based selective prediction. (For example, if we only predict instructions with latency 5 cycles or larger, then the low, medium, and high thresholds are applied to instructions with latency in the ranges $\geq$ 20 cycles, 10-19 cycles, and 5-9 cycles, respectively).

Figure 6 shows the impact of adaptive confidence estimation on the AVF of our three hardware struc-

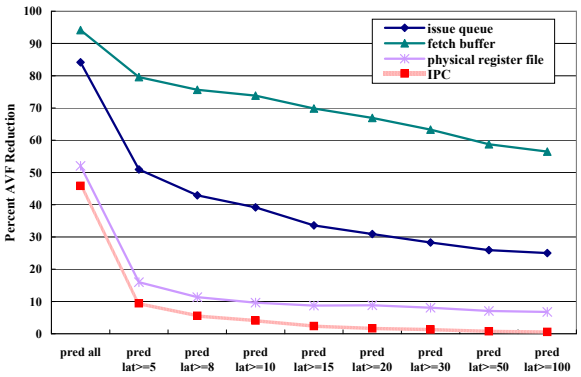


Figure 6. Percent AVF reduction in 3 hardware structures averaged across 9 SPEC2000 integer benchmarks by applying value prediction and adaptive confidence estimation to instructions with varying latencies. Confidence threshold used for each prediction (high, medium or low) varies according to the instruction's latency. The curve labeled "IPC" reports the percent IPC reduction for the same. All reductions are computed relative to no value prediction.

tures, as well as on IPC. (This figure uses the exact same format as Figures 3 and 5). As suggested by the above discussion, in these experiments we combine latency-based selective prediction with adaptive confidence estimation. In other words, we only consider for prediction those instructions that meet the latency threshold given along the X-axis of Figure 6, and for a given candidate instruction, we only predict it if its saturating counter meets the corresponding confidence threshold for its latency. As Figure 6 shows, adaptive confidence estimation incurs a relatively small performance degradation similar to the baseline confidence estimation technique shown in Figure 5. A particularly small performance degradation, about 5.4%, is achieved when limiting prediction to instructions with a latency of at least 8 cycles or larger. However, adaptive confidence estimation achieves a much better reliability improvement (AVF reduction) than the baseline confidence estimation, and approaches the reliability improvement achieved by value prediction without confidence estimation shown in Figure 3. For example, under selective prediction with baseline latency of 30 cycles, the AVF for the fetch queue, issue queue, and register file is reduced by about 63.3%, 28.3%, and 8.1%, respectively, while the performance is only degraded about 1.3%. Thus, by more aggressively predicting only the longer latency instructions, adaptive confidence estimation can cover the most critical instructions for reliability without sacrificing too much on performance.

4 Related Work

This work is related to several areas of research in fault tolerance. The first area includes studies which exploit explicit redundancy–by duplicating program execution either in hardware [1, 2, 3, 4, 5] or software [6, 7, 8]–to detect or recover from faults. In contrast, we study value prediction to explore the redundancy inherent in programs. Our technique avoids the overhead from explicitly duplicating computation for fault detection. However, value prediction cannot achieve 100% correctness, thus it cannot ensure failure-free execution while explicit duplication can. Our goal is to reduce the fault rate in a more cost-effective way, which is still meaningful for most systems that do not require failure-free execution. In addition, our technique considers fault vulnerability at the instruction level which is ignored by most existing techniques. By quantifying instruction's vulnerability, we selectively protect instructions that are most susceptible to faults, thus reducing the impact of mispredictions while still maintaining acceptable reliability.

In the area of exploiting inherent program redundancy, the work most related to ours is [15]. Racunas *et al* make use of value perturbation to prevent possible faults. Their technique tries to identify the valid value space of an instruction, which is done by tracking the instruction's past results. Future outputs that are not within the recorded valid value space are considered as potentially corrupted. Compared to value perturbation, value prediction tries to predict an instruction's result exactly. Outputs that are not equal to predicted values are considered as potentially corrupted. Compared to detecting value perturbations, value prediction can be more precise in finding discrepancies. For example, an instruction's past value space may be so big that corrupted values may still fall in the valid value space, and hence, cannot be detected.

Our technique is also related to the area of partial fault protection. Recently, some studies [12, 16] propose that traditional full-coverage fault-tolerant techniques are only necessary for highly-reliable and specialized systems, while for most other systems, techniques which tradeoff performance and reliability are more desirable. For example, Weaver *et al* [12] try to reduce error rate by flushing the pipeline on L2 misses. Gomaa *et al* [16] propose a partial-redundancy technique which selectively employs redundant thread or instruction-reuse buffer

for fault detection. The triggering of their redundancy technique is determined by program performance. Compared to their work, we exploit program's inherent redundancy for detecting possible faults. In addition, by characterizing instruction vulnerability, we selectively protect the most fault-susceptible instructions to achieve better coverage.

5 Conclusion

This paper investigates applying value prediction for improving fault tolerance. We make the observation that value predictability is a low-cost (albeit imperfect) form of program redundancy. To exploit this observation, we propose to use the output of a value predictor to check the correctness of predicted instructions, and to treat any mismatch as an indicator that a fault has potentially occurred. On a mismatch, we trigger recovery using the same hardware mechanisms provided for mispeculation recovery. To reduce the misprediction rate, we characterize fault vulnerability at the instruction level and only apply value prediction to instructions that are highly susceptible to faults (*i.e.*, those with long latency). We also employ confidence estimation, and adapt the confidence estimator's threshold on a per-instruction basis tuned to the instruction's latency. Instructions with higher latency are predicted more aggressively, while instructions with lower latency are predicted less aggressively. Our results show significant gains in reliability with very small performance degradation are possible using our technique.

6 Acknowledgements

This research was supported in part by NSF CAREER Award #CCR-0093110, and in part by the Defense Advanced Research Projects Agency (DARPA) through the Department of the Interior National Business Center under grant #NBCH104009. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

[1] R. W. Horst, R. L. Harris, and R. L. Jardine, "Multiple instruction issue in the NonStop Cyclone processor," in *Proc. of the 17th Int'l Symp. on Computer Architecture*, May 1990.

[2] Y. Yeh, "Triple-triple redundant 777 primary flight computer," in *Proc. of the 1996 IEEE Aerospace Applications Conference*, Feb. 1996.

[3] S. K. Reinhardt and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," in *Proc. of the 27th Annual Int'l Symp. on Computer Architecture*, June 2000.

[4] J. Ray, J. C. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," in *Proc. of the 34th annual IEEE/ACM Int'l Symp. on Microarchitecture*, Dec. 2001.

[5] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proc. of the 29th annual Int'l Symp. on Computer Architecture*, May 2002.

[6] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," in *IEEE Transactions on Reliability*, March 2002.

[7] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," in *IEEE Transactions on Reliability*, March 2002.

[8] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. Aug., "SWIFT: Software implemented fault tolerance," in *Proc. of the 3rd Int'l Symp. on Code Generation and Optimization*, March 2005.

[9] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors," in *Proc. of the 13th annual IEEE/ACM Int'l Symp. on Microarchitecture*, Dec 1997.

[10] B. Calder, G. Reinman, and D. Tullsen, "Selective Value Prediction," in *Proc. of the 26th Annual Int'l Symp. on Computer Architecture*, May 1999.

[11] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factor for a High-Performance Microprocessor," in *Proc. of the 36th annual IEEE/ACM Int'l Symp. on Microarchitecture*, Dec. 2003.

[12] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor," in *Proc. of the 31st Annual Int'l Symp. on Computer Architecture*, June 2004.

[13] D. Burger, T. Austin, and S. Bennett, "Evaluating future microprocessors: the simplescalar tool set," Tech. Rep. CS-TR-1996-1308, Univ. of Wisconsin - Madison, July 1996.

[14] B. Goeman, H. Vandierendonck, and K. de Bosschere, "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency," in *Proc. of the 7th Annual International Symp. on High-Performance Computer Architecture*, 2001.

[15] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-Based Fault Screening," in *Proc. of the 2007 IEEE 13th Int'l Symp. on High Performance Computer Architecture*, Feb 2007.

[16] M. Gomaa and T. N. Vijaykumar, "Opportunistic Transient-Fault Detection," in *Proc. of the 32nd Annual Int'l Symp. on Computer Architecture*, June 2005.

Multicore Power Management: Ensuring Robustness via Early-Stage Formal Verification

Anita Lungu[1], Pradip Bose[2], Daniel J. Sorin[3], Steven German[2], and Geert Janssen[2]

[1]Dept. of Computer Science
Duke University
anita@cs.duke.edu

[2]IBM T.J. Watson Research Center
{pbose,sgerman,geert}@us.ibm.com

[3]Dept. of ECE
Duke University
sorin@ee.duke.edu

Abstract

Power management is important for multicore architectures. One important challenge for multicore DPM schemes is verifying that they are both safe (cannot lead to power or thermal catastrophes) and efficient (achieve as much performance as possible without exceeding power constraints). The verification difficulty varies among designs, depending, for example, on the particular power management mechanisms utilized and the algorithms used to adjust them. However, verification effort is often not considered in the early stages of DPM scheme design, leading to proposals that can be extremely difficult to verify.

To address this problem, we propose using formal verification (with probabilistic model checking) of a high-level, early-stage model of the DPM scheme. Using the model checker, we estimate the required verification effort, providing insight on how certain design parameters impact this effort. Furthermore, we supplement the verifiability results with high-level estimates of power consumption and performance, which allow us to perform a trade-off analysis between power, performance, and verification. We show that this trade-off analysis uncovers design points that are better than those that consider only power and performance.

1. Introduction

The prevalence of multicore architectures coupled with demands for low power systems motivate the development and evaluation of efficient power management solutions targeted specifically at multicores. Power is managed for several reasons, including to: improve power-efficiency, avoid power spikes, increase battery life, reduce the cost of providing power to the chip, and manage temperature. In this work, we investigate dynamic power management (DPM) schemes that can cap the peak power usage of a multicore. Providing a DPM scheme that caps the peak power can reduce system cost by decreasing the cooling and packaging requirements, or it can relax the power constraints placed on other system components.

One critical aspect in the development of a new DPM scheme is its verification. There are three properties that we wish to verify. First, we want to verify that the DPM scheme is *safe*. A DPM scheme can be unsafe, for example, if it allows the power usage to often exceed the allocated budget, or if it allows a core to be assigned a voltage or frequency outside of the desired range. Second, we wish to verify that the DPM scheme is *efficient* in achieving as much performance as possible while not exceeding power constraints or violating priority rules for provisioning power. A buggy DPM scheme might sacrifice more performance than expected. Third, we want to verify that the DPM scheme is *functionally correct*, such that the same results are obtained with and without the DPM scheme. In this paper, we consider verification of the first two features. As a concrete example of the importance of DPM verification, concerns over Intel's Foxton DPM scheme [16] led to it being disabled in the first Montecito chips [4].[1]

The current industrial workflow in the development of a new DPM scheme is illustrated in the unshaded portion of Figure 1. At an early stage, the focus is restricted to maximizing the efficiency of the DPM scheme, with limited consideration of its verification. Later, the scheme is implemented in detailed, low-level simulators, and verification[2] primarily checks whether the scheme achieves its efficiency goals.

The problem with this current workflow is that it is prone to missing bugs. First, simulation is by definition incomplete as a verification solution, because only the states that are reached in a particular simulation path are ascertained to be bug-free. Second, if verification feasibility is not considered at design time, the reachable state space of the resulting DPM scheme can be enormous, which is problematic. Workflows often have goals for achieving minimum coverage, so having more states

1. Intel has not officially stated whether the concerns were over safety, efficiency, or functionality bugs.

2. Using a simulator to "verify" a design is sometimes referred to as "validation" instead of verification.

requires more simulation cycles. If no coverage goal is specified, having more states increases the probability that undiscovered bugs remain in the design and decreases confidence in DPM correctness.

To address the above concerns, we propose the introduction of an additional, early step in the development of a new DPM scheme. We illustrate this added step in the shaded portion of Figure 1. This additional step creates, at an early design stage, a high-level model of the proposed power management policy which is then verified for efficiency and safety using probabilistic model checking, an exhaustive formal verification method. By performing a high-level verification early in the development process, we identify problems when they are easier to solve. A high-level model is also much easier to develop and modify than a detailed simulator, so we can quickly explore numerous designs.

With the use of the model checker, we estimate the effort required to verify the DPM scheme (measured as number of reachable states and transitions) enabling a better understanding of the impact on verification effort of scaling certain design parameters. Furthermore, we supplement the verifiability results with a high-level estimate of power consumption and performance, which enables us to perform a trade-off analysis between reaching power, performance, and verification goals.

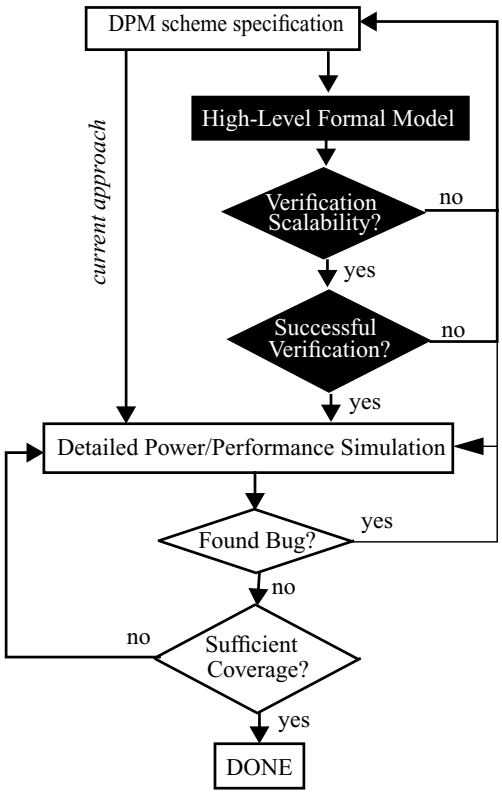


Figure 1. Workflow for Development of New DPM Scheme. Shaded portions indicate proposed additions.

Model checking does not eliminate the need to later simulate a detailed implementation of the DPM scheme, but it can catch bugs early and help the simulation reach desired state coverage goals.

Our main contributions are the following:

- We propose the use of verification effort as an additional metric to be considered, together with performance, in the early stages of DPM scheme design.
- We investigate and compare the effort necessary to verify different DPM algorithms as a function of the available mechanisms for adjusting power usage.
- We evaluate the trade-offs between verification effort, efficiency, and safety of the DPM schemes mentioned above.

The rest of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we present the type of DPM scheme we investigate and its parameters of interest. In Section 4, we explain our experimental methodology. In Section 5, we present our results, and we conclude in Section 6.

2. Background and Related Work

Power management is an important issue and thus there has been a significant amount of prior work in this area. In this section we first present multicore-specific power management schemes (Section 2.1). We then discuss prior work in power management verification (Section 2.2). Lastly, we discuss verification-aware design in general (Section 2.3).

2.1 Multicore Power Management

The most straightforward way to manage power in a multicore chip is to simply apply well-known single-core techniques to every core. However, Isci et al. [5] observed that such "local" (per-core) management was potentially inefficient because it could not take advantage of peak power averaging effects that occur across multiple cores. They introduce global schemes in which a single, centralized, "global" controller determines the power budget and settings (e.g., voltage and frequency) for every core. Sharkey et al. [18] provide a more detailed evaluation of these global schemes in terms of their efficiency. Sartori and Kumar [17] present a proactive scheme for managing peak power in multicore chips. They observe that distributed algorithms can be used to select the power level allocation for cores and that they would be more scalable than algorithms based on having a centralized global controller. However, no multicore DPM scheme has been analyzed to determine its verification effort and to trade-off verifiability against other design goals.

2.2 Verifying Power Management Schemes

There has been a limited amount of prior work in verifying DPM schemes. One representative piece of

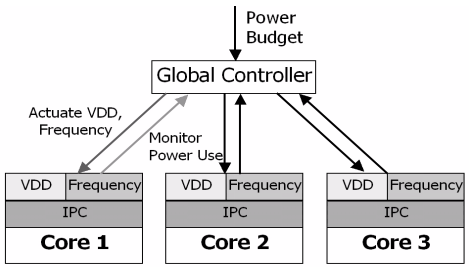


Figure 2. DPM Scheme with Global Controller

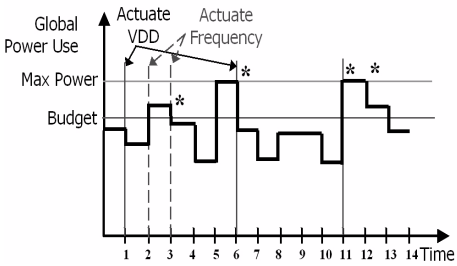


Figure 3. DPM Scheme Power Utilization

work by Shukla and Gupta [20] uses the SMV model checker [12] to verify a DPM scheme. We are interested in DPM for multicores, whereas their focus is on solutions for unicore systems. Furthermore, we use model checking to estimate verification effort and verify a set of correctness properties, while they use it to stress the optimality bounds of the DPM scheme by constructing a worst case task trace. Dubost et al. [3] present a high-level argument for specifying power management schemes in the Esterel language, which facilitates using a model checker to verify the designs. They do not discuss any specific DPM scheme or verification.

One interesting approach to DPM verification is the use of *probabilistic model checking*. With a traditional model checker, such as Murphi [2], one can prove absolute invariants. For example, one can prove that the power never exceeds a 50W power budget. However, with DPM, it may be tolerable that a 40W "soft power budget" is occasionally exceeded if that happens infrequently. Two recent research papers [15, 7] have used the PRISM probabilistic model checker [6] to analyze DPM schemes. They target unicore systems and use PRISM to find optimal power management policies for given task arrival distributions and constraints on expected wait queue size. In contrast, we are interested in analyzing the trade-off between verifiability and other metrics for multicore schemes.

2.3 Verification-Aware Design

Lungu and Sorin [8] quantified the effort required to formally verify parts of microprocessors. Martin [9] and Marty et al. [10] discussed the verification effort required for different cache coherence protocols. Our work differs from this prior work by focusing on power management schemes.

3. DPM Design Space Exploration

A wide variety of DPM solutions have been proposed in response to different requirements. In this section we describe the particular type of solution we analyze and its design parameters.

3.1 High Level View of DPM Design Space

We target DPM schemes that can cap the peak power usage of a multicore chip by using dynamic voltage and frequency scaling (DVFS). Figure 2 depicts the system we consider. The overall goal of the global DPM controller is to maintain the power usage of the system below the budget target set by a user (which could be the OS) with a minimum performance penalty. We use the expression "power budget" in a manner similar to prior work [5, 18]. The budget is the desirable power consumption level for the chip (shown in Figure 3). The budget differs from the Maximum Power for the chip, in that the budget is a somewhat soft limit. Exceeding the hard Maximum Power limit could lead to a thermal emergency and even burn the chip. However, exceeding the power budget occasionally, while still keeping the power below Maximum Power, can be tolerated. Budget overshoots cause the policy's goal to be temporarily unmet, but they cause no thermal emergencies. Recently developed DPM schemes also allow temporary budget overshoots [5, 18].

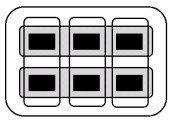
To keep the chip under its budget, the global controller periodically monitors the power usage of all cores and actuates their voltages and frequencies such that the total power consumption is maintained below the specified budget. We consider two actuation intervals: one for changing both voltage and frequency and one for changing only the frequency.

Figure 3 illustrates the power consumption of the chip over time. The Max Power horizontal line represents the maximum power the chip can consume given the worst case activity factors of all cores. The Budget line represents the constraint imposed on the power use of the chip. The global controller uses this power budget value as the target for its feedback mechanism. In setting the voltage and frequency levels, the global controller makes the prediction that the cores will maintain their current activity factors for the next interval. When this is a misprediction, the actual power use can temporarily overshoot, as shown in Figure 3 at the times marked with stars. On the next actuation point the controller tries again to bring the power use below budget.

3.2 Design Goals and Parameters

Of the multiple design goals that such a DPM scheme can target, we investigate *efficiency* (reducing the performance hit induced by decreasing core fre-

Figure 4. Possible Assignments of Cores to Controllers



quency through DVFS), *safety* (decreasing time and power spent over budget) and *verifiability* (decreasing required verification effort).

To reach these goals, designers can make decisions on many parameters. We consider here only a subset of them to keep our analysis tractable. Specifically, we compare a heterogeneous policy, which allows the controller to assign different voltage and frequencies across the cores, to a homogeneous policy, where the same voltage and frequency is set for all cores. For both policies, we analyze the design space along 3 parameters: number of voltage levels (**VL**) into which the voltage range is split, number of frequency levels (**FL**) that can be allocated for a given voltage level, and number of cores assigned to a single DPM controller. Figure 4 illustrates this cores per controller (**CPC**) design parameter. If we consider a 6-core chip, a DPM solution might use a single controller assigned to all chips (the outer boundary), or 2 controllers each monitoring 3 cores (the two horizontal groupings), or 3 controllers each supervising 2 cores (the three vertical groups).

3.3 Motivating Early Formal Analysis

Designers certainly have some intuitive *a priori* understanding of how choosing different design points in the above parameter space affects their goals. For example, one might expect that a heterogeneous solution with more CPC will outperform a solution with fewer CPC, because the peak power use of more cores should be decreased due to averaging effects. But what is the *quantitative* gain in performance when going from 2 CPC to 3 CPC, for example? Is that performance gain worth the impact on verification effort? How does the safety of the solution change in response to CPC? Do the answers vary between homogeneous and heterogeneous policies? In addition to questions about CPC, designers want to answer similar questions about other parameters, such as VL and FL, and possible interactions between parameters. Will a change in VL impact design goals differently depending on the value of CPC?

These are the type of questions to which we seek answers via performing the proposed early stage formal analysis. These answers enable designers to make more informed decisions, and we show concrete examples of these benefits in Section 5.

4. Methodology for Formal Analysis

We begin this section with our motivation for using probabilistic model checking to verify the analyzed DPM schemes and a brief overview on this method. Then we provide details on the particular methodology we use to conduct our experiments.

4.1 Probabilistic Model Checking

We use probabilistic model checking with PRISM [6] to explore the design space of our DPM schemes and analyze trade-offs between efficiency, safety, and verifiability.

Using a model checker allows us to quantify the verification effort for a system. We chose a model checking tool over a simulator because a model checker is a complete verification solution which traverses the entire reachable state space of a design in ascertaining correctness. In contrast, a simulator is incomplete because it touches only a limited subset of all reachable states. We obtain a better verifiability measure for a design when we can exercise its entire reachable state space and all state transitions. The choice of probabilistic model checking over traditional, non-probabilistic model checking was motivated by characteristics of the problem we want to verify. For the verification of a DPM scheme we are not only interested whether a power overshoot can happen, but also how often this is expected to happen under typical conditions. These types of correctness characteristics depend on the changing activity factor of the workloads, which can be captured in a probabilistic framework.

The inputs to the probabilistic model checker are: the *state elements* of the system, the *probabilistic transition rules* (a description of how the behavior can change from one state to the next), and the *correctness properties* (the requirements which, if met, assure the system's correctness). In addition, it is possible to evaluate the expected values of certain quantities in the system, such as power and performance, by associating *rewards* with system states. Rewards are similar to tokens, in that the states that satisfy a certain condition are assigned tokens. It is not our goal to use model checking for a better estimate of power usage and performance impact; rather, we use the rewards to obtain high-level measures of power and performance and analyze their trade-off with verifiability. Based on the probabilistic state machine description, the model checking tool traverses the entire reachable state space of the design and verifies whether the correctness properties are met. When rewards are specified it also calculates their expected values over a certain bounded number of system transitions.

4.2 DPM Model Construction

For our DPM scheme, the *state elements* are: the current voltage, frequency, and activity factor of each core and an incrementing counter triggering when the global controller should actuate both voltages and frequencies as opposed to only frequencies.

Table 1. Microprocessor Configuration

Feature	Description
pipeline width	4 decode/issue/commit
ROB/LSQ sizes	150 entries / 32 entries
branch pred.	2 level, 3 16K-entry BHTs
functional units	4 FXU, 4 FPU, 1 BR
L1I cache	64KB, 2-way, 16B blocks, 1cycle
L1D cache	64KB, 2-way, 16B blocks, 1cycle
L2 cache	1MB, 8-way, 64B blocks, 9 cycles
memory	100 cycles

Table 2. Benchmarks

	Low Ave IPC	High Ave IPC
Stable IPC	*mcf*	*eon, crafty*
Variable IPC	*art, parser*	*bzip2*

The *probabilistic transition rules* specify how the activity factor changes for the cores and how the voltages and frequencies change in response to controller actuations. We approximate each core's activity factor using its instructions per cycle (IPC), because IPC is strongly correlated with the activity factor and it is easy to obtain. This correlation is not perfect, but obtaining the exact activity factor would require a low-level implementation that is unlikely to exist early in the design cycle. To make our analysis tractable with PRISM, we quantize the IPC values into four distinct ranges, and we choose the mean IPC of a range to represent the activity factor of a core in that range.

We obtain the transition probabilities using Turandot [13], a detailed, cycle-accurate simulation model. The microprocessor's configuration is shown in Table 1. For benchmarks, we chose six SPEC 2000 benchmarks, shown in Table 2, that have very different behavior, both in terms of their average activity factor and in how much their activity factor changes over time. The appropriate SimPoint [19] intervals for these benchmarks were traced using Aria [14]. For each benchmark, the simulator produces the average IPC for each time quantum of 100µs (400,000 cycles at 4GHz). The sampling period of 100µs reflects the safe specification parameter of the power manager, in terms of the longest duration of allowable power spikes. Given that chip-level thermal time constants are in the range of milliseconds or tens of milliseconds [1], 100µs is a very safe, conservative setting of this parameter.

We wish to point out that we obtain the benchmark IPC values from a simulation of a single-core processor, rather than from a simulation of a multicore processor. The intuitive reason for this decision is that PRISM will inherently construct all possible combinations of IPCs and IPC transitions for all cores running the benchmarks.[3] Moreover, it is not obvious that we even *could* simulate every possible combination, since it is extremely difficult to compel the simulated system into each combination of core states.

4.3 DPM Scheme Properties

We verify the behavior of the system against a set of correctness properties that must be true in every state. We also specify a set of reward structures that enable us to quantify performance, power use, and safety.

Correctness properties. The correctness properties we consider for our DPM scheme are:

- No deadlock state can ever be reached.
- The voltages and frequencies for all cores are always maintained within a pre-specified range.
- There is no mismatch between the voltage and frequency assigned to a core (e.g., we never match a very high frequency with a very low voltage).

Reward structures. We use rewards to keep track of power, performance, and the states in which the system is over budget. PRISM computes the expected rewards over a bounded interval, and we set the bound to 1000 transitions in our experiments.

4.4 Quantifying Performance, Power, Safety, and Verifiability

We now describe the models and metrics we use to quantify performance, power, safety, and verifiability for our early stage formal analysis.

Performance. In our model, the performance of a core is a linear function of its frequency, *f*. That is, if we increase *f* by X%, then the performance is also improved by X%. This is an approximation, because the performance benefit of a large increase in *f* is limited by the unchanged memory performance. Nevertheless, for a high-level model that is considering small adjustments in *f*, we think this assumption is reasonable.

Our model considers the latency required to transition between voltage levels, and it assumes that a core functions at its lowest frequency during a voltage transition (1µs per 10mV). The latency of transitioning between frequency levels is much shorter—on the order of one or two processor cycles [11]—because it can be done with on-chip digital PLL mechanisms. This latency is orders of magnitude shorter than a 100µs actuation interval, and thus we do not model it.

3. One caveat is that a simulation of a multicore chip might (a) exhibit transitions that are *never* exhibited by a single-core chip, or (b) *never* exhibit transitions that are exhibited by a single-core chip. These scenarios, although unlikely, could result from contention for resources that occurs in multicore chips.

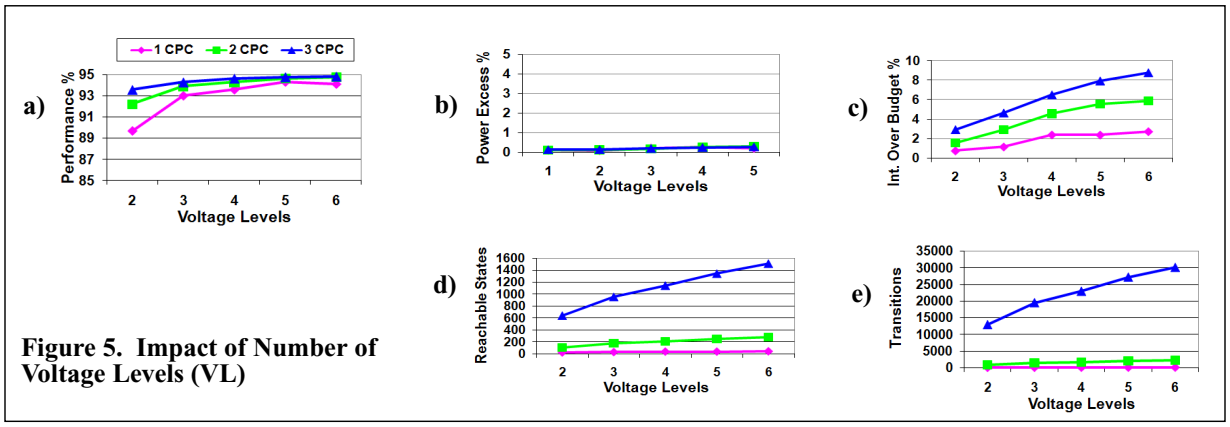


Figure 5. Impact of Number of Voltage Levels (VL)

Power. In our model, the power consumption of a core is a function of the core's frequency (f), voltage (V), and activity factor (A). We model both active and leakage power, with active power consumption formulated using the usual $\sim f*A*V^2$ dependence equation. The leakage power is modeled approximately as a cubic function of V, as this has been found to capture the behavior quite well for the particular supply and threshold voltage ranges appropriate for current CMOS technologies (65nm or 45nm). The power model used is admittedly abstract, but deemed to be good enough for the DVFS-driven power management policies considered in this paper (as in Isci et al. [5] or Sharkey et al. [18]).

Safety. We consider two safety metrics: the percentage of time the system is expected to be over budget, and the percentage of power used over budget.

Verifiability. We consider two metrics for quantifying verification effort. The first is the total number of reachable states of the design. The second is the number of possible transitions between states.

Because we use a simulator to generate the state transition probabilities, our performance and safety results are a function of the benchmark suite, because they depend on rewards computation. The verifiability results are also a function of benchmark suite as the number of reachable states and transitions depends on the changing behavior of the applications. For benchmarks with radically different behavior, these results might be different. We state this perhaps obvious characteristic of our work—after all, benchmark dependence is common in microarchitectural studies—because it differs from traditional (non-probabilistic) model checking. Note that the correctness properties mentioned in Section 4.3 are proved correct independent of the benchmark suite.

5. Experimental Evaluation

We now detail the two specific DPM schemes we modeled for our analysis and their design parameters. Then we describe the performance, safety, and verifiability trade-offs we find in this design space.

5.1 Scope of Analysis

We analyze *heterogeneous* and *homogeneous* DPM schemes. For the heterogeneous schemes, the controller uses a priority based greedy algorithm for distributing the power budget. It allocates the largest voltage that fits in the power budget for the first core (while provisioning enough power to run the rest of the cores at lowest voltage) then allocates the largest possible voltage for the second core and so on. This heterogeneous policy is very similar to current state-of-the-art DVFS policies, such as the "Priority" scheme analyzed by Isci et al. [5]. For homogeneous schemes, the controller allocates the single greatest voltage level that keeps the chip below the power budget, assuming all cores maintain their current activity factors. This homogeneous policy is very similar to the "Chip-Wide DVFS" scheme proposed by Isci et al. [5].

All of our DPM schemes use two actuation intervals: a 500µs one to change both voltage and frequency of cores (the frequency is set to the highest value permitted for the voltage level selected) and a 100µs one to change only the frequency. We vary the voltage range from 1.05V to 0.78V and we scale the frequencies linearly with the voltage from 4.2GHz to 3.15GHz.

When analyzing the impact of increasing VL, we maintain the same voltage range and divide it into more levels (from 2 to 6 in our experiments). When varying FL, we divide the frequency range corresponding to a particular voltage level into more values (from 1 to 5). We also vary CPC from 1 to 3. Note that this is different from comparing a 1-core chip to a chip with 2 or 3 cores; we consider a chip with the same number of cores, 6 for example, which has 6, 3 or 2 controllers.We do not model a 6-core system with a single controller (having CPC of 6) because the associated state explosion makes the verification through model checking

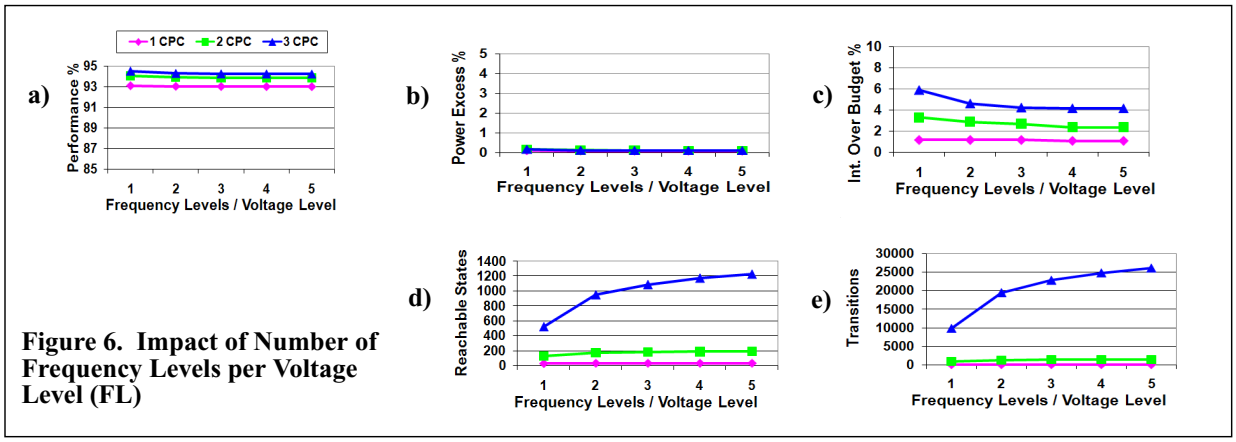


Figure 6. Impact of Number of Frequency Levels per Voltage Level (FL)

impractical and our results show little overall performance improvement beyond 3 CPC.

In our analysis, the global controller uses the power model described in Section 4.4 to estimate the power use of the system (a function of activity factor, voltage and frequency). The global controller predicts that the cores will maintain their current activity factor during the next interval.

We perform a range of experiments setting the power budget to 25, 40, 50, 70 and 100% of the maximum power the chip can consume (corresponding to a 4 IPC activity factor across all cores). The results we present are averaged across the different budget levels and benchmarks.

5.2 Impact of Number of Voltage Levels

The first design parameter we explore is VL. We consider a heterogeneous scheme and fix FL to 2 for clarity (the results were similar for the other FL values). Figure 5(a) shows the impact of VL on performance with respect to a chip without DPM. Figure 5(b,c) show safety, and Figure 5(d,e) show verifiability. We notice a strong interaction between VL and CPC; on many of our metrics of interest, the impact of increasing VL varied across different levels of CPC. Hence we present data for CPC=1, 2 and 3 on the same graph.

We notice several interesting phenomena. First, in terms of performance, the trend corroborates our intuition that increasing VL benefits performance. However, we notice a saturation around VL=5 and performance remains almost flat afterwards. Prior work [17] proposed using VL=10 in an experimental setup that used 4 cores, simulating various SPLASH benhmarks. Our results, albeit in a different setup, suggest that such a large value of VL offers little marginal benefit.

The impact of CPC on performance also matches our intuition in that we achieve better performance by increasing CPC. In fact the CPC=1 solution lags behind the CPC=2 and CPC=3 solutions at all voltage levels. However, the difference between the CPC=2 and CPC=3 solutions is minimal. They differ somewhat for low values of VL (2 or 3) but after that point there is very little difference in performance. The intuition is that the presence of 2 cores with activity factors that differ achieves a good enough average effect on the aggregate peak power to make throttling unnecessary. In prior work [5], the authors foresaw the motivation and need for centralization of the multicore power management problem. In this work we have seen that centralization is indeed better than local per-core control, but clustering of cores per controller beyond two may not yield additional performance. This insight is an important additional input to future architectural design of multicore power management protocols.

In terms of safety, the percentage power spent over budget is minimal, ranging from 0.1% to < 0.5% of the power usage of a solution without DVFS. The percentage of intervals spent over budget varies from ~0.5% to ~9%. An increase in CPC allows the controller to make more aggressive decisions in matching the power budget resulting in more mispredictions. The same can be said about increasing VL. Whether the amount of time spent over budget is deemed tolerable or not depends on the particular constraints of the application. However, considering the tiny percentage of power spent over budget, we conclude that VL does not greatly impact safety.

Given only the performance and safety analysis of the design space, one might conclude that the greatest difference can be noticed when going from CPC=1 to CPC=2 and that there is a minimal difference between CPC=2 and CPC=3. However, if we add verifiability to the picture, the conclusion changes dramatically. The verification effort, measured both in number of reachable states and transitions, increases dramatically with CPC. We see a strong interaction between CPC and VL in terms of verifiability effects. For both the CPC=1 and CPC=2 solutions, the verification effort does not

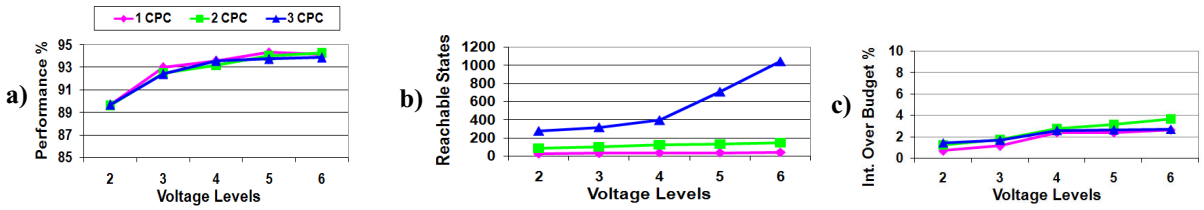


Figure 7. Impact of Homogeneous Policy

increase significantly with VL, unlike the case for the CPC=3 solution.

In conclusion, the performance improvement gained from going from CPC=2 to CPC=3 is insignificant (particularly for larger VL) while the increase in verification effort is extremely large. Our data suggest that the better design solution consists of having multiple controllers each assigned to a small number of cores (2) which can be set to 4-5 voltage levels as opposed to a design with a large CPC at low VL.

5.3 Impact of Number of Frequency Levels

The second design parameter we address is FL, the number of frequency levels that can be set for a given voltage level. Our hypothesis was that the 100µs actuation of the controller can take advantage of the increased frequency granularity and better track the power budget between consecutive voltage actuations.

Figure 6 shows our results when we consider a heterogeneous policy and fix VL=3 for performance with respect to a chip without DPM (a), safety (b, c) and verifiability (d, e). Our results indeed show a slight improvement in safety due to the increased flexibility in frequency levels. However, this improvement is minimal and accomplished with a performance penalty. The reason is that the frequency decrease is a lot less efficient in decreasing the overall power usage than the voltage. The impact of FL on verification, however, is very large both in reachable states and transitions. We conclude that the frequency knob should be used only when the safety margins of being over budget are tight, because a significant cost in verifiability will be paid. Also, FL=2 seems to suffice for getting most of the safety benefit. Our conclusion is specific to the type of system we analyzed, where it is possible to set both voltage and frequency of individual cores at different levels. For this case, using many frequency levels for one voltage level does not seem to represent a good design alternative from a verifiability, performance, and safety trade-off. For the class of systems that allocate the same voltage across all cores, the impact of frequency levels is likely to be more beneficial.

5.4 Impact of Using a Homogeneous Policy

We now explore the impact of choosing a homogeneous policy. We wish to discover whether homogeneity helps or hurts our pursuit of better design points. Figure 7 shows the results for a homogeneous policy when we vary VL. We notice a slight decrease in performance for an increase in CPC. This result is due to the fact that the homogeneous policy is more restrictive and all cores assigned to the controller are throttled to a single voltage level to match the budget. Second, the performance impact of increasing VL is more significant compared to the heterogeneous case. The safety is improved for the homogeneous solution as the percentage of intervals spent over budget decreases significantly.

6. Conclusions

Power management is important for multicore processors, and DPM scheme designers would like to have confidence that their schemes are both safe and efficient. We have shown the insight that can be gained by using formal methods—in this case, probabilistic model checking—to analyze high-level descriptions of DPM schemes. We have used PRISM to determine the effort required to verify DPM schemes, and we have compared these schemes with respect to their efficiency.

One conclusion we draw from this work is that global schemes (i.e., CPC>1) offer significant benefits in performance due to the ability to balance power across more cores. However, we must be careful to avoid scaling them to more cores than necessary. Linear increases in CPC cause exponential increases in the size of the reachable state space. Thus it is important to find the system configuration where both the verification is tractable and we obtain the majority of the benefits of a global solution. Our data shows that much of the benefit is achieved at just CPC=2; increasing CPC further provides little additional performance gain. In terms of safety, we found no significant difference between percentage energy spent over budget as a function of CPC, but a larger value of CPC resulted in the system spending more time over budget. Thus we recommend designs

in which chips are divided into small clusters of cores, where each cluster uses a global control scheme.

A second conclusion is that the use of fine-grained frequency tuning is likely not worth its costs for systems where it is possible to set both voltage and frequency of individual cores at different levels. The results show that having a large FL has an extremely large impact on verification effort. It is not clear that its modest safety benefits justify these verification costs.

Acknowledgments

This work was initiated as a 2007 summer internship project at IBM T. J. Watson Research Center. The work at IBM was supported in part by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. At Duke University this research was supported by the National Science Foundation under Grants CCF-0444516 and CCF-0811920. We thank Alvy Lebeck and Costi Pistol for helpful discussions about this work.

References

[1] J. Choi et al. Thermal-aware Task Scheduling at the System Software Level. In *Proc. of the Int'l Symposium on Low Power Electronics and Design*, Aug. 2007.

[2] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *1992 IEEE Int'l Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[3] G. Dubost, S. Granier, and G. Berry. An Esterel-Based Formal Specification Methodology for Power Manager Development. Presented at the SAME Forum, Oct. 2007.

[4] D. Dunn. Intel Delays Montecito in Roadmap Shakeup. *EE Times*, October 24 2005.

[5] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proc. of the 39th Annual IEEE/ACM Int'l Symposium on Microarchitecture*, Dec. 2006.

[6] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 2.0: A Tool for Probabilistic Model Checking. In *Proc. of the 1st Int'l Conference on Quantitative Evaluation of Systems*, pages 322–323, Sept. 2004.

[7] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic Model Checking and Power-Aware Computing. In *Proc. of the 7th Int'l Workshop on Performability Modeling of Computer and Communication Systems*, pages 6–9, Sept. 2005.

[8] A. Lungu and D. J. Sorin. Verification-Aware Microprocessor Design. In *Proc. of the Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 83–93, Sept. 2007.

[9] M. M. K. Martin. Formal Verification and its Impact on the Snooping versus Directory Protocol Debate. In *Proc. of the Int'l Conference on Computer Design*, Oct. 2005.

[10] M. R. Marty et al. Improving Multiple-CMP Systems Using Token Coherence. In *Proc. of the Eleventh Int'l Symposium on High-Performance Computer Architecture*, pages 328–339, Feb. 2005.

[11] R. McGowen et al. Power and Temperature Control on a 90-nm Itanium Family Processor. *IEEE Journal of Solid-State Circuits*, 41(1):229–237, Jan. 2006.

[12] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[13] M. Moudgill, P. Bose, and J. H. Moreno. Validation of Turandot, a Fast Processor Model for Microarchitecture Exploration. In *Proc. of the IEEE Int'l Performance, Computing and Communications Conference*, pages 451–457, Feb. 1999.

[14] M. Moudgill, J.-D. Wellman, and J. H. Moreno. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, 19(3):15–25, May/June 1999.

[15] G. Norman, D. Parker, M. Kwiatkowska, and S. Shukla. Using Probabilistic Model Checking for Dynamic Power Management. *Formal Aspects of Computing*, 17(2):160–176, Aug. 2005.

[16] C. Poirier, R. McGowen, C. Bostak, and S. Naffziger. Power and Temperature Control on a 90nm Itanium-family Processor. In *Proc. of the IEEE Int'l Solid-State Circuits Conference*, Feb. 2005.

[17] J. Sartori and R. Kumar. Proactive Peak Power Management for Many-Core Architectures. Technical Report CRHC-07-04, UIUC CRHC, Oct. 2007.

[18] J. Sharkey, A. Buyuktosunoglu, and P. Bose. Evaluating Design Tradeoffs in On-Chip Power Management for CMPs. In *Proc. of the Int'l Symposium on Low Power Electronics and Design*, Aug. 2007.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proc. of the Tenth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[20] S. Shukla and R. K. Gupta. A Model Checking Approach to Evaluating System Level Dynamic Power Management Policies for Embedded Systems. In *Proc. of the High-Level Design Validation and Test Workshop*, pages 53–57, 2001.