Using Underutilized CPU Resources to Enhance Its Reliability

Avi Timor, Avi Mendelson, Yitzhak Birk, and Neeraj Suri

Abstract—Soft errors (or Transient faults) are temporary faults that arise in a circuit due to a variety of internal noise and external sources such as cosmic particle hits. Though soft errors still occur infrequently, they are rapidly becoming a major impediment to processor reliability. This is due primarily to processor scaling characteristics. In the past, systems designed to tolerate such faults utilized costly customized solutions, entailing the use of replicated hardware components to detect and recover from microprocessor faults. As the feature size keeps shrinking and with the proliferation of multiprocessor on die in all segments of computer-based systems, the capability to detect and recover from faults is also desired for commodity hardware. For such systems, however, performance and power constitute the main drivers, so the traditional solutions prove inadequate and new approaches are required. We introduce two independent and complementary microarchitecture-level techniques: Double Execution and Double Decoding. Both exploit the typically low average processor resource utilization of modern processors to enhance processor reliability. Double Execution protects the Out-Of-Order part of the CPU by executing each instruction twice. Double Decoding uses a second, low-performance low-power instruction decoder to detect soft errors in the decoder logic. These simple-to-implement techniques are shown to improve the processor's reliability with relatively low performance, power, and hardware overheads. Finally, the resulting "excessive" reliability can even be traded back for performance by increasing clock rate and/or reducing voltage, thereby improving upon single execution approaches.

Index Terms—Transient faults, soft errors, superscalar, fault tolerance, microarchitecture, double execution.

1 Introduction and Background

The reliability impact of permanent or soft errors is an increasingly significant concern for upcoming processors. Permanent faults (node stuck-at-1/0, transistor open, shorted transistors, etc.) arise during fabrication or result from aging and destroy the intended function of the circuit. Soft errors, in contrast, occur for a relatively short duration in an otherwise correctly operating circuit. They may arise in digital systems due to internal noise sources such as power transients, capacitive, and inductive crosstalk or due to external noise sources such as cosmic particle hits [1], [2], [3]. Aging may also initially produce soft errors. While much of the research on reliable processors has focused on permanent faults, soft errors are becoming an important issue for mainstream processor and system-level reliability for several reasons:

1. Process scaling—every two years, on the average, new process technology is introduced, featuring smaller transistor geometries and a lower supply voltage. (This reduces power consumption, increases operating frequencies, and permits more transistors in a device.) Since less energy is needed in order to change the state of the transistor, soft errors occur more frequently per transistor; the larger number of transistors further increases the soft-error rate, with a resultant significant reduction in CPU reliability [2], [4], [5], [6], [7].
2. Operating with a higher than acceptable soft error rate, e.g. due to a reduction in operating voltage below that required by a given technology or for an acceptable soft error rate (due to reduced circuit stability) in conjunction with effective "fault recovery" mechanisms within the microarchitecture may yield a significant improvement in performance-to-power ratio while retaining the required bottom-line processor reliability [8].

Soft errors can thus no longer be ignored. Main memory, caches, array structures, and buses are often protected against data-level soft errors using mechanisms such as ECC. Therefore, this paper focuses on the handling of logic-circuit and control-flow soft errors within the CPU. While detection efficiency is paramount, the relative rarity of these errors makes the efficiency of the recovery mechanism a secondary concern [9], so the paper only considers detection.

Multiple soft error occurrences are, to a large extent, statistically independent events. Consequently, the probability of the exact same soft error occurring in two consecutive executions of an instruction is practically zero. In fact, when an increased soft error rate is the result of an intentional "excessive" reduction in the supply voltage, one can always choose the voltage such that the probability of double faults is sufficiently low. In designing soft-error detection mechanisms, one may thus assume at most a single such error at any given time.

• *A. Timor and A. Mendelson are with the Electrical Engineering Department, Technion, Haifa 31096, Israel, and also with Intel Corp., Haifa 31096, Israel. E-Mail: {aviel.timor, avi.mendelson}@intel.com.*
• *Y. Birk is with the Electrical Engineering Department, Technion, Haifa 32000, Israel. E-mail: birk@ee.technion.ac.il.*
• *N. Suri is with the Department of Computer Science, Darmstadt University of Technology, 64289 Darmstadt, Germany. E-mail: suri@cs.tu-darmstadt.de.*

Manuscript received 11 Feb. 2007; revised 22 Sept. 2007; accepted 29 Mar. 2008; published online 8 May 2008.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0015-0207.
Digital Object Identifier no. 10.1109/TDSC.2008.31.

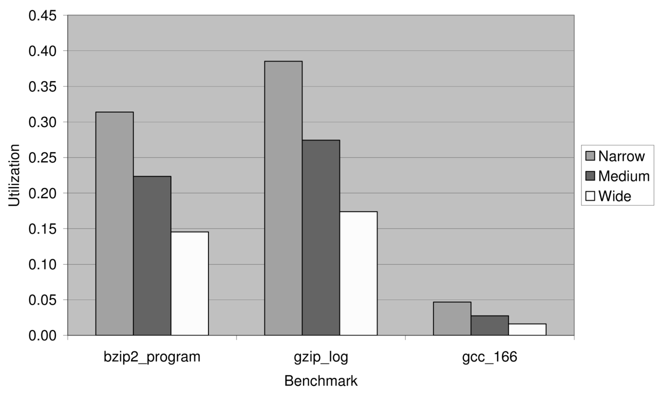


Fig. 1. Integer ALU utilization.

Most modern general purpose processors employ variations of "out of order" execution (also known as dynamic scheduling) to enhance the internal parallelism within the execution of serial code. Two basic mechanisms are 1) dynamic scheduling of the hardware that detects independent instructions and sends them for execution in their dependency order rather than in the original "program order" and 2) a recovery mechanism that allows the system to execute instructions speculatively and "roll back" if found to be wrong. Correct execution of a "truly" out of order machine can be very complicated, particularly if it is required to support precise exceptions (e.g., for error reporting), so most current processors divide the out of order execution into three stages—in-order front end, out-of-order execution, and in-order commit. This gives the user a virtual in-order machine while allowing the system to achieve the performance of "nearly optimal" parallel execution within a given window of instructions.

Key observation and drivers behind our approach: CPU resource underutilization. A primary goal of most processor architectures is increasing the number of instructions per cycle (IPC) by increasing the instruction-level parallelism (ILP). Techniques such as speculation, dynamic scheduling, and loop unrolling reduce the impact of data and control hazards, thereby enabling the processor to better exploit parallelism. The ideal IPC of a processor is the maximum parallelism that the microarchitecture allows for a given program [10]. In modern processors, hazards keep the IPC well below the ideal IPC. This implies low average utilization of processor resources such as execution units and internal structures such as arrays, which are consequently idle most of the time.

Fig. 1 depicts the integer ALU utilization for several benchmarks on three machine configurations (Narrow, Medium, and Wide pipelines) using the SimpleScalar architecture simulator (see Section 5 for details). It is evident that the integer ALUs are idle most of the time (average utilization is lower than 0.5), and we can assume that many other CPU resources (e.g., arrays, memories, and ALUs) also display similarly low utilization. The IPC of the bzip2_program and the gzip_log benchmarks is between 1 and 2 in all machine configurations, and still, at least in these examples, the ALU utilization is below 0.4.

The existence of underutilized CPU resources, along with the already existing recovery mechanism as part of the Out-of-Order micro-architecture, suggests that executing each instruction twice on the same hardware (and comparing the results) may enable achieving high reliability with relatively small performance loss. It is on this basis that we propose two novel schemes, **Double Execution** and **Double Decoding,** which utilize the inherent underutilization within processors in order to enhance their resilience to soft errors. In view of the relatively infrequent rate of soft errors, the performance impact of overcoming them is secondary. However, soft error detection must be active at all times and must therefore be efficient in all respects; this is precisely the focus of these two techniques.

The remainder of the paper is organized as follows: Section 2 describes contemporary approaches to handling soft errors; Sections 3 and 4 present our two microarchitecture techniques along with analysis of their correctness. Sections 5 and 6 present comparative simulations assessing these techniques for performance, memory latency, array sizes, queuing policies, and power consumption. Finally, Section 7 offers concluding remarks.

2 Related Work

A number of approaches for handling soft errors have been suggested and implemented [11]. Except for circuit design techniques to reduce the soft error probability, which are beyond the scope of this work, most solutions are based on redundancy and belong to two broad classes:

- *Spatial redundancy.* Hardware is added to the processor to detect or correct soft errors.
- *Temporal redundancy.* The soft errors are detected by using the same hardware to perform the same calculations more than once.

We briefly review some current approaches within this broad classification.

2.1 Spatial Redundancy

2.1.1 Error Detection and Correction Codes (ECC)

A given bit sequence is encoded into an expanded bit sequence. Unlike the original sequence, all values of which are "legal," the space of legal expanded sequences is sparse. Upon the detection of an "illegal" expanded sequence, one can either declare an error ("detection") or decide that the intended legal sequence was the closest one to the received one ("correction") [12].

Adding ECC to combinational logic blocks is both complicated and usually requires adding logic and calculations to the already timing-critical paths. Therefore, ECC is typically used only in memory arrays and in bus communication. Our proposed techniques target places that cannot be easily covered by ECC and are thus complementary to it.

2.1.2 Dynamic Implementation Verification Architecture (DIVA)

DIVA [13] is a microarchitecture-based technique that permits detection and recovery from both permanent and soft errors. A DIVA processor (Fig. 2) splits a traditional out-of-order processor design into two parts: the deeply speculative DIVA core and the functionally robust DIVA checker. The DIVA core is composed of the entire microprocessor design except the commit stage. The DIVA checker contains independent in-order computation logic that verifies the correctness of all core computations, only permitting a correct result to pass through to the commit stage. If any error is

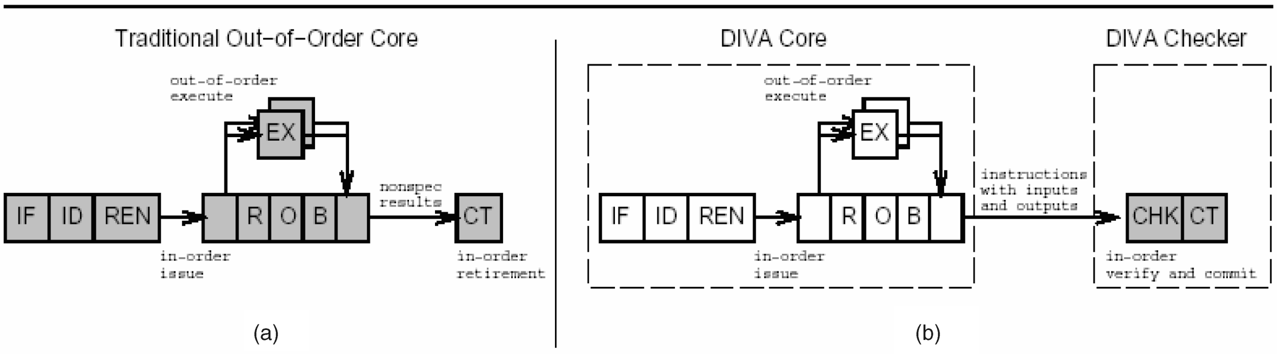


Fig. 2. (a) Traditional Out-of-Order core. (b) DIVA micro architechure. [13].

detected in the core computation, the checker fixes the computation, flushes the processor's pipeline, and then restarts the processor at the next instruction.

The DIVA architecture assumes that the DIVA checker is reliable, namely, that the memory structures have appropriate coding techniques, the design of the checker is fault free, and the communication between the core and the checkers is protected by coding techniques.

The DIVA architecture entails several costs: silicon area, power consumption, complex design, and performance degradation. In [13], only the performance degradation was evaluated and showed an average slowdown of 3 percent. Besides the direct performance loss, the DIVA checker is a significant hardware addition that comes at the expense of other potential performance-enhancing features. In our techniques, there is much less additional hardware, and the performance loss can be "switched off" whenever enhanced reliable operation is not required. Weaver and Austin [14] present an alternate fault-tolerant architecture based on principles similar to DIVA.

2.1.3 Replicated Hardware

One of the most common and simple fault-tolerant solutions is to use duplicate hardware and compare the results. The duplication can be done by using two identical CPUs or by duplicating execution units as part of the micro architecture. Systems like Compaq NonStop Himalaya detect soft errors by running identical copies of the same program on two identical cycle-synchronized microprocessors. In each cycle, both processors are fed identical inputs and a checker circuit compares their outputs [15]. In IBM S/390 systems, the execution units are duplicated and execute identical instructions and data. In every cycle, signals and instruction results from the duplicated execution units are compared for error detection in separate comparison logic [16]. The problem with these replication-based solutions is the cost of duplicating and synchronizing hardware, and the consequent increased power consumption without any performance gain.

2.2 Temporal Redundancy

In this section, we describe the temporal redundancy technique Simultaneous and Redundantly Threaded Processor (SRT). Other temporal redundancy techniques include [17], [18], [19], [20], [21], and [22].

2.2.1 Simultaneous and Redundantly Threaded Processor (SRT) [23], [24]

The basic idea of SRT is to run each program twice, as two identical threads, on a simultaneous multithreaded (SMT) processor. An SMT machine allows multiple independent threads to be executed simultaneously, even in the same cycle, in different functional units.

The SRT processor offers several benefits over hardware redundancy techniques. It can potentially outperform an equally sized hardware-replicated solution, as it partitions resources across execution copies dynamically rather than statically. Also, an SRT processor may require less hardware as it can share the same data path whenever redundancy is unnecessary (e.g., when the data path is protected by ECC). SRT processors may also be more economically viable as one can design a single processor that can be configured in either SRT or regular SMT mode, depending on the target system. Performance studies indicate that the average performance degradation of SRT relative to regular SMT with the same characteristics is about 32 percent and that SRT outperforms hardware-replicated processors by 16 percent on the average for equal processor areas. Later work suggested some performance improvement by using Slipstream Processors [25]. We will show how, by implementing the temporal redundancy inside the microarchitecture rather than in software, one can simultaneously attain the benefits of spatial redundancy (low-performance degradation) and those of temporal redundancy (little additional hardware).

3 Double Execution

3.1 Overview

In this section, we propose the Double Execution soft-error detection and recovery technique. It is based on utilizing existing out-of-order execution mechanisms and is superior to the solutions described in Section 2 in terms of device area, power, and performance.

Double Execution entails the execution of every instruction twice on the same hardware (temporal redundancy) and a comparison among the results of the two executions of each instruction for soft-error detection. The scheme exploits the fact that resources such as execution units and Out-of-Order arrays have low average utilization due to hazards. This permits the execution of every instruction twice with only a small impact on performance and power. The idea of executing each instruction twice, which is inherent in temporal redundancy, was introduced in a study of time-redundant fault-tolerant techniques for high-performance pipelined computers [26]. However, our Double Execution technique's novelty lies in the implementation level. Specifically, it exploits the Out-Of-Order microarchitecture hardware and enables better parallelism (and by that—performance). The basic idea of Double Execution was first introduced in [27] as the O^3RS approach, though primarily at a conceptual level without an actual implementation level study. Shared Resource Checker (SHREC) [28] carried the O^3RS approach a further step but kept the redundant execution in order, as implemented in DIVA [13]. In this work, we expand upon the basic ideas, establish their correctness, and evaluate them through extensive simulation of performance, as well as issues such as power consumption.

Using an Out-of-Order microarchitecture simplifies the implementation of Double Execution. When an instruction's first execution is completed, but before it reaches the commit stage, it is dispatched again for a second execution. An instruction is committed only after the second execution's result is ready and only if the two results are identical.

An Out-of-Order micro-architecture is suitable for Double Execution in several respects:

- The result of an instruction's execution in a conventional architecture is kept in the Reorder Buffer (ROB) in order to enforce In-Order commit. Double Execution uses the same array to keep the result of the first execution for comparison with that of the second one.
- The In-Order commit stage enables the speculative use of the first execution's result by instructions that depend on it, thereby obviating the need to wait for the second execution. If a soft error is detected after the second execution, all the instructions that followed the bad instruction are flushed from the pipeline as part of the error recovery flow (in the same way that the Out-of-Order architecture handles false speculations during the normal program run).
- When an instruction is dispatched for the second time, all the resource dependencies are guaranteed to have already been fulfilled (else, the instruction could not have been executed the first time). The Reservation Station (RS) structure is already designed to check these dependencies. It will mark the instructions, in their second executions, as ready for execution as soon as they are dispatched. Note that in its second execution, an instruction cannot be speculative if the first execution had not malfunctioned (due to a soft error) and is guaranteed to achieve the correct execution Thus, as long as soft errors are rare, the second execution can always be executed more efficiently.
- Branch instructions are also executed twice. The first execution's result can be used for branch misprediction resolution. (In the event of a fault, the pipe may be falsely flushed, but this does not affect correctness and, being rare, does not affect performance.)
- Load and Store instructions are executed twice (the address calculation part), but the actual memory access takes place once (data paths and memory are assumed to be ECC protected). In store instructions, the data is written to the memory (or to the store buffer) only after the second execution to prevent memory corruption in the event of a soft CPU error.

Double Execution dispatches an instruction for the second execution only when it reaches the commit stage. A falsely speculated instruction is thus often discovered to be one prior to its second dispatch, which is then cancelled. This reduces the resource-consumption penalty of false speculation. Ignoring soft errors that do not affect the program outcome is an established idea [29], [30].

Double Execution can easily be implemented on an Out-of-Order microarchitecture, without any new arrays, data paths, or other structures. The only required changes are the following:

- Mark each instruction in the Out-of-Order arrays as being dispatched for the first or second execution.
- Update the control logic in order to execute each instruction twice.
- Add a comparison mechanism in the ROB for soft-error detection.
- Add a recovery flow in order to correct a soft error once it has been detected.

3.2 Correctness

The "logical" correctness of Double Execution is obvious. In this section, we further establish that the scheme is deadlock and livelock free. This is done for two implementations of the dispatch and commit arrays. As mentioned earlier, we assume that all the memory arrays and main communication busses are fault-protected using ECC.

Implementation 1: Single array. Here, a single array is used for both dispatch and commit, serving both the first and second executions. An array entry is allocated to an instruction once it has been decoded (for the first time if decoded twice). This entry is used for both executions until the instruction completes the commit stage, at which time it is released.

Proposition 1. *Double Execution cannot cause a deadlock when using the same array for both dispatch and commit.*

Proof. By contradiction. Assume that the CPU is in a deadlock: no instruction is in execution, and all the instructions that are not committed cannot be executed due to unfulfilled dependencies. There are two complementary cases:

1. There are no instructions waiting for second execution (having completed their first execution). Because the first executions are allowed to proceed speculatively without waiting for second executions, this would occur regardless of Double Execution, thus contradicting the assumed deadlock-freedom of the original single-execution CPU.
2. There are instructions waiting for the second execution (the first execution has been completed). The completion of the first execution guarantees that all dependencies for these instructions have been fulfilled, so their second execution can begin. However, as mentioned above, as all execution units are free, these instructions will be executed in contradiction to

the deadlock assumption. (Note that an instruction does not "hang up" during its execution phase.) □

Implementation 2: Separate arrays. Here, separate arrays are used for dispatch and for commit, with each array serving both the first and second executions. After an instruction is decoded, it is allocated an entry in the dispatch array. This entry serves the instruction before, while, and after the execution stage. (An instruction requires no other "RSs" or other state-holding resources for which it may need to wait or which it may have to release in order to allow other instructions to proceed.) When an instruction reaches the commit stage following its first execution, it is allocated an entry in the commit array, and its entry in the dispatch array is released. For the second execution dispatch, a new entry in the dispatch queue needs to be allocated. The commit array entry remains allocated until the second execution is completed (keeps the first execution result).

Proposition 2. *Double Execution cannot cause a deadlock if and only if at least one entry in the dispatch array is reserved for second executions.*

Proof (necessity). If no entries in the dispatch array is reserved for the second executions, no instruction can commit pending second execution, no instruction can be dispatched for the second execution because the dispatch array is full with the first execution instructions, and these cannot release their dispatch array entries because they cannot be allocated entries in the commit array, which is filled with instructions awaiting the second execution. □

Proof (sufficiency). By contradiction, assume that the CPU is in a deadlock: no instruction is in execution, and all the instructions that are not committed cannot be executed due to unfulfilled dependencies. There are two complementary cases:

1. There are no instructions waiting for second execution (having completed their first execution): same as Case 1 in *Proposition 1*.
2. There are instructions waiting for second execution. As at least one entry in the dispatch array is reserved for second executions, there is at least one instruction in the dispatch queue that is ready for second execution. As all execution units are free and all dependencies have been fulfilled, this instruction will be executed, contradicting the deadlock assumption. □

Proposition 3. *Double Execution cannot cause livelock.*

Proof (Case 1). There are no soft errors inside the CPU. In this case, every instruction is executed only twice and is thus unable to circulate indefinitely in the CPU, so livelock is impossible. □

Proof (Case 2). There is a soft error in the first execution of an instruction. If the error goes undetected (the same error occurs in the second execution), execution will proceed as it would in a single-execution CPU, and no livelock will arise. (Of course, the result is incorrect, and in any case, this situation contradicts the assumption that a fault does not repeat in both executions of a given instruction.) If the error is detected following the second execution, the pipe is flushed, and the faulty instruction along with those that followed it (and were not allowed to commit because of their dependence on this unverified instruction) are reexecuted. Therefore, livelock can only occur if a soft error occurs repeatedly in every first or second execution of the same instruction in contradiction to the basic soft-error assumptions (see Section 1). □

Proof (Case 3). There is a soft error in the second execution of an instruction. Hence, the fault is detected when the instruction reaches the commit stage and the pipe is flushed, similar to Case 2. □

Due to the assumption of a single fault in the system at any given time (see Section 1), Cases 1-3 cover all possible scenarios, and therefore, no livelock is possible.

3.3 Instruction Queuing Policies

In this section, we suggest several queuing policies for use by the RS in dispatching the instructions to the execution units. As explained above, an instruction that depends on the result of another instruction can use the results of the first execution of that instruction. However, the second execution still affects the program runtime for the following reasons:

- Memory store operations are executed only after the commit stage (In-Order), so they must wait for the second execution to be completed (calculations of both the data and address).
- Double Execution causes the Out-of-Order arrays (such as RS and ROB) to keep instructions for longer periods of time, causing the average occupancies of these arrays to be higher. Whenever the arrays are fully occupied, the pipeline is stalled, which degrades performance.

There is thus a dilemma in choosing an execution queuing policy: advancement of the program favors assigning higher priority to the first executions, as the second executions are not needed for the program flow (from a data dependency point of view); however, freeing up the out-of-order arrays in order to make room for new instructions and prevent pipeline stalls favors assigning higher priority to the second executions. In Section 6, we assess several execution queuing policies:

1. **Program order.** The instructions are executed in their program order as long as their sources are ready, and the execution units are free. Second executions thus usually have higher priority than first ones (of other instructions).
2. **Low-priority second execution.** An instruction is dispatched for the second execution only if no instruction can be dispatched for the first execution in the same execution unit.
3. **First come first served (FCFS).** Instructions to be executed for the second time are treated as new arrivals in the RS, and there is no distinction between the first and second executions. In this case, there is no preferring of the first over the second execution, or vice versa.
4. **Random order.** All instructions (both first and second executions) are executed in random order.

4 DOUBLE DECODING

4.1 Introduction

An instruction decoder usually has a significant impact on the processor's performance and is usually a large timing-critical block that uses many speculations in parallel. Unlike the execution units, however, the decoder is heavily utilized, so an identical approach to Double Execution is inapplicable. In this section, we propose a scheme, based on spatial redundancy, for this unit. Our scheme, Double Decoding, attempts to protect the decoder with less additional hardware and a smaller performance penalty than brute force duplicate decoders.

4.2 Scheme Overview

Double Decoding combines three main elements:

- *Duplicate instruction decoding.* Each instruction is decoded twice by different hardware.
- *Low-performance redundant decoder.* The original decoder's result can be used speculatively (until the comparison). Thus, the redundant decoder, while functionally identical to the original, may have higher latency and be simpler. It does not require the speculation capability and parallel logic that the original decoder utilizes for performance enhancement. Simulations (Section 6) confirm that this decoder can be much smaller and consume less power than the original decoder, without impacting overall performance.
- *New decoding check stage that precedes the instruction commit stage.* The redundant decoder is placed in a new pipe stage, right before the commit stage (at the end of the pipe). By so doing, we only need to decode twice those instructions that reach the commit stage, ignoring those that were fetched by false speculations, thereby reducing power consumption. This requires a new array for holding the instructions that were fetched (before decoding).

4.3 Correctness

We now prove that a program always recovers correctly from a speculative run caused by a soft error in the decoding stage. The concern is that an error in the decoder may affect the program flow, and the CPU will not be able to recover from it. For the correctness proof, it does not really matter at what stage the soft error is detected, as long as it is detected before the commit stage.

Proposition 4. *With Double Decoding, any single soft instruction decoding error is recoverable.*

Proof. There are three cases of a single soft error in the decoding stage:

- A soft error in the decoder with no effect on the program flow. In this case, the CPU may perform wrong instructions or wrong calculations speculatively, but once the error is detected by the second decoding, the pipe will be flushed and the calculations will be reexecuted correctly. The wrong calculations do not have any effect on the CPU state after the pipe is flushed because speculative instructions are not allowed to commit.
- A soft error in the decoder changing the program flow. In this case, the CPU may speculatively perform a totally wrong program. However, the speculative instructions will not be committed before the error is detected and therefore will be flushed and reexecuted correctly. The speculative program flow may adversely affect the branch prediction tables, but these tables only affect performance and not program correctness.
- A soft error in the redundant decoder. In this case, there is no real error in the program run, though the CPU will assume such an error in the comparison stage and will reexecute the program correctly.

The adverse effect of Double Decoding on the branch prediction tables is typically negligible in view of the rarity of soft decoding errors and can thus be ignored. Alternatively, in order to permit efficient operation even when these errors are not sufficiently rare (as will be discussed later in Section 6), we propose to update the branch prediction tables only in the commit stage. The false speculation instructions caused by soft errors will then be detected (and flushed) before the branch prediction tables are affected. □

In the next sections, we present a simulation study of Double Execution and Double Decoding.

TABLE 1
Basic Simulation Parameters

Feature	Simulation Parameters
Level 1 data cache	32K, 4-way, LRU, 1-cycle latency
Level 1 instruction cache	32K, 4-way, LRU, 1-cycle latency
Level 2 combined data & instruction cache	512K, 4-way, LRU, 8-cycle latency
Branch Predictor	Bi-modal, 2-level
Instruction TLB	64K, 4-way, LRU
Data TLB	128K, 4-way, LRU
Memory access latency	200 CPU cycles
Simulation length	10^9 instructions
Simulation "warm up"	$2*10^8$ instructions

5 SIMULATION PLATFORM

The previous sections have outlined the Double Execution and Double Decoding schemes and established their correctness. In this section, we describe the simulation platform that is used to explore the various performance and related impacts of the proposed schemes.

5.1 Architecture Performance Simulator

The performance simulator used in this study was derived from the *SimpleScalar 3.0 tool set* [31], a suite of functional and timing simulation tools for the Alpha AXP instruction set architecture. We use the *sim-outorder* (version 7) simulator, a computer simulation tool that provides detailed simulations of a high-performance dynamic-scheduling modern microprocessor. Several of the simulator's procedures were altered (Sections 5.3 and 5.4 provide details) in order to simulate the new schemes. The simulations were performed on the "SPEC CPU 2000" benchmark set using the system

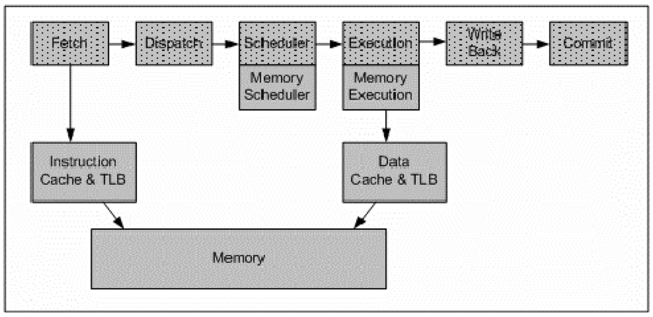


Fig. 3. Sim-outorder simulator pipe stages (taken from the “SimpleScalar Hacker’s Guide” [32]).

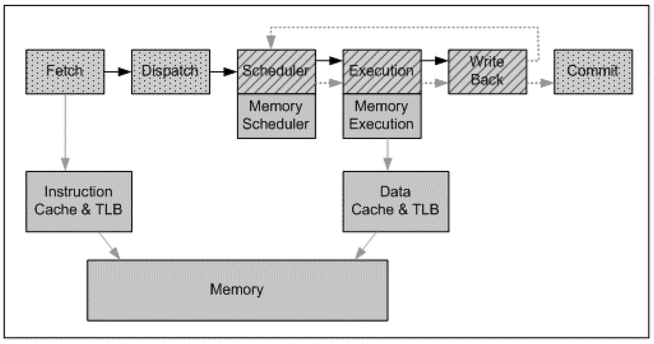


Fig. 4. Sim-outorder Double Execution implementation.

configuration depicted in Table 1. These parameters were chosen to represent reasonable figures based on the current CPUs in the market. In some simulations (e.g., array sizes impact on performance), we show that in most cases, these parameters do not change the result significantly.

Fig. 3 depicts the *sim-outorder* simulator pipe stages. The stages in undotted gray only apply to memory instructions.

5.2 Architecture Power Simulator

The dynamic power consumption simulator used in this study was derived from *Wattch version 1.02* power-performance simulator [33]. *Wattch* augments the *SimpleScalar* cycle-accurate simulator (*sim-outorder*) with a cycle-by-cycle tracking of power dissipation by estimating unit capacitances and activity factors. The same alterations that were made for the *SimpleScalar* performance simulations were also made to the *Wattch* power simulator (Sections 5.3 and 5.4).

5.3 Double Execution Implementation

When an instruction completes the write-back stage following the first execution, it is sent back to the “ready for execution” instruction queue. Instructions that depend on the output of this instruction are marked as ready (for execution) after its first execution. The *SimpleScalar* architecture uses one array for both dispatch and commit stages, and therefore, the Double Execution scheme cannot cause any deadlocks (see Section 3.2). The *readyq_enqueue()* function has several implementations for the different queuing policies. Fig. 4 depicts the *sim-outorder* simulator with support for Double Execution. The dotted arrows describe the instruction flow after the first execution. Diagonal stripes denote stages that are executed twice.

5.4 Double Decoding Implementation

The Double Decoding implementation in *SimpleScalar* includes the new decoding check pipe stage, carried out by a low-performance low-power decoder, before the instruction commit stage. The low-performance decoder is expressed by the addition of several cycles to the latency (the new decoder’s latency) and the twofold reduction of throughput relative to the machine widths (or the commit stage throughput, in our case). See Section 4 for details.

Our conjecture was that delaying the commit stage by a few cycles entails negligible performance impact. This was proven correct in the Double Execution implementation, where the commit stage was delayed for multiple cycles with very small impact on performance. The impact of the low-performance decoder’s throughput was simulated by reducing the commit stage throughput by 50 percent. This has the same performance impact as simulating the new pipe stage with the lower throughput. Since the new low-performance decoder’s implementation is outside the scope of this work, the reduction of the throughput is mainly to prove the concept rather than for accurate modeling. In summary, Double Decoding was simulated by only changing a *SimpleScalar* machine parameter (commit stage width), without any code alterations.

5.5 Machine Configurations

Three basic machine (CPU) configurations were defined for the simulations, differing in pipeline width and hardware resources: *Narrow, Medium, and Wide* machines. The Narrow and Medium machines represent current CPU designs: the Narrow machine represents standard or simple processors, while the Medium machine is similar to the higher end commercial processors. The Wide configuration represents potential future high-end processors. Table 2 describes the basic characteristics of each machine configuration and their differences.

5.6 Simulations

All simulations were done on both the original *SimpleScalar* (or *Wattch*) code (reference model) and on the new Double

TABLE 2
Basic Characteristics of Machine Configurations

Topic	Parameter	Machine Width		
		Narrow	Medium	Wide
Pipeline width	Fetch	2	4	8
	Decode	2	4	8
	Issue	2	4	8
	Commit	2	4	8
Array sizes	Register Update Unit (RUU) size	64	128	256
	Load Store Queue (LSQ) size	32	32	32
Number of ALUs	Integer adder	2	4	8
	Integer Multiplier	1	1	1
	FP adder	1	1	2
	FP Multiplier	1	1	1
	AGU (Address Gen. Unit)	1	1	1

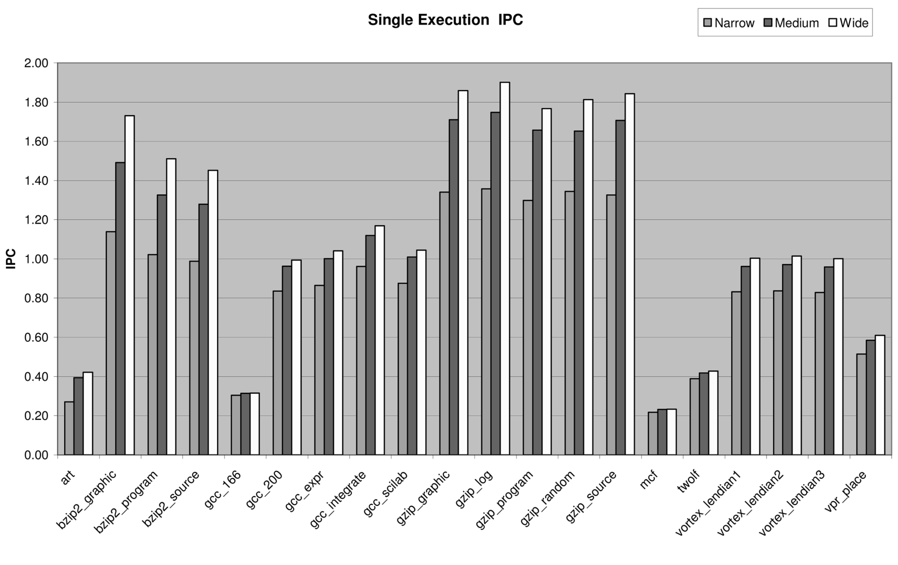


Fig. 5. Baseline (single execution) IPC with Narrow, Medium, and Wide configurations.

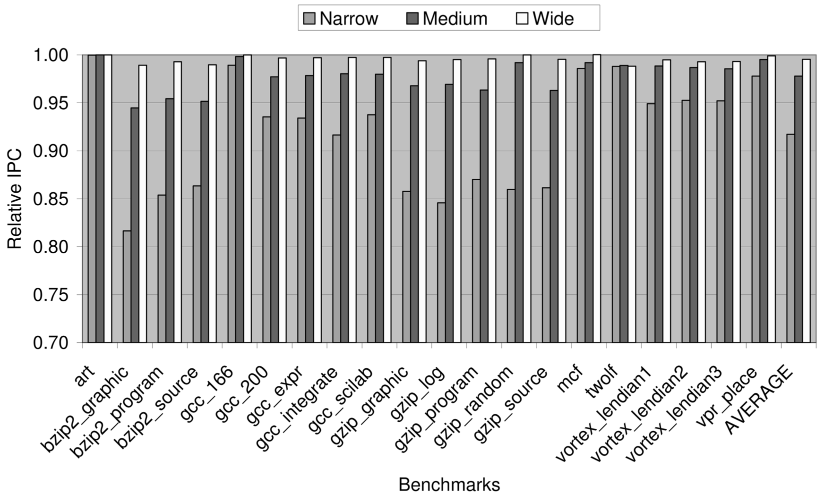


Fig. 6. Double Execution versus single execution performance.

Execution and/or Double Decoding models. The simulations were performed using the three machine configurations (Narrow, Medium, and Wide).

The following simulations were performed:

1. Double Execution:
   - Twenty different benchmarks, comparing the reference model with the Double Execution model (using the overall best queuing policy).
   - The influence of memory access latency on the overall average performance.
   - The influence of the RUU and LSQ array sizes on the overall average performance.
   - Overall average performance and instantaneous performance along the program run with the different queuing policies.
   - Effect of soft-error probability on the average performance when using Double Execution.
   - Performance degradation of an "ideal" processor.
2. Double Decoding performance relative to the original model, with and without Double Execution.
3. Dynamic power consumption with Double Execution relative to the reference model, comparing the total energy and average cycle power.

6 Results and Analysis

6.1 Baseline IPC

Prior to discussing the impact of double execution on overall system performance, the baseline IPC for various applications with Narrow, Medium, and Wide configurations are required.

As depicted in Fig. 5, the "baseline" IPC achieved when running on a "native" system without Double Execution increases with the width of the machine. Some applications such as ART, GCC, and MCF are known to be memory bounded and so present a relatively low IPC, while others such as GZIP and BZIP can utilize the resources much better.

It should also be noted that our configuration includes an address generation unit (AGU) that allows more address calculation operations to be performed concurrently with other operations. This feature is less important when the resources of the machine are lightly utilized but becomes

TABLE 3
Double Execution Performance Summary

Performance (IPC ratio)	Narrow	Medium	Wide
Average	0.917	0.978	0.995
Worst Case	0.817	0.945	0.988

more important when resource utilization is higher, for example, in the case of Double Execution. The AGU is therefore expected to mitigate the performance degradation of Double Execution relative to single execution.

6.2 Double Execution—Performance

We compared Double Execution with the original *SimpleScalar* on 20 different benchmarks. Fig. 6 depicts the ratio of the new mean number of IPC to the baseline (single execution) IPC for each benchmark and for the three machine widths that were described in Section 5.5. The queuing policy is a "Low-priority second execution" (see Section 3.3), which resulted in the best enhancements.

The results clearly indicate that in almost all benchmarks, the Wider the machine, the smaller the performance degradation due to Double Execution. As summarized in Table 3, the degradation is very small for both the Wide and Medium machines.

6.3 Double Execution—Memory Access Latency

These simulations assess the influence of the memory access latency (in the range 10-640 CPU cycles) on Double Execution's relative performance using the "low-priority second execution" queuing policy (Section 3.3). Several benchmarks were run, and we found *gcc_scilab* to offer representative results. Fig. 7 depicts the performance (IPC) of Double Execution relative to that of the single execution scheme using the different memory latency values and the three machine widths that were described in Section 5.5. The level 2 cache miss rates in these simulations were around 5 percent out of the total memory accesses.

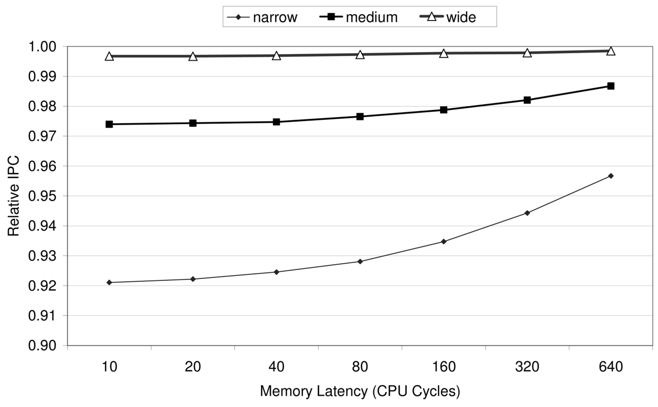


Fig. 7. Memory latency impact on performance degradation.

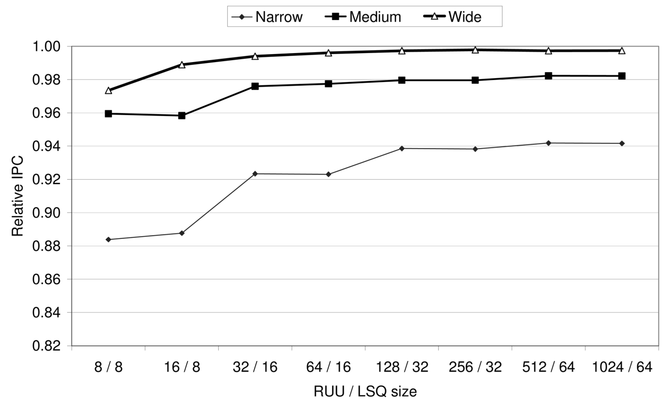


Fig. 8. Array size impact on performance degradation.

The results show a clear trend: the longer the memory access latency, the smaller the performance loss due to Double Execution. The largest impact is on the Narrow machine, whose performance degradation ranges from around 8 percent (memory access latency of 10 cycles) to 4.3 percent (latency of 640 cycles). Over the years, CPU frequencies have been rising faster than those of the memory, so memory access latencies (in CPU cycles) are increasing. Therefore, the performance loss due to Double Execution will become smaller with time as long as this trend continues.

6.4 Double Execution—Array Sizes

These simulations assess the impact of the out-of-order array sizes (RUU and LSQ arrays in *SimpleScalar*) on the relative performance of Double Execution. RUU sizes of 8-1,024 entries and LSQ sizes of 8-64 entries were simulated. For these simulations, we used the "*gcc scilab*" benchmark with the "low-priority second execution" queuing policy (see Section 3.3). We ran these simulations on various benchmarks and found *gcc_scilab* to be representative.

Fig. 8 depicts the relative IPC of Double Execution with different array sizes and the three machine widths of Section 5.5.

The results show that the larger the arrays, the smaller the performance loss of Double Execution. In some cases, in fact, enlarging the arrays can overcome the performance loss altogether. As the design trend is for these arrays to become larger, the performance loss will become smaller with time.

6.5 Double Execution—Queuing Policies

These simulations compare the performance (IPC) of the different instruction queuing policies (Section 3.3). We ran all benchmarks with the different machine configurations (Narrow, Medium, and Wide) with the four queuing policies: program order execution, FCFS, low-priority second execution, and random priority among the first and second executions.

The results for the *Narrow and Medium* machines are consistent—the "Low-priority second execution" queuing policy yields the best result (average relative IPC of 91.4 percent in Narrow and 97.4 percent in Medium; see Figs. 9 and 10). The "FCFS" policy is always second best (an average of 89.4 percent in Narrow and 96.4 percent

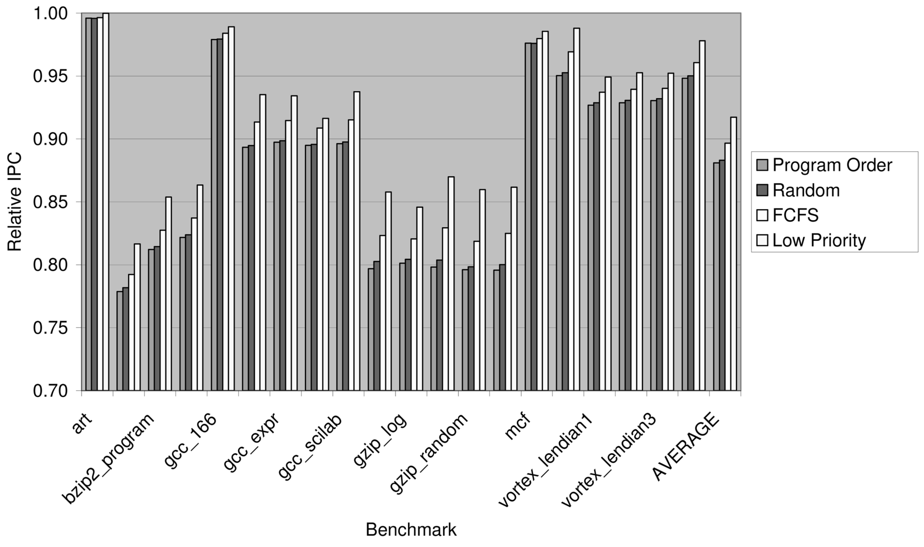


Fig. 9. Relative performance with different queuing policies (Narrow machine).

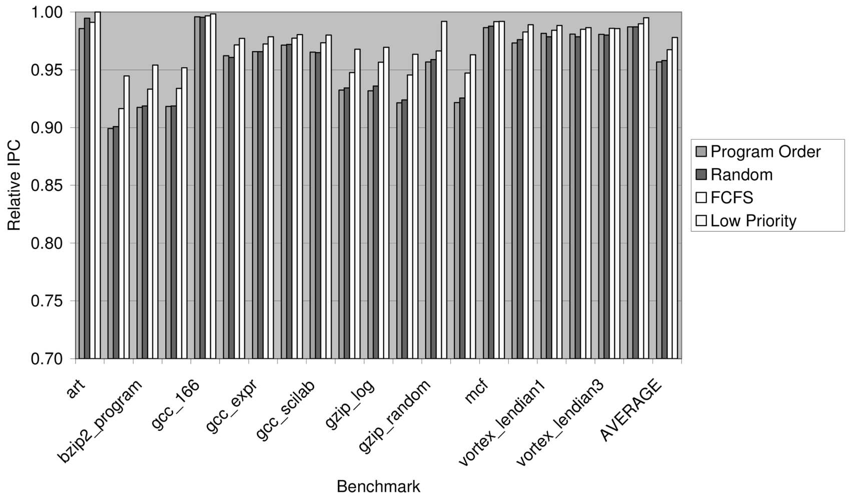


Fig. 10. Relative performance with different queuing policies (Medium machine).

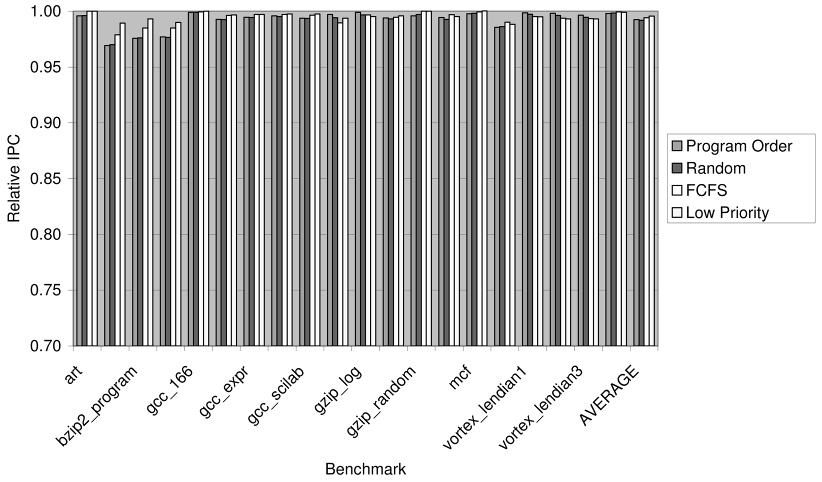


Fig. 11. Relative performance with different queuing policies (Wide machine).

in Medium). The "Program order" policy has the worst performance in all tests (an average of 87.8 percent in Narrow and 95.3 percent in Medium). The random queuing results are in the middle - better than "Program order" and worse than "Low-priority second execution."

For the *Wide* machine, the results are less conclusive (Fig. 11). Though on the average over all benchmarks the different queuing policies came in the same order as for the Narrow and Medium machines, there are some benchmarks for which the results were different:

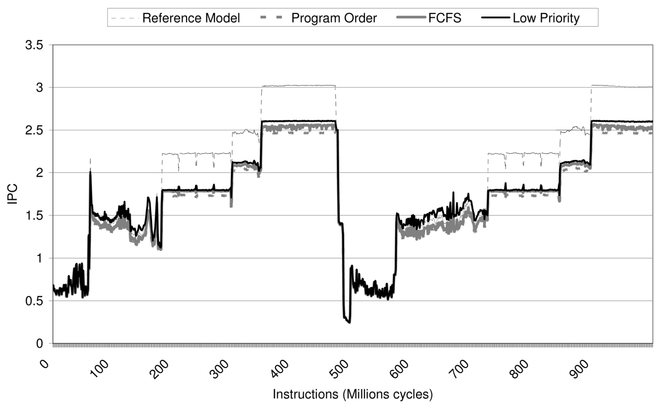


Fig. 12. IPC along program execution with the various queuing policies (BZip benchmark; Medium machine).

- In 20 percent of the benchmarks, the “FCFS” gives the best performance.
- In 25 percent of the benchmarks, the “Program order” gives the best performance.
- In the rest (55 percent) of the benchmarks, the results are in the same order as for the Narrow and Medium configurations (“Low-priority second execution” gives the best results).

The reason for the less conclusive result is that in the Wide machine there are many free resources almost all the time, so the Double Execution penalty is extremely small. Hence, the advantages of “Low-priority second execution” are minor for several benchmarks. The bottom line is therefore that one may still assign lower priority to second executions in all cases.

6.6 Double Execution—The Potential Benefit of a Dynamic Queuing Policy

In the previous section, we showed that “low-priority second execution” is the best result in most cases. Here, we study the potential benefit of dynamically changing the queuing policy along the program run based on program characteristics. We compare the instantaneous (using a sliding window) performance with the different queuing policies along the program runs and derive the would-be performance with an ideal dynamic scheduler. Specifically, we calculated the average IPC over every 10^6 consecutive instructions along the 10^9 instructions of the benchmark programs. We used all three machine configurations and several of the common benchmarks. Fig. 12 depicts the IPC of a Medium width machine while running the BZip benchmark. The reference model is the single execution scheme.

In this representative example, we can see that the different queuing policies’ performance is relatively stable along the program runs (meaning that the best policy is the same policy most of the time). This means that, at least in these cases, there is very limited potential for performance gain by using a dynamic queuing policy. There were several benchmarks in which performance was less consistent (in each period, a different policy gave better performance). However, because these were only sporadic cases, a dynamic queuing policy mechanism does not seem worthwhile. Nevertheless, it is possible that studies using wider ranges of workloads and conditions will show greater justification for the use of a dynamic queuing policy.

6.7 Double Execution—Impact of Soft-Error Probability on Performance

We now explore driving the processor technology to the limit and intentionally operating in a regime that retains independence (single failure at a time) but has more frequent soft errors. This is done to assess the trade-off between the added performance and/or reduced power consumption brought about by pushing the technology and the degradation brought about by the need for error recovery. In these simulations, we examine the performance impact of soft-error probability, assuming that the errors are recovered using Double Execution. We used the “*bzip2 program*” benchmark with the “low-priority second execution” queuing policy (Section 3.3). Three machine configurations were simulated across a broad range of soft-error probabilities ($5 \cdot 10^{-10}$ to 10^{-1}). Fig. 13 depicts the relative performance (IPC) of Double Execution versus error probability. The higher the soft error rate, the larger the performance degradation—for each soft error, the pipe is flushed and instructions must be reexecuted. Also, for extremely high soft-error rates (one every 5,000 instructions or less on average), the wider the machine, the worse

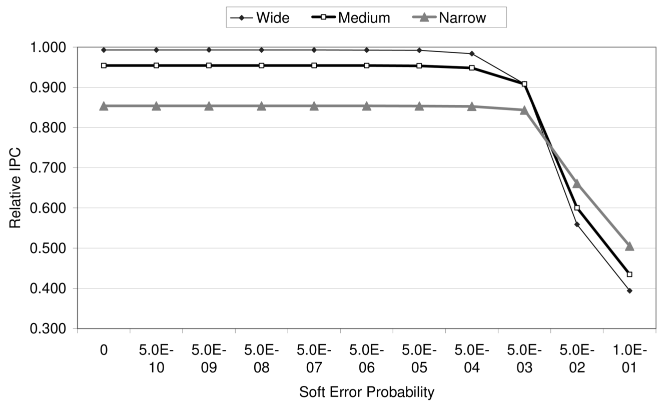


Fig. 13. Soft-error probability impact on performance.

TABLE 4
Double Execution Performance of "Ideal" Processor Summary

Performance (IPC ratio)	Narrow	Medium	Wide
Average	0.776	0.892	0.973
Worst Case	0.675	0.781	0.885

the performance. The reason is the larger pipe-flush and refill penalty for the wider machine.

However, the results suggest that even for a soft error probability of $5 \cdot 10^{-5}$, which is extremely high (> 50,000 soft errors per second in a 2-Ghz Processor), the performance loss is under 0.1 percent relative to the Double Execution performance numbers (with no faults). Thus, in a Medium machine, for example, the performance loss of running Double Execution with soft error probability of $5 \cdot 10^{-5}$ is around 5.7 percent relative to a single execution machine with no soft errors. Hence, Double Execution can be used to "squeeze" the CPU technology. The CPU can be "stretched" (e.g., higher frequency, lower voltage, and smaller noise margins), causing more errors that will be detected and recovered using Double Execution. The overall performance, power, or other parameters can be made better than the original CPU, without exposing the user to the higher "raw" error rate. Note that the evaluation of this approach must be carried out under conditions that equate the probabilities of undetected soft errors. The relevant error probability for Double Execution is the probability that both executions of a given instruction are faulty, which is approximately the square of the probability for a single soft error in one instruction. Thus, the "raw" error probability for Double Execution may be set as high as the square root of the tolerable soft-error probability of a single-execution machine. A similar approach is common in other fields such as storage and CPU cache design [8], where error detection and recovery mechanisms are widely used, and has been suggested for other parts of CPU design as well [34].

6.8 Double Execution—Performance of an "Ideal" Processor

Via simulations, we estimate the lower bound on Double Execution performance (Table 4). One might claim that since Double Execution is based on the resources' low utilization, the performance will not be as good as shown so far when advanced features are being used (e.g., using advanced compiler optimizations, multithread processor, etc.). In these simulations, we used a perfect branch predictor (100 percent hit rate) and no cache miss penalty (the delay of reading data from the memory is the same as from the level 2 cache). These parameters are, of course, unrealistic. However, these simulations provide a lower bound on the performance of Double Execution relative to that of single execution. Fig. 14 depicts the ratio of the new mean number of IPC to the baseline (single execution) IPC for each benchmark and for the three machine widths that were described in Section 5.5.

As expected, the results indicate that Double Execution's relative performance is worst when using an "ideal" processor since the resource utilization is higher when there are no branch mispredictions and no cache miss penalty. However, even when using this kind of unrealistic processor, the performance degradation is not as bad as in

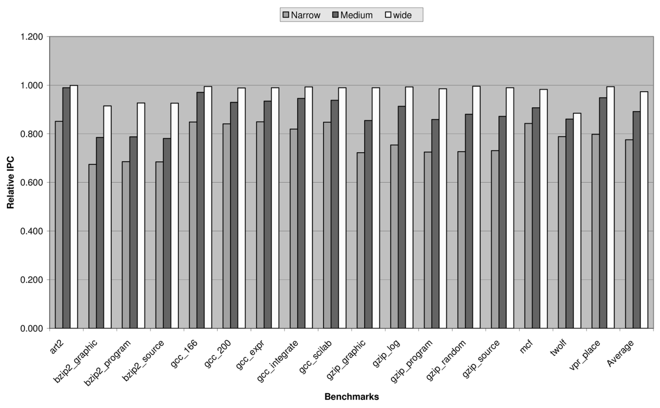


Fig. 14. Double Execution performance of an "ideal" processor.

most of the known transient fault solutions—an average degradation of only 10.8 percent in the Medium machine configuration and 2.7 percent in the Wide configuration. In the Narrow configuration, where the execution units and other resources are almost fully utilized (average of 1.75 IPC), the degradation is more significant.

A closer look at the results for the Narrow configuration raises a different question: how can the performance drop be quite moderate (22.4 percent on average) despite the high resource utilization in single execution? One possible explanation is that the baseline IPC (for a single execution) was not perfect due to data dependencies, so many of the "reexecuted" instructions could be executed within these free execution slots.

Several of our benchmarks had a near perfect "ideal IPC," as depicted in Table 5. Therefore, we examined closely those applications whose IDEAL_IPC (Narrow configuration) is greater than 1.85. We found out that two main factors contribute to these results: 1) not all the instructions are executed twice (in our system, loads and stores are executed only once since we assume that they are protected via other mechanisms), and 2) the system contains additional resources such as Floating point and AGU (Table 2), so the theoretical "perfect IPC," relative to which the degradation was computed, is actually greater than 2.

As seen in Table 6, the fraction of load and store instructions (that do not execute twice) is indeed quite significant in the aforementioned applications. Correspond-

TABLE 5
Basic versus IDEAL IPC

	IPC					
	Narrow		Medium		Wide	
Applications	basic	IDEAL	Basic	IDEAL	Basic	IDEAL
art	0.27	1.71	0.39	2.41	0.42	2.57
bzip2_graphic	1.14	1.91	1.49	3.29	1.73	5.19
bzip2_program	1.02	1.90	1.32	3.31	1.51	5.15
bzip2_source	0.99	1.90	1.28	3.34	1.45	5.16
gcc_166	0.30	1.67	0.31	2.24	0.31	2.55
gcc_200	0.84	1.53	0.96	2.22	0.99	2.59
gcc_expr	0.86	1.50	1.00	2.14	1.04	2.47
gcc_integrate	0.96	1.69	1.12	2.36	1.17	2.68
gcc_scilab	0.88	1.54	1.01	2.20	1.04	2.55
gzip_graphic	1.34	1.87	1.71	3.05	1.86	3.82
gzip_log	1.36	1.75	1.75	2.58	1.90	3.15
gzip_program	1.30	1.85	1.66	2.95	1.77	3.64
gzip_random	1.34	1.86	1.65	2.93	1.81	3.58
gzip_source	1.33	1.82	1.71	2.83	1.84	3.47
mcf	0.22	1.70	0.23	2.54	0.23	2.91
twolf	0.39	1.72	0.42	2.53	0.43	2.84
vpr_place	0.51	1.80	0.58	2.70	0.61	3.09

TABLE 6
Memory Operations and Basic IPC

	IDEAL IPC	%MEM
bzip2_graphic	1.91	30.1%
bzip2_program	1.90	29.1%
bzip2_source	1.90	29.3%
gzip_graphic	1.87	32.8%
gzip_program	1.85	32.0%
gzip_random	1.86	31.4%

ingly, Fig. 15 presents the results of the optimal execution with and without a separate AGU (also refer to Table 7).

It is evident that when all AGU operations are being executed on the general purpose integer units, the utilization of these resources is higher, and the performance loss due to double execution with Narrow configuration is indeed significant. With Medium and Wide configurations, the impact of the AGU on the overall slowdown is reduced and becomes relatively small even for the ideal case, when all benchmarks are considered. Finally, when actual memory latencies and branch predictions were considered, we found that the impact of the AGU on overall performance degradation due to Double Execution is negligible.

6.9 Double Execution—Power Simulations

These simulations used the *Wattch* power simulator to estimate the relative increase in the CPU active power consumption (expressed in percent of power consumption of single execution) of Double Execution using 20 different benchmarks. The results are the total relative active power increase (bars) and the average per cycle active power increase (the lines in the graphs). The average cycle power is important to determine the thermal characteristics of the CPU. The *Wattch* simulator assumes power cost for every CPU block of every active cycle and sums up the total power consumption. This, of course, is not accurate modeling, and its purpose is only to provide the power consumption growth trend. Figs. 16, 17, and 18 depict the

TABLE 7
Average Slowdown with and without AGU

Narrow		Medium		Wide	
AGU	No AGU	AGU	No AGU	AGU	No AGU
77.60%	63.70%	89.20%	78.20%	97.30%	93.90%

active power increase for the different benchmarks with the three machine configurations.

The results show that the overall active power, which represents the total energy consumption per program execution, increases on the average by 15 percent-17 percent for all machine sizes. However, the average per-cycle power increase, which is important for the heat-dissipation requirements, is around 5.4 percent in the Narrow machine and around 16.4 percent in the Wide machine. The reason for these results is the fact that the Wider machine has better parallelism, causing the execution units to be more active in the Double Execution scheme. It is important to note that the power simulations only calculated the active power. Double Execution does not significantly affect the leakage power (there is almost no additional hardware), which means that the overall power increase (both total and average per cycle) is actually smaller. Considering the clear trend of leakage power constituting a growing fraction of total power over the years, we predict that the effect of Double Execution on total power will become less significant over process evolutions.

6.10 Double Decoding—Performance

These simulations compare the performance of Double Decoding with and without Double Execution with the original *SimpleScalar* code for 20 different benchmarks. The results are the relative IPC. The queuing policy for the Double Execution model is "low-priority second execution" (Section 3.3), which gave the best results. Figs. 19, 20, and 21 depict the performance results of all benchmarks using the three machine widths (also refer to Table 8).

The results clearly indicate that in almost all benchmarks, wider machines exhibit smaller performance degradation. Also, the use of Double Execution along with Double Decoding incurs a very small additional performance penalty over that of Double Decoding alone. This is probably

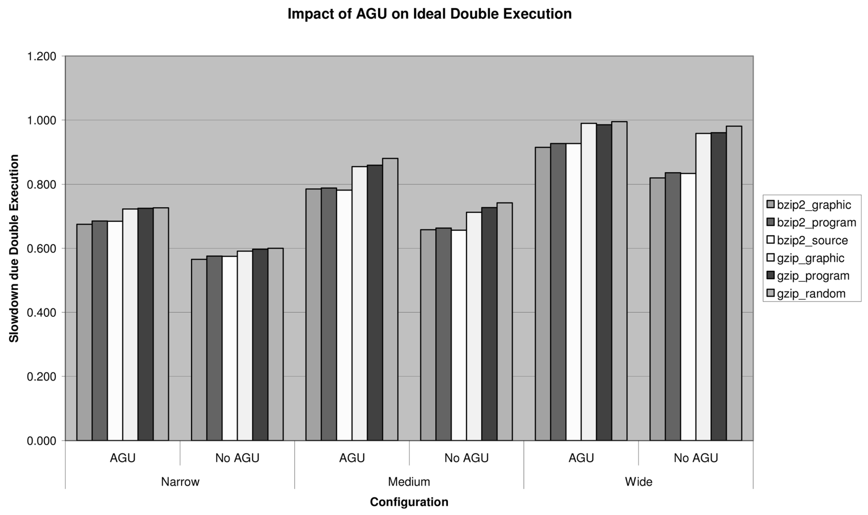


Fig. 15. Impact of AGU on Double Execution.

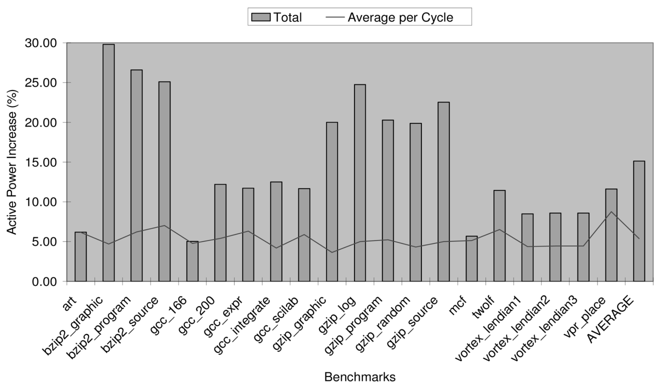


Fig. 16. Double Execution relative active power increase (Narrow machine).

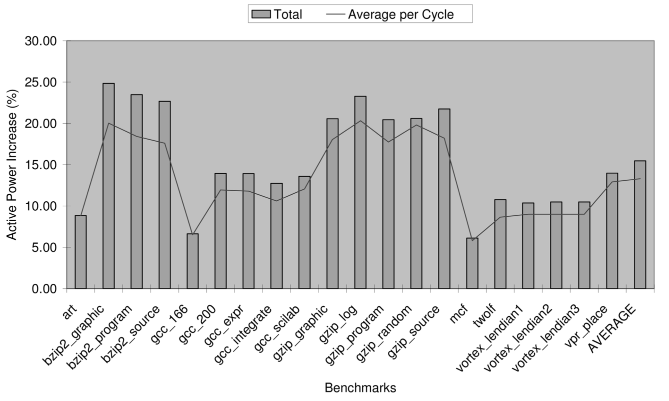


Fig. 17. Double Execution relative active power increase (Medium machine).

because the new Double Decoding pipe stage reduces the overall resource utilization, thereby increasing the availability of resources for the second execution.

In the Wide machine, the average performance loss is 2.2 percent-2.5 percent, and the worst case is less than 8 percent. In the Narrow machine, the performance loss is much more significant with an average of 17.9 percent-18.4 percent and a worst case of more than 32 percent.

Note that in Narrow machines, the cost of duplicating the original high-performance decoders instead of using low-performance decoders is usually less significant, and in that case, there is almost no performance loss (see Section 4.2 for more details).

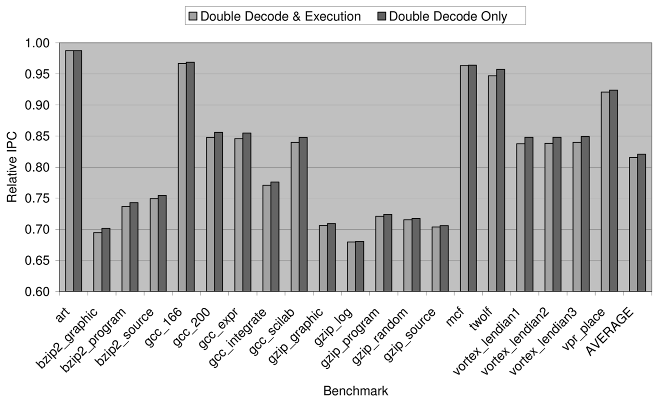


Fig. 19. Double Decoding relative performance (Narrow machine).

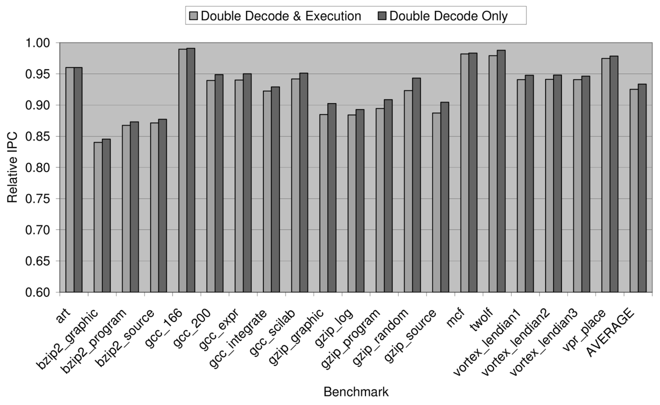


Fig. 20. Double Decoding relative performance (Medium machine).

7 Conclusions

With modern commodity processors increasingly prone to soft errors due to smaller feature sizes, reduced voltage levels, higher transistor counts, and reduced noise margins, efficient detection of soft errors is becoming a mainstream approach. As long as the soft-error rate remains relatively

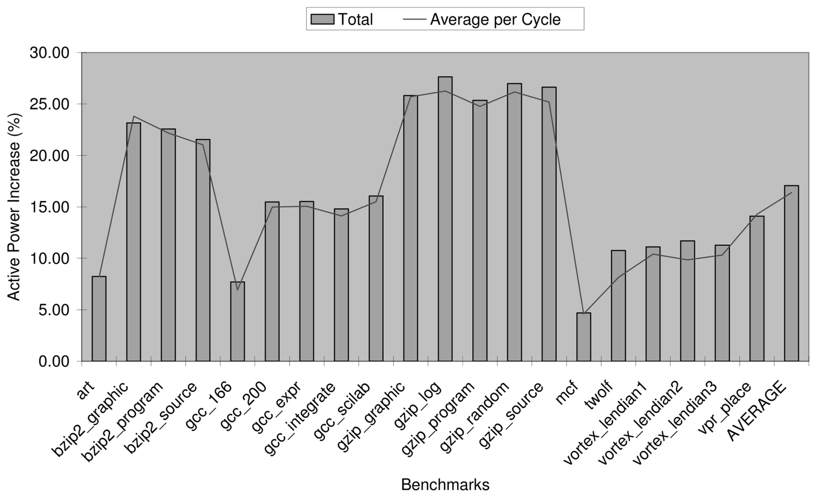


Fig. 18. Double Execution relative active power increase (Wide machine).

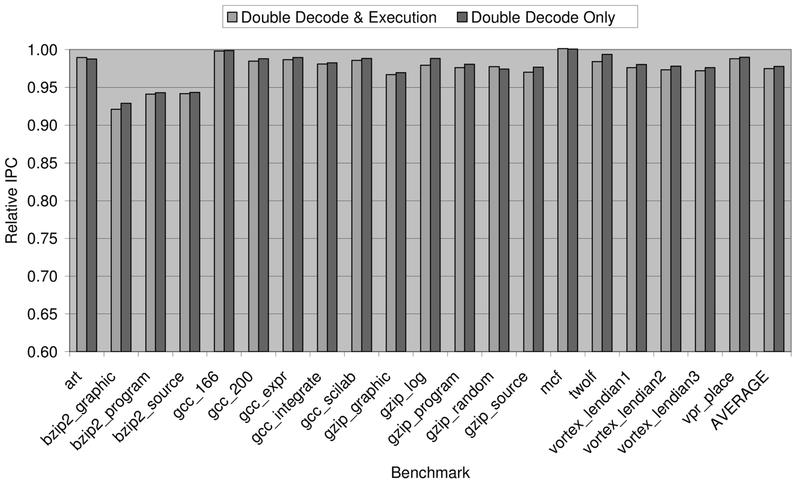


Fig. 21. Double Decoding relative performance (Wide machine).

TABLE 8
Double Decoding Performance Summary

Performance	Without Double Execution			With Double Execution		
	Narrow	Medium	Wide	Narrow	Medium	Wide
Average	0.821	0.933	0.978	0.816	0.925	0.975
Worst case	0.681	0.845	0.929	0.679	0.840	0.921

low, the cost of recovery is not a major factor. Only when the fault rate becomes significant, either as a result of future processes or as a result of intentionally using even lower voltage in order to save power, the performance overhead of the recovery becomes a key consideration. In this paper, we have presented two techniques—**Double Execution** and **Double Decoding**—that together with error correction codes can provide the entire processor similar levels of soft error coverage to those of commercial and other known solutions, with a significantly smaller performance loss and smaller power consumption than other proposals.

With **Double Execution,** soft errors in the Out-Of-Order part of the processor are detected by executing each instruction twice (temporal redundancy using underutilized execution resources). This implementation is very simple, requiring only minor hardware alterations. We show that the average performance degradation of this technique is 2.2 percent in Medium width processors (and better in wider machines), with an increase of approximately 15 percent in active power. We also show that the trend of increasing memory access latencies (in processor clock cycles) and array sizes is likely to mitigate the technique's adverse performance impact, and the trend of increase in leakage power fraction of overall power consumption is likely to reduce the technique's power impact. Another commercial advantage of the Double Execution is that it can easily be turned off by software, achieving a higher performance whenever the reliability is not important.

With **Double Decoding,** soft errors in the instruction decoders' hardware are detected by using separate small, low-performance, and low-power decoders (spatial redundancy), which are used in a new pipe stage before the commit stage and thus only need to decode nonspeculative instructions. We have shown a resulting average performance degradation of around 2.2 percent in high-performance processors. The combined use of Double Execution and Double Decoding reduces the overall performance by only 2.5 percent on the average in high-performance processors.

Double Execution and Double Decoding, along with error correction codes in the arrays and buses, provide excellent soft error coverage with much lower performance, power, and hardware cost than any contemporary solution. Our techniques could result in "excessive" (i.e., higher than required) reliability. Consequently, it is viable to use a design or operating parameters that incur a higher soft-error rate, thereby trading away the excessive reliability for a reduction in power consumption (lowering the voltage), higher performance (increasing the clock frequency), or both. Application of our techniques may thus even yield a solution that dominates single execution, simultaneously providing a lower undetected soft error rate, lower power consumption, and higher performance than those of single execution.

References

[1] J. Leray et al., "Atmospheric Neutron Effects in Advanced Microelectronics, Standards and Applications," *Proc. IEEE Int'l Conf. Integrated Circuit Design and Technology (ICICDT '04),* pp. 311-321, 2004.
[2] C. Constantinescu, "Impact of Deep Submicron Technology on Dependability of VLSI Circuits," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '02),* pp. 205-209, 2002.
[3] F.L. Yang and R.A. Saleh, "Simulations and Analysis of Transient Faults in Digital Circuits," *IEEE J. Solid-State Circuits,* vol. 27, pp. 258-264, 1992.
[4] C. Constantinescu, "Trends and Challenges in VLSI Circuit Reliability," *IEEE Micro,* vol. 23, pp. 14-19, 2003.
[5] P. Shivakumar et al., "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '02),* pp. 389-398, 2002.
[6] P. Hazucha and C. Svensson, "Impact of CMOS Technology Scaling on the Atmospheric Neutron Soft Error Rate," *IEEE Trans. Nuclear Science,* vol. 47, pp. 2586-2594, 2000.
[7] G. Saggese et al., "An Experimental Study of Soft Errors in Microprocessors," *IEEE Micro,* vol. 25, pp. 30-39, 2005.

[8] T. Suzuki and Y. Yamagami, "A Sub-0.5-V Operating Embedded SRAM Featuring a Multi-Bit-Error-Immune Hidden-ECC Scheme," *IEEE J. Solid-State Circuit,* pp. 152-160, 2006.

[9] N. Wang et al., "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '04),* pp. 61-70, 2004.

[10] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* second ed., pp. 220-261. Morgan Kaufmann, 1996.

[11] R. Iyer et al., "Recent Advances and New Avenues in Hardware-Level Reliability Support," *IEEE Micro,* vol. 25, pp. 18-29, 2005.

[12] W. Stallings, *Computer Organization & Architecture: Designing for Performance,* Section 5.2, sixth ed. Pearson Inc., pp. 148-153, 2003.

[13] T.M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *Proc. Ann. Int'l Symp. Microarchitecture (Micro '99),* pp. 196-207, 1999.

[14] C. Weaver and T. Austin, "A Fault Tolerant Approach to Microprocessor Design," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '01),* pp. 411-420, 2001.

[15] B. Pei and Y. Ming, "An Embedded Fail-Safe Interlocking System," *Proc. Pacific Rim Int'l Symp. Fault-Tolerant Systems (PRFTS '97),* pp. 22-27, 1997.

[16] T. Slegel, R.L. Averill, and M.A. Check, "IBM's S/390 G5 Microprocessor Design," *IEEE Micro,* vol. 19, pp. 12-23, 1999.

[17] M. Nicolaidis, "Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies," *Proc. VLSI Test Symp. (VTS '99),* pp. 86-94, 1999.

[18] S. Mitra et al., "Robust System Design with Built-In Soft-Error Resilience," *Computer,* vol. 38, pp. 43-52, 2005.

[19] N. Wang and S. Patel, "ReStore: Symptom Based Soft Error Detection in Microprocessors," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '05),* pp. 30-39, 2005.

[20] N. Oh et al., "Error Detection by Duplicated Instructions in Super-Scalar Processors," *IEEE Trans. Reliability,* vol. 51, pp. 63-75, 2002.

[21] T. Vijaykumar et al., "Transient-Fault Recovery Using Simultaneous Multithreading," *Proc. Ann. Int'l Symp. Computer Architecture (ISCA '02),* pp. 87-98, 2002.

[22] M. Rashid et al., "Power-Efficient Error Tolerance in Chip Multiprocessors," *IEEE Micro,* vol. 25, pp. 60-70, 2005.

[23] S.K. Reinhardt and S.S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," *ACM SIGARCH Computer Architecture News,* vol. 28, no. 2, pp. 25-36, May 2000.

[24] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS '99),* pp. 84-91, 1999.

[25] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A Study of Slipstream Processors," *Proc. 33rd Ann. IEEE/ACM Int'l Symp. Microarchitecture (Micro '00),* pp. 269-280, 2000.

[26] G.S. Sohi, M. Franklin, and K. Saluja, "A Study of Time-Redundant Fault Tolerance Techniques for High-Performance Pipelined Computers," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS '89),* pp. 436-443, 1989.

[27] A. Mendelson and N. Suri, "Designing High Performance and Reliable Superscalar Architectures—The Out of Order Reliable Superscalar (O3RS) Approach," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '00),* pp. 473-481, 2000.

[28] J. Smolens, J. Kim, J. Hoe, and B. Falsafi, "Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures," *Proc. Int'l Symp. Microarchitecture (Micro '04),* pp. 257-268, 2004.

[29] C. Weaver et al., "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor," *Proc. Ann. Int'l Symp. Computer Architecture (ISCA '04),* pp. 264-275, 2004.

[30] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture (Micro '03),* pp. 29-40, 2003.

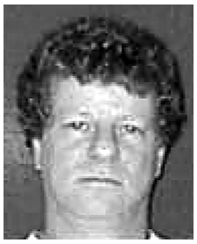
[31] D. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," Univ. of Wisconsin-Madison, http://www.simplescalar.com/docs/users_guide_v2.pdf, 1997.

[32] T. Austin, "SimpleScalar Hacker's Guide (release 2.0)," *SimpleScalar LLC,* http://www.simplescalar.com/docs/hack_guide_v2.pdf, 2008.

[33] D. Brooks et al., "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. Ann. Int'l Symp. Computer Architecture (ISCA '00),* pp. 83-94, 2000.

[34] D. Roberts et al., "Error Analysis for the Support of Robust Voltage Scaling," *Proc. Symp. Quality of Electronic Design (ISQED '05),* pp. 65-70, 2005.

Avi Timor received the BSc and MSc degrees from the Israel Institute of Technology (Technion) in 2002 and 2006, respectively. Since 1999, he has been a design engineer in the Intel Mobile Platform Group, Haifa, Israel.



Avi Mendelson received the BSc and MSc degrees from the Israel Institute of Technology (Technion) and the PhD degree from the University of Massachusetts, Amherst. He is a principal engineer in the Intel Mobile Platform Group, Haifa, Israel, and an adjunct professor in the Computer Science and the Electric Engineering Departments, Technion. He has been with Intel for eight years: he started as a senior researcher in Intel Labs and then moved to the Microprocessor Group, and he serves as the CMP architect of the Intel Core Duo Technology. His work and research interests are in computer architecture, fault-tolerant systems, low-power design, parallel systems, OS-related issues, and virtualization.

Yitzhak Birk received the BSc (cum laude) and MSc degrees in electrical engineering from Technion in 1975 and 1982, respectively, and the PhD degree in electrical engineering from Stanford University in 1987. He has been on the faculty of the Electrical Engineering Department, Technion, since 1991, and heads its Parallel Systems Laboratory. From 1986 to 1991, he was a research staff member at IBM's Almaden Research Center. His research interests include computer and communication systems. He is particularly interested in parallel and distributed architectures for information systems, including communication-intensive storage systems, with special attention to the true application requirements in each case. The judicious exploitation of redundancy for performance enhancement in these contexts has been the subject of much of his recent work. He is also engaged in research into various facets of processor architecture, attempting "cross fertilization" between his various areas of research. For more information visit http://www.ee.technion.ac.il/people/birk.

Neeraj Suri received the PhD degree from the University of Massachusetts, Amherst. He currently holds the TU Darmstadt chair professorship in "Dependable Embedded Systems and Software" at TU Darmstadt, Germany. His earlier academic appointments include the Saab Endowed Professorship and faculty at Boston University. His research interests span design, analysis, and assessment of dependable and secure systems/software/services. His group's research activities have garnered support from the European Commission, DARPA, DFG, NSF, NASA, NAWC, ONR, Boeing, Microsoft, Intel, Saab, and Hitachi among others. He is also a recipient of the NSF CAREER Award. He serves as an editor for the *IEEE Transactions on Dependable and Secure Computing*, *IEEE Transactions on Software Engineering*, *ACM Computing Surveys*, and the *International Journal of Security and Networks*. He has been an editor for the *IEEE Transactions on Parallel and Distributed Systems*. He is a member of IFIP WG 10.4 on Dependability and a member of Microsoft's Trustworthy Computing Academic Advisory Board. He has served as the PC chair for SRDS, HASE, ISAS, and Microsoft-TUD RAF and will serve as the PC chair for the upcoming DSN/DCCS 2008. More professional details are available at http://www.deeds.informatik.tu-darmstadt.de/suri.