Appears in the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)
Anchorage, Alaska, June, 2008

Detouring: Translating Software to Circumvent Hard Faults in Simple Cores

Albert Meixner
Dept. of Computer Science
Duke University
albert@cs.duke.edu

Daniel J. Sorin
Dept. of Electrical and Computer Engineering
Duke University
sorin@ee.duke.edu

Abstract

CMOS technology trends are leading to an increasing incidence of hard (permanent) faults in processors. These faults may be introduced at fabrication or occur in the field. Whereas high-performance processor cores have enough redundancy to tolerate many of these faults, the simple, low-power cores that are attractive for multicore chips do not. We propose Detouring, a software-based scheme for tolerating hard faults in simple cores. The key idea is to automatically modify software such that its functionality is unchanged but it does not use any of the faulty hardware. Our initial implementation of Detouring tolerates hard faults in several hardware components, including the instruction cache, registers, functional units, and the operand bypass network. Detouring has no hardware cost and no performance overhead for fault-free cores.

1. Introduction

As CMOS fabrication technology continues to scale to ever-smaller dimensions, processor cores are increasingly susceptible to hard (permanent) faults [20]. These hard faults may be introduced during fabrication or they may occur in the field. Rather than discarding cores with hard faults, it would be preferable to tolerate their defects, with perhaps a slight degradation in performance.

Tolerating a hard fault requires some mechanism for circumventing the faulty circuitry. In a high-performance, superscalar, speculative core, there exists a significant amount of redundancy that can be used for fault tolerance [2, 3, 19, 18]. For example, if an ALU is determined to have a permanent fault, the core can simply stop using that ALU. The core continues to operate correctly, using its remaining ALUs, but its performance is somewhat degraded.

Unlike superscalar cores, simple cores have little hardware redundancy for tolerating hard faults. This lack of fault tolerance is emerging as a problem, because simple, in-order cores with shallow pipelines are a popular option for multicore chips—such as Sun's Niagara [9], Cisco's Silicon Packet Processor [5], and the SPEs in the Cell Processor [8]. The area-efficiency and power-efficiency of simple cores enable multicore chips to stay within their area, power, and thermal budgets. Adding spare hardware for fault tolerance would undermine the advantages of these cores and is not an attractive option for commodity systems.

In this paper, we propose *Detouring*, a software-based scheme for improving the aggregate chip performance (which we will refer to as "throughput") of multicore processors (also known as chip multiprocessors or CMPs) with simple cores in the presence of permanent faults. The key idea is to modify the executed code when a hard fault is present, such that it does not use any of the faulty hardware yet has the same functionality as the original code. We illustrate a high-level view of Detouring in Figure 1. This code modification can be performed in various contexts—via static compilation, dynamic compilation, binary translation [11], or a virtual machine—with differing tradeoffs. In our implementation of Detouring, we have developed software modifications, called *Detours*, to tolerate hard faults in several hardware components, including registers, functional units, the instruction cache, and the operand bypass network.

Detouring is a software-only solution that has no performance cost for fault-free cores and no hardware cost. Our results show that our current set of Detours can tolerate 42.5% of all possible hard stuck-at faults in the core itself and nearly all faults in the instruction cache. The performance of a core using a Detour varies as a function of the workload and the particular Detour, as we show in Section 6. When applied to a chip multi-

Figure 1. High-Level View of Detouring

processor (CMP), Detouring can significantly increase the chip's throughput in the presence of hard faults.

The rest of the paper is organized as follows. In Section 2, we compare Detouring to prior work. In Section 3, we describe our system model and state our assumptions. In Section 4, we present Detouring, including Detours to tolerate specific faults. In Section 5, we discuss the modifications that we made to the compiler. We experimentally evaluate Detouring in Section 6, and we conclude in Section 7.

2. Related Work

The areas of work most related to Detouring are software-based fault tolerance, hardware-based fault tolerance, software handlers for unimplemented instructions, and microcode patches.

2.1. Software-Based Fault Tolerance

Prior work has used instruction replication to detect [15] and correct transient errors [17]. Instruction redundancy can detect errors due to some hard faults, but it is not well-suited to tolerating hard faults. To tolerate a hard fault, no more than one of the copies of each instruction must use the faulty circuitry, which is only possible on cores with redundant hardware resources. Detouring specifically targets permanent, (rather than transient) errors, provides no error detection capability, and works on cores without redundant hardware. As a consequence of the different focus, Detouring has much lower impact on runtime, because it changes only those portions of the code that are affected by erroneous circuitry. Detouring does not modify code running on fault-free cores.

Software schemes have also been proposed to detect permanent hardware faults (e.g., [7, 16, 22]). These schemes are complimentary to Detouring, which does not provide error detection or diagnosis, but relies on the presence of some error detection mechanism to obtain information about faulty hardware.

2.2. Hardware-Based Fault Tolerance

High-performance superscalar cores have multiple instances of many of their resources (e.g., ALUs, fetch/decode lanes). This redundancy is used to increase performance, but it can also be leveraged to tolerate hard faults [2, 3, 19, 18]. Bower et al. [2, 3] develop schemes for diagnosing hard faults in microprocessor components and then deconfiguring these faulty components. Shivakumar et al. [19] and Schuchman and Vijaykumar [18] improve chip yield by using hardware redundancy to tolerate fabrication defects.

A hardware approach to providing hard fault tolerance for storage, but not logic, is to use error correcting codes (ECC). Architects can add ECC to registers and other storage elements, but ECC is costly. ECC adds multiple bits to each storage element, plus hardware to compute and check the ECC (which may be on the critical path).

Unlike hardware-based fault tolerance, Detouring uses software to provide hard fault tolerance. Detouring uses software because it focuses on cores that do not have significant amounts of hardware redundancy and cannot afford spares due to cost or power.

2.3. Software Handlers

There is a long history of using software to emulate instructions that are not implemented in hardware. Certain instructions may be complicated to implement in hardware (e.g., floating point square root) or they may be used so rarely that a relatively slow software emulation is sufficient (e.g., legacy instructions). When these instructions are encountered, the processor traps to a software routine or to a microcode implementation.

Detouring differs from this work in a few aspects. First, Detouring has a different goal. Instead of emulating non-existent hardware, Detouring tolerates faults in existing hardware. Second, Detouring is not strictly instruction-based. For example, Detouring can tolerate a fault in a register that could be used by many instructions. Third, Detouring seeks to provide functionality for common operations, not just rare instructions. Fourth, unlike traditional software emulation, Detouring does not necessarily use simple units to emulate more complex ones. Detouring may use complex units to emulate simple ones, and it may use partially broken units to emulate fault-free ones.

2.4. Microcode Patches

Software, generally in the form of microcode, has been used to overcome design bugs that are uncovered too late in the product cycle to be fixed in hardware [6]. Detouring differs in that it tolerates hard faults that occur dynamically due to physical phenomena. Detouring also does not require that the core has microcode.

3. System Model

To simplify the system issues associated with Detouring, we assume a specific target system model in which a control processor is coupled with a number of simple in-order cores. The control processor runs the operating system and has full control over the assignment of tasks to the simple cores. The cores execute small to mid-size kernels and refer system calls to the

control processor. In the context of Detouring, the control processor is in charge of recompiling programs for individual cores and scheduling tasks such that faults have minimal impact. Note that the control processor can be logical (i.e., physically identical to the simple cores but used differently) or physical. A physical control processor can sit on the same die as the other cores, such as the PPE on the IBM Cell [8]. The control processor can also be external, such as a commodity out-of-order processor (control processor) coupled with a graphics [14] or physics [23] accelerator board that contains a number of simpler processors.

This system model suits a number of applications—including networking, multimedia, and certain types of scientific computing—in which task throughput is the goal and multiple simple tasks can be spread across the cores. For example, a network router chip can use cores for encryption, packet routing and analysis, error correction, etc. Detouring improves throughput by using faulty cores that would otherwise be disabled.

We next describe the processor core model (Section 3.1) and system requirements (Section 3.2).

3.1. Processor Core Model

Detouring's ability to tolerate hardware faults is fundamentally limited to faults in circuitry under direct software control. In complex out-of-order cores, software has little control over the circuitry and thus Detouring's fault coverage is minimal. The optimal cores for Detouring are VLIW processors, which are designed to give the compiler tight control over program execution. Traditional single-issue, in-order RISC cores, which are the focus of this work, are between these two extremes, but closer to VLIW.

Our Detouring implementation is built on top of the open-source OpenRISC 1200 (OR1200) microarchitecture [10]. The OR1200 is a 32-bit scalar (1-wide), in-order RISC core with a 4-stage pipeline (Fetch, Decode, Execute, Writeback) and 32 general purpose registers. It has an instruction cache and data cache, which we assume in this paper to both be 16KB and 2-way set-associative. The data cache is write-back, write-allocate, and it blocks on misses. The OR1200 core has an integer ALU, a non-pipelined integer multiplier/divider, and a load/store unit, but no floating point hardware. There is a single branch delay slot and no branch penalty, so no branch prediction is needed.

3.2. System Requirements

Detouring assumes that the system is capable of detecting and diagnosing hard faults, as well as recompiling software.

Error Detection and Diagnosis. Detouring requires a system to have the ability to detect errors and diagnose hard faults (i.e., determine where the faults are). The implementation of error detection and fault diagnosis is a topic that is orthogonal to Detouring, and we assume the use of existing techniques. Low-cost options, such as software testing and diagnosis [7, 16, 22] and built-in self-test (BIST) are a natural fit, but more heavyweight solutions could also be applied. Unlike Detouring itself, an error detection mechanism will impact fault-free performance such that fast mechanisms are preferable. The speed of fault diagnosis is less critical, because the appearance of a new permanent fault is an infrequent event and the recompilation necessary to adapt to the discovered fault is in itself a time-consuming process. Thus, software-based diagnosis is a good option for Detouring.

Recompilation. The list of hard faults obtained from diagnosis is used as an input to the Detouring software translation. Detouring could be implemented in the static compiler, a dynamic compiler, or a binary translator. Detouring could also be implemented as a virtual machine that acts as an interface between the software written for the architectural (fault-free) ISA and the potentially faulty hardware. In this paper, the Detouring is performed by re-compiling the source code. A perhaps more elegant model would have software distributed in an intermediate format (e.g., LLVA [1]), such that the core later compiles it to an executable. In this software distribution and usage model, Detouring would just require re-compiling from the intermediate format (either statically or dynamically).

4. Detouring

In this section, we first describe the specific Detours that we have developed or adapted from previously known emulation routines (Section 4.1-Section 4.5). We then explain how a Detour for a given piece of hardware can coincidentally tolerate faults in other parts of the chip (Section 4.6), describe how to compose multiple Detours (Section 4.7), and discuss faults for which we do not yet have Detours (Section 4.8).

4.1. Functional Unit Detours

There are many different functional units in even a simple core. The OR1200 core does not have floating point hardware, so we do not consider it. However, it is well-known how to emulate floating point arithmetic with software that uses only integer arithmetic.

For any functional unit, it is possible that a fault either effectively disables the entire unit or just a small

fraction of it. In the latter case, we try to leverage the remaining hardware that is not faulty.

4.1.1. Multiplier

The multiplier is a large circuit with many possible locations for faults. Being able to Detour around multiplier faults is thus beneficial. The multiplier in our core takes two 32-bit operands as inputs and returns a 64-bit product. A multiplier is a highly regular structure and, even if portions of it are faulty, it can often be used as a narrower multiplier to speed up emulation. For example, if a fault affects only the 37th bit of the product, then we can salvage a fault-free 16x16 multiplier. We only resort to the typical slow shift-and-add emulation if a fault affects too many result bits.

In our experiments we found that over 90% of single permanent faults affect only either the upper or lower 32-bits of the results, such that we still have a fault-free 16x16 multiplier available. We also support 16x8 and 8x16 cases that require 24 correct output bits. We explored a few other similar Detours, including one that used fault-free 8*8 multiplication, but the advantage over shift-and-add emulation was minimal.

4.1.2. Divider

The divider is another large functional unit, but we have not discovered any clever Detours for tolerating faults within it. Currently, we use a standard shift-and-subtract algorithm for our division Detour. This Detour provides correct operation, but its performance is probably only acceptable for software that does not frequently perform divisions.

4.1.3. Shifter

The OR1200 provides both left and right shift instructions. The three operands are the data to be shifted (*X*), direction to shift, and shift amount (*shamt*).

Left Shift. Left shifting *X* by *shamt* is equivalent to multiplying *X* by a scale factor of 2^{shamt}. Thus, our Detour must be able to determine 2^{shamt}. For constant shifts, it is easy to compute this scale factor at compile time. For variable shifts, we must determine 2^{shamt} at runtime. If the fault does not affect right shifting, then we use this hardware to compute the scale factor as *0x80000000 >> (31-shamt)*. Otherwise, we look it up in a table.

Right Shift. We first compute a scale factor of *1<<(32-shamt*), either by using left shifting (if this capability is fault-free) or by looking it up in a table. We multiply the scale factor and *X* to produce the desired result in the upper 32 bits of the multiplier's result. We illustrate an example in Figure 2.

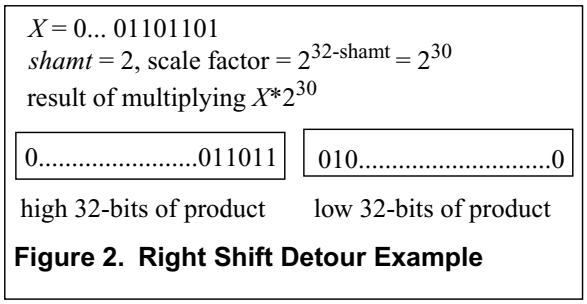

Figure 2. Right Shift Detour Example

4.1.4. Adder/Subtractor

Addition and subtraction are performed with mostly the same logic. Unfortunately, our attempts to develop an efficient Detour to tolerate hard faults in this shared circuitry have thus far been unsuccessful. We have devised correct Detours, but they involve far too many instructions to achieve reasonable performance, especially because this functional unit is used so frequently.

For the small subset of faults that affect either addition or subtraction, but not both, we can provide efficient Detours. The key to the Detour for faulty addition (subtraction) is to negate the appropriate operand and then perform a subtraction (addition).

4.1.5. Bitwise Logical Operators

The OR1200 ISA has the following bitwise operators: AND, OR, XOR, and NOT (implemented using XOR). A fault in any one of these operations can be tolerated by expressing it in terms of the other operations. The necessary expressions are derived by applying simple Boolean logic and DeMorgan's Laws.

4.1.6. Partial-Word Loads/Stores

A fault could cause partial-word loads or stores to be incorrect, even though whole-word accesses are unaffected by the fault. Example of such faults are faults in the alignment and extension hardware. Our Detours are similar to the software used to support partial-word accesses in architectures, like the Alpha 21064 [13], that do not support them in hardware. The key difference is motivation: the 21064 wanted to avoid hardware that could slow the critical path, whereas Detouring's goal is to tolerate faults.

Partial-Word Load. Our Detour first computes the word-aligned address and loads the full word. We shift the word to the desired byte or half-word and then sign/zero extend it.

Partial-Word Store. As with the partial-word load Detour, this Detour first computes the word-aligned address and loads the full word. We compute the mask of the bits that will be overwritten and clear those bits with an AND. We then take the desired byte or half-word to be written, shift it to the desired position, and OR it with the loaded word, thereby overwriting the

Before

```
      jal FOO
      nop # delay slot
```

After

```
      movhi rX, hi(FOO_RETURN#K)
      ori rX, rX, lo(FOO_RETURN#K)
      j FOO
      nop # delay slot
FOO_RETURN#K:
```

Figure 3. Detour for Link Register (r9). We use another register (rX) instead of r9. K is a unique number for each call site.

bits that had been cleared with the AND. Lastly, we store this word to memory.

4.1.7. Zero/Sign Extension Unit

The OR1200 has a functional unit for extending partial words, and it can perform either zero-extension or sign-extension. Zero-extension can be emulated using simple bitwise AND with a constant. The sign-extension Detour uses a left-shift (to move the desired sign bit to the beginning of the word) followed by an arithmetic right-shift (to propagate the sign bit).

4.2. Register File Detours

Detouring avoids faulty entries in the register file by excluding the affected registers from register allocation. In general, this Detour is straightforward, but certain registers have special purposes or usage conventions. For example, register zero is often hardwired to be all-zero. In the OR1200 architecture, register nine (r9) is hardwired to be the link register for the jump-and-link (jal) instruction. We address this issue with the link register Detour shown in Figure 3.

Other registers are, by ABI (application binary interface) convention, expected to be the stack pointer, frame pointer, call return address, etc. If one of these registers is faulty, the Detour will tolerate the fault but it will necessarily break the ABI and will thus require recompilation of all software, including libraries.

Most single faults in the register file (e.g., faulty SRAM cell) will render a single register unusable, but certain faults could also disable multiple registers. For example, a coupling fault (short circuit) could cause a bit of one register to always equal a bit in another register or a fault in the wordline decoder could cause accesses to both register *K* and register *K*+1 to access register *K*+1. In the case of multiple broken registers, as with a single faulty register, we can provide Detours by not allocating any of the faulty registers. However, the performance of the resulting software may suffer if too few registers are usable. Faults that break all available registers, including faults in sense amps and bit lines, cannot be tolerated by this approach.

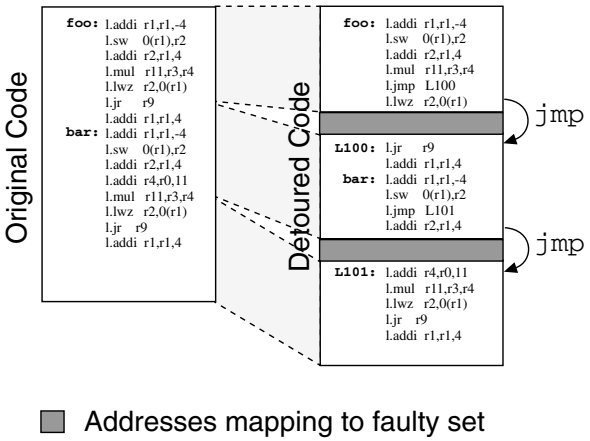

Figure 4. Detour for I-Cache Faults

4.3. Instruction Cache Detours

The fault model for the instruction cache (I-cache) is quite similar to that of the register file, because they are both SRAM storage structures. A fault can either lead to one faulty block, multiple faulty blocks, or a situation in which the entire I-cache is unusable.

As with the register file, we have developed Detours for those faults that do not disable the entire I-cache. The key to the I-cache Detours is to pad the code such that faulty blocks in the I-cache are not used (see Figure 4). This padding is performed by the linker, which avoids addresses that map to a faulty cache set when placing instructions. To keep the core from touching faulty sets, the linker inserts an unconditional jump right before each potentially faulty block. To determine which instructions will map into the faulty cache lines, the linker must control where the code is loaded into memory. In most formats for binary executables, such as ELF and COFF, sufficient control is provided through specification of alignment constraints.

4.4. Operand Bypass Network Detours

Pipelined microprocessors bypass the result of an instruction to consumer instructions that are in earlier stages of the pipeline. The operand bypass network consists of the wires and multiplexors that provide this capability, and faults could occur in this hardware. The OR1200 has two bypassing paths: from Execute (EX) to Execute and from Writeback (WB) to Execute.

We provide Detours by scheduling instructions and inserting NOPs into the code such that faulty bypass paths are not exercised. The NOPs degrade perfor-

Before

```
     bf FOO # branch on condition flag
     nop # delay slot
```

After

```
     movhi rX, hi(FALLTHROUGH#K)
     ori rX, rX, lo(FALLTHROUGH#K)
     addi rY, rX, FOO-FALLTHROUGH#K
     cmov rY, rY, rX
     jr rY
     nop # delay slot
FOO_RETURN#K:
```

Figure 5. Conditional Branch Emulation

mance, with respect to a fault-free processor, but they ensure correct execution.

It is possible that forwarding works for one of the two operands (left or right) but not the other. In this case, we could provide a Detour for associative instructions by simply switching the operands. We have not yet implemented this feature.

4.5. Jump/Branch Detour

RISC processors typically support three types of jumps or branches: conditional direct branches, unconditional direct jumps, and unconditional register-indirect jumps. We can emulate unconditional direct jumps by loading the target address into a register and using a register indirect jump for the control transfer. On systems that support conditional moves or predication, we can emulate conditional branches similarly, as shown in Figure 5.

Register-indirect jumps can be replaced with a decision tree implemented using conditional branches, in some cases (e.g., switch statements). However, a general Detour would require self-modifying code that patches an unconditional jump before every invocation, similar to the trampolines used previously in runtime binary patching [4]. The effectiveness of this Detour depends highly on the specific processor and the implementation of its memory system.

4.6. Coincidental Detours

Some of the Detours we have already described can also be used to tolerate hard faults in portions of the core other than their intended targets. We refer to such Detours as *Coincidental Detours*, and we now provide two examples.

First, faults in the instruction decoding logic can manifest themselves in many ways. Some faults may hopelessly corrupt decoder outputs—pipeline control signals, opcode, operand identifiers, etc.—in a way that cannot be efficiently tolerated with a Detour. However, some decoder faults may cause only one control signals for certain operation to get corrupted (e.g., operand routing for all multiply instructions). Many of these faults can be tolerated with Detours we have already described, because they allow us to avoid the affected operations. Second, some pipeline latch faults can be tolerated with previously described Detours. An example would be a fault in the latch holding a register identifier; a fault in the most-significant bit could be tolerated by configuring our register Detour to allocate only half of the available registers.

4.7. Combining Multiple Detours

If the processor has multiple hard faults, we can use multiple Detours. In general, Detours compose. However, if a given Detour uses hardware that is also faulty, then Detouring will not work. For example, the 16x16 and 16x8 multiplier Detours use the shifter and adder. If either the adder or shifter is faulty, these Detours cannot be used to tolerate multiplier faults.

4.8. Faults Currently Without Detours

As we show in Section 6, the set of Detours we have presented is capable of covering roughly 42.5% of the possible hard faults in the OR1200 core (excluding caches). We now discuss the reasons why our set of Detours does not cover the remaining 57.5%.

Feasible Future Work. The biggest potential for future Detours does not lie within the core, but within the data cache. The D-cache is by far the single largest structure currently without a Detour. Our attempts at providing a Detour for the D-cache using only compilation changes have been unsuccessful so far. All of our attempts required specific hardware support, such as the ability to enable and disable the data cache quickly under user control or the presence of instructions to pin or invalidate cache lines. In future work, we plan to investigate mechanisms that can avoid faulty data cache sets by using a combination of changes in the compiler, OS, and memory allocation routines.

Within the core, there are some faults for which we believe efficient Detours exist, but we have not yet had time to implement them. Examples are faults in the PC generation logic (branches and jumps) and certain faults in the instruction decoding logic; a decoder fault that corrupts the immediate field could be tolerated by a Detour that uses an instruction to fix the immediate.

Unacceptable Performance. There are other faults for which Detours exist but they would be too detrimental to performance. As mentioned earlier, the majority of faults in an adder cannot be efficiently tolerated.

Impossible. A large portion of the processor core is not salvageable using recompilation, because it is completely outside software control. Notable examples are the exception generation circuitry, most pipeline control logic, instruction fetch unit, and portions of the datapath not used for bypassing. This logic puts a fundamental upper bound on Detours error coverage. Determining its portion of the total logic is difficult, but we estimate it to be at least 40%-45%.

5. Compiler Implementation Details

For our evaluation, we implemented Detouring in the OpenRISC [10] backend of GCC 3.4.4 [21]. Table 1 provides a list of the Detours implemented in the compiler and evaluated in Section 6. We did not implement all Detours described in Section 4, because it is not cost-effective, in terms of development effort, to create Detours that cover only a negligible fraction of possible faults or incur prohibitive performance penalties. However, we presented them in Section 4 because they may be useful for other cores.

The exact compiler changes required and the compilation stage during which they are executed depend on the Detour. Detours for general purpose registers, described in Section 4.2, are implemented by modifying register definitions such that broken registers are never used by the compiler. Operand bypass Detours (Section 4.4) are realized during instruction scheduling. They are implemented by either re-ordering the code or, if that fails, inserting NOPs during a final code organization pass before assembly code is output.

Table 1. Summary of Implemented Detours

name	faulty functionality	short description
mul_AxB	multiplication	use fault-free AxB multiplication
mul_shift	multiplication	use shifts and additions
div	division	use shifts and subtractions
sext	sign extension	use shifts and masks
shl/shr/shift	left/right/both shifting	use shr/shl/table to compute shift amount, then multiply
K_regs	*K* general purpose registers	do not allocate *K* faulty registers
icache_K	*K* blocks in I-cache	pad code to avoid using faulty I-cache blocks
link_reg	link reg (r9)	use other reg for linking
exbyp	bypass from EX to EX	insert NOP to avoid this operand bypassing
wbbyp	bypass from WB to EX	insert NOP to avoid this operand bypassing

Functional unit and branch Detours (Sections 4.1 and 4.5) are inserted into the code in three separate places. First, complex Detours, such as multiplication, are added to the GCC library (libgcc), which is linked to every compiled program by default. Second, shorter Detours, such as constant shifts, are inserted during the translation from a parse tree to RTL. Third, simple Detours, such as the link register Detour, are created during the translation from RTL to assembly language. Finally, the I-Cache Detour is implemented in the linker as described in Section 4.3.

All Detours are controlled using command line flags during the compiler invocation. Our modifications have no impact on code generation unless they are explicitly enabled using these flags.

6. Experimental Evaluation

The ultimate goal of our experimental evaluation is to determine how much Detouring improves the chip's aggregate throughput in the presence of hard faults.

To better understand the throughput results (presented in Section 6.3), we first evaluate the fault coverage of this set of Detours (Section 6.1) and the performance of a core with each Detour (Section 6.2).

6.1. Fault Coverage

To test Detouring's fault coverage, we combined analytical evaluation with error injection experiments. For SRAM structures, such as the register file and I-cache, which have a regular well-known structure, we analytically determined the percentage of repairable faults. For all other hardware, we performed 5000 experiments in which we injected a different single hard fault in the structural, gate-level Verilog design of the OR1200 core. The OR1200 consists of roughly 25000 gates, excluding the register file. Each hard fault caused the output of one randomly-chosen gate to be stuck-at either zero or one, and the effect of each single fault can propagate to numerous downstream latches. For each injected hard fault, we ran a diagnosis microbenchmark that compares the outcome of routines that use Detourable hardware to routines implemented with the corresponding Detour. A Detour is considered successful if (a) the corresponding hardware causes failures, (b) the Detour works correctly for all inputs, and (c) all tests not related to the Detour or the Detoured hardware also succeed.

Core Coverage. We observe that Detouring covers 42.5% of the possible hard faults in the processor core (including the register file but excluding caches). The

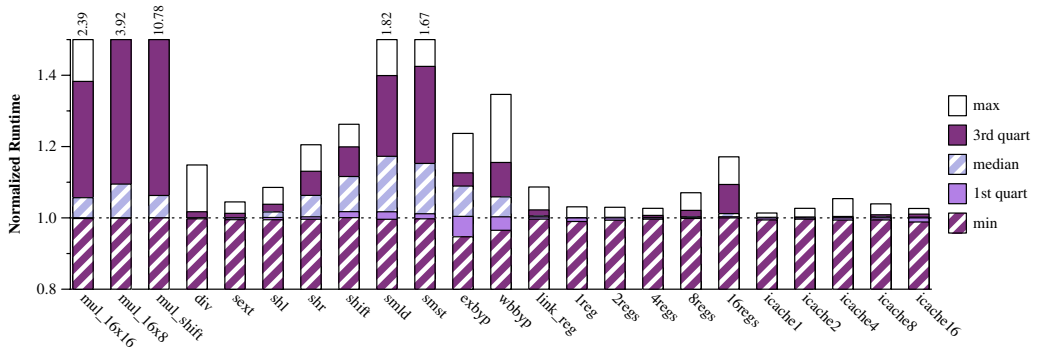

Figure 6. Performance of different Detours across benchmarks

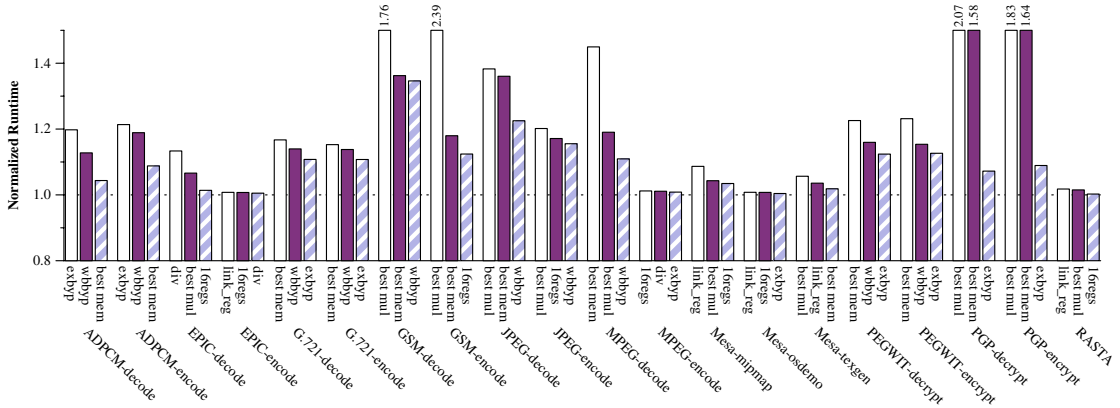

Figure 7. For each benchmark, the three Detours to which it is most sensitive. We only present the best-performing multiplier, I-cache, and shift Detours ("best mult", "best ic", "best shift").

majority of Detouring's fault coverage is provided by the register and multiplier, which is not surprising because they comprise a large fraction of the area of the OR1200 core. Of those faults that are covered, the register and multiplier Detours constitute 20% and 65% respectively. The shifter, divider, and bypass Detours account for 7%, 5%, and 1% of the covered faults, respectively. The Detours for sign-extension and for partial-word loads and stores cover less than 1% of tolerated faults. Some of these low-coverage Detours will be more relevant for cores other than the OR1200. For example, the operand bypassing Detours will provide more benefit for VLIW cores (with their additional bypass paths), and the partial-word load Detour will provide more benefit to cores with store buffers.

I-Cache Coverage. The I-cache Detour covers all faults in the I-cache except for faults in the sense amps and bitlines shared by all cores (>99% coverage).

6.2. Per-Detour Performance

To determine the performance impact of each Detour, we used the OR1200 simulator and the MediaBench benchmark suite [12]. Figure 6 plots the slowdown of each Detour with respect to the original binary (without Detours) running on a fault-free processor.[1] The graph shows minimum slowdown, maximum slowdown, and the three quartiles, as measured across all benchmarks. Register and I-Cache Detours are shown for different number of Detoured entries.

Figure 6 supports our initial intuition that Detouring is most useful for a CMP performing a number of different tasks. As expected, the maximum overhead of each Detour is quite significant. If this were not the case, the Detoured piece of circuitry would be unnecessary. However, for each Detour, at least one benchmark shows no relevant slowdown and at least half of the benchmarks have slowdowns less than 13%.

Three non-intuitive phenomena in the results warrant further explanation. First, many Detours show a slight speed-up for some benchmarks. This phenomenon is in all cases due to a reduction in I-cache misses caused by code reorganization and (for register Detours) stack reorganization. The second non-intuitive result is that Detouring is slower for right shifts than for left shifts. The performance discrepancy is because left shifts by

1. All of our performance results include the mult_16x8 Detour, but not the mult_8x16 Detour, because they have identical performance.

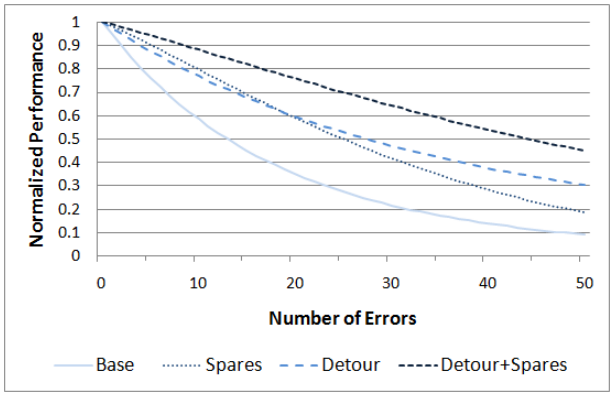

Figure 8. Throughput vs. number of errors for a 16-core CMP

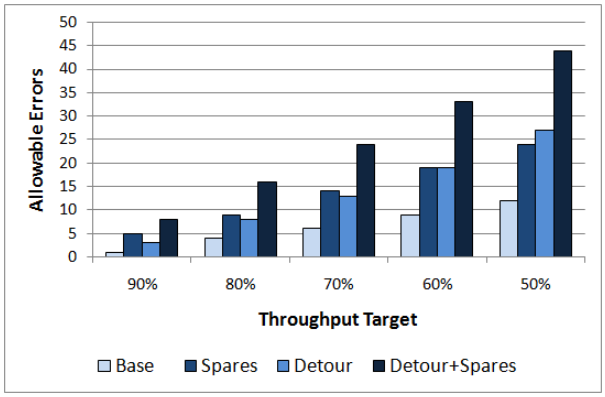

Figure 9. Expected number of errors that can be tolerated for a given throughput target

one or two, which are very common, are optimized into one or two additions and thus require no multiplication. No such shortcut exists for right shifts. Third, shift-based multiplication can be faster than our other multiplication Detours if the number of loop iterations is small (i.e., the second multiplicand is small).

To gain further insight into these results, we examined the sensitivity of each benchmark to each Detour. We studied, for each benchmark, the three Detours to which the benchmark is least sensitive and most sensitive. Figure 7[2] shows the three Detours to which each benchmark is most sensitive. The combination and rank of Detours in the top three varies greatly between benchmarks. An analogous experiment to identify the Detours to which each benchmark is least sensitive (graph not shown due to space constraints) shows that each benchmark has at least three Detours to which it is insensitive (slowdown less than 2.6%). From these results, we conclude that (a) there is no single Detour that has a large detrimental effect on all benchmarks and (b) there is no single benchmark that is sensitive to all Detours. Thus, given a Detouring-aware scheduler and a diverse workload, *the results show that Detouring can salvage an otherwise useless core such that it can be used at almost full performance*.

6.3. Aggregate Throughput

The ultimate goal of Detouring is to improve CMP throughput in the presence of faults. To evaluate throughput, we have developed a Monte Carlo-based analytical model that computes the estimated performance as a function of the number of faults present in a CMP. For this analysis we assumed faults to be uniformly distributed. Most Detours tolerate faults by circumventing circuitry at a very coarse grain level; therefore, a small cluster of faults would have similar impact as a single fault in most cases.

The results of our analysis are shown in Figure 8, which compares four different fault tolerance configurations of a 16-core CMP. The baseline system (*Base*) tolerates hard faults in a core by disabling it entirely. A hard fault in one of the caches is tolerated by disabling the affected cache, but will allow the associated core to continue execution. *Spares* is identical to the baseline configuration, except that it assumes a single spare row per cache that can be mapped into a faulty cache. *Detour* and *Detour+Spares* show the results for the two previously described systems when they use Detouring techniques.

In all configurations, throughput decreases dramatically as the number of faults increases, but the use of Detours can significantly slow down the performance decline. These results also show that spare cache rows are no substitute for Detouring, but that the two techniques are complementary.

Another way of looking at this data is to determine how many hard faults a CMP can contain before it fails to meet a certain throughput target. This question is more relevant than pure throughput in applications that must provide a certain level of performance (e.g., to display a movie at full frame rate or route packets at full link speed) to function properly. This data is shown in Figure 9. For any throughput target, the *Detour* configurations tolerate at least twice as many errors as the *Base* system and the *Detour+Spare* system can still tolerate 60%-80% more errors than the *Spare* system.

7. Conclusions and Future Work

As simple, low-power cores become more common in multicore chips, we will need to develop mechanisms to enable them to operate correctly in the pres-

2. Among the *K*_regs Detours in Figure 7, we consider only the 16regs Detour because it has the most significant impact.

ence of hard faults. Because these cores have little hardware redundancy, fault tolerance must be provided by software. Detouring is an all-software solution that leverages the fault-free hardware to improve the throughput of CMPs with permanent faults. This performance improvement is achieved by allowing some faulty cores, that would otherwise have to be disabled, to keep running at reduced performance.

We believe that additional Detours, most notably for the data cache, could be developed to further improve coverage, both for the OR1200 and for other simple processor cores, and we plan to pursue this avenue of research in our future work. We also believe that Detouring could achieve better performance—by increasing fault coverage and instantaneous performance—with the assistance of a small amount of hardware. We plan to develop hardware "hooks" that will enable new Detours and improve the instantaneous performance of other Detours.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grant CCR-0444516, the National Aeronautics and Space Administration under grant NNG04GQ06G, Toyota InfoTechnology Center, and an equipment donation from Intel Corporation. We thank Alvy Lebeck for feedback on this work.

References

[1] V. Adve et al. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003.

[2] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating Hard Faults in Microprocessor Array Structures. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 51–60, June 2004.

[3] F. A. Bower, D. J. Sorin, and S. Ozev. A Mechanism for Online Diagnosis of Hard Faults in Microprocessors. In *Proc. of the 38th Annual Int'l Symposium on Microarchitecture*, Nov. 2005.

[4] B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[5] Cisco Systems. Cisco Carrier Router System. http://www.cisco.com/application/pdf/en/us/guest/products/ps5763/c1031/cdcco% nt_0900aecd800f8118.pdf, Oct. 2006.

[6] R. P. Colwell. *The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips*. IEEE Computer Society Press, 2006.

[7] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-Based Online Detection of Hardware Defects: Mechanisms, Architectural Support, and Evaluation. In *Proc. of the 40th Annual Int'l Symp. on Microarchitecture*, pages 97–108, Dec. 2007.

[8] M. Gschwind et al. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, Mar/Apr 2006.

[9] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.

[10] D. Lampret. OpenRISC 1200 IP Core Specification, Rev. 0.7. http://www.opencores.org, Sept. 2001.

[11] J. R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–335, Dec. 1997.

[13] E. McLellan. The Alpha AXP Architecture and the 21064 Processor. *IEEE Micro*, 13(3):36–47, May/June 1993.

[14] J. Montrym and H. Moreton. The GeForce 6800. *IEEE Micro*, 25:41–51, March/April 2005.

[15] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error Detection by Duplicated Instructions in Super-Scalar Processors. *IEEE Transactions on Reliability*, 51(1):63–74, Mar. 2002.

[16] M. Psarakis et al. Systematic Software-based Self-test for Pipelined Processors. In *Proc. of the 43rd Design Automation Conference*, pages 393–398, July 2006.

[17] G. A. Reis et al. SWIFT: Software Implemented Fault Tolerance. In *Proc. of the International Symposium on Code Generation and Optimization*, Mar. 2005.

[18] E. Schuchman and T. N. Vijaykumar. Rescue: A Microarchitecture for Testability and Defect Tolerance. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 160–171, June 2005.

[19] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting Microarchitectural Redundancy For Defect Tolerance. In *Proceedings of the 21st Int'l Conference on Computer Design*, Oct. 2003.

[20] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *Proc. of the International Conference on Dependable Systems and Networks*, June 2004.

[21] R. M. Stallman and the GCC Developer Community. GNU Compiler Collection Internals. http://gcc.gnu.org/onlinedocs/gccint.pdf, 2005.

[22] G. Xenoulis et al. On-line Periodic Self-Testing of High-Speed Floating-Point Units in Microprocessors. In *Proc. 22nd IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Sept. 2007.

[23] T. Y. Yeh, P. Faloutsos, and S. J. Patel. ParallAX: An Architecture for Real-Time Physics. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 232–243, June 2007.