Emμcode: Masking Hard Faults in Complex Functional Units

Nicholas Weaver, John H. Kelm and Matthew I. Frank
Center for Reliable and High-Performance Computing, Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL-61801, USA
{nrweaver,jkelm2,mif}@illinois.edu

Abstract

This paper presents Emμcode, a technique for masking hard faults in modern microprocessors that provides graceful performance degradation. Emμcode employs microcode traces with control flow that replace an original instruction once a fault is detected. Emμcode adds lightweight microarchitectural hardware to assist in correcting hard faults in larger structures, such as SIMD execution units found in contemporary microprocessors, where replication is infeasible. Key challenges in implementing microcode traces include maintaining proper architectural state and the optimization of trace code. We are able to significantly optimize traces by exploiting dynamic trace behavior and by performing minor modifications to the microarchitecture.

We find that removing hard to predict branches is important for optimizing traces. Emμcode uses partial predication, new microcode operations, and the full use of the microcode's flexibility and visibility to create fast traces. This paper studies the viability of implementing SIMD floating point arithmetic operations found in modern x86 processors using Emμcode traces. Our results show that for programs with 1 to 5 percent of the dynamic instructions replaced by Emμcode, a graceful performance degradation of only 1.3x to 4x is achievable.

1. Introduction

As process fabrication technology improves, hard faults are becoming a primary concern of computer architects. A hard fault occurs when a transistor or wire is either improperly fabricated or degrades to the point where it no longer operates correctly. Hard faults also manifest themselves as stuck-at faults where a transistor or node is persistently stuck at either a logical one or zero. Smaller manufacturing processes exacerbate the problems that have been observed already. Examples of degradation include electromigration [3], negative bias temperature instability [22], and the hot-electron effect [16]. These problems become more common as process variation increases with the decrease in device sizes, leading to an increased likelihood of defects due to fabrication [4]. Moreover, hard faults are persistent and non-local.

Previous work suggests many methods of correcting these hard faults. A form of graceful performance degradation whereby redundancy, which already exists in processors for the purpose of increasing performance through parallelism, enables execution to continue after certain units have failed [20] . Structural duplication (SD) [20] involves fabricating spare structures on the chip that can be be activated when a failure is discovered. Similarly extra pipeline stages to detect and correct errors have been studied [2]. However, not all of these techniques are useful for specialized structures like floating point units, or for correcting instructions that require the use of the entire execution pipeline, such as small-vector SIMD on contemporary CPUs, where there is no redundancy in the architecture. Due to the prohibitive cost in terms of area of adding spare floating point units, we pursue an alternate approach. Emμcode is a technique presented to mask these hard faults once detected. Emμcode corrects hard faults in accelerated structures by replacing faulty instructions with microcode that uses a different pipeline.

Emμcode is a graceful performance degradation (GPD) technique where hard faults are masked by executing an Emμcode *trace* in place of the original instruction that failed. An Emμcode trace is similar in nature to traces found in the Trace Cache architecture as presented by Rotenberg et al. [17]. Floating point Emμcode traces, for instance, are microcoded versions of the original instruction that utilize only the integer pipeline, which is more easily protected by other methods such as SD. For the purposes of this work, SSE floating point arithmetic [21] operations are chosen as a representative set of instructions that could be protected by Emμcode. Since duplicating the SSE pipeline is prohibitive because it is large, having another method of correcting hard faults over the instructions that use the SSE

978-1-4244-4421-2/09/$25.00

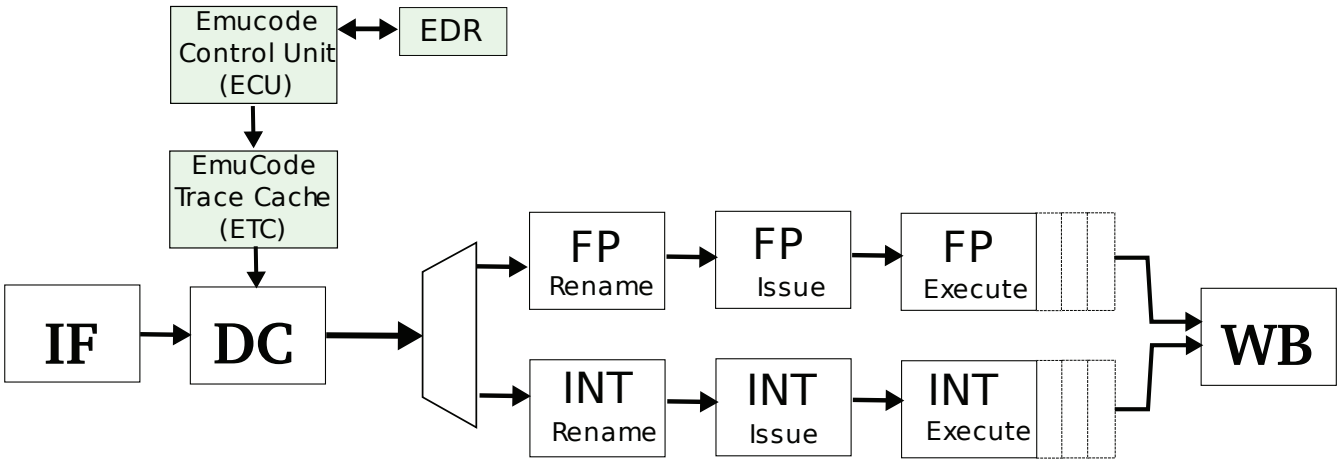

Figure 1. Architecture block diagram showing Emμcode. Components include: Emμcode Control Unit (ECU), Emμcode Trace Cache (ETC), and Emμcode Destination Register (EDR).

pipeline is necessary.

In order to facilitate Emμcode, low-cost microarchitecture modificiations are necessary. Emμcode traces must execute in a non-intrusive manner. They cannot change the original architected state as this might have an adverse effect on the original program. Another important aspect of Emμcode is the creation of the traces that will supplant an original instruction. A naive implementation is to simply compile a software floating point library and use the code generated as a trace. But there is potential for much optimization. These optimizations exploited knowledge of the actual trace, as well as the flexibility to make minimal but important modifications to the microarchitecture. Predication, extra architected registers only visible to the microcode, the microcode's flexibility with source and destination registers, and the implementation of new micro-ops were used to optimize the traces to be at best 6x faster than the naive implementation and 3.79x faster in the average case for the traces themselves.

Overview This paper is organized as follows. Section 2 motivates and describes the Emμcode implementation and linkage. Section 3 describes trace optimizations. Section 4 evaluates Emμcode in the context of a contemporary architecture. Section 5 gives an overview of related work. Section 6 concludes.

2. Emμcode

As process sizes shrink and the entire processing pipeline becomes increasingly susceptible to hard faults, protections for all instruction types must be addressed. As additional system instructions, such as new hardware virtualization support and special-purpose hardware like SSE variants [21], become prevalent, there arises a need for a catch-all technique that masks hard faults in parts of the chip not protected by duplication mechanisms. Emμcode is a technique that extends previous work to increase the reliability of complex contemporary microprocessors.

Emμcode provides GPD for accelerated instructions without the need to duplicate the structure by using software emulation rather than hardware duplication. As Figure 1 shows, Emμcode consists of the Emμcode controller that is used for orchestrating the use of Emμcode traces, which are microcoded versions of the instructions Emμcode protects. Emμcode traces are stored in the Emμcode Trace Cache, and relevant information about the current instruction being Emμcoded is stored in the Emμcode Destination Register (EDR). Once a fault is detected, Emμcode is invoked and replaces the hardware-accelerated instruction with a software-only microcode trace. Thus, the Emμcode trace is required to use parts of the pipeline which are protected by other techniques such as GPD or SD. Using this technique, Emμcode is able to cover a set of instructions that would otherwise incur high marginal cost per instruction if protected using SD alone.

Emμcode traces are designed to use structures not used by the original instruction, specifically, for this study protecting floating point instructions, the integer pipeline which can be protected more cheaply by SD. The traces should be optimized to facilitate Emμcode since performance is still a primary concern. This paper presents techniques that can produce optimized traces to be used in a microcode technique such as Emμcode.

2.1. Case Study: SSE Floating Point

Floating point arithmetic is inherently different from regular integer arithmetic. Since there is a floating point unit in most modern microprocessors, floating point arithmetic will provide a good evaluation for Emμcode traces because a floating point unit is an expensive piece of hardware to duplicate. Specifically, the SSE instructions that utilize both of the floating point units in the pipeline simultaneously, provide good examples of instructions that would be hard to duplicate with SD, because it would essentially require implementing an entire alternate floating point pipeline. Even if only considering a single floating point arithmetic operation, reproducing an entire alternate floating point unit for the purpose of reliability is area-expensive. Emμcode can be used to supplant these kinds of instructions in case of error by utilizing only the integer pipeline.

2.2. Emμcode Design

The Emμcode Control Unit (ECU) is the control logic for the Emμcode design. The Emμcode ECU takes over once a hard fault has been detected for a certain instruction. A bit is set in a bit-mask of all operation codes (op codes) that can be Emμcoded signifying that a fault has been detected. If the bit is set for the op code of the currently decoding instruction, Emμcode takes over and implements the original instruction with an Emμcode trace. Emμcode needs to operate in an environment where the original architected state does not change to ensure the proper execution of the program. Implementing and trapping into this space presents a challenge.

To support multiple code paths and branching within Emμcode traces, it is necessary to take control of an address space to differentiate between basic blocks of the Emμcode trace. Therefore a small address space was sectioned off for Emμcode use only. Emμcode traces requires ending the current instruction decode with a branch to the Emμcode trace, and branching back at the end of the trace. The overall process of trapping from the original instruction stream to the Emμcode trace and back is presented in Figure 2.

Emμcode linkage includes a set of registers called the Emu registers that are meant to be intermediary registers to store source operands and necessary information to return back to the original instruction stream. They are necessary because Emμcode cannot change the original state of the program, and therefore cannot change the architected register file. Along these same lines a new set of registers, called Emutrace registers, are used when executing an Emμcode trace. The Emu registers and Emutrace registers are treated the same as the architected registers with respect to the microarchitecture. The SSE instructions do not change flags, so the flags must be saved and restored to return the program to the exact state that was expected after the execution of the SSE instruction.

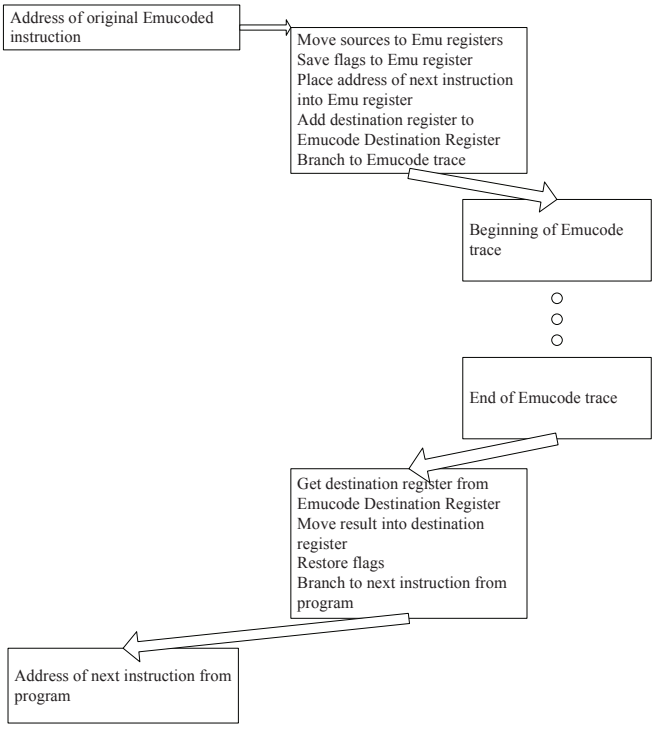

Figure 2. Overview of linking from original instruction stream to the Emμcode trace.

Two new micro-ops were implemented in order to facilitate saving and restoring the flags. These micro-ops have a delay of one cycle. Adding these two micro-ops is sufficient to allow microcode control flow for Emμcode traces.

As Figure 2 shows, the original instruction is decoded with a branch to the Emμcode trace, therefore the address of the next instruction is saved in an Emu register, and the destination of the instruction is saved in the Emμcode Destination Register. Next the Emμcode trace will take over and execute in a scratch space that does not modify the original architected state. Once the Emμcode trace has completed it will read in the destination register of the original instruction from the Emμcode Destination Register, move the result into it, and indirectly jump to the next instruction in the original instruction stream.

3. Trace Optimization

A baseline implementation of Emμcode traces is generated by compiling Soft Float 2b [8] with `gcc -O3 -finline_functions` and converting the assembly into micro-ops. These traces proved to be too detrimental to

overall performance. Since these traces have predictable dynamic behavior in actual practice, such as the sources or destination numbers not being NaN or denormal, we were able to target a predictable path that gcc is not able to do. We were also make minor modifications to the microarchitecture to improve performance. We also have more knowledge about the actual implementation of traces and know that for our case, predication to avoid hard-to-predict branches is a good idea whereas the value of predication is hard to determine statically.

3.1. Register Spill Reduction

In standard compilation, compilers will spill to memory under the assumption of a very large stack. The reason gcc requires the use of local variables is that the target ISA, x86, only has seven registers that gcc can use, in addition to the stack register. With so few registers, gcc cannot store all the data needed for the computation, including intermediate data in registers. We can get around this limitation by creating new architected registers that are not visible at the external ISA. Therefore, increasing the number of architected and available registers can improve the optimizations that we can do on the trace.

In microcode there is limited actual memory that can be used for temporary information without compromising the architected state. Therefore, in order to implement the trace the creation of an Emμcode scratch space is necessary, which is 1 kB in size and assumed to exist at the same level as the level 1 data cache. Although it never actually uses close to a full 1 kB, it is not known what future traces may use. The optimized traces do not use the scratch space, as they use seven additional architected registers instead. The traces converted from `gcc` compilation do use the scratch space.

3.2. Preserving ISA Semantics

Many CISC architectures implement a separate internal ISA by utilizing many RISC micro-operations [9]. Since gcc targets the external ISA it cannot exploit features of the internal RISC ISA. However, the internal RISC ISA does not necessary have the same limitations as the externally visible ISA. Therefore optimization is possible by using the RISC micro-ops which can allow for more registers than allowed by the target ISA.

Since gcc targets the external ISA, and in this case the x86 ISA, required many instructions to have one source operand, and one source and destination operand. However, the microcode is not limited to one source operand and one source and destination operand. The microcode, we assume, allows up to three source registers and a different destination register for many operations. An example of an ISA level instruction that can have two source operands and a different destination operand is the `lea` instruction on x86, which does allow this level of flexibility with an add operation. However, there are not necessarily complementary instructions like `lea` for other standard operations like `or`, `and`, and `xor` at the ISA level that can exist at the microcode level. Since gcc does not know this, it generates code that will move data that needs to be saved to the stack so that it can perform an operation that will clobber the register.

3.3. Predication in Trace Optimization

There are many hard-to-predict branches in the Emμcode traces. This leads to problems dealing with branch misprediction.

```
1   if (0 <= (int32_t) (zSig0<<1)) {
2       zSig0 <<= 1;
3       -- zExp;
4   }
```

Figure 3. The check for the mantissa of 32 bit floating point product result being too large.

Consider the piece of code in the 32 bit floating point multiply of Soft Float 2b in Figure 3. Mispredicting this branch changes the execution time of the trace from 25 cycles to 48 cycles, a 23 cycle penalty. On a hard-to-predict branch this causes concern for the average case.

Predication can be used to improve performance in these cases. Predication is the idea of having a Boolean guard around the issue of an instruction. Hsu and Davidson present the idea of having a guard expression over a store or branch instruction [10]. By implementing predication, it is possible to fetch and decode instructions unrelated to the condition before the condition is calculated. With predication, instructions can be moved around and implemented under the assumption that instructions will only be allowed to execute once the result of a Boolean guard is calculated. By doing this they create a new schedule of instructions that improves performance dramatically. In the example in Figure 3 the guard would be the boolean in the if statement on line 1 and the conditional instructions would be statements 2 and 3.

Many contemporary architectures support partial predication instructions. Partial predication as explained by Mahlke et al. [13] is the ability for instructions to take a set of condition codes. If the condition code turns out to be true, the result of execution is written back. If the condition code is not true, the result of the execution is thrown away and not used. This differs from full predication because full

predication would have squashed the instruction before it was issued. Therefore, partial predication support suffers the loss of actually executing the instruction, but gains the benefit of being easily implemented in modern microprocessor designs.

After if-conversion on that piece of code in Figure 3, the predicted path, in which neither the inputs nor the result is NaN or denormal, results in an average of 33 cycles and no hard-to-predict branches.

```
1   reg1 <- reg1 xor reg1
2   reg0 <- zSig0 shl 1
3   reg0 <- reg0 subtract 0
4   reg1 <- setz
5   zSig0 <- zSig0 shl reg1
6   zExp <- zExp add reg1
```

Figure 4. The check for the mantissa of 32 bit floating point product result being too large after if-conversion.

If-conversion in this case is solved by using the set conditional instruction. This instruction allows for a register to be set to a 1 if the condition code is true, else it does not change the register. Thus the simple if-conversion looks like Figure 4.

Another instruction that many architectures implement is a move conditional instruction. This allows for the conditional move of a value from a register/memory location to another register/memory location. This instruction could be used for optimization, however, it did not prove useful in the hand optimized traces as the places where it could have been added did not involve hard-to-predict branches.

3.4. New Micro-Operations

In the traces there are a small set of subroutines that account for many of the dynamic uops seen in practice. Many of these operations can be implemented in the internal RISC ISA with little hardware overhead. For example a *jamming right shift* was performed many times during the traces [8]. The `jshr` instruction differs from a `shr` by performing a logical or of all the bits shifted off and the least significant bit in the result already, and placing the result of that or into the least significant bit. Implementing this instruction is inexpensive and will probably not add to the critical path of the execute unit. As Figure 5 shows, to actually implement this functionality without the `jshr` would include quite a few micro-ops as well as a potential branch if predication was not available. The `jshr` micro-op replaces eight micro-ops that are necessary to microcode it, assuming that operating on the original source is allowed; otherwise it takes nine micro-ops. Each of these micro-ops takes one cycle to complete. By having hardware support for this micro-op, a potential branch is also avoided. Since this micro-op is very similar to a shift, it has been assigned the same delay as a normal right shift micro-op: one cycle.

```
1 reg0 <- src
2 reg1 <- reg0
3 reg0 <- reg0 shr shift_count
4 reg2 <- shift_count
5 reg2 <- neg reg2
6 reg2 <- shift_size plus reg2
7 reg1 <- reg1 shl reg2
8 reg3 <- setnz
9 reg0 <- reg0 or reg3
```

Figure 5. The microcode that would be necessary to implement a jshr instruction without modification to the shift unit. Src is the source that needs to be jamming right shifted. Shift_count is the number to shift the source by. Shift_size is the size of the register being operated on.

Another set of micro-ops implemented are the `pfloat` micro-ops. These micro-ops take a mantissa, an exponent, and a sign. There is a separate pfloat for each precision of floating point numbers. Each pfloat shifts the exponent left by the appropriate number, the sign left by the appropriate number, and add in the mantissa. However, since the shifts are a static number based on the precision of the float, there is no real need to shift, just to move the bits to the right place. There is an add operation between the mantissa and the exponent. However, since the add is only relevant to the exponent, the size of the add needed is 8 bits for the single precision pfloat and 11 bits for the double precision pfloat. With the add being the only computation needed to pack a float, the `pfloat` micro-ops were assigned the same delay as a regular `add` micro-op, which is one cycle.

3.5. Optimizing the Expected Path

Traces generated using an optimizing compiler without the benefit of the optimization techniques described in this work were found to provide unacceptable performance. Therefore, another set of Emμcode traces was created by hand. Examining the control flow of a 64 bit floating point multiply from the Soft Float 2b library will yield a control flow graph similar to Figure 6 under a few assumptions. The first assumption is that the rounding mode implemented is round to nearest even, so there is no need to check for which rounding mode to use. The second assumption is that there

are no IEEE floating point exceptions to implement as the x86/amd64 architectures do not implement most of them, and the ones implemented are not enabled by default.

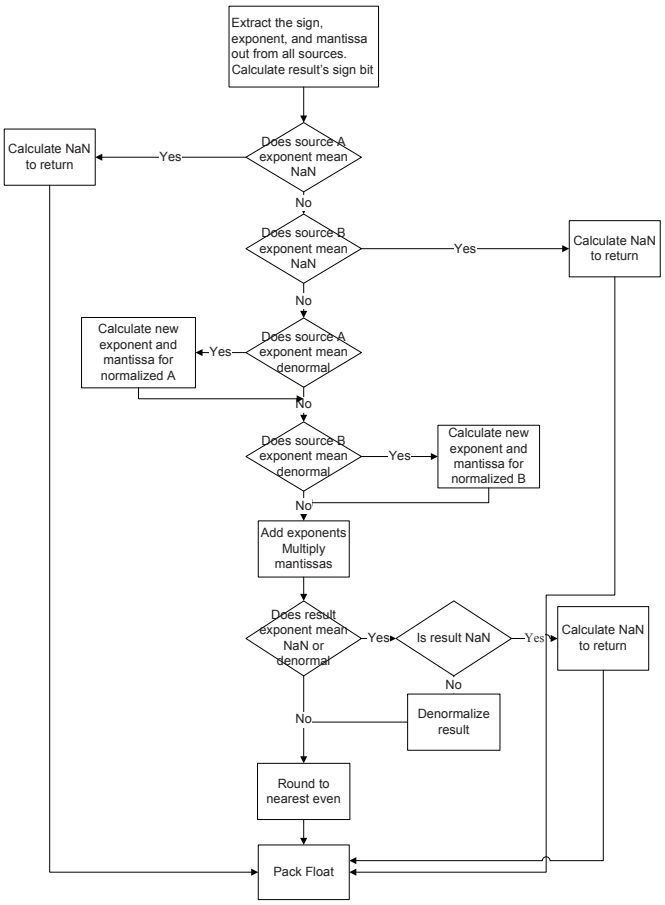

Figure 6. The control flow graph of a 64 bit floating point multiply assuming round to nearest even. A and B are the two input 64 bit floating point numbers.

As can be seen, there is a lot of branching associated with special cases for NaN and denormalized results or sources. In general these cases are rare, and the path that should be optimized for is the case where the inputs are neither NaN nor denormalized, and the resulting floating point number is neither a NaN nor denormalized. By optimizing for the expected path, with a good branch predictor which would be able to determine that NaNs and denormals are not expected cases, it is possible to implement a much simpler solution for manual traces. With current compiler technology, it is infeasible to perform the dynamic analysis to figure out if NaNs or denormals are expected, the optimization of the trace down this expected path is another optimization that is done. If the IEEE floating point exceptions are needed, most of them are determined along the special case paths as well. The floating point inexact exception is determined if rounding is needed at all and can be done fairly quickly along the main path if necessary.

4. Evaluation

For this paper, PTLSim [23] is used as a cycle-accurate x86/amd64 simulator for evaluation of Emμcode and direct execution of x86/amd64 instructions. PTLSim implements a superscalar architecture similar to the Intel Pentium 4 [9]. The changes are described in 2.2.

Soft Float 2b, as implemented by Hauser [8], is an implementation library of floating point arithmetic using only integer instructions that follows the IEEE binary floating point standard [1]. Soft Float 2b includes a software implementation of 32 bit, 64 bit, 80 bit, and 128 bit floating point arithmetic operations. Soft Float 2b has defined its own set of functions to calculate integer multiplies, a special kind of right shift called a jamming right shift, denormalizing floating point numbers, and rounding. It includes support for all four IEEE floating point rounding modes, all of the IEEE floating point exceptions, and proper NaN resolution. Since it includes everything necessary to implement any kind of IEEE floating point arithmetic from a base set of 32 bit integers, it is a perfect match for the kind of microcode traces needed to implement Emμcode.

Soft Float 2b was the basis for creating Emμcode traces. When creating the naive traces, Soft Float 2b was compiled under gcc -03 with inline functions turned on. When creating the hand optimized traces, Soft Float 2b was used as a reference for what needed to be done. Emμcode provides the accuracy specified in the IEEE standard, Soft Float 2b provides this as well, all three methods produced the same answer for the results presented.

4.1. Trace Evaluation

A microbenchmark is used in order to evaluate specific Emμcode trace implementations. The microbenchmark consists of invoking a random number generator to create a random set of floating point values. These values are designed to have a random sign, a random exponent between zero and 32 for single precision floating point numbers or zero and 64 for double precision floating point numbers, and a random mantissa. The window for number of exponents is done to ensure that when executing a floating point add, the result is not exactly the same as one of the two sources, since the full spectrum of exponents would result in many of these cases. Figure 7 shows the results of these microbenchmarks for the two types of traces created.

All of these instructions take six cycles natively. The manual trace implementations result in only 6x slowdown

Benchmark	Single Precision Add (%)	Single Precision Sub (%)	Single Precision Mul (%)	Double Precision Add (%)	Double Precision Sub (%)	Double Precision Mul (%)
equake	0	0	0	3.29	3.65	4.62
vpr	0.37	0	0.64	0.35	0	0
ammp	0	0	0	8.19	7.28	14.12
swim	0	0	0	16.36	3.64	19.99
applu	0	0	0	12.80	2.85	15.25
mesa	0.82	0.42	1.19	0	0	0.15
mgrid	0	0	0	19.25	6.83	2.83
mcf	0	0	0	0	0	0

Table 1. Percent of arithmetic operations performed during the 5 million instruction window

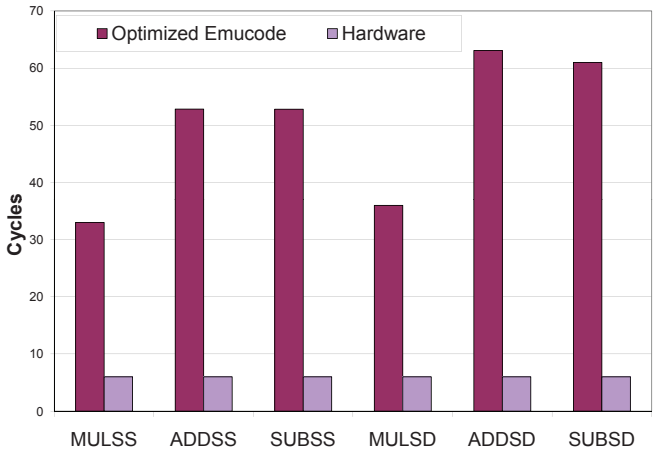

Figure 7. The average number of cycles each of the instructions takes to execute in Optimized Emμcode.

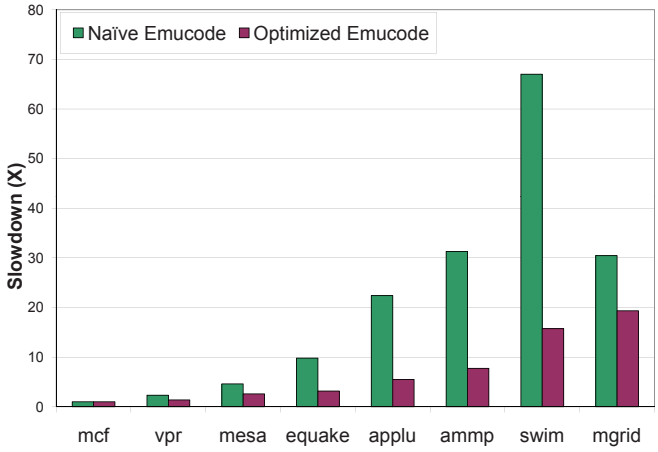

Figure 8. Performance impact of trace optimizations.

for the two kinds of multiplies and 10x slowdown for the two kinds of adds.

There are quite a few branches in the implementation of the floating point add and subtract instructions which degrade performance. There are two hard-to-predict branches in the add instructions: first, the sign of the two sources, which determines whether the instruction will add or subtract the two numbers, and second, which source has the larger exponent, which determines which mantissa to shift. If the branch predictor is correct it will take only 33 cycles, if wrong the next lowest is 46 cycles for mispredicting the larger exponent, then 57 cycles for mispredicting whether the signs agree, last is 68 cycles for mispredicting both. Since there is such a penalty, more predication support could drastically improve performance.

4.2. Program Results

We gather results from SPEC benchmarks being simulated inside PTLSim modified to support Emμcode. Table 1 shows the dynamic number of arithmetic operations encountered.

The results of the optimization of the traces can be seen in Figure 8. The naive versions exhibit a great deal more slowdown than the optimized versions. The results are arranged in the order of slowdown resulting from the optimized Emμcode traces. The notable exception is the `mgrid` trace, which is only about 1.5x worse than the optimized traces. The reason for this is because `mgrid` involves many operations where one or more of the source parameters is 0.0. Since the result of the operation is known once the zero has been detected, this is an early exit path for the naive traces, which will not perform any of the arithmetic operations until after this case is excluded. For the

optimized traces this is also not along the predicted path as 0.0 is a special case, which results in the optimized case being slightly worse than expected. Catching zero in a special case is done in the optimized trace, however, branch misprediction of the case causes the trace to take 51 cycles instead of the predicted 33 cycles. It is not feasible to perform predication over the entire operation to check for 0.0.

Since Emμcode traces are only relevant when programs actually use the instructions covered by Emμcode and Emμcode does not effect the control flow of programs, a simple model can be used to predict the slowdown of programs when Emμcode is on. For each arithmetic operation being Emμcoded, simply replace 1 cycle of time with the average execution time of the corresponding arithmetic operation's trace. It is important to distinguish between instructions and arithmetic operations since SSE instructions can reference multiple arithmetic operations. The equation for this model can be seen in Formula 1.

$$GPD = \frac{Cyc_{orig} - \sum_{i \in Emucoded} N_i * (EmuCyc_i - 1)}{Cyc_{orig}} \quad (1)$$

As Figure 9 shows, the prediction is quite close to the observed slowdown. The prediction underestimated the slowdown of `mesa` and `mgrid` by the most. This is because `mesa` and `mgrid` have many of arithmetic operations where 0 is added or multiplied to another number. This was not the path that Emμcode traces were optimized for. Since the average number of cycles on traces was predicted around assuming nonzero operands, it is not surprising that for `mesa` the average number of cycles taken for a single precision multiply is 45 cycles instead of 33 cycles. The other arithmetic operations slow down respectively in `mesa` and `mgrid`. Along the right hand side of Figure 9 the dynamic percentage of floating point operations performed during the 5 million instruction window can be seen. As the number of operations increases the amount of slowdown increases.

In `mcf`, there were zero dynamic floating point instructions during the 5 million instruction window that the benchmark was run for. Since many of the SPEC CPU2000 benchmarks cover this case, it is representative of all of those that do not use floating point operations. Mcf resulted in no slowdown, showing that Emμcode does not instill overhead if not used. For `mesa` and `vpr`, Emμcode resulted in a slowdown of 1.36x and 2.58x respectively. These benchmarks had a relatively small number of dynamic floating point operations that were Emμcoded. Since the alternative for a floating point unit failure is to not be able to run the program these two make good cases for why Emμcode is a valuable technique. It allows programs with a small number of floating point operations to continue running with a small amount of slowdown. However, with

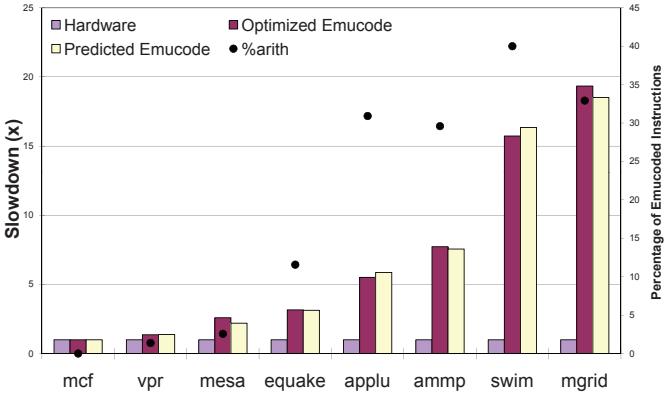

Figure 9. Performance impact of Emμcode for a 5 million instruction run. We plot the fraction of Emμcoded instructions on secondary axis. Emμcode provides graceful degradation of performance while preserving correctness.

many of the other benchmarks run, there was a larger slowdown observed, especially for the SPEC FP benchmarks. Since these programs primarily focus on floating point operations Emμcode instills a relatively large slowdown on them. However, if a floating point unit has failed then these programs probably would not be able to obtain such performance levels, as software emulation is very closely involving traps into the operating system.

5. Related Work

Correction of Hard Faults A large body of research exists in the area of hard fault correction. Graceful performance degradation (GPD) can be built around the redundancy that already exists within current microprocessors in order to take advantage of parallelism. For example, many superscalar processors, such as the Intel Pentium 4 or the AMD Athlon, have multiple arithmetic logic units (ALU). If one such unit should become unusable, it is still possible to execute code using one of the other units, although this would decrease performance. To implement GPD requires that the functional units be somewhat modular. This means that it is possible to disconnect the unit from the rest of the pipeline, which can be done easily in modern microprocessors setting the unit flagged as busy at issue as was done by Srinivasan et al. in [20]. They show that GPD provides 1.42x the reliability with a performance cost of less than five percent. However, in large accelerator structures there is little to zero redundancy built in, so this kind of GPD will not provide reliability over those structures whereas Emμcode can.

Structural duplication is similar to the concept of dual modular redundancy (DMR), where reliability is achieved by using two microprocessors to run the same program and comparing the two results. If the results differ it is necessary to determine which one is incorrect. To isolate the failed unit, triple modular redundancy (TMR) can be employed where there are three copies of the structure being protected. The consensus of the three is concluded to be the correct result. However, it is possible to add extra functional units and simply turn them on as needed. As with GPD, this form of SD requires that the architecture allow for the turning on and off of units when it is determined that they have failed. A limitation of SD is that it incurs a large overhead in chip area. In the extreme case of TMR it requires 200% overhead to guarantee reliability, a cost too high for most modern architectures. Srinivasan et al. [20] show that SD improves reliability to 3.17x the base value for 2.25x the cost. However, duplication of expensive accelerator structures such as a floating point unit is expensive in terms of area and done more for performance than reliability. Emμcode allows the coverage of these structures without the need for duplication.

Bulletproof by Shyam et al. [19] is another method of GPD where a test suite is run on the mircroprocessor with checkpointing for recovery from failures, and faulty structures are disabled. While Bulletproof would need DMR in order to be able to recover from all points of failure, which is expensive in terms of area, Emμcode does not. DIVA by Austin [2] uses extra stages in the pipeline to implement a more reliable checker core that re-executes instructions and corrects if the checker finds a problem, with the checker core's answer taken as the correct one. In order for DIVA to be able to provide reliability over larger accelerator structures, it would be required to have implementations of the units used to calculate the result. Doing so requires an extra copy of expensive accelerator structures, Emμcode does not. Meixner et al. show how to recompile code to circumvent structures with hard faults in what they call Detouring [14]. Emμcode does not require recompiling code. Core Salvage by Joseph [11] utilizes redundancy within a multicore system between the cores by having certain instructions trap into a virtual machine monitor (VMM). Emμcode obtains the same result without the need for an expensive trap operation. This paper also considers the optimization of microcode routines, whereas Detouring and Core Salvage do not.

In modern microprocessors there are numerous accelerated instructions that would be costly to duplicate in SD or would require expanding the capabilities of other checkers. Examples of these are instructions that modify protected state of the architecture, such as an INT instruction on x86 which necessitates a context switch, or the SSE instructions which utilize the entire floating point pipeline to perform one operation. The common implementation of GPD cannot be used with the standard implementations for these instructions. Duplication of these structures would be expensive because adding an extra floating point unit requires a lot of chip area, justified more for performance reasons than for reliability. Therefore another method is necessary for correcting these types of instructions. This study explores such techniques.

Detection of Hard Faults Emμcode assumes an underlying hard fault detection mechanism. Hard error detection has been explored in many ways. The use of redundant threads on SMT machines was explored by Schuchman and Vijaykumar in [18]. Attaching a self-checking mechanism is explored by Yilmaz et al. in [15]. Software propagation of hard faults is performed by Li et al. [12]. Online detection using firmware that periodically checks for errors by using the scan latches is explored by Constantinides et al. [5]. Emμcode assumes that one of these low-overhead effective methods of detection can be used to determine when Emμcode must mask a hard fault.

Other Related Work Dally has shown in [7] that it is possible to optimize over multiple floating point operations. He has shown that by analyzing the data flow graph of programs it is possible to combine operations common among adds and multiplies over the same numbers and reduce execution time by 40 percent. However, the traces that he used are designed to be used with a floating point unit where the various sub-operations it performs are open for optimization. Since Emμcode is designed to provide reliability over floating point units, this information is not useful for creating Emμcode traces, targeting the integer pipeline.

Corliss et al. in [6] presents a framework for inserting micro-code routines into decoded instructions for performance reasons. They present many of the important structures necessary for the insertion of Emμcode routines without affecting the ISA state. They introduce structures such as pattern matching based on the instruction bits for parameterized replacement, dedicated registers, and a DICEPC for branching within their application customization functions (ACF). It would be possible to utilize a DISE-like framework to insert Emμcode traces as an ACF if the instruction op-code was used as the DISE trigger after the hard fault was detected.

6. Conclusion

This paper presents Emμcode as a technique for masking hard faults in hardware structures that are either seldom used or expensive to duplicate. We evaluate the cost of using microcode traces in place of the original instruction. We

show that it is possible to achieve a 6x slowdown when microcoding a floating point arithmetic operation. A 6x slowdown for floating point operations for these instructions is acceptable for many programs, especially those that do not have many floating point instructions. For workloads with fewer than five percent floating point operations covered by Emμcode a slowdown of only 1.36x to 2.56x is achievable, and for workloads with seven percent to 40 percent floating point operations covered by Emμcode a slowdown of 3.16x to 19.33x with Emμcode enabled is observered. There is no performance overhead if no floating point instructions covered by Emμcode are used. However, Emμcode is meant for expensive structures to replicate as well as infrequently used instructions.

Supporting Emμcode requires adding minimal hardware to a conventional processor. It requires extra structures to store instruction information. It also necessitates a lot of indirect branching and storing of data that is no longer available after the original instruction is decoded. These problems result from traces requiring microcode branches, which we find to be hard-to-predict, complicating achieving performance. Elimination of the branches is a primary concern for this kind of microcode trace generation, as they are very costly to execution time. We show that partial predication, additional registers, the use of the microcode's flexibility with source and destination registers, and the addition of a few new inexpensive micro-ops can prove useful for optimizing the traces to be at best 6x faster than the naive implementation and 3.79x faster in the average case for the traces themselves.

References

[1] Ieee standard for binary floating-point arithmetic. ANSI/IEEE Std 754-1985, Aug. 1985. [Online]. Available: http://ieeexplore.ieee.org/stamp/stampjsp?arnumber=30711&isnumber1316.

[2] T. M. Austin. Diva: A reliable substrate for deep submicron microarchitecture design. In *MICRO 32*, pages 196–208, 1999.

[3] D. T. Blaauw, C. Oh, V. Zolotov, and A. Dasgupta. Static electromigration analysis for on-chip signal interconnects. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27:39–48, 2003.

[4] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25:10–16, Nov.-Dec. 2005.

[5] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-based online detection of hardware defects mechanisms, architectural support, and evaluation. *MICRO 40*, pages 97–108, Dec. 2007.

[6] M. L. Corliss, E. C. Lewis, and A. Roth. Dise: A programmable macro engine for customizing applications. In *ISCA 30*, pages 362–373, 2003.

[7] W. J. Dally. Micro-optimization of floating-point operations. In *ASPLOS*, pages 283–289, 1989.

[8] J. R. Hauser. Soft float 2b, May 2002. [Online]. Available: http://www.jhauser.us/arithmetic/SoftFloat.html.

[9] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, Q1:1–13, 2001.

[10] P. Y. T. Hsu and E. S. Davidson. Highly concurrent scalar processing. In *ISCA 13*, pages 386–395, 1986.

[11] R. Joseph. Exploring salvage techniques for multi-core architectures. In *HPCRI*, 2005.

[12] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *ASPLOS 13*, pages 265–276, New York, NY, USA, 2008. ACM.

[13] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu. A comparison of full and partial predicated execution support for ilp processors. In *ISCA*, pages 138–149, 1995.

[14] A. Meixner and D. J. Sorin. Detouring: Translating software to circumvent hard faults in simple cores. In *DSN 38*, pages 80–89, 2008.

[15] S. Ozev, D. Sorin, and M. Yilmaz. Low-cost run-time diagnosis of hard delay faults in the functional units of a microprocessor. *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pages 317–324, Oct. 2007.

[16] J. M. Rabaey. *Digital Integrated Circuits: A Design Perspective*. Prentice-Hall, Upper Saddle River, NJ, 1996.

[17] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *MICRO 29*, pages 24–35, Washington, DC, USA, 1996. IEEE Computer Society.

[18] E. Schuchman and T. N. Vijaykumar. Blackjack: Hard error detection with redundant threads on smt. In *DSN 37*, 2007.

[19] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra low-cost defect protection for microprossesor pipelines. *ASPLOS*, pages 113–119, 2006.

[20] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *ISCA*, pages 520–531, 2005.

[21] A. M. D. T. Staff. *AMD x86-64 Architecture Programmers Manual, Volume 4: 128-Bit Media Instructions*. Advanced Micro Devices, 2002.

[22] J. H. Stathis and S. Zafar. The negative bias temperature instability in mos devices: A review. *Microelectronics and Reliability*, 46:270–286, Feb. 2006.

[23] M. T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *IPASS*, pages 23–34, 2007.