AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors

Eric Rotenberg
Computer Sciences Department
University of Wisconsin - Madison
ericro@cs.wisc.edu

Abstract

This paper speculates that technology trends pose new challenges for fault tolerance in microprocessors. Specifically, severely reduced design tolerances implied by gigaherz clock rates may result in frequent and arbitrary transient faults. We suggest that existing fault-tolerant techniques -- system-level, gate-level, or component-specific approaches -- are either too costly for general purpose computing, overly intrusive to the design, or insufficient for covering arbitrary logic faults. An approach in which the microarchitecture itself provides fault tolerance is required.

We propose a new time redundancy fault-tolerant approach in which a program is duplicated and the two redundant programs simultaneously run on the processor. The technique exploits several significant microarchitectural trends to provide broad coverage of transient faults and restricted coverage of permanent faults. These trends are simultaneous multithreading, control flow and data flow prediction, *and* hierarchical processors -- *all of which are intended for higher performance, but which can be easily leveraged for the specified fault tolerance goals. The overhead for achieving fault tolerance is low, both in terms of performance and changes to the existing microarchitecture. Detailed simulations of five of the SPEC95 benchmarks show that executing two redundant programs on the fault-tolerant microarchitecture takes only 10% to 30% longer than running a single version of the program.*

1. Introduction

The commercial success of general purpose computers, from personal computers to servers and multiprocessors, can be attributed to the proliferation of high performance single-chip microprocessors. Both *technology advances* -- circuit speed and density improvements -- and *microarchitecture innovations* -- exploiting the parallelism inherent in sequential programs -- fuel the rapid growth in microprocessor performance that sustains general purpose computing. And interestingly, both technology and microarchitectural trends have implications to microprocessor fault tolerance.

Technology-driven performance improvements will inevitably pose new challenges for fault tolerance in microprocessors [1,2]. In particular, there may come a time when clock rates and densities are so high that the chip is prone to *frequent, arbitrary transient faults* [2]. High clock rate designs require (1) small voltage swings for fast switching and power considerations, and (2) widespread use of high performance, but relatively undisciplined, circuit design techniques, e.g. dynamic logic. This combination is problematic because dynamic logic is susceptible to noise and crosstalk, and low voltage levels at charged nodes only make it more so. Pushing the technology envelope may even compromise conservative, static circuit designs, due to *reduced tolerances in general* (consider, for example, the difficulty in managing clock skew in gigaherz chips).

Specialized fault-tolerant techniques, such as error correcting codes (ECC) for on-chip memories [3] and Recomputing with Shifted Operands (RESO) for ALUs [4,5], do not adequately cover arbitrary logic faults characteristic of this environment. And while self-checking logic techniques [6,7] can provide general coverage, chip area and performance goals may preclude applying self-checking logic globally (for example, integrating self-checking into the design methodology and cell libraries). Finally, system-level fault tolerance, in the form of redundant processors, is perhaps too costly for small general purpose computers.

Therefore, we propose a *microarchitecture-based fault-tolerant approach*. That is, instead of perpetuating the rigid separation between computer architecture and fault tolerance, broad coverage of transient faults, with low to moderate performance impact, can be achieved by exploiting microarchitectural techniques -- techniques that are already incorporated in the microprocessor for performance reasons.

In this paper, we propose and evaluate a time redundancy fault-tolerant approach called *Active-stream/Redundant-stream Simultaneous Multithreading*, or AR-SMT. AR-SMT exploits several recent microarchitectural trends to provide low-overhead, broad coverage of transient faults, and restricted coverage of some permanent faults.

1.1. AR-SMT time redundancy

Time redundancy is a fault-tolerant technique in which a computation is performed multiple times on the same hardware [4,8]. This technique is cheaper than other fault-tolerant solutions that use some form of hardware redundancy, because it does not replicate hardware. Of course, simple time redundancy can only detect transient faults that are present during one or more of the redundant computations, but not all of them. Frequent and relatively short-lived transient faults are the primary focus of this paper, and therefore time redundancy is a viable approach. (However, detection of long-lived transient faults and some permanent faults are also addressed in our proposal.)

The disadvantage of time redundancy is the performance degradation caused by repetition of tasks. In a general purpose computer, full programs are the tasks to be performed. If we assume 2x redundancy, *program-level time redundancy* effectively doubles the execution time of a program because the same program is run twice back-to-back, as shown in Figure 1(a).

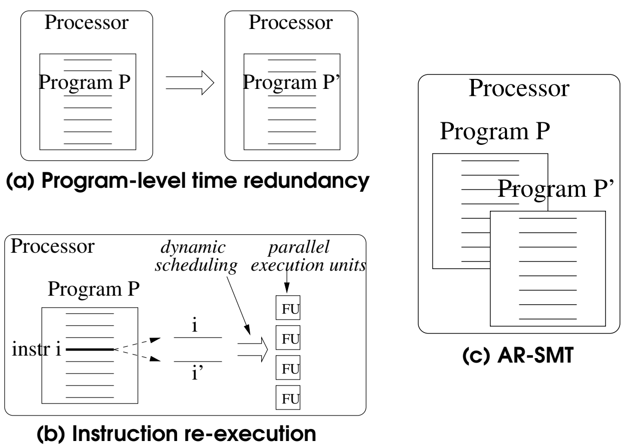

FIGURE 1. Time redundancy techniques.

Instruction re-execution [5] addresses the performance degradation caused by time redundancy. With instruction re-execution, the program is not explicitly duplicated. Rather, when an instruction reaches the execution stage of the processor pipeline, two copies of the instruction are formed and issued to the execution units (Figure 1(b)). Because instructions are duplicated within the processor itself, the processor has flexible control over the scheduling of redundant computations. Dynamic scheduling logic combined with a highly parallel execution core allows the processor to “scavenge” idle execution cycles and execution units to perform the redundant computations. This is possible because there are not always enough independent operations in the program to fully utilize the parallel resources. That is, *dynamically scheduled superscalar processors* [9] are designed for the irregularity of instruction-level parallelism in ordinary, sequential programs, and consequently the peak parallelism supported by the microarchitecture is greater than sustained parallelism of programs.

The downside of instruction re-execution is that it provides limited hardware coverage. Only a *single stage* of a complex processor pipeline is conceptually duplicated: the functional units, i.e. ALUs. In modern superscalar processors, this comprises only a fraction of the overall complexity of the chip.

AR-SMT combines the full processor coverage of program-level time redundancy with the performance advantages of instruction re-execution. In AR-SMT, *two explicit copies of the program run concurrently on the same processor resources*, as shown in Figure 1(c). The two copies are treated as completely independent programs, each having its own state or program context. Consequently, as with program-level time redundancy, the entire pipeline of the processor is conceptually duplicated, providing broad coverage of the chip. The performance advantages of instruction re-execution are retained, however, due to the concurrent sharing of processor resources by the two program threads. This technique is made possible by a recent microarchitecture innovation called *simultaneous multithreading (SMT)* [10,11]. SMT leverages the fine-grain scheduling flexibility and highly parallel microarchitecture of superscalar processors. As we mentioned before, often there are phases of a single program that do not fully utilize the microarchitecture, so sharing the processor resources among multiple programs will increase overall utilization. Improved utilization reduces the total time required to execute all program threads, despite possibly slowing down single thread performance. In AR-SMT, half of the program threads happen to be duplicates for detecting transient faults.

AR-SMT is an example of exploiting the microarchitecture, in this case simultaneous multithreading, to achieve fault tolerance. In the following two sections, we first describe the basic mechanisms behind AR-SMT, and then point out other microarchitecture trends that are exploited for (1) reducing the performance overhead of time redundancy even further and (2) improving fault coverage.

1.2. Basic operation of AR-SMT

Figure 2 shows an abstraction of AR-SMT. The solid arrow represents the dynamic instruction stream of the original program thread, called the *active stream* (A-stream). As instructions from the A-stream are fetched and executed, and their results committed to the program’s state, the results of each instruction are also pushed onto a FIFO queue called the *Delay Buffer*. Results include modifications to the program counter (PC) by branches and any modifications to both registers and memory.

A second *redundant instruction stream* (R-stream), represented with a dashed arrow in Figure 2, lags behind the A-stream by no more than the length of the Delay Buffer. The A-stream and R-stream are simultaneously processed using the existing SMT microarchitecture. As the R-stream is fetched and executed, its committed results are compared to those in the Delay Buffer. A fault is detected if the comparison fails, and furthermore, the committed state of the R-stream can be used as a checkpoint for recovery.

The Delay Buffer ensures time redundancy: the A-stream and R-stream copies of an instruction are executed at different times, in general providing good transient fault coverage. A fault may cause an error in the A-stream, the R-stream, or both. An error in the A-stream is detected after some delay through the Delay Buffer. An error in the R-stream is detected before committing the first affected instruction. A fault may induce errors in both streams, in which case only the R-stream plays a role in detection.

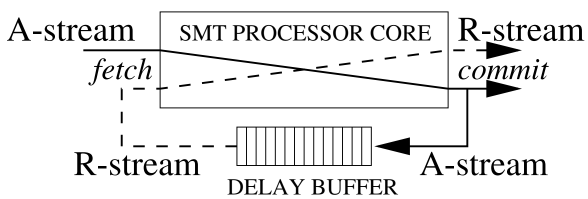

FIGURE 2. High level view of AR-SMT.

1.3. Exploiting other microarchitecture trends

The AR-SMT model exploits simultaneous multithreading to implement time redundancy. We now describe two additional microarchitectural trends that can be leveraged for both higher performance and improved fault tolerance.

1.3.1. Improved performance through control and data "prediction". The order in which instructions may execute is dictated by *data dependences* and *control dependences* among instructions. If instruction *i* produces a value that is used by instruction *j*, *j* cannot execute until *i* completes. This is called a data dependence. Likewise, branch instructions introduce control dependences: the next instructions to be fetched after a branch instruction depends on the outcome of the branch instruction.

High performance processors attempt to execute multiple instructions in parallel each cycle. Unfortunately, *instruction-level parallelism* is limited or obscured by both control and data dependences in the program. Consider the sequence of five instructions shown in Figure 3(a). Instructions i2 through i4 all have a data dependence with an immediately preceding instruction, which means they must execute serially as shown in Figure 3(b). Furthermore, although instruction i5 has no data dependences, it can not be fetched until the outcome of branch i4 is known. The control dependence with i4 is a severe performance penalty because it both serializes execution *and* exposes some number of cycles to fetch instruction i5 into the processor, labeled in the diagram as "pipeline latency".

Control dependences are typically alleviated with branch prediction [12,13]: the outcomes of branches can be predicted with high accuracy based on the previous history of branch outcomes. Branch prediction allows the processor to fetch instructions ahead of the execution pipeline, so i5 can execute as early as cycle 1 if the branch i4 is predicted to be taken, as shown in Figure 3(c).

More recently, researchers have even suggested predicting data values [14,15,16]. At instruction fetch time, the source operands of instructions are predicted. In this way, they do not have to wait for values from their producer instructions -- instructions may execute in parallel under the assumption that the predicted values are correct. In Figure 3(c), values for r1, r2, and r3 are predicted, so instructions i1 through i4 may execute in parallel during cycle 1. In cycle 2, the computed results produced during cycle 1 are used to *validate* the predicted values for r1, r2, and r3 (special recovery actions are required in the event of mispredictions [14,17]).

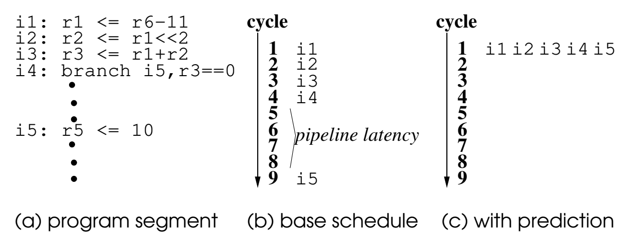

FIGURE 3. Control and data dependences.

With both control and data prediction, the processor exposes more instruction-level parallelism and speeds execution of the program. AR-SMT can exploit this same concept to minimize the execution time of the R-stream, because *the Delay Buffer contains data and control information from the first run of the program* (i.e. results from the A-stream). The state in the Delay Buffer provides the R-stream with *perfect* control and data flow prediction: instructions can execute essentially free from all control and data dependences. Note that this does not reduce fault coverage of the processor in any way, since instructions still pass through all stages of the pipeline. They simply pass through the pipeline quicker.

Furthermore, *the hardware used for validating data and control predictions is in fact the hardware for detecting faults*. Normally, predictions are compared with computed results to detect mispredictions. In the case of the R-stream, these comparators validate so-called "predictions" obtained from the A-stream against values computed by the R-stream. If the validation fails, then a transient fault must have occurred in either stream.

In summary, existing prediction techniques can be leveraged to reduce the overhead of time redundancy, both in terms of performance and hardware: "predictions" from the Delay Buffer speed execution of the R-stream, and existing validation hardware is used for fault detection logic.

1.3.2. Improved fault tolerance through hierarchy and replication. In the search for ever-increasing amounts of instruction-level parallelism, high performance processors have become exceedingly complex. There has been a recent interest among researchers to reduce complexity by dividing up large, centralized structures and wide datapaths [18,19,20,21,17]. By reducing complexity in an intelligent way, a high clock rate can be achieved without sacrificing the exploitation of instruction-level parallelism.

Complexity can be reduced through hierarchy [21]. Instead of having one large processing structure work on, say, 128 or 256 individual instructions at a time, 8 smaller processing elements can each work on 16 or 32 instructions.

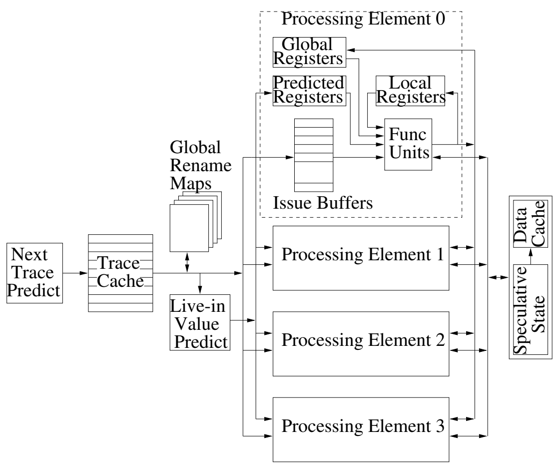

FIGURE 4. A trace processor [17].

A *trace processor* [17], shown in Figure 4, is one example of a hierarchical microarchitecture. It dynamically partitions the instruction stream into larger units of work called *traces*. A trace is a dynamic sequence of instructions; typical trace lengths are 16 or 32 instructions, and there can be any number of branches within a trace. Traces are considered the fundamental unit of work, and the processor is organized as such. In particular, the execution resources are distributed among multiple processing elements (PEs), each PE resembling a moderate-sized superscalar processor. Traces are explicitly predicted and fetched as a unit (i.e. trace prediction instead of individual branch prediction), and subsequently dispatched to a PE. Each PE is allocated a single trace to execute. Inter-trace data dependences are predicted to enhance the parallel processing of traces.

It is beyond the scope of this paper to discuss the complexity and performance advantages of trace processors: suffice it to say that sequencing and executing the program at the higher level of traces is potentially more efficient than processing instructions individually. However, we can point out one particular advantage this microarchitecture offers for AR-SMT: *the replicated PEs inherently provide a coarse level of hardware redundancy*. We propose detecting permanent faults within PEs by guaranteeing that a trace in the A-stream and its corresponding redundant trace in the R-stream execute on different PEs. This requires storing a few extra bits of information per trace in the Delay Buffer to indicate which PE executed the trace. Furthermore, the PE allocation constraint imposed on the R-stream does not degrade performance with respect to arbitrary allocation. If only one PE is currently free, but it cannot be used by the R-stream because it is the same PE on which the trace first executed, then the *next* PE to become available is guaranteed to be a different PE; meanwhile, the A-stream can make use of the available PE.

The coarse level of hardware redundancy is especially appealing for *re-configuring* the trace processor in the event of a permanent PE fault. It is easier to remove a PE from the resource pool than to remove an individual instruction buffer from among hundreds of closely-integrated buffers. The small number of PEs and the fact that PEs are relatively isolated from each other (modularity) makes re-configuration conceptually simple.

1.4. Related work

A spectrum of time redundancy techniques is presented in [22]. The key realization is that all time redundancy approaches essentially duplicate the program and they differ only in the granularity at which redundant computation is *interleaved*. This paper is the culmination of that earlier research.

Recently and independently, the Stanford ROAR project [23] proposed *Dependable Adaptive Computing Systems*. The architecture is composed of a general purpose computer and a reconfigurable FPGA coprocessor, and encompasses at least three significant concepts. First, the use of redundant but *diverse* modules significantly increases dependability during common-mode failures and, furthermore, the reconfigurable coprocessor is an ideal platform for synthesizing diverse modules. Second, the FPGA can be reconfigured down to the gate level, so recovery from failures does not require swapping in large spare modules. Third, SMT is suggested for achieving low-overhead fault tolerance. [23] is an overview of the ROAR project and, consequently, a fault-tolerant SMT implementation is not put forth and evaluation is based on analytical estimates and a compression algorithm multithreaded by hand.

2. AR-SMT implementation issues

In this section key implementation issues are presented. Section 2.1 reviews SMT hardware techniques. Where SMT policy decisions are required, we describe how these policies are tailored to AR-SMT. Section 2.2 discusses new issues that arise because the dynamically created R-stream is not a true software context, requiring minor operating system and hardware support.

2.1. Implementing SMT

Most of the design is derived from work on simultaneous multithreaded machines [10,11]. The techniques are well established and understood, and recent research shows that SMT can be incorporated into existing high performance processors rather seamlessly. The following discussion focuses on two important aspects of any SMT machine: (1) separating register and memory state of multiple threads, and (2) sharing critical processor resources.

2.1.1. Handling register values from multiple threads.

Each thread must be provided its own register state, and register dependences in one thread must not interfere with register dependences in another thread. The approach in [11] leverages *register renaming* to transparently and flexibly share a single, large physical register file among multiple threads.

Register renaming overcomes the limitation of having too few general-purpose registers in the instruction-set architecture (e.g. 32). Typically, the processor provides many more *physical registers* so that multiple writes to the same logical register can be assigned unique physical registers. This allows the writes, and their dependent instruction chains, to proceed independently and in parallel. A *register map* maintains the most current mapping of logical to physical registers.

In SMT, there is still a single, large physical register file but each thread has its own register map. The separate maps guarantee the same logical register in two different threads are mapped to two different physical registers.

The approach in [11] has two advantages. First, the most complex part of the processor -- the instruction issue mechanism -- is unchanged. The fact that instructions from multiple threads co-exist in the processor is transparent to the instruction issue and register forwarding logic because it uses physical register specifiers, and renaming ensures the physical registers of various threads do not overlap. Second, managing a shared centralized register file instead of dedicated per-thread register files allows some threads to use more registers than other threads. Section 3.2 shows the R-stream requires fewer resources than the A-stream.

2.1.2. Handling memory values from multiple threads.

The *memory disambiguation unit* is the mechanism for enforcing data dependences through memory. It ensures that a load gets its data from the last store to the same memory address. The disambiguation hardware consists of load and store buffers to track all outstanding memory operations, and logic to detect loads and stores having the same memory address. Like the register file, this buffering is shared among the SMT threads, i.e. loads and stores from multiple threads co-exist in the disambiguation unit.

As with register dependences, memory dependences from different threads must not interfere with each other. Thus, memory addresses must be augmented with a *thread identifier* if disambiguation is based on virtual addresses. The same virtual address used by two different threads is distinguishable using the thread id. Thread ids need not be stored in the data cache, however, if physical tags are used.

2.1.3. Concerning instruction fetch for the R-stream.

Conventional SMT requires multiple program counters (PCs), one for each of the threads. Furthermore, branch predictor structures must be shared by multiple threads for predicting control flow.

AR-SMT also requires multiple program counters, but the control flow predictor does not have to be shared between the A-stream and R-stream. Recall that the PCs of retired A-stream instructions are stored in the Delay Buffer, in a sense providing control flow predictions for the R-stream. Therefore, the control flow predictor structures are dedicated to the A-stream: predictor accuracy and complexity remain unaffected.

2.1.4. Sharing processor bandwidth. The trace processor pipeline is shown in Figure 5. At each pipeline stage, we show how AR-SMT shares processor bandwidth between the A-stream and the R-stream. Some parts of the pipeline are *time-shared* and others are *space-shared*. Time-shared means in any given cycle, the pipeline stage is consumed entirely by one thread. Space-shared means every cycle a fraction of the bandwidth is allocated to both threads.

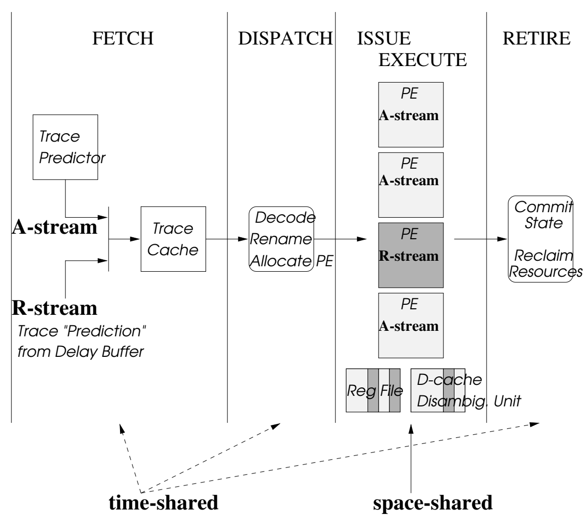

FIGURE 5. How threads share pipeline stages.

The instruction fetch/dispatch pipeline is time-shared due to the nature of traces. Traces are fetched and dispatched as an *indivisible unit*, at a rate of 1 per cycle. Clearly, this bandwidth cannot be split and shared between two threads -- traces belong to one thread or the other.

Likewise, traces are retired from the processor at the rate of 1 per cycle. Retirement is essentially the dual of dispatch in that resources are reclaimed, e.g. the PE, physical registers, load/store buffers, etc. Therefore, the retire stage is also time-shared.

Of course, execution resources are space-shared. In trace processors, the unit of space-sharing is a PE. For example, Figure 5 shows 3 PEs allocated to the A-stream and 1 PE allocated to the R-stream.

An important design decision is how to allocate bandwidth to multiple threads to minimize overall execution time. For pure SMT, many sophisticated policies are possible [11]. Adaptive heuristics allocate bandwidth dynamically based on control prediction accuracies, amount of instruction-level parallelism exhibited by each thread, etc.

For AR-SMT, however, there is significantly less scheduling flexibility because the A-stream and R-stream are tightly coupled via the Delay Buffer. More specifically, (1) the R-stream cannot run ahead of the A-stream and (2) the A-stream can only run ahead of the R-stream by an amount equal to the length of the Delay Buffer. It is not clear how much added benefit an SMT-like algorithm can yield over a simple scheduler given these constraints; clearly this is an area that demands further study.

The AR-SMT scheduling rules are consequently trivial. The rules are based on keeping the Delay Buffer full and only involve arbitration for the fetch/dispatch and retirement stages.

1. **Fetch/dispatch pipeline arbitration.** If the Delay Buffer is full, the R-stream is given priority to access the fetch/dispatch pipeline.
2. **Retirement stage arbitration**: If the Delay Buffer is not full, the A-stream has priority to retire a trace.

These rules cause deadlock if it were not for the following definition of “full” in rule #1: the Delay Buffer is considered full when the number of free entries left (in terms of traces) is equal to the number of PEs in the trace processor. Thus, there is always room to drain the A-stream from the PEs into the Delay Buffer, in turn allowing the R-stream to proceed.

2.2. New issues and operating system support

AR-SMT introduces new issues that do not arise with pure SMT. The R-stream is not a true software context. It is created on the fly by hardware and the operating system (O/S) is unaware of it. Yet the R-stream must maintain a separate physical memory image from the A-stream and exceptional conditions must be properly handled.

2.2.1. Maintaining a separate memory image. The R-stream, because it is delayed with respect to the A-stream, needs a separate memory image (just as there is a separate register “image” in the physical register file). A simple solution is proposed here.

- The O/S, when allocating a physical page to a virtual page in the A-stream context, will actually allocate two contiguous physical pages. The first is for the A-stream to use, and the second is for the R-stream to use. In this way, there is still the appearance of a single address space with a single set of protections, but simple redundancy is added to the address space.
- Address translations are placed in the Delay Buffer for use by the R-stream. This is the only virtual address translation mechanism for the R-stream because no page table entries are explicitly managed on its behalf. Addresses are translated by taking the original translation and adding 1 to it.

Another solution is to make the O/S aware of the R-stream as a true context (pure SMT).

2.2.2. Exceptions, traps, and context switches. Exceptions, traps, and context switches are handled by synchronizing the A-stream and R-stream. When any such condition is reached in the A-stream, the A-stream stalls until the Delay Buffer completely empties. At this point the two contexts are identical and the R-stream is terminated. Now only the A-stream is serviced, swapped out, etc., which is required if the operating system has no knowledge of the redundant thread. When resuming after a context switch (or upon starting the program in general), the O/S must guarantee that the duplicated pages have the same state. This is required for the R-stream to function properly.

2.2.3. Real time I/O support. The method of synchronizing the A-stream and R-stream may not support critical I/O applications in which real time constraints must be met. One solution is to include the synchronization delay in real time guarantees.

3. Performance evaluation

3.1. Simulation environment

A detailed, fully execution-driven simulator of a trace processor [17] was modified to support AR-SMT time redundancy. The simulator was developed using the *simplescalar* simulation platform [24]. This platform uses a MIPS-like instruction set and a gcc-based compiler to create binaries.

The simulator only measures performance of the microarchitecture. *Fault coverage is not evaluated.* It is beyond the scope of this paper to characterize transient faults in future microprocessors, and then develop and simulate a fault model based on this characterization. This is left for future work.

Trace processor hardware parameters are not tabulated here due to space limitations but can be found in [22][17]. The parameters most relevant to this paper are as follows.

- The maximum trace length is 16 instructions. Traces are terminated only at indirect jumps/calls and returns.
- Depending on the experiment, the trace processor consists of 4 or 8 PEs.
- Each PE can issue up to 4 instructions per cycle.
- Results are presented for only two threads sharing the trace processor, the A-stream and the R-stream.

Five of the SPEC95 integer benchmarks (Table 1) were simulated to completion.

TABLE 1. SPEC95 integer benchmarks used.

benchmark	input dataset	dynamic instruction count
compress	400000 e 2231	104 million
gcc	-O3 genrecog.i	117 million
go	9 9	133 million
ijpeg	vigo.ppm	166 million
xlisp	queens 7	202 million

3.2. Results

Two trace processor configurations were simulated, one with 4 processing elements and one with 8 processing elements. Figure 6 shows AR-SMT execution time *normalized with respect to the execution time of a single thread.* With 4 PEs, executing two redundant programs with AR-SMT takes only 12% to 29% longer than executing one program; with 8 PEs, it takes only 5% to 27% longer. This is much better than the 100% overhead of program-level time redundancy.

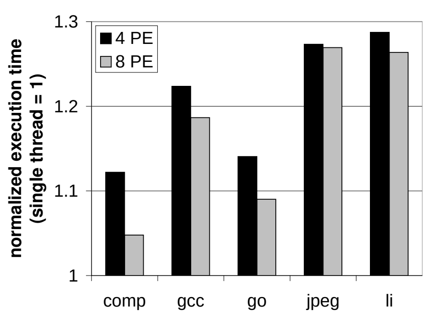

FIGURE 6. AR-SMT execution time normalized to the execution time of a single thread.

Two factors contribute to the good performance of AR-SMT. First, general purpose programs do not fully utilize the peak bandwidth of high performance processors. SMT exploits this by sharing processor bandwidth among multiple threads. Second, the R-stream requires a significantly lower fraction of the processor bandwidth because it executes faster than the A-stream -- the R-stream has the benefit of knowing in advance all control flow changes and data values.

This second point is clearly demonstrated in Figure 7, which shows the utilization of PEs by both the R-stream and A-stream, measured for the *gcc* benchmark running on an 8 PE trace processor. The graph shows the fraction of all cycles that 0 PEs are used, 1 PE is used, 2 PEs are used, etc. As expected, the R-stream utilizes much fewer PEs than the A-stream. Although the total number of traces executed by both streams is identical, R-stream traces are serviced much faster due to perfect control and data flow information. The vertical lines superimposed on the graph show average utilization for both streams. On average, only 1.5 processing elements are in use by the R-stream.

Notice the average utilizations do not add up to 8. This is because the A-stream "squashes" all traces after a mispredicted branch, and one or more PEs will be idle until they are dispatched new traces.

Two other conclusions can be drawn from Figure 6. First, the overhead of AR-SMT is greater for benchmarks that have higher instruction-level parallelism, namely *gcc*, *li*, and *jpeg*. These benchmarks have high trace prediction accuracy, and as a result the A-stream makes good utilization of the trace processor. Therefore, taking resources away from the A-stream has a larger impact relative to less predictable benchmarks (*go*, *compress*).

Second, the overhead of AR-SMT is always lower with 8 PEs than with 4 PEs. Adding more PEs yields diminishing returns for single thread performance, so the additional PEs can be used relatively uncontested by the R-stream. This is particularly true for benchmarks with poor trace prediction accuracy -- *compress* and *go* show a significant drop in AR-SMT overhead with 8 PEs. On the other hand, the A-streams in *li* and *jpeg* utilize 8 PEs and 4 PEs equally well.

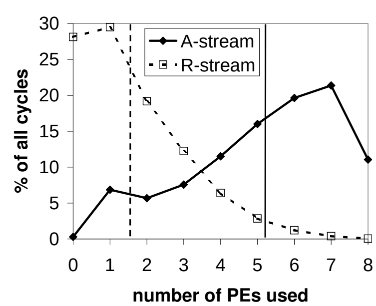

FIGURE 7. PE utilization (*gcc*, 8 PEs).

4. Summary

Both technology and microarchitecture trends have interesting implications for fault tolerance in future high performance microprocessors. On the one hand, technology-driven performance improvements potentially expose microprocessors to a new fault environment, one in which severely reduced tolerances result in frequent transient faults throughout the chip. On the other hand, microarchitecture trends can provide an overall solution for this new fault environment.

AR-SMT is the microarchitecture-based fault-tolerant solution put forth in this paper. Its development can be summarized as follows.

- AR-SMT is a time redundancy technique that combines the broad coverage of program-level time redundancy with the high performance and fast fault detection/recovery capability of instruction re-execution. It achieves this by creating two separate programs (like program-level redundancy) and running both programs simultaneously (like instruction re-execution).
- AR-SMT leverages three important microarchitecture trends -- advances that are likely to be implemented in future microprocessors for high performance and management of complexity. The primary mechanism, simultaneous multithreading, allows the active and redundant streams to co-exist within the processor and thus better utilize resources. Control flow and data flow prediction concepts are applied to speed execution of the redundant stream, and also exploit existing prediction-validation hardware for detecting faults. Hierarchical processors are organized around large, replicated processing elements; this coarse hardware redundancy is exploited to detect permanent faults and dynamically reconfigure the processor to work around the faults.
- Detailed simulations show that AR-SMT increases execution time by only 10% to 30% over a single thread. The low overhead is attributed to improved utilization of the highly parallel microprocessor and use of control/data flow information from the active thread to speed execution of the redundant thread.

Acknowledgements

Kewal Saluja is gratefully acknowledged for his constant encouragement.

References

[1] D. P. Siewiorek. Niche successes to ubiquitous invisibility: Fault-tolerant computing past, present, and future. *25th Fault-Tolerant Computing Symp.*, pages 26–33, June 1995.

[2] P. I. Rubinfeld. Virtual roundtable on the challenges and trends in processor design: Managing problems at high speeds. *Computer*, 31(1):47–48, Jan 1998.

[3] C. L. Chen and M. Y. Hsiao. Error-correcting codes for semiconductor memory applications: A state of the art review. In *Reliable Computer Systems - Design and Evaluation*, pages 771–786, Digital Press, 2nd edition, 1992.

[4] J. H. Patel and L. Y. Fung. Concurrent error detection in alu's by recomputing with shifted operands. *IEEE Trans. on Computers*, C-31(7):589–595, July 1982.

[5] G. Sohi, M. Franklin, and K. Saluja. A study of time-redundant fault tolerance techniques for high-performance pipelined computers. *19th Fault-Tolerant Computing Symp.*, pages 436–443, June 1989.

[6] N. K. Jha and J. A. Abraham. Techniques for efficient mos implementation of totally self-checking checkers. *15th Fault-Tolerant Computing Symp.*, pages 430–435, June 1985.

[7] N. Kanopoulos, D. Pantzartzis, and F. R. Bartram. Design of self-checking circuits using dcvs logic: A case study. *IEEE Trans. on Computers*, 41(7):891–896, July 1992.

[8] B. W. Johnson. Fault-tolerant microprocessor-based systems. *IEEE Micro*, pages 6–21, Dec 1984.

[9] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *Proc. IEEE*, 83(12):1609–24, Dec 1995.

[10] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *22nd Intl. Symp. on Computer Architecture*, pages 392–403, June 1995.

[11] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. *23rd Intl. Symp. on Computer Architecture*, pages 191–202, May 1996.

[12] J. E. Smith. A study of branch prediction strategies. *8th Symp. on Computer Architecture*, pages 135–148, May 1981.

[13] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. *19th Intl. Symp. on Computer Architecture*, May 1992.

[14] M. Lipasti. *Value Locality and Speculative Execution*. PhD thesis, Carnegie Mellon University, April 1997.

[15] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The performance potential of data dependence speculation and collapsing. *29th Intl. Symp. on Microarchitecture*, pages 238–247, Dec 1996.

[16] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. Technical Report 1080, Technion - Israel Institute of Technology, EE Dept., Nov 1996.

[17] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. *30th Intl. Symp. on Microarchitecture*, Dec 1997.

[18] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. *19th Intl. Symp. on Computer Architecture*, May 1992.

[19] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. *22nd Intl. Symp. on Computer Architecture*, pages 414–425, June 1995.

[20] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. *24th Intl. Symp. on Comp. Architecture*, pages 1–12, June 1997.

[21] J. Smith and S. Vajapeyam. Trace processors: Moving to fourth-generation microarchitectures. *IEEE Computer, Billion-Transistor Architectures*, Sep 1997.

[22] E. Rotenberg. Ar-smt: Coarse-grain time redundancy for high performance general purpose processors. *Univ. of Wisc. Course Project (ECE753), http://www.cs.wisc.edu/~ericro/course_projects/course_projects.html*, May 1998.

[23] N. Saxena and E. McCluskey. Dependable adaptive computing systems – the roar project. *Intl. Conf. on Systems, Man, and Cybernetics*, pages 2172–2177, Oct 1998.

[24] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar toolset. Technical Report CS-TR-96-1308, Univ. of Wisconsin, CS Dept., July 1996.