AR-SMT: Coarse-Grain Time Redundancy for High Performance General Purpose Processors

Eric Rotenberg

1.0 Introduction

Time redundancy is a fault tolerance technique in which a task -- either computation or communication -- is performed multiple times on the same hardware. This technique is cheaper than other fault tolerance solutions that require some form of hardware redundancy, because it does not *require* replicated hardware. However, fault coverage may be lower with time redundancy as it only captures certain classes of faults, and furthermore performance is degraded due to the repetition of tasks.

The purpose of this paper is to qualitatively study the fault coverage and performance impact of various time redundant techniques. The fault tolerant application is a high performance, single-chip uniprocessor running large general purpose programs. A primary contribution is articulating the concept of *granularity* of time redundancy -- how coarse or fine grained the redundant computation is. Time redundancy granularity suggests a spectrum of implementations with interesting tradeoffs among fault coverage, hardware coverage, performance, and design impact.

Using this framework, we propose a new time redundant technique called *Active-stream/Redundant-stream Simultaneous Multithreading*, or AR-SMT. The idea is to create a second *dynamic* instruction stream from the primary dynamic instruction stream as it retires; the two instruction streams simultaneously share the processor resources. The concept is similar to instruction re-execution [1, 2, 3], but the granularity of redundant computation is coarser.

1.1 Time redundancy spectrum

In this paper we consider schemes for executing instructions twice on the same processor in order to detect faults. At one extreme, the same program can be run twice back-to-back. Thus, with *program granularity* time redundancy, the unit of re-execution is an entire program. At the other extreme, a dynamic instruction can be executed twice before retiring from the instruction window. *Instruction granularity* time redundancy, for example instruction re-execution [1, 3], treats individual dynamic instructions as the unit of re-execution.

As shown in Figure 1, program granularity and instruction granularity represent two extremes in a spectrum of time redundant implementations. All share the property of creating two redundant dynamic instruction streams. Points in the spectrum differ in how the two redundant instruction streams are *interleaved*, i.e. the granularity of interleaving.

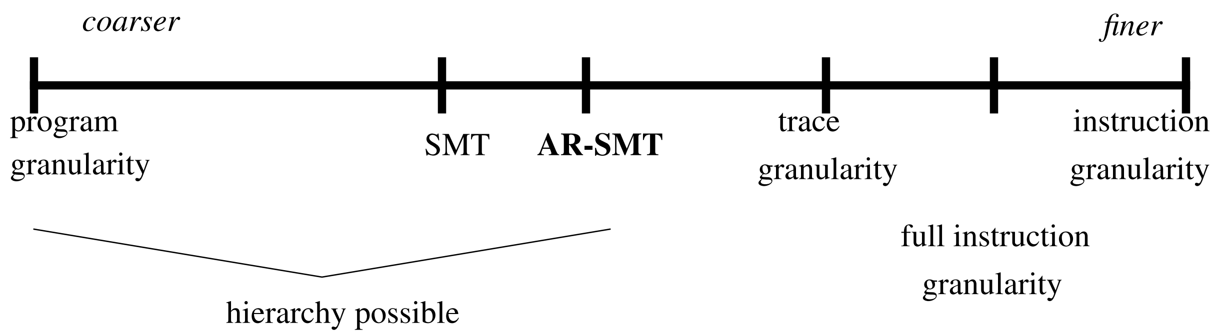


FIGURE 1. Time redundancy spectrum.

1.1.1 Instruction granularity

Instruction re-execution techniques were originally designed to detect faults in functional units, for example ALUs. Depending on the implementation, only certain types of faults will be detectable with time redundancy. Simple re-computation does not modify the input data of the computation and therefore covers only *transient faults* -- temporary, rather short-lived faults. A transient fault is detectable if it is active during only one of the computations. That is, during either the original or redundant computation, but not both [1].

By modifying the input data for the redundant computation, some subset of permanent faults will also be detectable, including long-lived transient faults that otherwise appear permanent. *Recomputing with Shifted Operands* (RESO), proposed by Patel and Fung [1], is one such technique. Although the inputs are modified, the function is preserving -- the original result can be derived from the modified result, allowing the two computations to be compared. As for fault detection, the basic idea is that a fault will manifest differently for different (e.g. shifted) inputs [4]. Thus, the operation is non-preserving in the presence of a fault.

Sohi, Franklin, and Saluja [3] applied RESO in the context of highly-pipelined processors with many functional units. They found that the hardware "redundancy" inherent in these processors can be exploited to improve the performance of time redundancy. Although each ALU operation issues twice, the overall performance degradation is typically less than 10% for the kernel benchmarks studied.

There are several reasons why time redundancy, if done at the instruction granularity, can provide high performance:

- The processor is not fully utilized in every cycle -- this is a result of true data dependences (instructions wait for values), control dependences (instruction fetch bottlenecks), and multiple pipelined functional units.
- Working at this granularity gives the scheduler very fine control of when to issue redundant instructions, allowing it to "scavenge" quite effectively for free cycles and idle ("redundant") resources that are already implemented as part of the microarchitecture.

- Not all instructions are candidates for re-execution; non-ALU instructions, in particular loads, stores, and branches, were not considered in [3].
- The highest performance implementations have the instruction duplication logic at the functional units. This performs better than duplicating at the instruction issue logic because issue bandwidth is a more critical resource.

Additional fault coverage can be obtained if the redundant computation is guaranteed to execute on a different functional unit than that of the original instruction. In fact, if this is enforced all of the time, then the data operands need not be modified to cover permanent faults or long transients (i.e. RESO is not needed). However, this requires the lower performance solution of duplicating the instruction at the issue stage; further, enforcing this policy will aggravate instruction stalls.

1.1.2 Program granularity

Running the same program twice back-to-back is the simplest form of time redundancy. It does not require any changes to the processor. However, it has two major drawbacks.

First, in general it can only detect transient faults. Nevertheless, in practice it may cover a significant number of permanent faults due to non-determinism in the computer -- caches, context switches, pseudo-random instruction scheduling policies, etc. This non-determinism will ultimately cause the redundant computations to pass through the processor differently.

Second, the performance of program granularity time redundancy is quite poor, at least two times the latency of executing the single program. Additional latency overhead will be incurred during the validation phase (comparing program outputs).

1.1.3 Qualitative comparison of fine and coarse time redundancy

In this section, the two extremes of the time redundancy spectrum are qualitatively compared. The comparison is not only interesting in itself, but also because it shows the factors to consider when evaluating any fault tolerance scheme. Further, the comparison points out strengths and weaknesses of both extremes, suggesting intermediate and hybrid approaches that combine the strengths.

Table 1 summarizes the comparison. The critical differences are highlighted with shading. Clearly, the performance of instruction granularity is superior. Unfortunately, this is at the expense of very limited hardware coverage: only functional units are covered, and simple ones at that (e.g. load/store buffers and cache ports are not covered), although other fault tolerance techniques can be applied throughout the processor.

Instruction granularity enables a small error latency and run-time recovery. Faults are detected almost immediately due to the close proximity (in time) of redundant computations, and checkpointed state already exists in the form of committed state (the faulting instruction is speculative).

TABLE 1. Comparison of instruction and program granularity.

fault tolerance factor	instruction granularity	program granularity
performance	good: about 10%	bad: about 100%
coverage		
hardware	bad: simple functional units only	good: all hardware
permanent faults	good: RESO, bad: simple re-execution	*unknown*
long-lived transients	good: RESO, bad: simple re-execution	good
short-lived transients	good	good
fault handling		
detection	good: small error latency	bad: large error latency
recovery	good: checkpoint = committed state	bad: re-run program
critical I/O (real time)	good: no impact	bad: unsupported
design impact	ok: RESO impacts the ALU (wider) good: simple re-execution has less impact	good: none

1.2 AR-SMT time redundancy

In this section we present a new implementation of time redundancy that lies somewhere between program and instruction granularity. A high-level view of AR-SMT is shown in Figure 2.

The *active stream* (A-stream) is the first instance of the program. This dynamic instruction stream is created normally as it is a true program context. The A-stream is fetched and *dispatched* into the instruction window of the processor, after which execution proceeds in an out-of-order and parallel fashion. Instructions from the A-stream are *retired* in order, i.e. state is committed in a precise fashion.

A second, *redundant stream* (R-stream) is dynamically created from the A-stream. As A-stream instructions are retired, summary information about the instructions are pushed into a FIFO queue called the Delay Buffer (i.e. a *form* of instruction duplication is performed at the retire stage). When it fills, summary information is popped from the Delay Buffer and used to fetch/dispatch the R-stream into the processor.

The Delay Buffer allows for the creation of a redundant context that, as far as the processor is concerned, is totally separate from the active context. In this respect, AR-SMT resembles program granularity. It is coarse-grained and consequently stresses all parts of the processor, including caches and all phases of instruction processing (good hardware coverage). Further, the Delay Buffer is designed such that long-lived transients are detectable: sufficient delay is inserted between the A-stream and R-stream.

However, unlike program granularity, AR-SMT interleaves the two contexts via *simultaneous multithreading* (SMT) [5]. In this model, the two streams space-share the processor. SMT leverages the parallelism provided by modern superscalar processors. Often there are phases of a single program that do not fully utilize the parallel resources. Thus, sharing the processor resources among multiple programs will increase overall utilization, despite slowing down single thread performance.

In this respect, AR-SMT resembles instruction re-execution. AR-SMT performs well because of high processor utilization, and because control and data flow information stored in the Delay Buffer speeds execution of the R-stream. Yet AR-SMT will likely perform worse than instruction re-execution because all phases of instruction processing are duplicated, from fetch through retirement.

As will be described later, aggressive forms of AR-SMT retain the fault detection and recovery model of instruction re-execution. Further, information may be stored in the Delay Buffer to guarantee R-stream instructions execute on different functional units the second time around. This increases coverage of permanent faults.

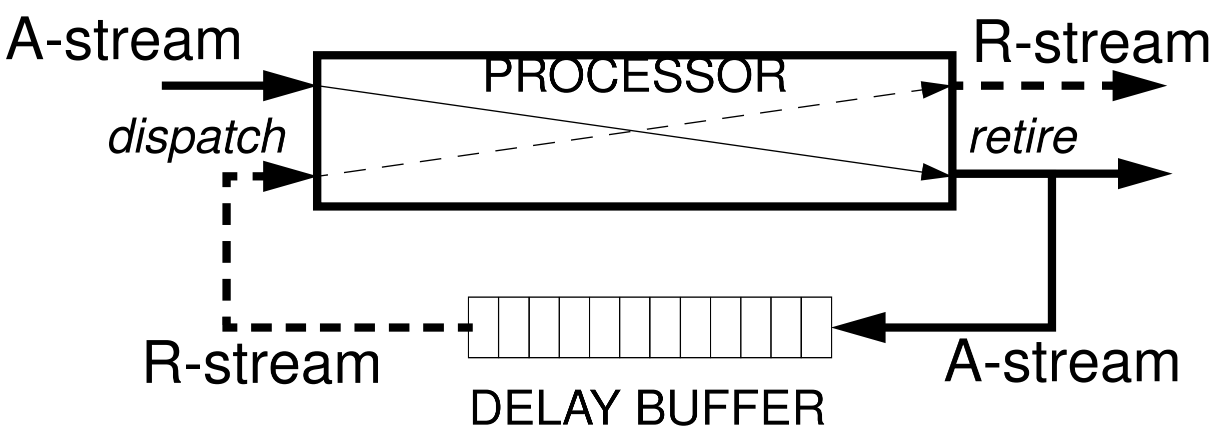


FIGURE 2. High-level view of AR-SMT time redundancy.

1.3 Other implementations

Other implementations in the time redundancy spectrum include full instruction re-execution, trace re-execution, pure SMT, and hierarchical time redundancy (Refer to Figure 1).

Full instruction re-execution replicates an instruction at all phases of the pipeline, so that more of the processor is covered. The same concept can be applied in trace processors (described later), but at the trace granularity. A pure SMT implementation extends AR-SMT into the software realm -- the operating system explicitly creates the R-stream. In this case, there is no Delay Buffer and no guaranteed delay between the redundant streams.

Finally, instruction re-execution can be implemented on top of AR-SMT to achieve hierarchical time redundancy. More generally, fine-grain and coarse-grain time redundancy can be combined.

1.4 Paper organization

The paper is organized as follows. In Section 2.0, an AR-SMT design is presented, covering the microarchitecture, the Delay Buffer, and operating system issues. Section 3.0 qualitatively describes the fault tolerance aspects of AR-SMT, including fault coverage, hardware coverage, performance, and design impact. In Section 4.0, trace processors are presented as a candidate for supporting both the general SMT and AR-SMT models. The performance of AR-SMT implemented in a trace processor is evaluated in Section 5.0.

2.0 AR-SMT microarchitecture

In this section details about the AR-SMT microarchitecture are presented. The discussion focuses on two important aspects of any SMT machine: (1) sharing critical processor resources (namely fetch, dispatch, issue/execute, and retire bandwidth) and (2) separating register and memory state for multiple contexts.

In addition, alternative Delay Buffer designs are discussed, including techniques to speed up the processing of the R-stream while at the same time providing an aggressive fault detection/recovery model.

2.1 Implementing SMT

Most of the design is derived from work on simultaneous multithreaded machines [5]. This is beneficial for several reasons. The techniques are well established and understood. Recent research shows that SMT can be incorporated into existing superscalar processors rather seamlessly. This work also shows that SMT flexibly exploits both fine (intra-thread) and coarse (inter-thread) parallelism. As a result, overall processor utilization improves as does overall performance.

All of these reasons lead us to believe that SMT will be incorporated in upcoming generations of wide-issue processors. Processor developers will continue to push for higher levels of instruction-level parallelism, and it is only natural to leverage this additional parallelism in as many ways possible.

2.1.1 Concerning instruction fetch for the R-stream

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

AR-SMT simplifies this by storing instruction PCs of retired A-stream instructions in the Delay Buffer. In fact, *it is this sequence of instruction PCs that dynamically creates the R-stream context*. Therefore, the Delay Buffer **at a minimum** contains the program counter values that drive instruction fetching for the R-stream. This reduces complexity because the branch prediction hardware remains dedicated to the A-stream (especially important for complex path-based control predictors [6,7]). In a sense, the Delay Buffer serves as the branch predictor for the R-stream.

2.1.2 Sharing processor bandwidth

A dynamic instruction passes through various stages of processing. Instruction fetch and dispatch bring the instruction into the instruction window. Before reaching the out-of-order issue engine, instructions are decoded, register dependences are established among the newly fetched instructions and those instructions in the window, and architectural registers are renamed to the larger physical register file.

Once dispatched into the instruction window, instructions wait for register operands to be broadcast on result buses (if they are not already in the physical register file). When all register operands are available, the instruction issues to a functional unit where it is executed. Cache ports are considered functional units as are ALUs. When execution completes, the instruction broadcasts its result on a result bus to wakeup dependent instructions and write values into the register file.

Completed instructions wait until they reach the head of the reorder buffer, at which time register and/or memory state may be committed, i.e. retired. At this point the instruction is no longer speculative (in terms of control and/or data speculation) and no prior instructions have caused an exception (precise exceptions).

In AR-SMT, the A-stream and R-stream can either time-share or space-share a given resource. I have decided on the following partitioning scheme.

- *Instruction fetch and dispatch*: Instruction fetch bandwidth is a very critical resource in ILP processors. As a result, many techniques aimed at improving instruction fetch bandwidth have made the frontend of processors complex. Therefore, I have chosen to time-share the instruction fetch engine between the A-stream and R-stream. This is where AR-SMT diverges from recent SMT proposals [5].

  This decision is also justified with the advent of low latency, high bandwidth instruction fetch mechanisms such as trace caches [8,9,10,11]. Although instruction fetch is multiplexed between the two streams, when a given stream does access the fetch unit, it receives a large group of instructions.

  Instruction fetch and dispatch are really part of the same pipeline, and so dispatch is treated similarly. That is, the entire frontend pipeline, from instruction fetch through dispatch, is time-shared. This simplifies the design considerably since the two streams arbitrate at a single point.

- *Instruction execution*: All execution resources, including the instruction issue buffers, issue bandwidth, functional units (including cache ports), and result buses are space-shared between the A-stream and R-stream. Thus, instructions from separate contexts co-exist in the instruction window simultaneously. However, the instruction issue logic is essentially unaware of multiple contexts due to the SMT register rename mechanism (described in the next section); this transparency is one of the attractive implementation features of SMT.

- *Instruction retirement*: Retirement is the dual of dispatch in that register state is committed -- physical registers are returned to the register freelist and register maps are also freed. Therefore, retirement, like dispatch, is time-shared between the A-stream and R-stream.

Scheduling algorithm for dispatch and retirement

In SMT, there is flexibility and choice in the allocation of resources to multiple threads. However, AR-SMT is a more restricted form of SMT in that the A-stream and R-stream are linked together via the Delay Buffer. This linkage reduces the number of choices that are otherwise an integral part of scheduling dispatch and retirement.

The scheduling rules are simple.

1. Dispatch: if the Delay Buffer is full, R-stream has priority.
2. Retire: if the Delay Buffer is not full, A-stream has priority.

More complex policies are possible if the Delay Buffer length is considered flexible. For example, rule 1 can be modified such that the R-stream is considered for dispatch if its length reaches a certain threshold, where the threshold is less than the full buffer length.

There are several policies that take advantage of this flexibility. One such policy estimates the confidence of branch predictions in the A-stream, and dispatches more R-stream instructions if the confidence is low [12,13,14].

2.1.3 Handling register values

Separate register spaces must be simultaneously maintained for the two instruction streams. AR-SMT uses the approach proposed in [5]. In this approach, a single large physical register file is used, as in a conventional processor. However, the existing register renaming mechanism can be used such that this single file holds the state of multiple contexts. The size of the physical register file must be increased accordingly, but no new datapaths are created because there is still only a single register file.

Register renaming ensures that multiple writes to the same *architectural register*, as specified by static instructions, are bound to distinct *physical registers* so as not to interfere. This is exactly the desired effect for distinguishing writes to the same architectural register in two different programs.

Register rename maps provide the mapping from architectural to physical registers. Multiple maps are required in processors employing speculation and precise interrupts; these maps provide checkpoints for misspeculation or exception rollback. The new support required for SMT is these maps must be space-shared among multiple contexts. Also, one additional rename map is required per context. (AR-SMT adds only one map table.)

The real benefit of this approach is that once instructions pass through renaming, the fact that there are multiple contexts is transparent to the instruction issue logic. The issue logic is based on physical names, and SMT renaming ensures a single name space.

2.1.4 Handling memory values

The memory disambiguation hardware -- responsible for buffering speculative store values and enforcing load-store dependences --is space-shared in AR-SMT. Memory dependences from different contexts must not interfere with each other. Thus, memory addresses must be augmented with a *context identifier* (assuming disambiguation is based on virtual addresses). The same virtual address used by two different contexts is distinguishable using the context id. In the case of AR-SMT, only a 1-bit id is required.

2.2 New issues and O/S support

AR-SMT introduces new problems 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 is unaware of it. Yet how does the R-stream maintain a separate physical memory image from the A-stream? What happens if the A-stream is context-switched out of the processor by the operating system, or if there are exceptions or synchronous traps to the operating system for such things as I/O?

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 we created 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, we maintain 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 translation mechanism for the R-stream; addresses are translated by taking the original translation and adding 1 to it.

There are other solutions. For example, if the A/R-stream delay is sufficiently small, the processor state may be retired only by the R-stream. That is, both streams share the same register and memory state (like instruction re-execution), which implies the state cannot be committed until the R-stream completes. This solution does not scale well. It places too much pressure on the physical register file and speculative store buffering, since it essentially increases the instruction window size many-fold (delays retirement).

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- and R-streams. 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 can essentially be terminated. Now only the A-stream is serviced, swapped out, etc., which is necessary if the operating system has no knowledge of the redundant context.

When resuming after a context switch (or upon starting the program in general), the duplicated pages must have the same state for the R-stream to function properly.

There remains the issue of traps and exceptions caused by the R-stream. The A-stream should reach synchronous traps/exceptions first, and so those in the R-stream can be ignored; after catching up, the R-stream is terminated.

2.2.3 Real time support (I/O)

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. The only solution is to include the synchronization delay in real time guarantees.

2.3 Delay buffer

In this section the contents of the Delay Buffer is discussed. Control flow information from the A-stream is used to create the R-stream, and other information may improve the performance and fault tolerance features of AR-SMT.

2.3.1 Control flow

As mentioned in Section 2.1.1, control flow information derived from the retired A-stream is placed in the Delay Buffer. This drives instruction fetching for the R-stream. Stated another way, this control flow information provides the branch "prediction" mechanism for the R-stream.

Notice that this branch prediction mechanism, under non-fault conditions, should be perfect. This will considerably speed up the processing of the R-stream, reducing the performance degradation of coarse-grain time redundancy. If the R-stream executes a branch and the outcome differs from the "prediction", there must be a fault either in the R-stream or the A-stream. (Additionally, the fault may have occurred in the Delay Buffer itself.)

This raises the question: what if data flow information is placed in the Delay Buffer as well? The ramifications are discussed in the following sections.

2.3.2 Register values

If the values of retired source registers, destination registers, or both are placed in the Delay Buffer, R-stream processing is further sped up by virtue of perfect value "prediction" [15,.16,17]. Instructions need not wait for data dependences to be resolved before issuing. Results of the "prediction" are validated after the fact as values are computed; the validation will fail only in the presence of a fault in the A-stream, R-stream, or Delay Buffer.

2.3.3 Load and store addresses

If the addresses of retired loads and stores are placed in the Delay Buffer, R-stream processing is further sped up by virtue of oracle memory disambiguation. As R-stream loads and stores are dispatched, the “predicted” addresses are used to synchronize stores with dependent loads, not unlike register dependency analysis. In this way, loads issue as early as possible because true memory dependences are known in advance. On top of that, loads may issue earlier simply because the addresses are available.

Again, computed addresses are used to validate the address “predictions” and the optimistic disambiguation mechanism. The validation will fail only in the presence of a fault in the A-stream, R-stream, or Delay Buffer.

2.3.4 Concerning error latency and fault recovery

Another benefit of including comprehensive control flow and data flow information is it provides a good fault recovery model, similar to that of instruction re-execution. In particular, storing values gives the R-stream something to compare against. If the comparison fails, a fault has occurred and the R-stream committed state serves as a checkpoint.

2.3.5 Improving permanent fault coverage

Partial information about where an instruction was initially processed (e.g. functional unit, processing element in a trace processor, result bus, cache port, etc.) can be used to ensure the redundant computation is routed to different resources. This may improve the coverage of permanent faults.

3.0 Characterizing AR-SMT fault tolerance

AR-SMT combines the advantages of instruction granularity and program granularity time redundancy. The following discussion summarizes the fault tolerant qualities of AR-SMT. Refer also to Table 2.

TABLE 2. Summary of AR-SMT fault tolerant qualities.

fault tolerance factor	AR-SMT
performance	good: about 5-25%
coverage	
hardware	good: all hardware
permanent faults	*unknown* to good: can guarantee different execution paths
long-lived transients	good: if Delay Buffer is sufficiently large
short-lived transients	good
fault handling	
detection	good: small error latency (related to Delay Buffer)
recovery	good: checkpoint = R-stream committed state
critical I/O (real time)	good: little impact (factor in maximum delay)
design impact	ok: if implementing on top of an SMT machine; cost = Delay Buffer bad: if non-SMT machine; cost = register file and maps, Delay Buffer

The performance of AR-SMT will be much better than that of program granularity. This is because SMT better utilizes the processor, and because the R-stream has perfect control prediction and perhaps perfect address disambiguation and value prediction. It will likely perform worse than instruction granularity, because in that model only instruction execution is duplicated. Performance of AR-SMT is evaluated in Section 5.0.

As with program granularity, AR-SMT stresses much of the hardware since it implements coarse-grained time redundancy. With sufficient information in the Delay Buffer, AR-SMT may achieve good permanent fault coverage by routing redundant computations along different execution paths. The delay introduced by interleaving the A- and R-streams may provide coverage of long latency transients, without the need for the explicit re-routing.

If comprehensive control and data flow information is passed along in the Delay Buffer, the R-stream can quickly detect faults via state comparisons. Further, the R-stream provides the checkpointed state that may be used by a trap handler to initiate recovery.

If an SMT machine is used as a platform for AR-SMT, clearly the design impact, in terms of cost and complexity, is minimal. However, augmenting a non-SMT machine requires increasing the size of the register file (which has cycle time implications), adding an extra register rename map, and designing the control logic and datapaths for SMT. All of this is in addition to the Delay Buffer storage[1].

4.0 Trace processors as a platform for AR-SMT

In this paper we use a new processor microarchitecture called *trace processors* [18,19,20,21] as a platform for AR-SMT.

A *trace* is a long, dynamic sequence of instructions captured and stored by hardware. It may contain any number of control transfer instructions. The primary constraint on a trace is a hardware-determined maximum length, but there may be any number of other implementation-dependent constraints.

The microarchitecture, shown in Figure 3, is completely organized around traces. Trace processors exploit control flow and data flow hierarchy to overcome complexity and architectural limitations of conventional superscalar processors by (1) distributing execution resources based on trace boundaries and (2) applying control and data prediction at the trace level rather than individual branches or instructions.

1. The Delay Buffer can use very dense storage cells. The cells are specialized shift register elements that do not require the area-consuming read/write ports of standard SRAM cells.

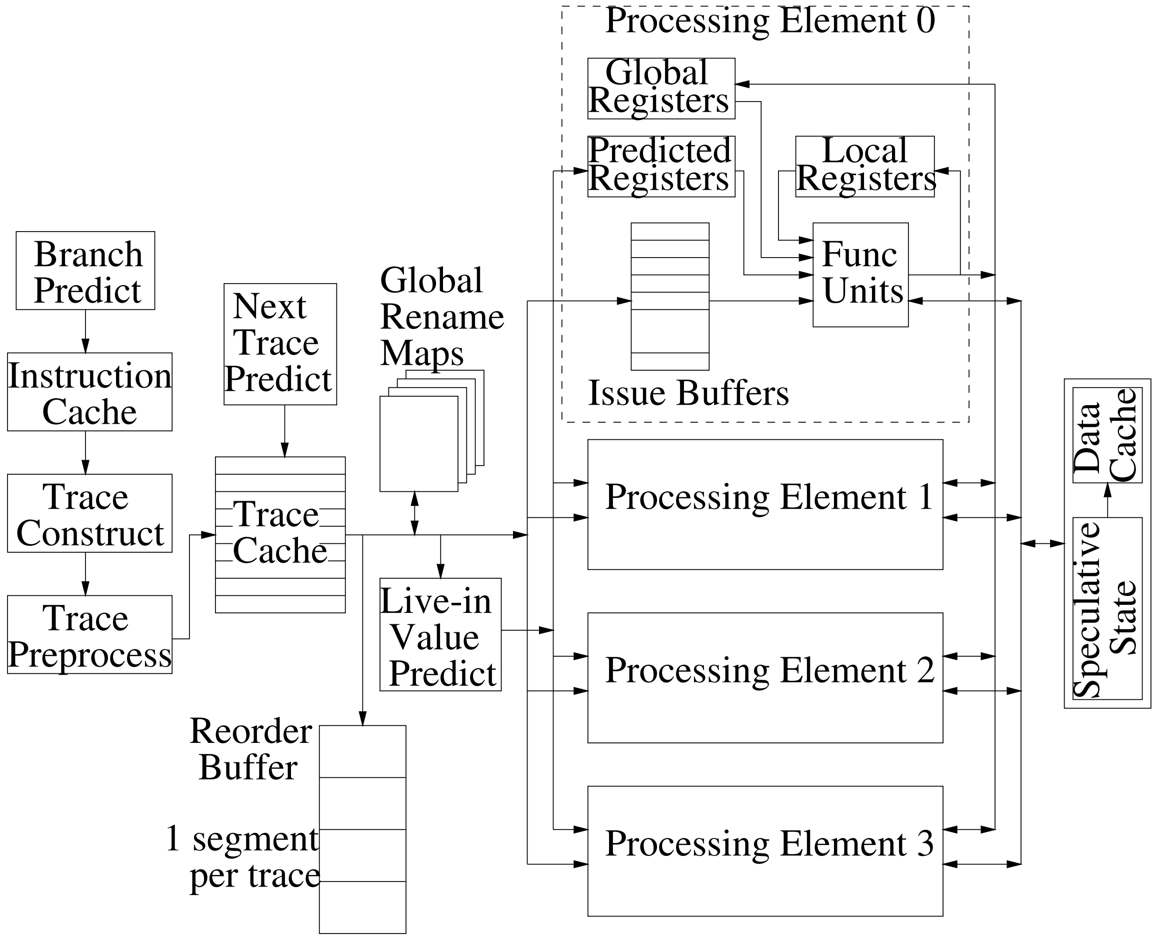


FIGURE 3. Trace processor microarchitecture (AR-SMT support not shown).

Although it is beyond the scope of this paper to fully describe or advocate trace processors, we argue that this approach lends itself naturally to both general SMT and AR-SMT.

- *Highly parallel*: Trace processors are designed to exploit large levels of instruction-level parallelism. As a result, the processor has significant inherent hardware redundancy.
- *Hierarchical, distributed processor*: The instruction window is physically partitioned into multiple processing elements (PEs), each with enough instruction buffers to hold an entire trace.

  This physical partitioning makes it conceptually simpler to implement SMT -- resources can be allocated to threads at the coarser granularity of PEs/traces. For example, managing 8 PEs is much simpler than managing 128 individual instruction buffers.
- *Hierarchical register file*: A trace uses and produces values that are *live-in* (produced by a previous trace), entirely *local* (produced and consumed solely within a trace), or *live-out* (consumed by subsequent traces). The result is a hierarchical register file: a dedicated local register file per PE/trace to hold local values, and a single global file for holding values live between traces (the global file is replicated for read bandwidth).

Because locals are not stored in the global physical register file, its size is reduced; the benefit is even greater for SMT machines in which the physical register file must be made larger to accommodate multiple contexts.

- *Replication*: Just as instruction re-execution can exploit replicated functional units, so can AR-SMT exploit replicated PEs. However, the advantage of PEs over functional units is that they are significantly more complex -- a lot more hardware is encapsulated. Specifically, a PE contains register files, dedicated functional units, local result buses, load/store buffers, and instruction buffers. Thus much more hardware is covered by “redundant” PEs than by “redundant” functional units.
- *Wide instruction fetch/dispatch*: The frontend fetches and dispatches a trace per cycle. Further, a trace can be retired every cycle. This is very high bandwidth, and a good match for the time-multiplexed dispatch/retirement model described in Section 2.1.2.

5.0 Performance evaluation

5.1 Simulation environment

A detailed, fully execution-driven simulator of a trace processor [21] was modified to support AR-SMT time redundancy. The simulator was developed using the *simplescalar* simulation platform [22]. This platform uses a MIPS-like instruction set (no delayed branches) and comes with a gcc-based compiler to create binaries.

AR-SMT was implemented as specified in Section 2.0. As in a real design, the R-stream compares its computed values against Delay Buffer values. This not only faithfully models a real implementation of AR-SMT, but it also serves to validate correctness of the timing simulator.

Trace processor parameters are shown in Table 3. Benchmarks, input datasets, and dynamic instruction counts are shown in Table 4.

TABLE 3. Trace processor parameters.

frontend latency	2 cycles (fetch + dispatch)
trace predictor	DOLC predictor table size = 2^{16} entries augmented with hybrid pred. and RHS
value predictor	*turned off*
trace cache	size/assoc/repl = 128kB(instr only)/8-way/LRU total traces = 2048 trace line size = **16 instructions** trace selection = stop at jump/call indirects and returns
branch predictor	predictor = 64k 2-bit sat counters BTB = 16k entries, dir map, no tags, 1-bit hyst.
instruction cache	size/assoc/repl = 128kB/4-way/LRU line size = 16 instructions 2-way interleaved miss penalty = 12 cycles
global physical registers	unlimited
functional units	*n* symmetric, fully-pipelined FUs (for *n*-way issue)
memory	unlimited speculative store buffering D$ size/assoc/repl = 64kB/4-way/LRU D$ line size = 64 bytes D$ miss penalty = 14 cycles D$ MSHRs = unlimited outstanding misses
execution latencies	address generation = 1 cycle memory access = 2 cycles (hit) integer ALU operations = 1 cycle complex operations = MIPS R10000 latencies validation latency = 1 cycle
number of PEs	4 or 8
execution bandwidths	issue bandwidth per PE = 4-way out-of-order issue local result buses = 4 global result buses = 4 # global buses that can be used by a PE = 4 # cache buses = 4 # cache buses that can be used by a PE = 4

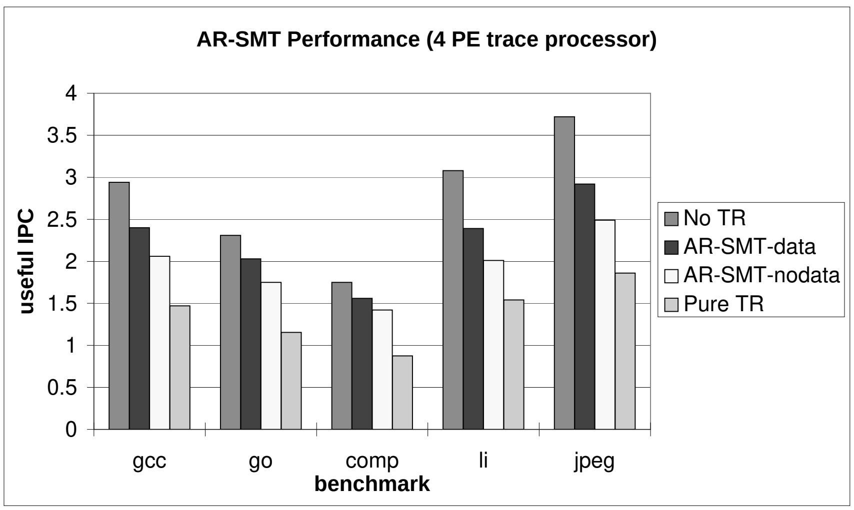
TABLE 4. SPEC95 integer benchmarks used.

benchmark	input dataset	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

5.2 Results

Two trace processor configurations were simulated, one with 4 processing elements and one with 8 processing elements. In each case, three runs were made:

- **No TR**: No time redundancy is modeled (i.e. only the A-stream executes).
- **Pure TR**: This is program granularity time redundancy. This was not actually simulated, but rather the useful IPC is computed as one half that of *No TR*.
- **AR-SMT-data**: This is AR-SMT with both control flow and data flow information passed in the Delay Buffer.
- **AR-SMT-nodata**: This is AR-SMT with only control flow information passed in the Delay Buffer. Therefore this model does not benefit from oracle memory disambiguation, perfect address prediction, or perfect value prediction.



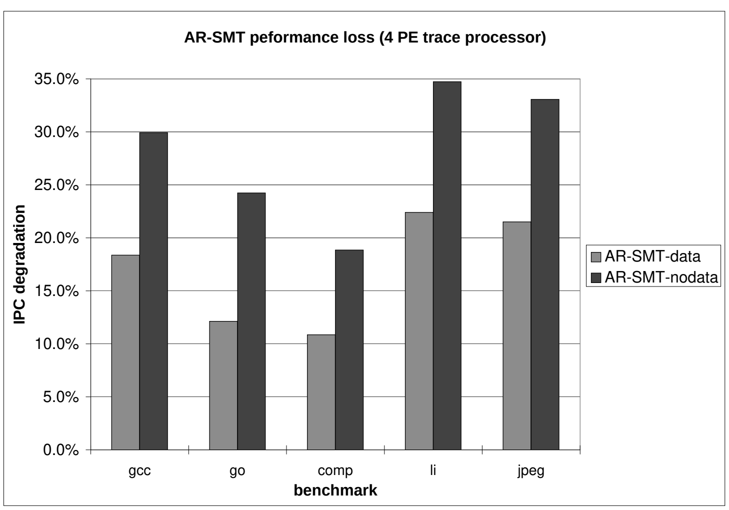
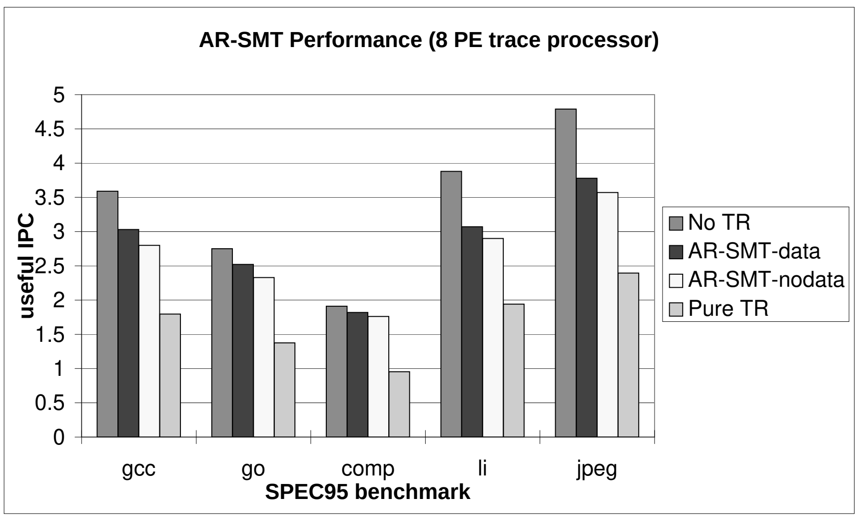


FIGURE 4. AR-SMT performance, 4 PE trace processor.

Figure 4 shows the results for 4 PEs. IPC is shown in the first graph, and the IPC degradation incurred by the AR-SMT models (relative to *No TR*) is shown in the second graph. AR-SMT clearly outperforms program granularity. This is because SMT better utilizes the processor resources, and because AR-SMT has at least perfect control prediction to speed up the R-stream. At the time of this writing, we were unable to isolate just the SMT contribution to performance.

AR-SMT-data degrades performance 10-20%. This increases to 20-35% without data flow information in the Delay Buffer. We conclude perfect value prediction and other data flow information has a significant performance impact when the number of PEs is low or moderate. This is because with a low number of PEs, there is significant contention between A- and R-stream, and speeding the processing of the R-stream has a correspondingly large impact.

The performance degradation is higher for the more parallel benchmarks (*gcc*, *li*, and *jpeg*). These benchmarks have good trace prediction accuracy, and as a result the A-stream makes good utilization of the trace processor; taking resources away from the A-stream has a large impact relative to less predictable benchmarks (*go*, *compress*).



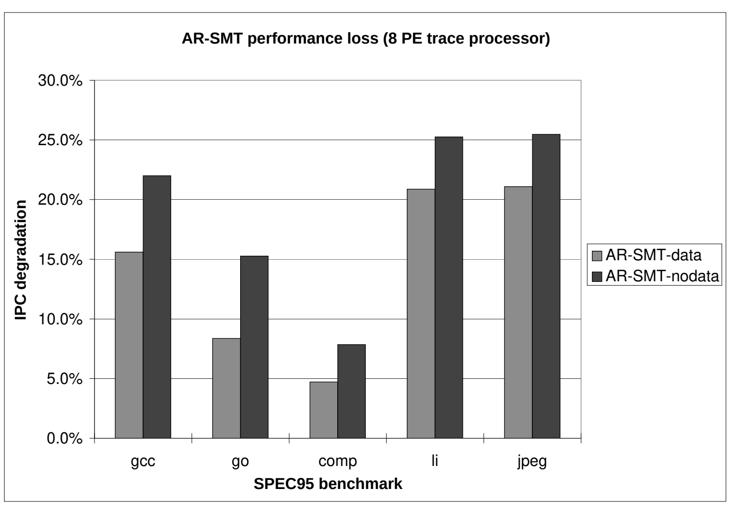


FIGURE 5. AR-SMT performance, 8 PE trace processor.

Figure 5 shows the results for 8 PEs. As expected, AR-SMT still outperforms program granularity. In fact, the performance gap is even larger for 8 PEs because adding PEs yields diminishing returns for single-threaded parallelism.

This is also why AR-SMT performance degradation is less with 8 PEs -- having 4 more PEs reduces contention between the two streams. AR-SMT-data degrades performance 5-20%. This increases to 8-25% without data flow information in the Delay Buffer. Notice perfect value prediction and other data flow information has less performance impact when the number of PEs is large. Having sufficient PE resources appears to compensate for not having data flow information. Also, the fact that global result buses and cache buses have not been added to the 8 PE trace processor may limit the gains of perfect value prediction ("*hurry up and wait* for a result bus").

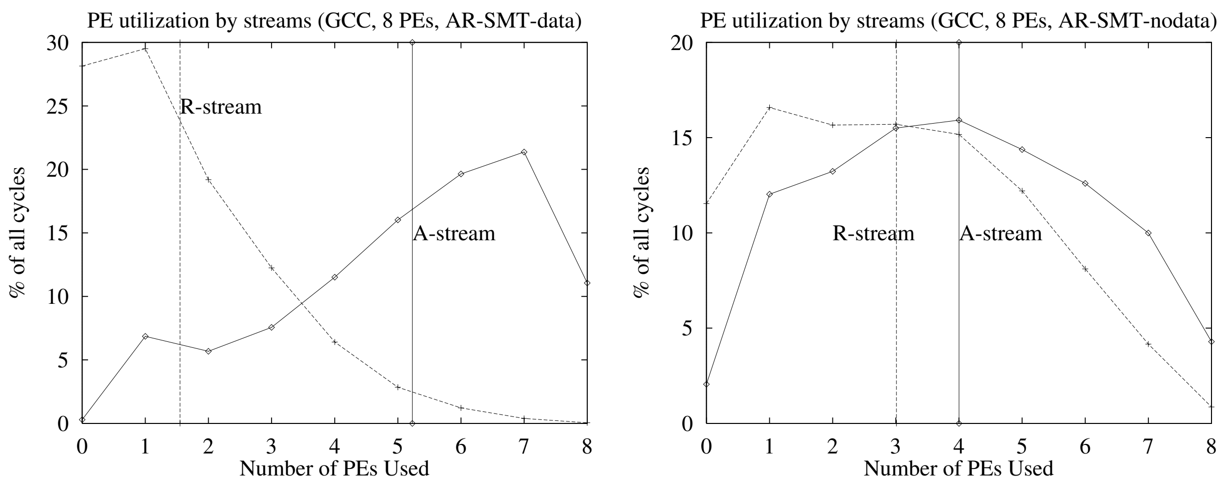


FIGURE 6. PE utilization by A- and R-streams, both with and without data flow information in the Delay Buffer. The benchmark is *gcc*, and the trace processor has 8 PEs.

Figure 6 shows the utilization of PEs by both the R-stream and A-stream (for *gcc*). The data is presented in the form of a histogram: the fraction of all cycles that 0 PEs are used, 1 PE is used, 2 PEs are used, etc. The first graph is for AR-SMT-data. 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. On average, only 1.5 processing elements are in use by the R-stream (indicated on the graph with a vertical line).

The second graph shows that without data flow information in the Delay Buffer, R-stream utilization doubles to 3 PEs on average. It is still lower than A-stream utilization due to perfect control prediction. Notice the average utilizations do not add up to 8; this is because the A-stream has control squashes, consequently some cycles experience idle PEs.

6.0 Summary

In this paper we introduced the concept of *granularity* of time redundancy. The concept suggests a spectrum of time redundant implementations, with instruction re-execution at one extreme (fine-grain) and program granularity at the other (coarse-grain). We qualitatively compared the advantages and disadvantages of the two extremes, in terms of fault coverage, hardware coverage, performance, and design impact.

This study motivated a new coarse-grain time redundant mechanism called AR-SMT. It retains the hardware coverage advantages of program granularity by creating two separate, full instructions streams, but the two instruction streams are interleaved in time to retain the performance advantages and recovery model of instruction re-execution.

A design for AR-SMT was presented. It leverages prior work in simultaneous multithreaded processors. It was argued that trace processors are amenable to the SMT and AR-SMT models, consequently this microarchitecture was chosen as a framework for implementing and evaluating AR-SMT. Detailed simulations of five of the SPEC95 integer benchmarks shows AR-SMT degrades performance by only 5-25% for an 8 PE trace processor.

7.0 Future work

The most important item that must be addressed before proceeding further with this research is **fault coverage** of AR-SMT. In particular, AR-SMT is much more viable if we can show that it covers many more faults throughout the processor, compared to other time redundant techniques.

Once fault coverage is established, there are many interesting things to be looked at.

- *Simulate instruction re-execution.* It would be nice to compare instruction re-execution and AR-SMT in the same environment (i.e. same processor model and benchmarks).
- *Design other time redundant mechanisms.* In particular, hierarchical redundancy, i.e. instruction re-execution on top of AR-SMT. Also, full instruction re-execution, trace re-execution, etc.
- *Better A/R-stream scheduling.* I feel that with a flexible Delay Buffer length and branch confidence estimation for throttling the A-stream, we can improve AR-SMT performance significantly. Also, I have to study the scheduling policy with respect to global result buses and cache ports, both critical resources.
- *Memory state.* The hardware/software mechanisms for establishing the separate memory context of the R-stream have to be defined more precisely and shown to be sufficient.
- *Nasty issues.* Exceptions, interrupts, traps, I/O, etc.
- *Better permanent fault coverage.* Implement mechanisms that guarantee redundant traces are routed through different execution paths. For example, guarantee routing to a different PE, and measure the performance impact if any.
- *Partial data flow information.* What happens to performance and the fault recovery model if not all values are placed in the Delay Buffer, but only live-ins of traces?
- *Isolate SMT contribution to performance.* How much does SMT-ness contribute to AR-SMT performance, versus the Delay Buffer information for speeding up the R-stream? This question is difficult to answer because it requires implementing pure SMT, and there are many scheduling options to be explored.

- *Trace processor re-configuration.* If faults persist in a PE, re-configure the dispatch mechanism to skip that PE. Note that global result buses and other shared resources can be similarly re-configured given sufficient run-time information.
- *Fault situation in the future.* Characterize faults in future high performance processors, considering the insane densities, the relatively longer wires, the widespread use of dynamic logic in places where static logic currently rules, the insane clock rates, the insane noise margins, and in general the much higher susceptibility to transient faults in random logic such as delay faults and crosstalk. Here is one reference on transient fault characterization [23].

8.0 References

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

[2] J. H. Patel and L. Y. Fung. Concurrent error detection in multiply and divide arrays. *IEEE Transactions on Computers*, C-32(4):417–422, April 1983.

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

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

[5] 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.

[6] Q. Jacobson, E. Rotenberg, and J. Smith. Path-based next trace prediction. *30th Intl. Symp. on Microarchitecture*, Dec 1997.

[7] Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith. Control flow speculation in multiscalar processors. *3rd Intl. Symp. on High Perf. Computer Architecture*, pages 218–229, Feb 1997.

[8] J. Johnson. Expansion caches for superscalar processors. Technical Report CSL-TR-94-630, Computer Science Laboratory, Stanford, CA, June 1994.

[9] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. U.S. Patent Number 5,381,533, Jan 1995.

[10] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. *29th Intl. Symp. on Microarchitecture*, pages 24–34, Dec 1996.

[11] S. Patel, D. Friendly, and Y. Patt. Critical issues regarding the trace cache fetch mechanism. Technical Report CSE-TR-335-97, University of Michigan, EECS Department, 1997.

[12] E. Jacobsen, E. Rotenberg, and J. Smith. Assigning confidence to conditional branch predictions. *29th Intl. Symp. on Microarchitecture*, pages 142–152, Dec 1996.

[13] G. Tyson, K. Lick, and M. Farrens. Limited dual path execution. Technical Report CSE-TR-346-97, University of Michigan, EECS Department, 1997.

[14] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. *25th Intl. Symp. on Computer Architecture*, June 1998.

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

[16] Y. Sazeides and J. Smith. The predictability of data values. *30th Intl. Symp. on Microarchitecture*, Dec 1997.

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

[18] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin, Nov 1993.

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

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

[21] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. *30th Intl. Symp. on Microarchitecture*, pages 138–148, Dec 1997.

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

[23] X. Castillo, S. McConnel, and D. Siewiorek. Derivation and calibration of a transient error reliability model. *IEEE Transactions on Computers*, C-31(7):658–671, July 1982.