Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection

Cheng Wang, Ho-seop Kim, Youfeng Wu, Victor Ying

Programming Systems Lab
Microprocessor Technology Labs
Intel Corporation
{cheng.c.wang,ho-seop.kim,youfeng.wu,victor.ying}@intel.com

Abstract

As transistors become increasingly smaller and faster with tighter noise margins, modern processors are becoming increasingly more susceptible to transient hardware faults. Existing Hardware-based Redundant Multi-Threading (HRMT) approaches rely mostly on special-purpose hardware to replicate the program into redundant execution threads and compare their computation results. In this paper, we present a Software-based Redundant Multi-Threading (SRMT) approach for transient fault detection. Our SRMT technique uses compiler to automatically generate redundant threads so they can run on general-purpose chip multi-processors (CMPs). We exploit high-level program information available at compile time to optimize data communication between redundant threads. Furthermore, our software-based technique provides flexible program execution environment where the legacy binary codes and the reliability-enhanced codes can co-exist in a mix-and-match fashion, depending on the desired level of reliability and software compatibility. Our experimental results show that compiler analysis and optimization techniques can reduce data communication requirement by up to 88% of HRMT. With general-purpose intra-chip communication mechanisms in CMP machine, SRMT overhead can be as low as 19%. Moreover, SRMT technique achieves error coverage rates of 99.98% and 99.6% for SPEC CPU2000 integer and floating-point benchmarks, respectively. These results demonstrate the competitiveness of SRMT to HRMT approaches.

1. Introduction

Transient Faults, also known as Soft-Errors or Single-Event Upsets (SEUs), are intermittent faults that do not occur consistently. These faults are often caused by external events such as neutron and alpha particles striking the chip or internal noise on power supply and interconnections [3]. Although these faults do not cause permanent hardware damage, they may result in incorrect program execution.

As the silicon integration technology advances, transistors are becoming increasingly smaller and faster. As a result, noise margin is getting tighter, which in turn makes processors more susceptible to soft-errors [14]. Recent studies [14] show that, in the near future, soft-error rate in combinational logic will be comparable to that of memory elements, and protecting the entire chip, instead of only the memory elements, will be among the designers' top considerations.

Several Hardware-based Redundant Multi-Threading (HRMT) approaches [6][9][12][16] have been proposed to detect and recover transient faults. The basic idea of HRMT approach is to use specialized hardware to replicate an application execution thread into two redundant ones at run time, the leading thread and the trailing thread. The trailing thread repeats computations performed by the leading thread, and values produced by these two threads are compared for error detection and recovery. Required special hardware support in these approaches, however, usually increases hardware complexity and cost.

As chip multi-processors (CMPs) become ubiquitous, using the readily available processor cores for enhancing reliability becomes feasible. In this paper, we present a Software-based Redundant Multi-Threading (SRMT) approach for replicating an application thread into two and comparing their results for transient fault detection. This software-based approach leverages multi-core resources available in CMPs to coordinate the replicated execution. As such, our SRMT approach not only avoids the hardware design and validation complexities of the HRMT approaches, but also explores source code level

International Symposium on Code Generation and Optimization (CGO'07)
0-7695-2764-7/07 $20.00

information via sophisticated compiler analysis to reduce error checking overhead.

Moreover, SRMT is flexible: applications enhanced with SRMT and regular applications without SRMT can run simultaneously on the same CMP system for different reliability and performance requirements. Our SRMT implementation also supports combining SRMT code and existing non-SRMT binary code (OS provided libraries, for example) in the same application, which makes it possible to dynamically adjust the application reliability and performance trade-off based on run-time information and user selectable policies.

Note that we cannot simply run the same application twice and compare the results for transient fault detection. The application may have permanent updates to system storage, such as data base and memory mapped I/O. There may also be non-deterministic behavior such that different runs of the same program produce different executions. In general, simply running the program twice is not suitable for error detection. Our approach deterministically replicates the application thread execution through compiler analysis and transformation, and hence, is different from process-level redundant execution [10], which tries to run an application in two separate processes and synchronize the two processes at system call boundaries. Non-determinism caused by program bugs such as uninitialized variables, or data racing on shared memory accesses in multithreaded applications such as spin-locks, may prevent the two processes from seeing the same sequence of system calls, and thus the process-level redundant execution may result in false positive error reports.

This paper makes the following contributions.

- We believe that this is the first attempt to use a second core for software-based redundant execution and guarantee no false positive errors even in the presence of non-determinism caused by data racing in multi-threaded applications.
- Our SRMT technique uses compiler analysis and optimizations to filter out data references that do not need communication between leading and trailing threads and identify variables whose side effects are harmless in the event of an error. These optimizations can reduce the communication demand from 5.2 bytes per cycle in HRMT [6] to about 0.61 bytes per cycle in SRMT (an 88% reduction).
- We demonstrate a novel technique to integrate SRMT code with non-SRMT binary code in the same application. This is important for applications where parts of them are not sensitive to faults or have built-in fault tolerance features, or are library code available only in binary form.
- For general purpose chip multi-processors with an intra-processor communication queue, our experiment shows that the SRMT approach incurs only about 19% overhead. This is competitive to more complicated HRMT approaches [12].
- Currently, our SRMT implementation achieves error coverage rates of 99.98% and 99.6% for SPEC CPU2000 integer and floating-pointer programs, respectively, which are again competitive with the hardware and other software approaches.

The rest of the paper is organized as follows. Section 2 overviews the related work. Section 3 describes compiler transformations for SRMT. Section 4 discusses the run-time thread communications in SRMT. Section 5 presents the experimental results. Section 6 concludes the paper and discusses future work.

2. Related Works

Chip-level Redundant Threading (CRT) [9] and Chip-level Redundantly Threaded multi-processor with Recovery (CRTR) [6] are proposals for transient fault detection and recovery, respectively, based on chip multi-processors. These techniques rely mostly on special purpose hardware to replicate the program into redundant execution and compare their results.

Simultaneous and Redundantly Threaded (SRT) processors [12] and Simultaneously and Redundantly Threaded processors with Recovery (SRTR) [16] are proposals for transient fault detection and recovery, respectively, based on Simultaneous Multi-Threading (SMT) processors [23]. Again, they rely on special hardware extensions.

Recently, several lightweight HRMT approaches [25] [26] [27][28] have been proposed to duplicate only a subset of the dynamic instruction streams at the cost of possibly lower error detection and recovery rate. The idea behind these "partial redundant threading" approaches is to improve the overall cost-effectiveness to make reliable computing practical. We believe that the cost-effectiveness can be further improved with software or software-hardware hybrid approaches like SRMT.

There are other types of software-based fault tolerance schemes. Instruction-level fault tolerance techniques [2][13], such as SWIFT [17], provide transient fault tolerance by duplicating program execution at the instruction granularity within the same thread. SWIFT targets Itanium Processor Family (IPF) processors where the large number of general-purpose registers (GPRs) permits relatively small overhead for instruction-level redundant execution. The scenario is

Table 1. Comparison among Different Fault Tolerance Approaches

Issues	SRT/SRTR	CRT/CRTR	Instruction-level redundancy	Process–level redundancy	Thread-level redundancy (SRMT)
Special hardware	Yes	Yes	No	No	No
Limited by single processor resource	Yes	No	Yes	No	No
False positive due to non-determinism	No	No	No	Yes	No

very different for the IA-32 architecture, however, which only has 8 GPRs. Based on our experiments, and also confirmed by experimental results in [13], instruction-level redundancy on IA-32 incurs much higher overhead than on register-rich architectures such as IPF. Our SRMT approach targets for IA-32 and explores CMP capabilities so that its overhead is not limited by single processor resources.

Process-level fault tolerance techniques, such as Somersault [10], replicate the application in two independent processes, and rely on a special software layer to synchronize system calls between the two processes. Somersault assumes that the two processes will issue the same system calls to the software layer, which then executes those system calls only in one process and duplicates return values to the two processes. However, many programs have non-deterministic behaviors and therefore two processes may not be able to generate exactly the same sequences of system calls and hence may report false positive errors.

As a summary, we compare different fault tolerance approaches in Table 1. SRMT is the only approach that does not require specialized hardware, is not limited by a single processor resource, and can handle non-deterministic events correctly.

3. Compiler Transformation for SRMT

Like HRMT approaches, our SRMT approach replicates original program execution into two communicating threads, the leading thread and the trailing thread. The leading thread performs all operations in the original program with additional operations to communicate with the trailing thread. The trailing thread transparently replicates computations of the leading thread and compares its results with those from the leading thread to detect transient faults. For correctness, our compiler treats the leading thread as the original thread in the program and the trailing thread as a helper thread which only helps to detect transient fault but never affects the correctness of the program execution.

The fault model in our SRMT approach can be defined in term of Sphere of Replication (SOR) [12]. The operations within SOR are replicated to run in both threads and the operations out of SOR run only in the leading thread. For simplicity, we also call the operations within SOR as repeatable operations and operations out of SOR as non-repeatable operations.

In our SRMT approach, we define the following non-repeatable operations:

System calls for I/O operations: I/O operations must be outside SOR due to their non-repeatable external effects (e.g. a user may see the same output string twice on the monitor if we replicate the output system calls).

Shared memory access operations: In some applications (for example, multithreaded applications), data racing on shared memory access may lead to non-deterministic application behaviors. Therefore, in our SRMT approach, we also put the shared memory access operations outside SOR. As with other approaches [2] [6] [9] [12] [13] [16] [17], we rely on existing storage array protection mechanisms, such as ECC, for transient fault detection in shared memory.

For the SRMT scheme to work, we need to address the following issues: 1) what needs to be duplicated for redundancy; 2) what needs to be checked for transient fault detection; 3) how to efficiently provide fail-stop property when a fault is detected; 4) how to generate efficient code for the leading and trailing threads and integrate binary functions. All these issues are addressed below with compiler optimizations to resolve them efficiently.

3.1 Values Duplicated for Redundancy

In the model of SOR, all the values come into SOR need to be duplicated. Therefore, we need duplicate the following types of values in the trailing thread:

Return values of shared memory load operations: These values come into SOR through shared memory loads.

Return values of system calls: These values come into SOR through system calls.

To duplicate the same value in the trailing thread, the leading thread needs send the value to the trailing thread. Figure 1 shows an example of a shared memory load operation, where the trailing thread uses the value r1 sent from the leading thread, rather than repeating the operation.

# original code	# leading thread	# trailing thread
… ld r1, [mem1]	… ld r1, [mem1] **send r1**- - - - - - - ▶	… **receive r1**
// use r1	// use r1	// use r1

Figure 1. Redundant computation of shared memory load operations

Global variables may be accessed by different threads. Therefore, loading and storing a global variable should be treated as a shared memory access. On the other hand, operations that operate on non-address-taken local variables can be duplicated since they operate only on non-shared data which could be either in registers or on local stack, depending on whether or not there are enough registers to hold them.

# original code	# leading thread	# trailing thread

Foo() {
// x is shared
int x;
// p is non-shared
int *p;
 	
p = &x;
 	
use x;
 	
use p
}
	Leading_foo() {
	
	int x;
	
	int *p;
	
	send &x;
	p = &x;
	
	send x;
	use x
	
	use p
	}
		Trailing_foo() {
		
		// no duplicate x
		// duplicate p
		int *p;
		
		receive &x;
		p = &x;
		
		receive x;
		use x
		
		// directly use p
		use p
		}

Figure 2. Transformation of local variables

For local variables that are accessed through pointers (address-taken local variables), however, we still have to treat them as shared memory because different threads may access them through pointers. For example, a parent thread may allocate data in its own stack and shares the data with child threads by passing the data pointer to child threads. In our SRMT approach, we allocate only a single copy of shared local data in the leading thread's stack and send the loaded value to the trailing thread if it is needed there. Furthermore, the trailing thread may need the address of the local variable for its computation. In this case, the leading thread needs to pass the address to the trailing thread. Figure 2 shows an example of a shared local variable x and a non-shared local variable p.

3.2 Values Checked for Error Detection

In the model of SOR, all the values go out of SOR need to be checked. Therefore, we need check the following types of values:

Addresses of shared memory load operations: These values go out of SOR through shared memory loads.

Addresses of shared memory store operations: These values go out of SOR through shared memory stores.

Values to be stored into shared memory: These values go out of SOR through shared memory stores.

Parameters passed to system calls: These values go out of SOR through system calls.

To check a value for transient fault, the leading thread sends the value to the trailing thread. The trailing thread computes its own value and compares it with the value from the leading thread. If a mismatch is found, an error is reported.

Figure 3 shows examples of error checking for a load address, a store address, and its store value. The operation "check x and y" is a shorthand for "if (x != y) signal error". Note that, unlike HRMT approaches, the store operation in this example is performed in the leading thread even if the trailing thread signals an error. We will address the fail-stop issue in next section.

# original code	# leading thread	# trailing thread

//compute
mem1
...
ld r1, [mem1]
 	
//compute r2
//compute
mem2
…
st r2, [mem2]
	//compute mem1
	…
	send mem1
	ld r1, [mem1]
	
	send r1
	//compute r2
	//compute mem2
	...
	send mem2
	
	send r2
	st r2, [mem2]
		//compute mem1
		…
		receive mem1'
		check mem1 and mem1'
		receive r1
		//compute r2
		//compute mem2
		…
		receive mem2'
		check mem2 and mem2'
		receive r2'
		check r2 and r2'

Figure 3. Values checked by the trailing thread

3.3 Fail-stop Optimizations

Fail-stop is an important requirement for all error detection techniques. When an error is detected, the program must stop the operations that may have adverse side effect to the outside world, such as changing critical data on disk or sending message to the network.

For an operation, e.g. a store, that must stop when its operands are affected by a fault, the leading thread has to wait for an acknowledgement from the trailing thread indicating that the operation is not affected by a

fault (see Figure 4). The wait operation is expensive as the leading thread cannot make forward progress while the acknowledgement travels from the trailing thread to the leading thread. We could let the trailing thread perform the store after it performs the error checking. Still, other operations in the leading thread may potentially be aliased with this store and have to wait for an acknowledgement from the trailing thread before they can proceed. In this subsection, we focus on compiler optimization to minimize the number of memory operations that are needed to fulfill the fail-stop requirements.

# original code	# leading thread	# trailing thread
//compute mem1	//compute mem1	// compute mem1
…	…	…
ld r1, [mem1]	send mem1	receive mem1'
		check mem1 and mem1'
	wait(ack) ◄- - - - -	**signal**(ack)
	ld r1, [mem1]	
	send r1	receive r1
// compute r2	//compute r2	//compute r2
// compute mem2	//compute mem2	//compute mem2
…	…	…
st r2, [mem2]	send mem2	receive mem2'
	send r2	receive r2
		check mem2 and mem2'
		check r2 and r2'
	wait(ack)◄- - - - -	**signal**(ack)
	st r2, [mem2]	

Figure 4. Acknowledgements for fail-stop

In order to minimize the number of memory operations that must be fail-stop, we note that compiler is allowed to move a regular store above a regular load operation, as long as data dependence is not violated. When the load operation generates a run-time error, e.g. via null pointer checking or array bound checking, however, the store might have already completed. In this case, the store is not fail-stop with respect to the original program order before the compiler optimizations. Similarly, our SRMT compiler does not need to guarantee fail-stop for regular loads/stores, which gives compiler more freedom in optimizations. For volatile loads/stores and shared stores, on the other hand, compiler is not allowed to move them across other memory operations. Note that some volatile variables are memory-mapped I/O ports, and both load and store to such a variable have externally visible side effects and therefore they must be made fail-stop. Similarly, some shared-memory locations are memory-mapped files, and fail-stop is needed to prevent the side effect from getting into the files.

Consequently in our SRMT implementation, we use variable attributes available to compiler to identify volatile and shared variables, and only generate acknowledgements for them. Since volatile and shared variables account for only a small portion of all variables, the acknowledgement overhead does not affect overall performance much. We believe this represents a significant advantage of our compiler-based approach over hardware and binary tool based approaches, where high-level language information is not available.

Figure 4 shows an example code with acknowledgements for fail-stop support, assuming [mem1] and [mem2] are volatile memory locations. We only need a single semaphore variable named as "ack" for the fail-stop synchronizations from the trailing thread to the corresponding leading thread.

In summary, we have the following types of variables:

Repeatable operations: operations that use registers or thread local stack variables whose addresses are not taken and used globally. These operations can be executed in both threads.

Non-repeatable non-fail-stop operations: operations that access global variables or stack locations whose addresses are taken and used globally. These operations are executed in the leading thread, without waiting for acknowledgement from the trailing thread.

Non-repeatable fail-stop operations: operations that access volatile and shared variables. These operations are executed in the leading thread, and must wait for acknowledgement from the trailing thread before they are performed.

Repeatable operations are the cheapest ones and our SRMT compiler tries to discover as many of them as possible, utilizing such optimizations as register promotion and partial redundancy elimination [8]. Non-repeatable fail-stop operations are the most expensive ones, but fortunately they are not used frequently in most applications.

Since we relax the fail-stop requirements for some memory operations, a soft-error may cause a program to generate an exception in the leading thread even if the error would be detected in the trailing thread. In our SRMT framework, compiler can install signal handlers to detect various exceptions (segmentation faults, illegal instruction, divide by zero, etc.). Any such error detected by a handler has the same benefit as being detected by the trailing thread: preventing silent data corruption (SDC) from happening.

3.4 Code Generation and Interaction with Binary Functions

Since we cannot duplicate non-repeatable operations in the trailing thread, it is inefficient for the leading

thread and the trailing thread to use the same code: we would have to put guarding conditions in the code for the two threads of execution, which would incur extra run-time overhead. In our SRMT approach, we use compiler to generate two differently specialized versions of code, the LEADING version for the leading thread and the TRAILING version for the trailing thread. Although this approach improves performance, it also poses challenges.

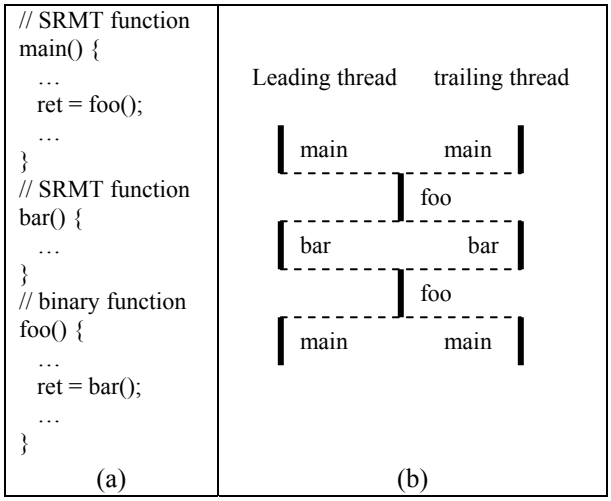

Figure 5. A scenario of SRMT code calling a binary function which calls back to an SRMT function

In applications compiled by our SMRT compiler, a LEADING version function often makes calls to LEADING version functions and a TRAILING version function to TRAILING version functions. However, the application may also call functions that must run in a single thread. This situation arises when part of the application is not sensitive to faults (e.g. playing DVD), is already hand-coded with fault-tolerance features [4], or is third party library code without source for recompilation (we refer these parts of the application *binary functions* in this paper). Furthermore, a binary function may also call back an SRMT function. Since the binary function can only be executed by the leading thread, the trailing thread will pause when a binary function is entered and resume when the function returns or calls back an SRMT function. This scenario is illustrated in Figure 5 (b).

Calling a binary function from an SRMT function is relatively easy to handle, as we may simply treat the binary function call as a non-repeatable operation. The leading thread calls the binary function and sends the return results to the trailing thread. The trailing thread uses the result without calling the binary function.

Handling a call back from a binary function to an SRMT function is more involved. For a function compiled with the SRMT compiler, we need an EXTERN version, a wrapper, of the function in addition to the LEADING version and the TRAILING version of the function. The EXTERN version has the same prototype as the original function so it can be directly called by a binary function. When an EXTERN

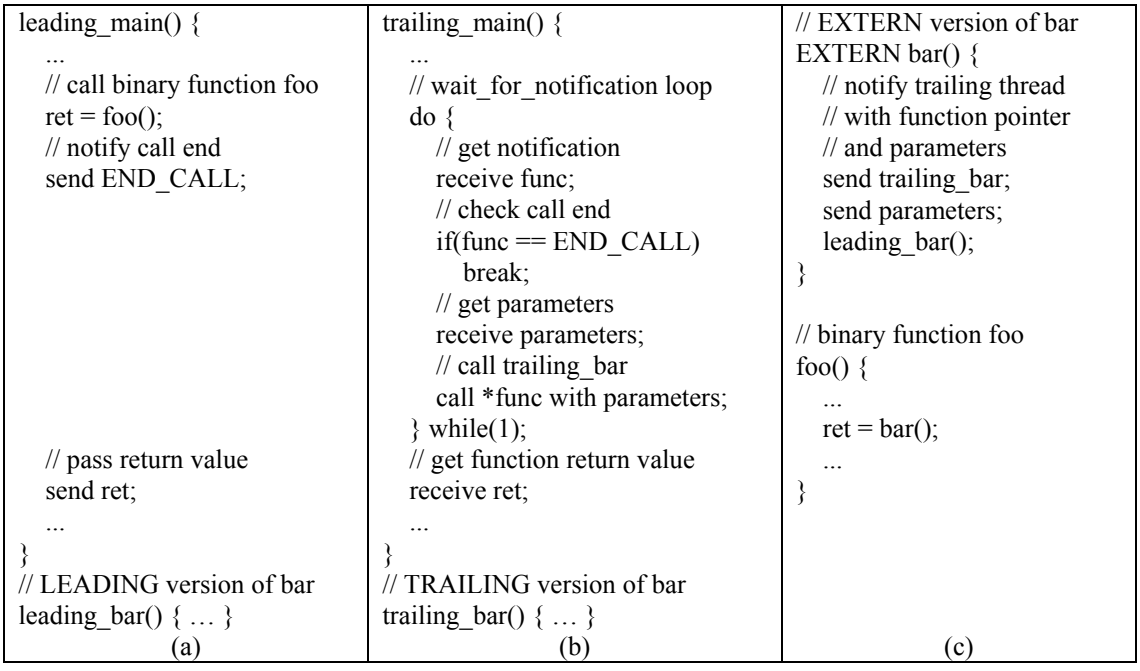

Figure 6. Code generation for handling call to and call back from a binary function

function is called by a binary function, it not only executes the leading version of the SRMT function, but also requests the trailing thread to execute the trailing version of the SRMT function.

Figure 6 shows the SRMT code capable of handling call-to and call-back from binary functions for the test case shown in Figure 5 (a). *leading_main* calls binary function *foo*. It sends the *END_CALL* and the return value *ret* to the trailing thread after the function call returns (Figure 6 (a)). *trailing_main* waits in a wait_for_notification loop until it receives *END_CALL* (Figure 6 (b)). When the binary function *foo* calls back SRMT function *bar*, the EXTERN version of function *bar* notifies the trailing thread by sending the corresponding function pointer *trailing_bar* and the corresponding parameters to the trailing thread so that the trailing thread can correctly make a call to *trailing_bar* with the function pointer and parameters (Figure 6 (c)).

The above scheme also works for function calls through pointers. A function pointer points to either an EXTERN version of an SRMT function or a binary function. The SRMT compiler generates code as if the indirect call is to a binary function (e.g. the code similar to those in Figure 6 (a) and (b)). If the callee function turns out to be an SRMT function, its EXTERN function will be called, which in turn calls the leading function and notifies the trailing thread to execute the trailing function.

In the above, we assume a callee function always returns normally. For the special non-return or abnormal-return functions such as *setjmp*, *longjmp* and *exit*, we provide our special versions of these functions so they can be called by either an SRMT function or a binary function.

Figure 7 shows our special leading (a), trailing (b), and EXTERN (c) versions of setjmp and longjmp, where _setjmp and _longjmp represent the inlined common bodies of the setjmp and longjmp, respectively. The key observation is that the leading and trailing versions of the setjmp/longjmp should use different environments so that the leading longjmp jumps to the environment (*env*) set by the leading setjmp, and the trailing longjmp jumps to the environment (*new_env*) set by the trailing setjmp. We maintain a hash table in the trailing thread for mapping the environments between the leading thread and the trailing thread. The *hash_alloc* function allocates a new entry in the hash table and the *hash_lookup* function searches the hash table for an entry. When the leading thread calls a binary function, say *foo*, which in turn calls an EXTERN version of setjmp or longjmp, the EXTERN version of setjmp or longjmp notifies the trailing thread to run the TRAILING version of these functions and also performs the setjmp or longjmp operations for the leading thread. Notice that, when the EXTERN version of setjmp/longjmp is called, the trailing thread must execute the wait_for_notification loop at the call site of the binary function *foo* (see Figure 6 (b))

4. Run-time Thread Communication

The main challenge to our SRMT approach is the

```
// LEADING version
leading_setjmp(env) {

  send env;
  _setjmp(env);
}



leading_longjmp(env) {

  send env;
  _longjmp(env);
}
```

(a)

```
// TRAILING version
trailing_setjmp(dummy) {
  receive env;
  hash_alloc(env, new_env);
  _setjmp(new_env);
}



trailing_longjmp(dummy) {
  receive env;
  new_env = hash_lookup(env);
  _longjmp(new_env);
}
```

(b)

```
// EXTERN version
setjmp(env) {
  // notify trailing thread to run trailing_setjmp
  send trailing_setjmp
  send dummy

  send env
  _setjmp(env);
}

longjmp(env) {
  // notify trailing thread to run trailing_longjmp
  send trailing_longjmp
  send dummy

  send env
  _longjmp(env);
}
```

(c)

Figure 7. SRMT version of setjmp and longjmp

communication overhead between the leading and trailing threads. In this section, we present a software queue technique to reduce the communication overhead on traditional shared-memory multi-processors (SMP) and explore chip multi-processor (CMP) supports, such as shared cache and communication queues to reduce the communication overhead.

4.1 Software Queue

For existing multi-processor systems, SRMT may use a circular software queue for data communications between the leading thread and the trailing thread. Specifically, the "send" operation can be implemented with an "*enqueue*" operation and the "receive" with a "*dequeue*". If implemented naively, however, those *enqueue* and *dequeue* operations can cause excessive amount of cache coherence traffic between the leading thread's cache and the trailing tread's cache.

```
enqueue(data) {
  buffer[tail_DB] = data;
  tail_DB = (tail_DB + 1)
          % QUEUE_SIZE;
  if(tail_DB % UNIT == 0) {
    while(tail_DB==head_LS)
      head_LS = head;
    tail = tail_DB;
  }
}
```

```
dequeue() {
  if(head_DB % UNIT == 0) {
    head = head_DB;
    while(head_DB== tail_LS)
      tail_LS = tail;
  }
  data = buffer[head_DB];
  head_DB = (head_DB + 1)
          % QUEUE_SIZE;
  return data;
}
```

Figure 8 Optimized software queue

We implemented two optimizations, Delayed Buffering (DB) and Lazy Synchronization (LS), to increase communication granularity and reduce synchronization overhead [24]. Figure 8 shows the implementation. With DB technique, the leading thread buffers enqueued data (through the use of *head_DB* and *tail_DB*) and communicates to the trailing thread only when enough (*UNIT*) data have been accumulated. The LS technique does not check the shared synchronization variables (e.g. the queue *head* and *tail* variables) for each *enqueue*/*dequeue* operation. Instead, we keep local copies of the shared variables (in *head_LS* and *tail_LS*) and trigger the communication of shared synchronization variables in a lazy way such that the accesses to shared variables are minimized. The data communication through the shared memory between processors leads to cache misses in the cache coherence protocol. With a simple Word Counter (WC) program, Our DB and LS techniques together reduce 83.2% L1 cache misses and 96% L2 cache misses.

Although the optimized software queue implementation is efficient enough for applications with coarse-grained communications, it is still quite heavyweight for those fine-grained communication patterns in SRMT. There are two sources of overheads with software queue in SRMT: Firstly, queue manipulation for each load/store requires many instructions; this leads to significant instruction count expansion. Secondly, the producer-consumer communication between two caches traversing the cache hierarchy is very expensive, as will be shown in the evaluation section.

4.2 Explore CMP hardware features for fast thread communications

Faster communication hardware in emerging chip multi-processor systems may help reduce SRMT communication overhead. For example, if the leading thread and the trailing thread can access a shared on-chip cache, rather than going out all the way to the shared main memory and back, our software circular queue implementation will result in much less coherence overhead. For another example, if the leading thread and the trailing thread can communicate through a hardware queue, both operations to manipulate the software queue and coherence overhead can be significantly reduced. A number of recent studies have

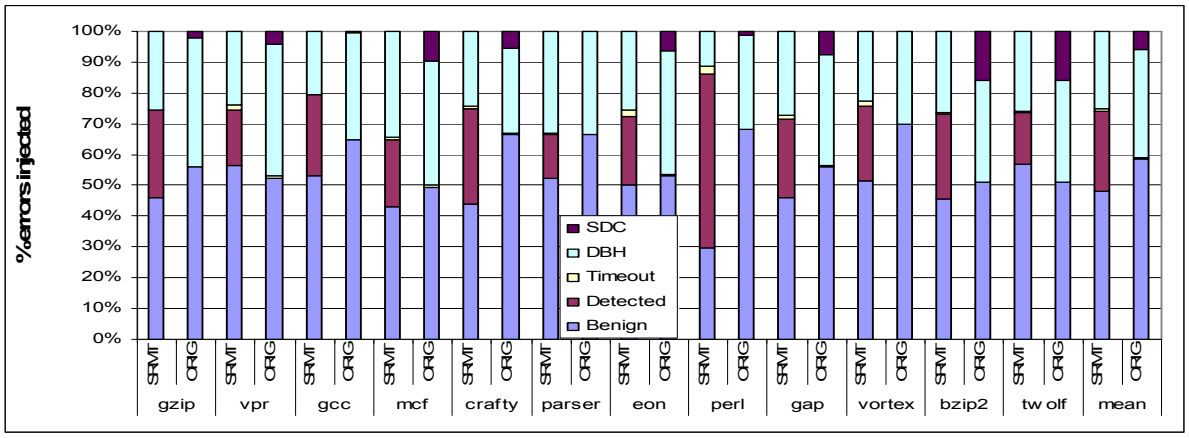

Figure 9. Fault injection distributions for SPEC2000 Integer Benchmarks

been advocating such communication queues for intra-chip core-to-core communications [11][15][19][29]. Note that such hardware is widely useful in any applications, not just SRMT, where more direct control over producer-consumer communication pattern is desirable. We experiment such general-purpose hardware support and present our experimental result in the next section.

5. Experiments

We implemented our SRMT approach in a research version of the Intel production compiler, ICC 9.0, and experimented with SPEC CPU2000 benchmark programs. We first measure the error coverage rate and then report the performance impact of SRMT-based error detection. For simplicity, we compile only the source code of the benchmarks with SRMT compiler; we treat all legacy library codes, including all system calls, as binary functions and execute with one thread. All performance measurements on real machines are run with the reference input set. The reduced input set from MinneSPEC [7] was used for simulator-based experiments and error coverage collection. We used an internal CMP simulator for this study.

5.1 Error coverage

To evaluate reliability of the applications generated by the SRMT compiler, we use a program instrumentation tool, PIN [18] to randomly inject one single bit of fault in one of application registers. Although in reality an error may occur in various places of the processor pipeline such as internal buffers and bypasses, in most cases the error will eventually influence application registers to affect program output. It is true that there are certain types of microarchitecture-specific faults that are difficult to inject using a software-only tool such as PIN. In return, software-based error injection techniques can be run on native hardware rather than detailed microarchitecture simulators (typically orders of magnitude slower) and allows much shorter experiment turn-around time. This type of software-based error injection technique was also used in other studies [13][17]. For error coverage measurement, each SPEC benchmark is run 1000 times and during each run one error is injected. We use software queue and run the SRMT compiler generated binaries on real machines (Intel® Xeon 2.2 GHz processor-based systems). After a fault is injected into the execution, the program may show one of the following five possible behaviors. If it generates an exception, we consider the error as Detected by Handler (DBH). If the program's output and exit code were identical to those of good execution, that fault is considered Benign (or unnecessary for architecturally correct execution [20]). If the output or exit code were different, then the fault has caused a Silent Data Corruption (SDC). There is also a slight chance that a fault may lead an application to an infinite loop. This infinite execution scenario can be detected with a timeout script; we consider errors detected this way as Timeout. For applications compiled with SRMT compiler, errors successfully detected by the trailing thread are referred to as Detected.

The numbers of DBH, Benign, Timeout, Detected, and SDC for SRMT versions and non-SRMT (ORIG) versions are plotted in Figure 9 (for integer) and Figure 10 (for floating-point).

SRMT versions, on average, have about 0.02% and 0.4% SDC for integer and floating-point benchmarks, respectively. On the other hand, non-SRMT versions have about 5.8% and 12.6% SDC for integer and floating-point benchmarks. Furthermore, non-SRMT versions have higher portions of DBH, especially for integer benchmarks: 35.3% for non-SRMT version vs. 25.0% for SRMT versions. This happens because

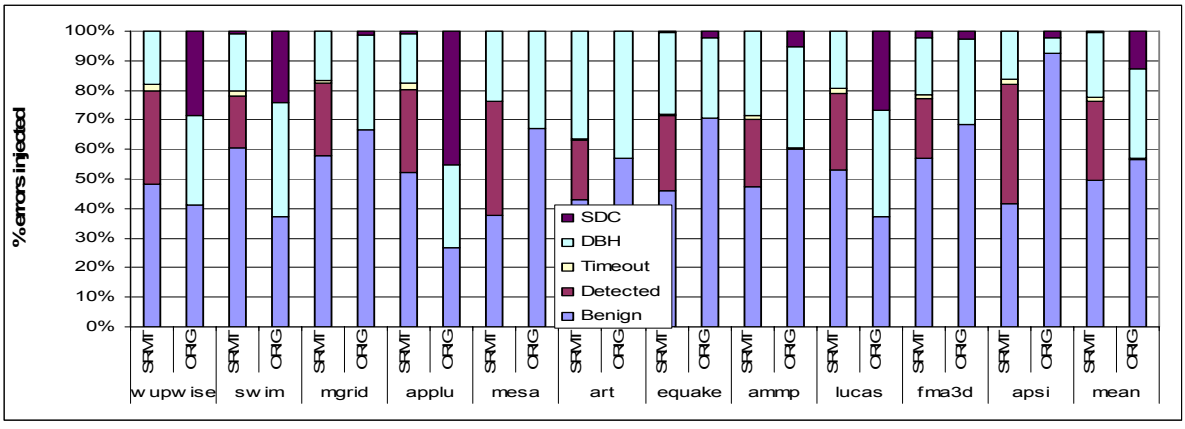

Figure 10. Fault injection distributions for SPEC2000 FP Benchmarks

undetected errors may result in invalid memory addresses, thereby leading to segmentation faults. On the other hand, SRMT programs have built-in error detection, and 26.1% and 26.8% of errors are detected for integer and floating-point benchmarks, respectively.

There are a few reasons why error coverage rate is not 100%. First, the binary code in the experiments is not replicated and hence, the errors injected into such code are not be detected. This is an example of trade-off between flexibility in software compatibility and error coverage. If desired, the entire software stack could be compiled with SRMT and made more reliable over all. Next, software-only techniques work at program-visible architected state level so there may be small window of vulnerability where a fault happens after the value is checked. For example, a value may be corrupted after it is sent to the trailing thread for checking but before being used by the leading thread [13].

5.2 Performance

Our SRMT targets emerging CMP processors that are more susceptible to soft-errors. We first evaluate the performance potential of our SRMT approach on a CMP prototype with a cycle-accurate simulator. The CMP prototype supports an inter-core communication queue, which has been proposed as a general communication mechanism in recent studies [11][15][19][29]. More specifically, our simulator supports a SEND instruction to put data into the hardware queue and a RECEIVE instruction to get data from the queue. These instructions automatically block when the queue is full or empty, respectively. The queuing mechanism is fully pipelined. We simulated six SPEC CPU2000 integer benchmarks. The latency for the data traveling in the queue from the leading thread to the trailing thread is also faithfully modeled.

Performance results are shown in Figure 11, relative to non-SRMT programs. It shows that we may achieve SRMT with only 19% overhead (see the bar marked with *CMP on-chip queue*), which is competitive with HRMT that uses special hardware extensively [12]. Note that it is possible that this overhead can even be further reduced with further support in the instruction set architecture (ISA). Figure 11 also shows dynamic instruction counts of the leading and trailing threads. The trailing thread always has less instruction executed, as some computations become dead after error checking. Therefore the leading thread dominates the SRMT execution time. Here, dynamic instruction count increase in the leading thread (about 37%) is higher than cycle time increase (19%). This is because the newly inserted instructions for SRMT are mostly for enqueue and register spill/restore. As such they are not as performance-critical as memory accesses and branches.

We next experiment our SRMT approach without the communication queue but with a shared on-chip cache. Although a few commercial CMP processors that contain a shared L2 cache are being introduced nowadays, we have not been able to run our SRMT compiler on such a system yet. Our experiment is conducted on the same simulator; the cores in the CMP prototype have private L1 caches and share the same on-chip L2 cache. Figure 12 shows dynamic instruction count increases in the leading thread (right-hand bars) and performance slowdown (left bars). The overall performance slowdown is about 2.86X and the instruction count increase is about 2.2X. Here the performance slowdown rate is higher than that of dynamic instruction count increase because there is still significant coherence overhead to move the queue data from leading CPU's L1 cache to trailing CPU's L1 cache through the cache hierarchy.

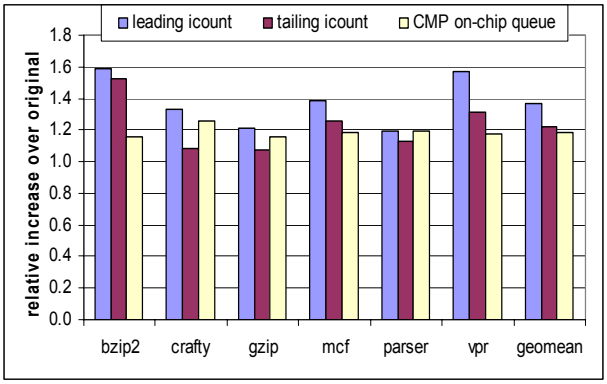

Figure 11. Performance impact of SRMT on Simulator for CMP machine with on-chip queue

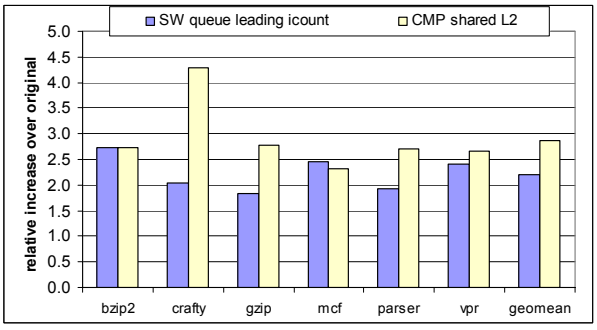

Figure 12. Performance of SRMT with SW queue on Simulator for CMP machine with shared L2

We also experiment our SRMT with software queue communication on a shared-memory system with eight Intel® Xeon 2.2GHz processors [22], to understand the effects of three different communication patterns on SRMT performance. Each processor supports two hyper-threads so we can run the leading and trailing

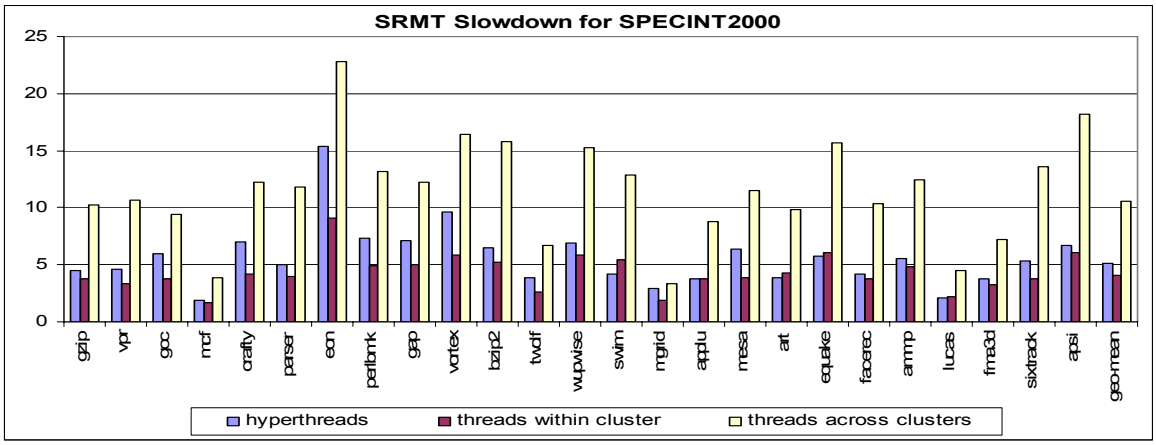

Figure 13 Overhead of SRMT with SW queue on SMP machine for Spec2000 Benchmarks

threads on the two hyper-threads of the same processor (config 1). Also, the eight processors are grouped into two clusters, with four processors in each cluster sharing an off-chip L4 cache. We can run the leading and trailing threads on two processors that are either in the same cluster sharing the L4 cache (config 2) or in different clusters accessing two different L4 caches (config 3).

Figure 13 shows performance results for SPEC CPU2000 integer and floating-point benchmarks. As expected, overhead with software queue on SMP machine is quite high, with average slowdown more than four times. This is mainly due to the cache coherence overhead reasons discussed in Section 4. It is interesting to notice that the configuration with two threads running on two processors sharing the same L4 cache, config 2, performs the best. Even though two hyper-threads share an L1 cache, they also share most of microarchitecture resources, such as execution units. The latter is the performance limiter for the hyper-threading configuration, config 1. On the other hand, config 1 performs better than the configuration with the two threads running on two processors accessing two different L4 caches, config 3. Here large cluster-to-cluster communication latency is clearly holding the whole system back.

5.3 Communication bandwidth

Figure 14 shows total number of bytes of data communicated between the leading and trailing treads divided by total cycle count of original program execution without SRMT. This gives average bandwidth requirements for our SRMT approach in order to not slow down original program execution. Average bandwidth requirement in our SRMT approach is only about 0.61 byte per cycle, compared to that of 5.2 bytes per cycle in HRMT approach [6]. This demonstrates a key benefit of our SRMT approach over HRMT and binary tool based approaches. Unlike HRMT approaches, which incur communication for each memory access, we do not need data communication for local memory accesses, such as register spills and reloads, which are common in CISC ISAs like IA-32.

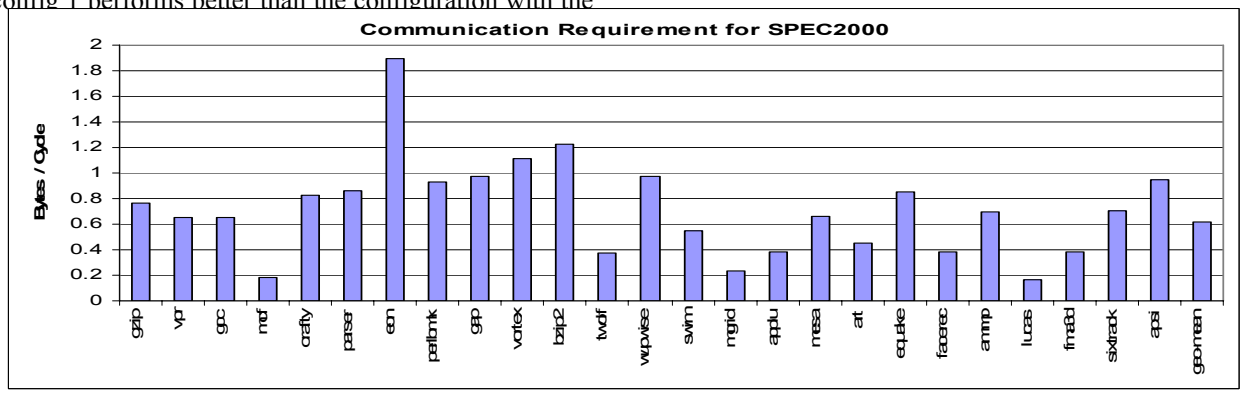

Figure 14. SRMT Bandwidth Requirement for SPEC2000 Benchmarks

Note that bandwidth requirements for different benchmark programs are roughly proportional to the performance slowdown shown in Figure 13. This again confirms that slow down in our SRMT approach with software queue on SMP machines mostly comes from data communication overhead.

6. Summary and Future Work

This paper presents a Software-based Redundant Multi-Threading (SRMT) approach for transient fault detection. We show that SRMT is a promising technique with the following advantages:

- Software-based approach can reduce design and validation complexity inherent in HRMT approaches.
- Software-based approach is flexible. Applications with SRMT and normal applications without SRMT can run simultaneously on the same hardware for different reliability and performance requirements. We also allow SRMT code and existing binary code to be linked and run in the same application.
- Sophisticated compiler analysis and optimizations can reduce error checking overhead. For example, SRMT incurs about 0.61 bytes per cycle communication bandwidth, comparing favorably to 5.2 bytes per cycle in HRMT [6].
- For emerging Chip-level Multi-processors (CMP) that have fast intra-chip communication queues between cores, we show that slowdown of SRMT over original program is about 19%. This is competitive with HRMT approaches [12].
- Our SRMT approach achieves error coverage rate of 99.98% and 99.6% for SPEC CPU2000 integer and floating-point benchmarks, respectively, making it useful in many applications.

Currently, we are pursuing the following extensions:

- SRMT can be extended to perform both error detection and recovery. One way to perform error recovery is to have two trailing threads, and use majority voting to recover from a single error. This may require us to check for errors that are currently detected by signal handlers. Another method is to explore hardware support to buffer store values for recovery.
- Our SRMT technique currently only allows the leading thread to perform non-local memory accesses, and send results to the trailing thread. We may balance memory references in the two threads and allow the trailing thread to perform some non-local memory references and send the result to the leading thread.
- There exist several binary translation works to improve reliability [2]. We may apply our SRMT technique through binary translation to improve reliability of legacy code without recompilation.
- We are also working closely with a processor reliability group to apply our SRMT approach in mean-time-to-fail (MTTF) and soft-error rate (SER) measurements with Neutron-induced soft-errors.

Acknowledgements

We would like to thank Intel's compiler team for developing the compiler infrastructure used in this study, Jesse Fang, John Crawford, Shubu Mukherjee, Nelson Tam, Guilherme Ottoni, Shiliang Hu and Wei Liu for their support and valuable comments, Peng Tu, Robert Cox, Jack Liu for helping with the compiler internals. We appreciate the comments from the anonymous reviewers that helped improve the quality of the paper.

References

[1] R. Baumann. Soft-Errors in Commercial Semiconductor Technology: Overview and Scaling Trends. *IEEE 2002 Reliability Physics Symposium*, Tutorial Notes, Reliability Fundamentals, IEEE press, 2002.

[2] E. Borin, C. Wang, Y. Wu, G. Araujo, Software-Based Transparent and Comprehensive Control-Flow Error Detection. In *Proceedings of the Fourth Annual International Symposium on Code Generation and Optimization*, 2005.

[3] C. Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro, 23(4), 14-19*, July 2003.

[4] J. Dugan and M. Lyu. System Reliability Analysis of N-version Programming Application. *IEEE Transactions on Reliability, 43(4), 513-519*, Dec. 1994.

[5] H. Ando, et al. A 1.3-GHZ Fifth-Generation SPARC64 Microprocessors. *IEEE Journal of Solid-State Circuits Conference, 38(11), 1896-1905*, Nov. 2003.

[6] M. Gomaa, C. Scarbrough. T. N. Vijaykumar, and I. Pomeranz. Transient-Fault Recovery for Chip Multi-processors. In *Proceeding of the 30th Annual International Symposium on Computer Architecture*, 2003.

[7] A. KleinOsowski and D.J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 1, June 2002.

[8] R. Lo, F. Chow, R. Kennedy, S.-M. Liu, and P. Tu. Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores. In *Proceeding of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[9] S. S. Mukherjee. M. Kontz, and S. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of the 29th Annual*

International Symposium on Computer Architecture, 2002.

[10] P. Murray, R. Fleming, P. Harry, and P. Vickers. Somersault Software Fault-Tolerance. *HP Labs whitepaper*, 1998.

[11] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, 2005.

[12] S. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[13] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. Mukherjee. Design and Evaluation of Hybrid Fault-Detection Systems. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[14] P. Shivakumar, M. Kisler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft-Error Rate of Combinational Logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, June 2002.

[15] J.-Y. Tsai, E. Ness, and P.-C. Yew. The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation. In *Proceedings of International Conference on Parallel Architecture and Compiler Techniques*, 1996.

[16] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery Using Simultaneous Multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[17] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

[18] C. K. Luk, R. Cohn, R. Muth, H, Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood. Pin: Building customized Program Analysis Tools with Dynamic Instrumentation. In *Proceeding of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005

[19] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic, "HeapMon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection," IBM Journal of Research and Development issue 50-2/3, 2006.

[20] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. "The Soft-Error problem: an architectural perspective." In Proceedings of the 11th International Conference on High-Performance Computer Architecture, pages 243–247, 2005.

[21] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In Proceedings of the 2004 International Conference on Dependendable Systems and Networks, pages 61–72, June 2004.

[22] IBM Corp, "Intel processor-based servers: x445", http://www-03.ibm.com/servers /eserver/xseries/x445.html

[23] S.J. Eggers; Emer, J.S.; Leby, H.M.; Lo, J.L.; Stamm, R.L.; Tullsen, D.M.; "Simultaneous multithreading: a platform for next-generation processors," IEEE Micro, Volume 17, Issue 5, Sept.-Oct. 1997 Page(s):12 - 19

[24] C. Wang and Y. Wu, "Efficient Software Queue with Delayed Buffering and Lazy Synchronization", US patent pending

[25] M. Gomaa and T.N. Vijaykumar. Opportunistic transient-fault detection. 32nd International Symposium on Computer Architecture, June 2005.

[26] N. J. Wang and S. J. Patel. ReStore: Symptom based soft error detection in microprocessors. International Conference on Dependable Systems and Networks, June 2005

[27] V. K. Reddy, S. Parthasarathy, E. Rotenberg. Understanding Prediction-Based Partial Redundant Threading for Low-Overhead, High-Coverage Fault Tolerance. 12th International Conference on Architectural Support for Programming Languages and Operating Systems. Oct. 2006

[28] A. Parashar, S. Gurumurthi, A. Sivasubramaniam. SlicK: Slice-based Locality Exploitation for Efficient Redundant Multithreading. 12th International Conference on Architectural Support for Programming Languages and Operating Systems. Oct. 2006

[29] R. Rangan, N. Vacharajani, A. Stoler, G. Ottoni, D. August, G. Cai. Support for High-Frequency Streaming in CMPs. The 39th Annual IEEE/ACM International Symposium on Microarchitecture 2006.