DAFT: Decoupled Acyclic Fault Tolerance

Yun Zhang[1], Jae W. Lee[1], Nick P. Johnson[1], and David I. August[1]

[1] Department of Computer Science, Princeton University, 35 Olden St., Princeton, NJ 08540. Email: {yunzhang, jl7, npjohnso, august}@princeton.edu

Higher transistor counts, lower voltage levels, and reduced noise margin increase the susceptibility of multicore processors to transient faults. Redundant hardware modules can detect such faults, but software techniques are more appealing for their low cost and flexibility. Recent software proposals have not achieved widespread acceptance because they either increase register pressure, double memory usage, or are too slow in the absence of hardware extensions. This paper presents DAFT, a fast, safe, and memory efficient transient fault detection framework for commodity multicore systems. DAFT replicates computation across multiple cores and schedules fault detection off the critical path. Where possible, values are speculated to be correct and only communicated to the redundant thread at essential program points. DAFT is implemented in the LLVM compiler framework and evaluated using SPEC CPU2000 and SPEC CPU2006 benchmarks on a commodity multicore system. Evaluation results demonstrate that speculation allows DAFT to reduce the performance overhead of software redundant multithreading from an average of 200% to 38% with no degradation of fault coverage.

KEY WORDS: Fault Tolerance, Compiler, Speculation

1. INTRODUCTION

As semiconductor technology continues to scale, the number of transistors on a single chip grows exponentially. This implies an exponential reduction in transistor size, degrading the noise margin of each transistor. In addition, extreme demands for energy efficiency drive aggressive voltage scaling, which leads to an even lower noise margin. While the fault rate per bit remains relatively constant over technology generations [1], these technology trends make processor chips more susceptible to transient faults than ever before.

Transient faults are caused by environmental events, such as particle strikes or fluctuating power supply [2–5], and are nearly impossible to reproduce. Transient faults occur randomly after deployment and are not necessarily attributed to design flaws. These soft errors do not cause permanent hardware damage, but may result in complete system failures. Sun Microsystems acknowledges that customers such as America Online, eBay and Los Alamos National Labs, experienced system failures caused by transient faults [6, 7].

A typical solution for transient fault detection is redundant computation. A program's execution is duplicated, in either hardware or software, and the results of the two instances are compared for validation. Hardware solutions are transparent to programmers and system software, but require specialized hardware (e.g., *watchdog* processor in [8]). Real systems, such as IBM S/390 [9], Boeing 777 airplanes [10, 11], and HP's NonStop Himalaya [12] incorporate hardware transient fault detection and recovery modules. However, redundant execution in custom hardware can increase a processor's transistor count by 20-30%, which leads to extra chip area and additional

verification cost [9, 13]. In addition, the scope and mechanism of protection are hardwired at design time under an assumed failure model and working environment, which may be suboptimal depending on deployment environments. Some hybrid techniques combine custom hardware extension and software redundancy for fault detection [14, 15]. The requirement of hardware extension severely limits the applicability of these techniques. Like hardware-only solutions, they cannot be deployed on commodity platforms.

By contrast, software redundancy is more flexible and cost-efficient in terms of physical resources. This approach can be applied to commodity systems that are already deployed and avoids expensive hardware and chip development costs. In addition, multicore design provides increasing parallel resources in hardware, making software redundancy solutions more viable than ever. Recent implementations of software redundancy, however, either double the usage of general-purpose registers [16], require specialized hardware communication queues [15], or double memory usage [17].

This paper presents DAFT, a software-only speculation technique for transient fault detection. DAFT is a fully automatic compiler transformation that duplicates computations in a redundant trailing thread and inserts fault detection instructions. DAFT speculates that transient fault checking never detects a fault so that cyclic inter-thread communications can be avoided. For misspeculation detection, DAFT generates specialized exception handlers and is capable of discerning transient faults from software exceptions that occur normally (e.g., bugs in the software). Volatile variables, such as memory-mapped I/O addresses, are handled with special care to

prevent speculative execution from triggering an externally observable side effect. As a result, DAFT exhibits very low performance overhead. Communication and code optimizations are then applied to further improve whole program performance. As a software-only approach, DAFT provides the programmer with the flexibility to choose the region of a program to protect.

In short, DAFT advances the state-of-the-art in software redundant multithreading by achieving the following desirable properties:

- Geomean performance overhead of 38% on a real multicore machine, compared to 200% for a non-speculative version of software redundant multithreading. This low overhead is comparable to those of hardware solutions but achieved without any hardware support.
- Ability to distinguish normal exceptions from transient faults and guarantee no false positives.
- 99.93% fault coverage on a mixed set of SPEC CPU2000 and SPEC CPU2006 benchmark programs in the transient fault simulation experiments. This is comparable to other hardware and software redundancy techniques.

The remainder of this paper is organized as follows: Section 2 surveys related work and compares DAFT with other approaches. Section 3 introduces the software speculation technique in DAFT and other optimizations to minimize performance overhead without compromising fault detection capabilities. Section 4 presents the automatic code transformation algorithm of DAFT. Section 5 presents experimental results along with analysis. Section 6 concludes the paper.

2. RELATED WORK

Early multithreaded techniques for fault tolerance rely on specialized hardware to execute redundant copies of the program for transient fault detection and recovery. Rotenberg's AR-SMT [18] is the first technique to use simultaneous multithreading for transient fault detection. An active thread (A) and a redundant thread (R) execute the same program at runtime, and their computation results are compared to detect transient faults. Mukherjee et al. improved AR-SMT with Chip-level Redundant Threading (CRT), which uses a multicore chip for redundant execution and value checking [19]. Simultaneous Redundant Threading (SRT), proposed by Reinhardt et al., detects transient faults based on simultaneous multithreading processors [14]. However, all these techniques rely on specialized hardware extensions, hence are not applicable to off-the-shelf commodity systems.

The pi bit [20] by Weaver et al. and dependence-based checking [21] by Vijaykumar et al. have been proposed as methods to only detect faults that affect program outcome. This is done by following the propagation of faults through the entire program. These techniques also require custom hardware extensions to dynamically track true register dependences between instructions towards the program output as the program is executed.

Software redundancy detects transient faults without any hardware support [15, 16, 19, 22, 23]. Techniques for software redundancy can be divided into three categories: *thread-local duplicated execution*, *software redundant multithreading*, and *process-level redundant execution*.

Thread-local duplicated execution techniques, such as EDDI [22] and SWIFT [16], redundantly execute instructions within a single thread. Thread-local duplicated execution has several advantages. No communication or synchronization is necessary between original execution and redundant execution traces. Both original and duplicate instructions are executed on the same processor, better utilizing the cache.

However, thread-local duplicated execution doubles register usage and relies solely on instruction-level parallelism (ILP) to reduce the performance overhead from an increased instruction count. On architectures with a small number of registers, this causes extra register spills. For this reason, SWIFT's overhead is low on architectures with many registers, such as the Itanium [24]. However, instruction-level redundancy has much higher overhead on x86_64 architecture having only 16 general purpose registers.

Software Redundant Multithreading (SRMT) executes identical code on difference processors using multiple threads [15], and compares the results to ensure correct execution. This approach maintains one shared copy of memory space, which implies that redundant multi-threading techniques lose redundancy at store instructions. Before a memory operation is executed, its operands must be communicated between threads and checked for consistency. If the values of the operands match, the *trail* thread sends a message to the lead thread, confirming the absence of transient fault. The lead thread then executes the memory operation, and proceeds. This barrier synchronization incurs significant performance cost. One such implementation of software redundant multi-threading, SRMT, is reported to have up to 4× slowdown on real machines without hardware extensions [15] partially for this reason.

When a real transient fault triggers an exception (for instance, by causing division by zero), SRMT invokes the program's exception handler to catch the fault, possibly leading to a false positive and changing the program's behavior. On real machines without hardware extension, the experiment results of SRMT report up to 4× slowdown. Like SRMT, DAFT takes a software-only redundant multithreading approach. DAFT speculates that all computations execute correctly and verifies them off the critical path, drastically reducing the overhead of fault detection. Since the inter-thread communication pattern is acyclic, DAFT is insensitive to the latency of inter-core communication. Finally, DAFT distinguishes between transient faults and normal exceptions avoiding false positives without degrading fault coverage.

Process-level redundant execution, duplicates the original program into several process instances [17, 25–28]. Private memory space owned by each process provides natural protection for non-externally visible memory operations, except for shared memory. Only externally visible values need to be verified when they escape user space. DieHard [26] by Berger et al. uses redundancy on general-purpose machines for memory fault tolerance, which can be used in combination with DAFT for whole system protection. Process-level Redundancy (PLR) presented by Shye et al. acts as a shim between user programs and the operating system [17]. In PLR, two identical program instances run simultaneously on multiple processors, and performs fault detection only on externally visible side effects, such as I/O operations and program termination. This approach guarantees that faults do not change the observable behavior. PLR checks fewer values and tends to have lower overheads than other software redundancy techniques, yet the memory usage of PLR is at least dou-

bled. PLR's memory footprint can be prohibitive for memory-bound applications or memory-constrained systems, such as embedded devices. In addition, PLR must be applied at the whole program granularity; programmers and tools cannot select critical sections of code that need protection.

Representative transient fault detection techniques are summarized in Table I. Compared with other techniques, DAFT provides broad fault coverage, presents little pressure on register files, requires no specialized hardware, and keeps memory overhead minimal.

[Table 1 about here.]

3. DECOUPLED ACYCLIC FAULT TOLERANCE

This section presents the design of DAFT with step-by-step development. Section 3.1 defines the Sphere of Replication (SoR) of DAFT, which determines the scope of protection. Section 3.2 introduces a non-speculative version of redundant multithreading. Section 3.3 describes the software speculation technique in DAFT to minimize performance overhead caused by redundant execution and error checking. While boosting performance, speculation poses new challenges for detecting faults and ensuring the correctness of program execution. Section 3.4 addresses these challenges with three fault detection mechanisms. Section 3.5 discusses how DAFT handles indirect function calls. Finally, Section 3.6 presents several communication and code optimization techniques to make DAFT even faster.

3.1. Sphere of Replication

The Sphere of Replication (SoR) [14] is a logical domain of redundant execution within which all activity and state is replicated, in either space or time. Like previ-

ous fault tolerance techniques [14, 16, 19, 22, 23], DAFT's SoR is the processor core. DAFT's SoR does not include the memory subsystem, such as caches and off-chip DRAMs, as it can be protected by error correction codes (ECC). In practice, all static instructions, except memory operations (i.e. loads and stores), need to be replicated in DAFT across the leading and the trailing threads.

Loads are excluded from replication because a pair of loads from the same memory address in a shared memory model are not guaranteed to return the same value, as there is always a possibility of intervening writes between the two loads from an exception handler or from other threads. Replicating load operations may lead to false positives in fault detection. The situation is the same for stores. Library functions are also excluded in cases when the DAFT compiler does not have access to the library's source code or intermediate representation.

DAFT executes each load only once in the leading thread and passes loaded values to the trailing thread via a software queue. Similarly, store instructions are executed once in the leading thread, with values and memory addresses being checked in the trailing thread. In this way, DAFT ensures deterministic program behavior and eliminates false positives. Because the source code of library functions is not available for DAFT to compile, calls to such functions are also only executed once. The return value of a library function call is similarly produced and consumed across the two threads like a loaded value. In Figure 1(a), for example, instructions 2, 4, 5 and 7 are replicable, whereas instructions 1 (library function call), 3 (load), 6 (store) and 8 (store) are not.

[Fig. 1 about here.]

3.2. Non-Speculative Redundant Multithreading

To execute a program with redundant multithreading, the compiler replicates the instructions in the SoR into the leading and trailing threads and inserts code for communication and fault checking. Figures 1(b) and (c) illustrate how the leading and trailing threads in *non-speculative* redundant multithreading are created based on the original program. Instructions for communication and fault checking are emphasized in boldface. Before every memory operation in the leading thread, the memory address and the value to be stored, if any, are sent to the trailing thread. The trailing thread compares these values to the corresponding locally-computed values. The result of fault checking is sent back to the leading thread. The memory operation is commited only if there is no fault; otherwise, the leading thread will stop execution and report a transient fault. Resuming correct program execution after a failure needs support from transient fault recovery scheme, which is not within the scope in this paper. As an example, checkpointing systems [29–31] can be used in concert with DAFT for resuming program execution after a transient fault is detected.

More importantly, these chains of `produce`, `consume`, `check`, `send`, and `wait` instructions create a cyclic communication pattern. As a result, the leading thread spends much of its time waiting for confirmation instead of performing useful work. In the code shown in Figures 1(b) and (c), there are three communication cycles among instruction 4 and 5, 11 and 12, and 16 and 17. Our measurements indicate that this non-speculative version of redundant multithreading has more than 3× slowdown over the original code (see Section 5). Moreover, performance is highly sensitive to the inter-thread communication cost. An increase in communication latency can

cause significant further slowdown. In one realistic setup, SPEC CPU benchmarks with software redundant multithreading slowed down almost by 3× due to an increase in the inter-thread communication cost [15].

Redundant computation and fault checking increase static and dynamic instruction counts, leading to significant performance overhead. Consequently, compiler optimizations should be performed before applying redundant multithreading. These pre-pass optimizations remove dead code and reduce the number of memory operations, leading to less code replication and lower checking/communication overhead.

3.3. Software Speculation in DAFT: Removing Cyclic Dependencies

Cyclic dependencies in the non-speculative redundant multithreading from Section 3.2 put inter-thread communication latency on the critical path of program execution, slowing down the leading thread significantly. Since a transient fault occurs rarely in practice, the trailing thread almost always signals *no fault* to the leading thread. Therefore, this inter-thread communication signal value can be speculated with high confidence.

Inspired by Speculative Decoupled Software Pipelining (Spec-DSWP) [32], DAFT exploits such a high-confidence value speculation to break the cyclic dependencies. Specifically, the communication dependence between `signal` and `wait` instructions is removed. Instead of waiting for the trailing thread to signal back, the leading thread continues execution. Consequently, program performance is insensitive to the inter-thread communication latency. Figures 1(d) and (e) illustrate the program code after speculation is applied. Through speculation, DAFT not only improves program

performance by allowing the leading thread to continue execution instead of busy waiting, but also reduces communication bandwidth use and code growth.

However, speculation poses new challenges for detecting faults and ensuring the correct execution of programs. For example, misspeculation on volatile variable accesses can cause severe non-reversible problems, such as sending a wrong value to an I/O device. Another potential issue is the difficulty of distinguishing a segmentation fault from a transient fault when a fault occurs in a pointer register. The next section discusses challenges and solutions to maintain broad fault coverage without losing the performance benefit of speculation. Figure 2 shows the structure of DAFT.

[Fig. 2 about here.]

3.4. Misspeculation Detection

With speculation, the problem of fault detection in DAFT is effectively translated to the problem of misspeculation detection. Figure 3 shows usage scenarios of a bit-flipped register value and the fault detection mechanisms of DAFT for all the scenarios (leaf nodes in the scenario tree). Some faults are detected by the leading thread, while others by the trailing thread. If the faulty value is never used by later computation, the fault can be safely ignored without affecting the correctness of the program, where "use" means the variable will affect a later store to a memory address, including memory mapped I/O addresses. Figure 3 presents three mechanisms for misspeculation detection in DAFT: in-thread operand duplication for volatile variable accesses, redundant value checking and custom signal handlers.

[Fig. 3 about here.]

3.4.1. In-Thread Operand Duplication

A volatile variable is defined as a variable that may be modified in ways unknown to the implementation or have other unknown side effects [33]. Memory-mapped I/O accesses are an example of volatile variable accesses. Misspeculating transient faults on volatile variable accesses may cause an externally visible side effect which cannot be reversed. Assuming `vaddr` in the example shown in Figure 1(a) is an I/O mapped memory address, `r3` and `vaddr` must be checked for correctness (by instructions 14 and 15) before the store to prevent potentially catastrophic effects. One conservative solution would be to fall back to the non-speculative, cyclic communication pattern of Figures 1(b) and (c). However, performance gains from speculative execution would be lost; communication latency would slow the critical path of program execution.

In this case, the more efficient solution is to verify the operands to the volatile store *in thread*; slowing the leading thread infrequently is a better strategy than cyclic communication. Dataflow analysis is used to compute the def-use chain of the volatile variable. DAFT replicates all instructions from the volatile variable's def-use chain in the leading thread, as shown in Figures 1(d) and (e). An automatic code generation algorithm to handle this case is described in Section 4.

3.4.2. Redundant Value Checking

If a transient fault occurs and flips a bit in a register to be used later, it is usually detected by redundant value checking. The trailing thread in DAFT contains value checking code for every non-volatile store and is responsible for reporting this kind of fault. Instruction 11 in Figure 1(e) illustrates an example of redundant value

checking. The `check` operation compares the two redundant copies, `r2` and `r2'`, and reports a transient fault if the two values mismatch.

3.4.3. Custom Signal Handler

A faulty value may raise an exception before redundant value checking detects the fault in the trailing thread. For instance, memory access to an unmapped address may cause a segmentation fault, or a division instruction may cause a division-by-zero exception. It is also possible that the original program would have to throw the same exception with or without a transient fault. To distinguish between these two cases, DAFT employs a custom signal handler, which is registered (via `sigaction()`) at the beginning of program execution.

When an exception occurs, it is first captured by the DAFT signal handler. The signal handler does not abort the program immediately but waits for the trailing thread to trigger the same exception. If the exception was triggered by a transient fault, the value checking code in the trailing thread eventually detects the fault. If no fault is reported before a timeout occurs, the signal handler assumes that this is a normal exception and calls the corresponding system signal handler. If the original program attempts to register a signal handler itself, DAFT wraps the application signal handler, ensuring that fault-detection logic runs first.

In the case of a valid address, the trailing thread will eventually detect the fault by comparing redundant copies of the faulty register. If the address is invalid, a segmentation fault exception will be triggered. In such a case, SRMT [15] relies on a system signal handler to abort the program. Unfortunately, this is not a safe solution. SRMT cannot distinguish normal program bugs from a transient fault, and

thus changes program behavior. The DAFT signal handler catches all segmentation faults as shown in Figure 2. For example, when a segmentation fault happens, the signal handler traps it and asks the leading thread to wait for a signal from the trailing thread. If the trailing thread confirms the address is correct, the exception is due to a bug in normal program exception, and the original signal handler is called. Otherwise, a transient fault is reported and the program is terminated. This is critical for program safety, especially for programs implementing custom signal handlers.

The program behavior for external signals is not changed, since the original process ID is kept. Any externally triggered signal, such as SIGINT interrupt, will be sent to both threads and the corresponding response will be issued.

3.5. Indirect Function Calls

The compiler *cannot* always determine the target of indirect function calls (e.g. function pointers). However, DAFT must ensure that the trailing thread follows the same path as the leading thread. DAFT overcomes this problem by using trampoline functions. Indirect calls will invoke the leading version of the callee function. Such calls may originate either in the leading thread or in the trailing thread; an extra flag is added to distinguish these cases. If the function is called from the trailing thread, the original function in the leading thread serves as a trampoline and invokes its corresponding trailing version of the function. For the leading thread, only one long jump is made as each call is not very computationally expensive. For the trailing thread, the trampoline in the leading thread is used to invoke the corresponding trailing function. If a wrapper function was used for indirect calls, every function

call instruction in both threads would have to go through two calls, which increases runtime overhead.

3.6. Communication and Code Optimizations: Making DAFT Faster

Speculation removes `wait` and `signal` communication, and takes communication latency off the critical path. However, the amount of communication in a program as well as the communication speed still plays an important role for program performance. To speed up DAFT, two optimizations are applied to DAFT-transformed code for minimal communication cost and fewer branches. Several optimization decisions are also made to speed up a single data communication.

3.6.1. Branch Removal

Since the trailing thread does not duplicate all instructions in the original program, it may sometimes contain basic blocks that contain only `consume` and `branch` instructions. This is not redundant code and cannot be removed through dead code elimination. Figure 4 explains a typical case where some branches can be removed to reduce the number of branch instructions in the trailing thread. In Figure 4(a), basic block `bb1` contains only one library function call and an unconditional branch to basic block `bb12`. The DAFT transformation in Figures 4(b) and (c) creates a basic block `bb1` in the trailing function containing only a `consume` and an unconditional branch. It is preferable to remove the basic block `bb1` entirely and move the communication to the basic block `bb` to avoid one unnecessary branch.

[Fig. 4 about here.]

3.6.2. Inter-thread Communication Lifting

`r1` in Figure 5 is a loop induction variable. Its value is used later in computing the memory address to load from. This pattern is typical in array-based operations. The custom signal handlers in DAFT capture exceptions caused by transient faults, such as segmentation faults or division by zero. If a loop performs only arithmetic computation and memory accesses, it is safe to move the memory address check out of the loop. Any transient faults that may occur during the loop execution can be detected via either the value checking outside of the loop, or the custom signal handler. In the example code in Figure 5, DAFT can remove one inter-thread communication and one value checking per iteration.

[Fig. 5 about here.]

3.6.3. Software Communication Queue

In DAFT, an unbalanced lock-free ring buffer software queue library is used for inter-thread communication [34]. This queue implementation shifts more work of communication onto the consumer thread. Since all communications in DAFT are uni-directional from the leading to trailing thread, the fast communication queue ensures low runtime overhead and latency tolerance.

The queue implementation uses streaming store and prefetching to achieve best performance on real machines. Streaming store is an SSE instruction for better bandwidth and performance stability. Streaming stores bypass L2 cache and write to memory directly. This optimization speeds up communication especially when two threads are not sharing an L2 cache. Prefetching is enabled for the consumer to prefetch queue data into its own cache before the values are needed.

4. DAFT AUTOMATIC CODE TRANSFORMATION

DAFT is implemented as an automatic compiler transformation in the LLVM compiler framework [35]. A program can be transformed to DAFT-protected code using Algorithm 1. This algorithm takes an intermediate representation (IR) of the program and the program dependence graph (PDG) [36] as inputs, produces a new program IR containing code for redundant computation using multiple threads, inter-thread communication, value checking, and signal handler registration.

Once the function `main` is invoked, a redundant thread is spawned with identical program inputs, shared memory space, and shared file system. The original program copy is called *Leading* thread, and the redundant program copy is called the *Trailing* thread.

4.1. Identifying Instructions for the Trailing Function

For each function in a program, DAFT first traverses the intermediate representation and partitions the instructions into three sets:

- Non-replicable
- In-thread replicable
- Redundant replicable

Non-replicable instructions are those which directly load from or store to memory, or are library function calls. In-thread replicable instructions are those which compute the address or value of a volatile memory operation, or I/O operations. Redundant replicable instructions are all other instructions: those which do not access memory, or are not calls to library functions. DAFT replicates in-thread redundant instructions

Algorithm 1 Automatic DAFT transformation

```
1: Register signal handlers for Leading and Trailing
2: for all Function func ∈ Program do
3:    IdentifyTrailingThreadInst(func)
4:    BuildRelevantBBs(func)
5:    for all Instruction inst ∈ func do
6:       if inst ∈ InThreadReplicableSet then
7:          copy inst to Leading
8:          copy inst to Leading
9:          if inst ∈ STORE_OPs then
10:            append check to Trailing
11:         end if
12:      else if inst ∈ RedundantReplicableSet then
13:         copy inst to Trailing
14:      else if inst ∈ NonReplicableSet then
15:         if inst ∈ LOAD_OPs then
16:            append produce to Leading
17:            append consume to Trailing
18:         else
19:            append produce to Leading
20:            append consume to Trailing
21:            append check to Trailing
22:         end if
23:      end if
24:   end for
25:   RedirectBranches(func)
26: end for
27: InsertInitialAndFinalCommunication()
```

Algorithm 2 IdentifyTrailingThreadInst (Function func)

```
1: // Building Replicable Instruction Set
2: RedundantReplicableSet = InThreadReplicableSet = NonReplicableSet = ∅
3: for all Instruction inst ∈ func do
4:    if inst ∈ Memory_OPs then
5:       NonReplicableSet = NonReplicableSet ∪ {inst}
6:       if inst ∈ Volatile Operations then
7:          for all Instruction prev_inst ∈ DefinitionChain(inst) do
8:             InThreadReplicableSet = InThreadReplicableSet ∪ {prev_inst}
9:          end for
10:      end if
11:   else if inst ∈ REG_OPs then
12:      RedundantReplicableSet = RedundantReplicableSet ∪ {inst}
13:   else if inst ∈ CALL_INST then
14:      callee = getCalledFunction(inst)
15:      if callee ∈ Program then
16:         RedundantReplicableSet = RedundantReplicableSet ∪ {inst}
17:      end if
18:   else
19:      NonReplicableSet = NonReplicableSet ∪ {inst}
20:   end if
21: end for
22: RedundantReplicableSet = RedundantReplicableSet \ InThreadReplicableSet
```

Algorithm 3 BuildRelevantBBs (Function func)

```
1: Work = Controls = ∅
2: Work = Work ⋃ RedundantReplicableSet
3: while Work ≠ ∅ do
4:    for all Instruction inst ∈ Work do
5:       temp = ControlDepend(inst)
6:       temp = temp \ Controls
7:       if temp ≠ ∅ then
8:          Work = Work ⋃ temp
9:          Controls = Controls ⋃ temp
10:      end if
11:      Work = Work \ { inst }
12:   end for
13: end while
14: RelevantInsts = Controls ⋃ RedundantReplicableSet
15: for all BasicBlock bb ∈ func do
16:   for all Instruction inst ∈ bb do
17:      if inst ∈ RelevantInsts then
18:         RelevantBBs = RelevantBBs ⋃ bb
19:         break
20:      end if
21:   end for
22: end for
```

Algorithm 4 RedirectBranches (Function func)

```
1: for each branch instruction branch ∈ Trailing do
2:    newTarget = closestRelevantPostDom(target(branch))
3:    redirect(branch, newTarget)
4: end for
```

into the leading thread, whereas redundant replicable instructions are replicated into the trailing thread. The process of identifying these sets of instructions is shown in Algorithm 2.

4.2. Identifying Relevant Basic Blocks

Next, we construct a new, empty function to serve as the trailing thread. Both threads must follow the same control flow. However, not every basic block will perform work in the trailing thread. For efficiency, we selectively copy only *relevant* basic blocks to the trailing thread.

We say that a basic block *bb* is relevant to the trailing thread if (i) any instruction from *bb* is in the redundant-replicable set, or (ii) an instruction from *bb* controls[1] any instruction relevant to the trailing thread.

The second rule is transitive. It is necessary to identify and replicate a skeleton of the control flow graph which is relevant to the trailing thread as to ensure control equivalence between the leading and trailing threads. This second rule is repeatedly applied until the relevant set converges. This procedure is described in Algorithm 3.

4.3. Automatic Code Generation

The code generation pass of the DAFT transformation consists five steps as follows.

4.3.1. Replicating Redundant Code

Whenever a value escapes the SoR and needs to be communicated from the leading thread to the trailing thread, a `produce` operation is inserted into the leading thread, and a `consume` operation is inserted into the trailing thread at the corresponding

[1] An instruction X controls an instruction Y if, depending on the direction taken at X, Y must execute along one path and may not execute along another path [37].

location. Lines 8 and 10 in Algorithm 1 demonstrate the situations when the original program code is replicated to the leading or trailing thread.

4.3.2. Building Redundant Program Structure

To achieve control equivalence between a pair of leading and trailing threads, conditional branches from each thread must branch the same direction. In other words, the branch predicate must be communicated from the leading thread to the trailing thread. If that branch condition is within the redundant-replicable or non-replicable set, the value should already be communicated to the trailing thread. Otherwise, in the case of in-thread replicable branch conditions, that value must be communicated to the trailing thread with an additional produce-consume pair.

4.3.3. Inserting Communication and Value Checking Codes

For a memory load instruction, one produce-consume pair is created for the memory address. Similarly, two produce-consume pairs are created for a `store` instruction for communicating value and address, respectively. Before each binary function call, each argument passed via register is produced to the trailing thread for value checking. If a library function call returns a value that is used in later computation, that value needs to be communicated, too. Our definition of relevant basic blocks ensures that produce and consume operations are always inserted at control-equivalent locations in each thread. After inter-thread communication instructions are inserted, value checking code (`check`) is inserted into the trailing thread for runtime transient fault detection.

4.3.4. Redirecting Branch Targets

All relevant basic blocks are copied to the trailing thread, first as an empty block. After redundant code replication and communication insertion, the control-flow instruction (branch, switch, return, etc) at the end of each basic block relevant to the trailing thread is duplicated and inserted into the trailing thread. Since the destination basic block may not be relevant to the trailing thread, we redirect those destinations to the closest post-dominating block which is relevant to the trailing thread, as in Algorithm 4.

4.3.5. Inserting Initial and Final Communication Codes

At the start of program execution, DAFT registers the custom exception handler via `sigaction()`, and spawns a new thread to serve as the trailing thread. DAFT invokes `main` in both the leading thread and the trailing thread. At the end of program execution, the trailing thread must join the leading thread. The custom signal handlers are implemented as a library, compiled and linked with DAFT-transformed program code.

4.4. Example Walk-through

The example in Figure 1(a) is used to demonstrate how Algorithm 1 works on a real piece of code. The first step is to identify replicable instructions. The algorithm scans all the instructions in the function. If the instruction is a regular computation statement such as instruction 2, it is inserted into *RedundantReplicableSet*. If the instruction is a memory load/store, or a binary function call, it is immediately marked *NonReplicable*. The rest of the instructions are inserted into *RedundantReplicableSet*.

A tricky case is volatile memory access. In this example, instruction 7 in Figure 1(a) is a regular computation at the beginning of analysis and therefore redundantly replicable. But as soon as instruction 8 is examined, DAFT realizes that `r3` is stored as a volatile variable. At this point, the def-use chain of `r3` is traversed. Instruction 7 is then removed from *RedundantReplicableSet* and inserted into the *InThreadReplicableSet*. This is why replicable instruction sets must be built before code duplication and communication insertion. Such information is stored in a data structure similar to Table II.

[Table 2 about here.]

Once the instructions are classified, the program code is ready for code duplication. All redundant replicable and in-thread replicable instructions are copied to the trailing and leading functions, respectively. Since branch and function call instructions are all redundant replicable, the trailing thread copies the control flow of the leading thread, too. After instruction replication, instructions 2, 7, and 8 in Figure 1(e) are inserted into the trailing thread version of that function. Similarly, instruction 14 in Figure 1(d) is replicated from in-thread replicable instruction 7 in the original program.

Communications are inserted for values that enter or exit the SoR. In this example, instruction 3 (`load`) in Figure 1(a) exits the SoR, therefore the address it loads from must be communicated for correctness checking. In Figure 1(e), instruction 3 is inserted into the trailing thread. Fault checking code is inserted immediately after the communication; `check` operations serve as correctness checking points and alert the user if a fault occurs.

Similarly, for instruction 6 (`store`), both the value and the memory address are communicated, followed by fault checking code. Volatile variable store such as instruction 8 triggers in-thread fault checking. No communication is needed for this store. The fault checking code is inserted into the leading thread immediately before the store commits. The return value of a call to an extern function, such as `r0` in instruction 1 in Figure 1(a), is a value that comes into the SoR, hence it is communicated to the trailing thread. Otherwise, calls to non-external functions such as instruction 4 need no communication or fault checking code since nothing escapes from the SoR.

5. EVALUATION

DAFT is evaluated on a six-core Intel Xeon X7460 processor with a 16MB shared L3 cache. Each pair of cores shares one 3MB L2 cache. A mixed set of SPEC CPU2000 and SPEC CPU2006 benchmark programs is used for reliability and performance analysis. All evaluations use the SPEC *ref* input sets. DAFT is implemented in the LLVM Compiler Framework [35]. DAFT uses fast, lock-free software queues with streaming write and prefetching for inter-thread communication [34].

5.1. Reliability Analysis

To measure the fault coverage of DAFT, Intel's PIN instrumentation tool [38] is used to inject single bit flip faults. Single-event-upset (SEU) model is assumed in the evaluation [14, 24, 39]. First, PIN monitors a profile run of the program to count the number of dynamic instructions. In the simulations, no fault is injected into the standard C libraries, since they are not compiled with DAFT and therefore lack transient fault protection. One dynamic instruction is selected randomly. One

register is selected randomly among general-purpose registers, floating point registers, and flag registers. PIN flips a random bit of the selected register after the selected instruction. Program output is compared against the reference output to ensure that externally visible behavior is unchanged. Each benchmark program is executed 5,000 times, with one transient fault injected in each trial.

This fault injection method cannot simulate faults occurring on the bus or latches. Simulating such faults would require complex hardware modeling support. PIN works at the software level and can simulate faults in architectural state which are the target of this paper. The memory system of the machine used for experimentation is protected by ECC and is outside of DAFT's SoR.

Injected faults are categorized into four groups based on the outcome of the program: (1) Benign faults; (2) Detected by DAFT; (3) Timeout; and (4) Silent Data Corrupt. After a fault is injected, it is possible that the program can still finish running normally with correct output. We call this kind of injected fault *Benign* because it does not affect the program's normal execution. Some injected transient faults can be detected by DAFT through either redundant computation and value checking, or specialized exception handling. This kind of soft error is *Detected by DAFT*. There is a chance that some faults may cause the program to freeze. We specify a scale and an estimated execution time of the program. If the program takes more than $scale \times ExecutionTime$ to finish, our instrumentation aborts the program and reports *Timeout* as an indication that transient fault happened. The fault coverage of DAFT is not 100% because transient faults can occur while moving from a redundant instruction to a non-replicable instruction. For example, if a transient fault

occurs on register `r1` in Figure 1(d) right after instruction 5 (`load`) and instruction 6 (`produce`), DAFT is not able to detect the fault (represented as *Silent Data Corrupt* in Figure 6). However, the possibility of such a fault occurring is extremely low – the fault coverage is evaluated to be 99.93% from simulation.

[Fig. 6 about here.]

5.2. Performance

Figure 7 shows the runtime overhead (vertical axis in the figure) of redundant multithreading with and without speculation, normalized to the original sequential program without any fault protection. The geomean performance overhead of DAFT is 38% (or 1.38× slowdown) on average. Compared with redundant multithreading without speculation (as described in Section 3.2), DAFT is 2.17× faster. Previous software solutions, such as SRMT [15], reported 4.5× program execution slowdown using a software queue on a real SMP machine. Compared to SRMT, DAFT performs favorably, and hence is more practical for real-world deployment.

[Fig. 7 about here.]

DAFT speeds up execution by almost 4× in `473.astar` to 2× in `435.gromacs`, compared to non-speculative redundant multithreading. In `473.astar`, memory loads and stores are closely located with each other in some hot loops. Without speculation, each of the two redundant threads has to wait for the other to pass values over. This back and forth communication puts the communication latency on the critical path, causing the program to slow down significantly. `181.mcf` and `164.gzip` have similar memory access patterns. `435.gromacs` does not contain a lot of closely located

memory loads and stores, but the number of memory operations is higher than in other benchmark programs. More memory operations mean more communications and redundant value checking, which translate to higher runtime overhead.

The whole program slowdown of DAFT mainly depends on the number of memory operations in a program. For one load instruction, DAFT inserts two produce/consume pairs: one before loading to check the correctness of the memory address; the other one after the load to pass values to the trail thread. For one store instruction, two produce/consume pairs need to be inserted: one for the value to be stored and the other for the memory address. Figure 8 indicates the number of communications (linear in the number of values communicated through software queue) normalized to the number of total instructions in a program.

[Fig. 8 about here.]

5.3. Binary File Size

Figure 9 shows the static size of the binary generated by DAFT normalized to the original program without transient fault tolerance. This size of the binary file is the sum of program executable code, and statically linked libraries, such as software communication queue. The geomean code size of DAFT-transformed program is about 2.4× larger than the baseline program. This increase came from the communication primitives inserted into the program, value checking code in the trailing thread, and the initialization code to register signal handlers and fork the trailing thread.

Specifically, every register computation instruction is duplicated into two identical copies. DAFT creates two produce-consume pairs for each load and store instruction, before code optimization. `435.gromacs` has a higher increase in binary file size be-

cause the unprotected program is smaller than other benchmarks. Compared with the non-speculative redundant multithreading approach, DAFT has a comparable increase in binary size, yet lower runtime overhead, due to DAFT's acyclic communication pattern.

[Fig. 9 about here.]

5.4. Memory Footprint

DAFT increases memory usage with additional code for leading and trailing threads, a runtime stack for each thread, and a communication queue. All benchmark programs were instrumented to measure DAFT's impact on memory pressure. At program termination, the leading thread dumps the program's peak runtime memory usage. This number is collected from the operating system through the `/proc` file system.

Figure 10 shows that DAFT uses a geomean of 1.13× memory footprint of the unprotected programs. The extra memory usage comes from the software communication queue, extra stack allocation and increased binary file size. The software queue uses a fixed amount of memory. As demonstrated in Figure 1(b), redundant multithreading without speculation requires bi-directional inter-thread communication between the leading thread and the trailing thread, resulting in using two software communication queues, while DAFT uses only one queue. DAFT uses 5% less peak runtime memory than non-speculative redundant multithreading across all benchmark programs.

[Fig. 10 about here.]

The memory overhead of DAFT-protected code is higher for programs with low memory usage, since the constant 16 MB overhead of the communication queue is significant. For other benchmarks, such as `164.gzip` which uses 329 MB memory, this overhead is less pronounced. This is an improvement over a previous approach that more than doubles the memory usage for every benchmark program [17].

6. CONCLUSION

Future processors will ship with more cores, more and smaller transistors, and lower core voltages. Short of a miracle in silicon fabrication technology, transient faults will become a critical issue for developers everywhere. However, the multicore revolution has brought redundant hardware to commodity systems, enabling low-cost software redundancy for fault detection.

This paper presents a fast, safe and memory-efficient redundant multithreading technique for transient fault detection. By combining speculation, custom signal handlers, and intelligent communication schemes, DAFT provides advanced fault detection on off-the-shelf commodity hardware. It features minimal runtime and memory overhead. Unlike some of previous software solutions, DAFT correctly handles exceptions and distinguishes program exceptions from transient faults. DAFT can provide reliability for off-the-shelf systems without specialized hardware or explosion in memory use.

ACKNOWLEDGMENTS

We thank the Liberty Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and

suggestions. This material is based upon work supported by the National Science Foundation under Grant No. 0627650. We acknowledge the support of the Gigascale Systems Research Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the Liberty Research Group and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

1. S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai, Impact of CMOS Scaling and SOI on Software Error Rates of Logic Processes, *VLSI Technology Digest of Technical Papers* (2001).

2. R. C. Baumann, Soft Errors in Advanced Semiconductor Devices-Part I: The Three Radiation Sources, *IEEE Transactions on Device and Materials Reliability*, **1**(1):17–22 (March 2001).

3. T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, I. C. J. Montrose, H. W. Curtis, and J. L. Walsh, Field Testing for Cosmic Ray Soft Errors in Semiconductor Memories, *IBM Journal of Research and Development*, pp. 41–49 (January 1996).

4. G. A. Reis, J. Chang, D. I. August, R. Cohn, and S. S. Mukherjee, Configurable Transient Fault Detection via Dynamic Binary Translation, *Proceedings of the 2nd Workshop on Architectural Reliability* (2006).

5. J. Segura and C. F. Hawkins, *CMOS Electronics: How It Works, How It Fails*, Wiley-IEEE Press (April 2004).

6. R. C. Baumann, Soft Errors in Commercial Semiconductor Technology: Overview and Scaling Trends, *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pp. 121_01.1 – 121_01.14 (April 2002).

7. S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender, Predicting the Number of Fatal Soft Errors in Los Alamos National Labratory's ASC Q Computer, *IEEE Transactions on Device and Materials Reliability*, **5**(3):329–335 (September 2005).

8. A. Mahmood and E. J. McCluskey, Concurrent Error Detection Using Watchdog Processors-A Survey, *IEEE Transactions on Computers*, **37**(2):160–174 (1988).

9. T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb, IBM's S/390 G5 Microprocessor Design, *IEEE Micro*, Vol. 19, pp. 12–23 (March 1999).

10. Y. Yeh, Triple-triple redundant 777 primary flight computer, *Proceedings of the 1996 IEEE Aerospace Applications Conference*, Vol. 1, pp. 293–307 (February 1996).

11. Y. Yeh, Design considerations in Boeing 777 fly-by-wire computers, *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium*, pp. 64 – 72 (November 1998).

12. R. W. Horst, R. L. Harris, and R. L. Jardine, Multiple Instruction Issue in the NonStop Cyclone Processor, *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 216–226 (May 1990).

13. H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama, A 1.3GHz Fifth Generation SPARC64 Microprocessor, *International Solid-State Circuits Conference* (2003).

14. S. K. Reinhardt and S. S. Mukherjee, Transient fault detection via simultaneous multithreading, *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 25–36, ACM Press (2000).

15. C. Wang, H.-S. Kim, Y. Wu, and V. Ying, Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection, *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pp. 244–258, IEEE Computer Society, Washington, DC, USA (2007).

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

17. A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance, *International Conference on Dependable Systems and Networks*, IEEE Computer Society, Los Alamitos, CA, USA (2007).

18. E. Rotenberg, AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors, *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, p. 84, IEEE Computer Society (1999).

19. S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, Detailed design and evaluation of redundant multithreading alternatives, *SIGARCH Comput. Archit. News*, **30**(2):99–110 (2002).

20. C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor, *Proceedings of the 31st Annual International Symposium on Computer Architecture* (2004).

21. T. N. Vijaykumar, I. Pomeranz, and K. Cheng, Transient-fault Recovery using Simultaneous Multithreading, *The 29th Annual International Symposium on Computer Architecture*, pp. 87–98, IEEE Computer Society (2002).

22. N. Oh, P. P. Shirvani, and E. J. McCluskey, Error Detection by Duplicated Instructions in Super-Scalar Processors, *IEEE Transactions on Reliability*, Vol. 51, pp. 63–75 (March 2002).

23. M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, Transient-fault recovery for chip multiprocessors, *Proceedings of the 30th annual international symposium on Computer architecture*, pp. 98–109, ACM Press (2003).

24. G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, Design and Evaluation of Hybrid Fault-Detection Systems, *Proceedings of the 32th Annual International Symposium on Computer Architecture*, pp. 148–159 (June 2005).

25. A. Avizienis, The N-Version Approach to Fault-Tolerant Software, *IEEE Transaction of Software Engineering*, **11**:1491–1501 (December 1985).

26. E. D. Berger and B. G. Zorn, DieHard: Probabilistic Memory Safety for Unsafe Languages, *Proceedings of the ACM SIGPLAN '06 Conference on Programming Language Design and Implementation* (June 2006).

27. S. S. Brilliant, J. C. Knight, and N. G. Leveson, Analysis of Faults in an N-Version Software Experiment, *IEEE Trans. Softw. Eng.*, **16**(2):238–247 (1990).

28. G. Novark, E. D. Berger, and B. G. Zorn, Exterminator: automatically correcting memory errors with high probability, *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pp. 1–11, ACM, New York, NY, USA (2007).

29. W. D. James and J. E. L. Jr, A User-level Checkpointing Library for POSIX Threads Programs, *The Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing* (1999).

30. K. Whisnant, Z. Kalbarczyk, and R. K. Iyer, Micro-Checkpointing: Checkpointing for Multithreaded Applications, *Proceedings of the 6th IEEE International On-Line Testing Workshop (IOLTW)*, IEEE Computer Society, Washington, DC, USA (2000).

31. M. Rieker and J. Ansel, Transparent user-level checkpointing for the Native POSIX Thread Library for Linux, *International Conference on Parallel and Distributed Processing Techniques and Applications* (2006).

32. N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, Speculative Decoupled Software Pipelining, *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pp. 49–59, IEEE Computer Society, Washington, DC, USA (2007).

33. ISO/IEC 9899-1999 Programming Languages – C, Second Edition, 1999.

34. T. B. Jablin, Y. Zhang, J. A. Jablin, J. Huang, H. Kim, and D. I. August, Liberty Queues for EPIC Architectures, *Proceedings of the 8th Workshop on Explicitly Parallel Instruction Computing Techniques* (April 2010).

35. C. Lattner and V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, p. 75, IEEE Computer Society, Washington, DC, USA (2004).

36. J. Ferrante, K. J. Ottenstein, and J. D. Warren, The program dependence graph and its use in optimization, *ACM Transactions on Programming Languages and Systems*, **9**:319–349 (July 1987).

37. G. Ottoni, R. Rangan, A. Stoler, and D. I. August, Automatic Thread Extraction with Decoupled Software Pipelining, *MICRO '05: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 105–118, IEEE Computer Society, Washington, DC, USA (2005).

38. 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, *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pp. 190–200, ACM, New York, NY, USA (2005).

39. D. Walker, L. Mackey, J. Ligatti, G. A. Reis, and D. I. August, Static typing for a faulty lambda calculus, *SIGPLAN Not.*, **41**(9):38–49 (2006).

LIST OF TABLES

Table I. Comparison of transient fault detection techniques

	SRT [14]	SWIFT [16]	SRMT [15]	PLR [17]	DAFT [This Paper]
Special Hardware	Yes	No	No	No	No
Register Pressure	1×	2×	1×	1×	1×
Fault Coverage	Broad	Broad	Broad	Broad	Broad
Memory Usage	1×	1×	1×	2×	1×
Communication Style	Cyclic	None	Cyclic	Cyclic	Acyclic

Table II. Replicability of instructions in Figure 1(a)

Replicability	Instruction
In-thread Replicable	7
Redundant Replicable	2, 4, 5
Non-replicable	1, 3, 6, 8

LIST OF FIGURES

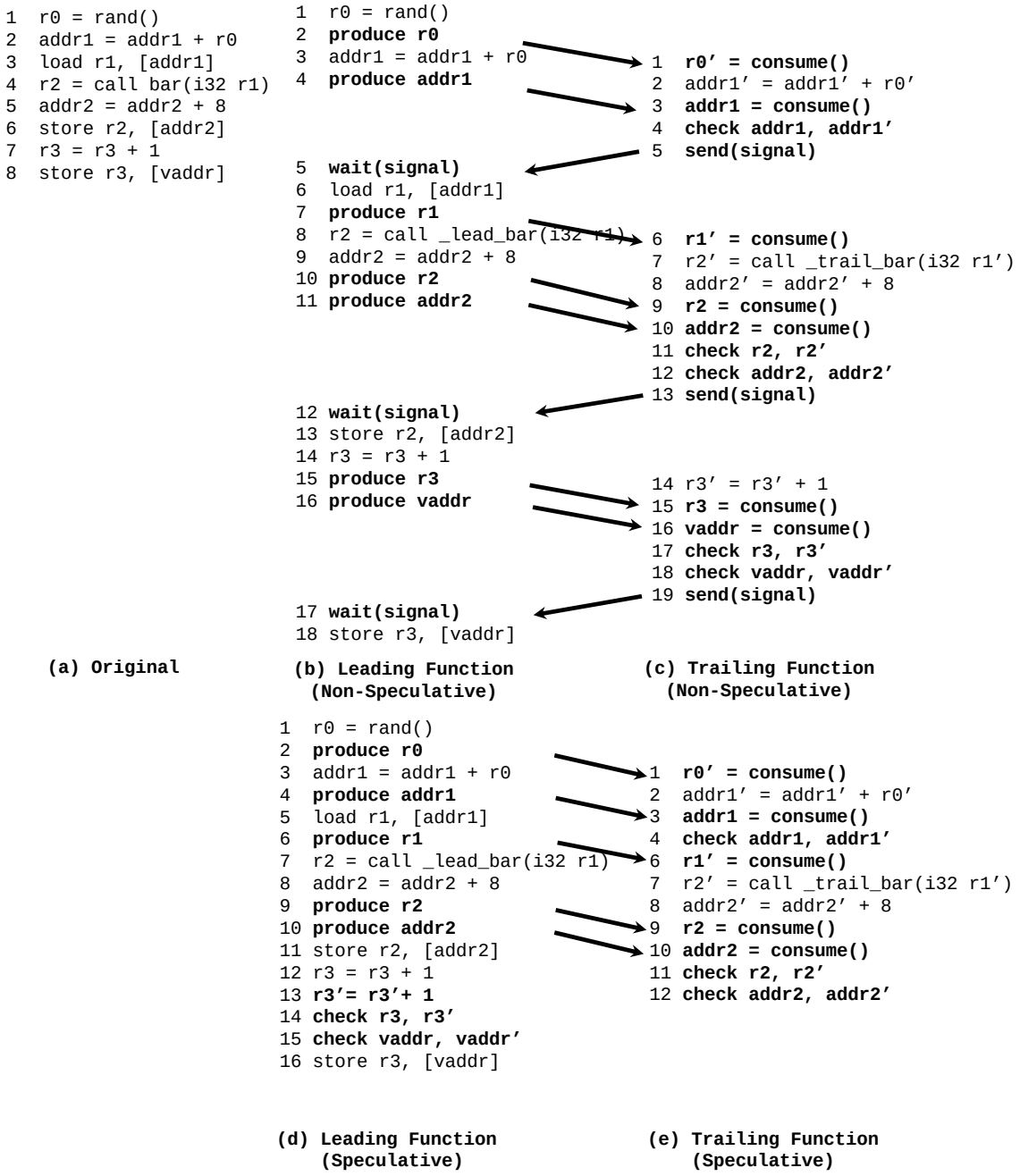

Fig. 1. Redundant multithreading with and without DAFT

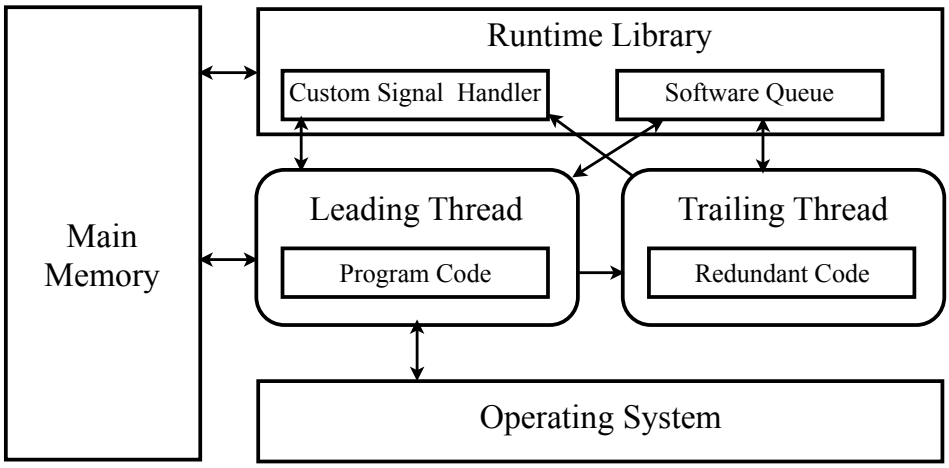

Fig. 2. Overall structure of DAFT

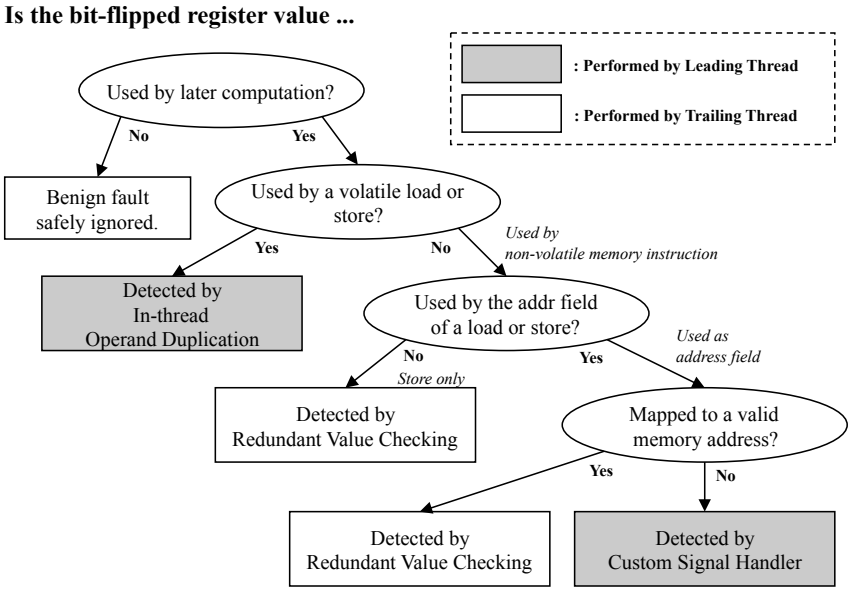

Fig. 3. Classification of possible usage scenarios of a bit-flipped register value and fault detection mechanisms in DAFT

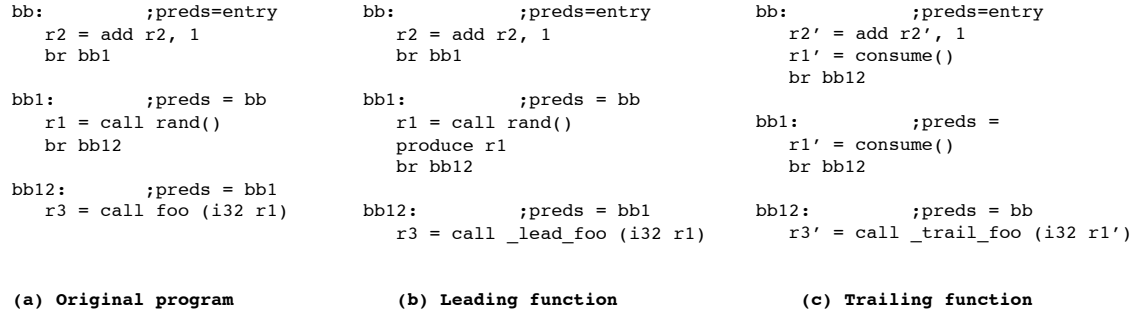

Fig. 4. Branch removal after DAFT code generation

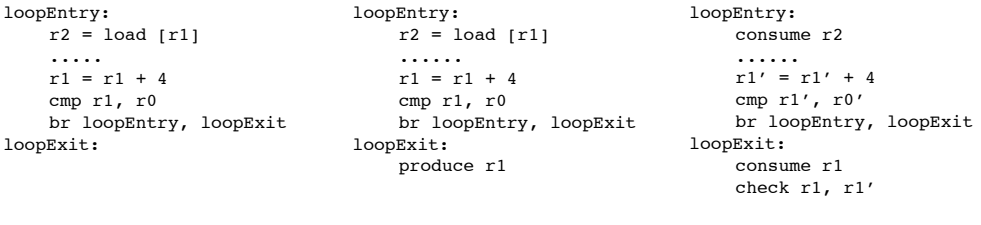

Fig. 5. Inter-thread communication lifting

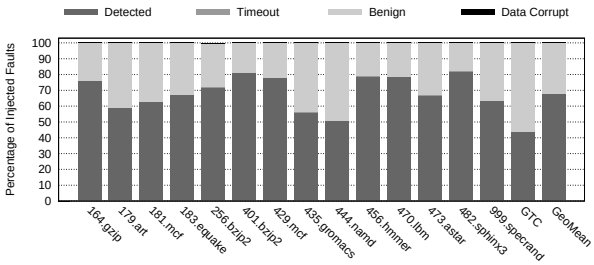

Fig. 6. Fault detection distribution

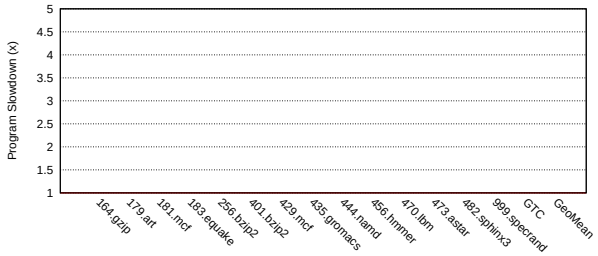

Fig. 7. Performance overhead of redundant multithreading with and without speculation

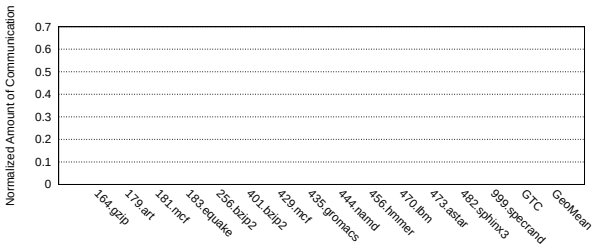

Fig. 8. Number of communication instructions (produce/consume) normalized to the total number of instructions executed

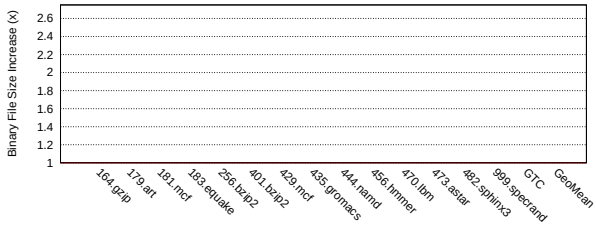

Fig. 9. DAFT-generated binary size normalized to the original binary size

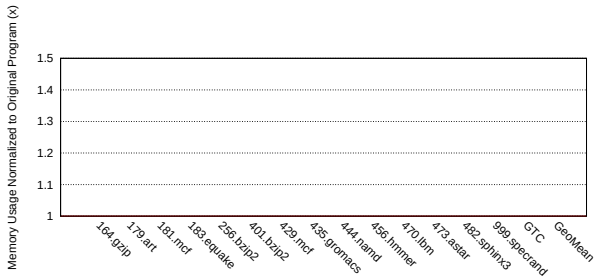

Fig. 10. Memory footprint of the DAFT-generated program normalized to that of the original unprotected program