DAFT: Decoupled Acyclic Fault Tolerance

Yun Zhang Jae W. Lee† Nick P. Johnson David I. August

Computer Science Department
Princeton University
Princeton, NJ 08540
{yunzhang, npjohnso, august}@princeton.edu

† Parakinetics Inc.
Princeton, NJ 08542
leejw@parakinetics.com

ABSTRACT

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 errors, but software transient fault detection techniques are more appealing for their low cost and flexibility. Recent software proposals double register pressure or memory usage, or are too slow in the absence of hardware extensions, preventing widespread acceptance. 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. Results demonstrate DAFT's high performance and broad fault coverage. 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.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Fault tolerance

General Terms

Reliability

Keywords

transient fault, multicore, 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. All of these technology trends make processor chips more susceptible to transient faults than ever before.

Transient faults are caused by either environmental events, such as particle strikes, or fluctuating power supply, and are nearly impossible to reproduce. Transient faults are not necessarily attributed to design flaws and occur randomly after deployment. These soft errors do not cause permanent hardware damage, but may result in a complete system failure. For example, Sun Microsystems acknowledges that customers such as America Online, eBay and Los Alamos National Labs experienced system failures caused by transient faults [8].

A typical solution for transient fault detection is through redundant computation. A program's execution is duplicated, in either hardware or software, and the results of the two instances are compared. Hardware solutions are transparent to programmers and system software, but require specialized hardware (e.g., *watchdog* processor in [7]). Real systems, such as IBM S/390 [16], Boeing 777 airplanes [20, 21], and HP's NonStop Himalaya [4] incorporate hardware transient fault detection and recovery modules. However, hardware redundant computing requires extra chip area, extra logic units, and additional hardware verification. The scope of protection and fault detection scheme are usually hardwired at design time, which limits the system's flexibility.

On the other hand, software redundancy is more flexible and much cheaper in terms of physical resources. This approach avoids expensive hardware and chip development costs. Multicore designs provide increasing parallel resources in hardware, making software redundancy solutions more viable than ever. Reinhardt et al. proposed redundant multithreading for fault detection, in which leading and trailing threads execute the program simultaneously and compare their outputs [11]. Recent implementations of software redundancy, however, double the usage of general-purpose registers [12], require specialized hardware communication queues [19], or double memory usage [15].

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 error checking instructions. DAFT speculates that transient faults do not happen so that cyclic inter-thread communications can be avoided. As a result, DAFT exhibits very low performance overhead. 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 IO addresses, are handled with special care to prevent speculative execution from triggering an externally observable side-effect.

Communication and code optimizations are then applied to further improve whole program performance. Because DAFT is a

PACT'10, September 11–15, 2010, Vienna, Austria.

software-only approach, it provides the flexibility to choose the region of a program to protect.

In short, DAFT advances the state-of-the-art in software redundant multithreading. DAFT achieves all of the following desirable properties:

- performance overhead of an average of 38% on a real multi-core 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.
- the 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, which 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 fault detection techniques rely on specialized hardware to execute redundant copies of the program for transient fault detection and recovery. Rotenberg's AR-SMT [14] is the first technique to use simultaneous multi-threading 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 chip-level redundantly threaded multi-processors for redundant execution and value checking [9]. Simultaneous Redundant Threading (SRT), proposed by Reinhardt et al., detects transient faults based on simultaneous multi-threading processors [11]. However, all these techniques rely on specialized hardware extensions.

Software redundancy using multithreading can also detect transient faults without any hardware support [3, 9, 12, 19]. SWIFT exploits unused computing power of multiple-issue processors by duplicated execution within the same thread [12]. The resulting code requires twice as many registers, potentially causing register spills. For this reason, SWIFT's overhead is low on architectures with many registers, such as the Itanium. However, instruction-level redundancy has much higher overhead on IA32 architecture having only 8 software-visible registers [13]. DAFT targets the IA32 architecture and exploits multicore to minimize runtime overhead.

Software-based Redundant Multithreading (SRMT) is a software solution that achieves redundancy with multiple threads. The SRMT compiler generates redundant code and minimizes inter-thread communication. SRMT allows the leading thread to continue on non-volatile variable accesses without waiting for verification from the trailing thread. Volatile variable accesses still need cyclic communications between original and redundant threads. However, when a real transient fault triggers an exception, SRMT invokes the program's exception handler to catch the fault, registering a false positive and possibly changing the program's behavior. Like SRMT, DAFT takes a software-only redundant multithreading approach. DAFT assumes speculatively 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 and guarantees no false positives in fault detection.

Lessons from n-version programming can also be applied to fault detection [1, 2, 10, 15]. Process-level Redundancy (PLR) presented by Shye et al. acts as a shim between user programs and the operating system [15]. Two instances of the program run simultaneously, and fault detection is performed on externally visible side effects, such as I/O operations or program termination. This approach guarantees that faults do not change the observable behavior. PLR checks fewer values, so it tends to have low overheads, yet the memory usage of PLR is at least doubled. PLR's memory footprint can be prohibitive for memory-bound applications or memory-constrained systems, such as embedded devices. Also, PLR must be applied at the program granularity; programmers and tools are not free to select critical sections of code that need protection.

Several transient fault detection techniques are summarized in Table 1. Compared with other techniques, DAFT provides broad fault coverage, presents little pressure on register files, requires no specialized hardware, and keeps memory overhead minimal.

3. DECOUPLED ACYCLIC FAULT TOLERANCE

This section presents the design of DAFT with step-by-step development, starting from a non-speculative version of redundant multithreading in Section 3.1. Section 3.2 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 3.3 addresses these challenges with three fault detection mechanisms. Finally, Section 3.4 presents several communication and code optimization techniques to make DAFT even faster.

3.1 Non-Speculative Redundant Multithreading

One question for software redundant multithreading, with or without speculation, is which instructions in the original program are replicated for redundant execution. DAFT replicates all static instructions in the original program except memory operations (i.e., loads and stores) and library function calls. 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 the possibility of intervening writes between the two loads, from an exception hander or from other threads, for example. This potentially leads to many false positives in fault detection. The situation is the same for stores. Library function calls are also excluded in cases when the DAFT compiler does not have access to the library source codes or intermediate representations.

To address this, 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, with value and memory address being checked in the trailing thread. In this way, DAFT ensures deterministic program behavior and eliminates false positives. Because the source of library functions is not

Table 1: Comparison of transient fault detection techniques

	SRT [11]	SWIFT [12]	SRMT [19]	PLR [15]	DAFT
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

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. The Sphere of Replication (SoR) [11] of DAFT is the processor core; the memory subsystem, including caches and off-chip DRAMs, is out of DAFT's SoR, as it can be protected by ECC.

The DAFT compiler replicates the instructions in the SoR into the leading and trailing threads and inserts instructions 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 and compared against the corresponding duplicate values. The result of fault checking on these values is sent back to the leading thread. The memory operation fires only if there is no fault; otherwise, the leading thread will stop execution and trap into the operating system to report a transient fault.

Redundant computation and fault checking in redundant multithreading increase static and dynamic instruction counts, which lead to significant performance overhead. Consequently, compiler optimizations should be performed before applying redundant multithreading. These pre-pass optimizations help by removing dead code and reducing the number of memory operations, leaving less code replication and less checking/communication overhead.

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. According to our evaluation, this non-speculative version of redundant multithreading shows 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 CPU2000 benchmarks with software redundant multithreading have slowed down almost by 3× solely because of an increase in the communication cost between threads [19].

3.2 Software Speculation in DAFT: Removing Cyclic Dependencies

Cyclic dependencies in the non-speculative redundant multithreading from Section 3.1 put the inter-thread communication latency on the critical path of program execution, thereby 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 makes a high-confidence target for speculation.

Inspired by Speculative Decoupled Software Pipelining (Spec-DSWP) [17], DAFT exploits such a high-confidence value speculation to break the cyclic dependencies. More 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. The performance of the program is no longer sensitive to the inter-thread communication latency. Figures 1(d) and (e) illustrate the 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 bloat.

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 problems, such as sending a wrong value to an IO 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.

3.3 Safe Misspeculation Detection

With speculation, the problem of fault detection in DAFT is effectively translated to the problem of misspeculation detection. Figure 2 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, and 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 memory. In what follows, we present three mechanisms for misspeculation detection in DAFT—in-thread operand duplication for volatile variable accesses, redundant value checking and custom exception handlers—as we walk through the scenario tree in Figure 2.

In-Thread Operand Duplication for Volatile Variable Accesses

A volatile variable is defined as a variable that may be modified in ways unknown to the implementation or have other unknown side effects. Memory-mapped IO accesses are an example of volatile variable accesses, and compiler optimizations should not reposition volatile accesses. Misspeculation on volatile variable accesses may cause an externally observable side-effect which cannot be reversed. Assuming `vaddr` in the example shown in Figure 1(a) is an IO mapped memory address, `r3` and `vaddr` must be checked for correctness (by instruction 14 and 15) before the store commits to prevent potentially catastrophic effects. However, if we become too conservative and fall back to the non-speculative solution used in Figure 1(b) and (c) with cyclic dependence and signaling, performance gains from speculation would be lost; communication latency would be put once again in the critical path.

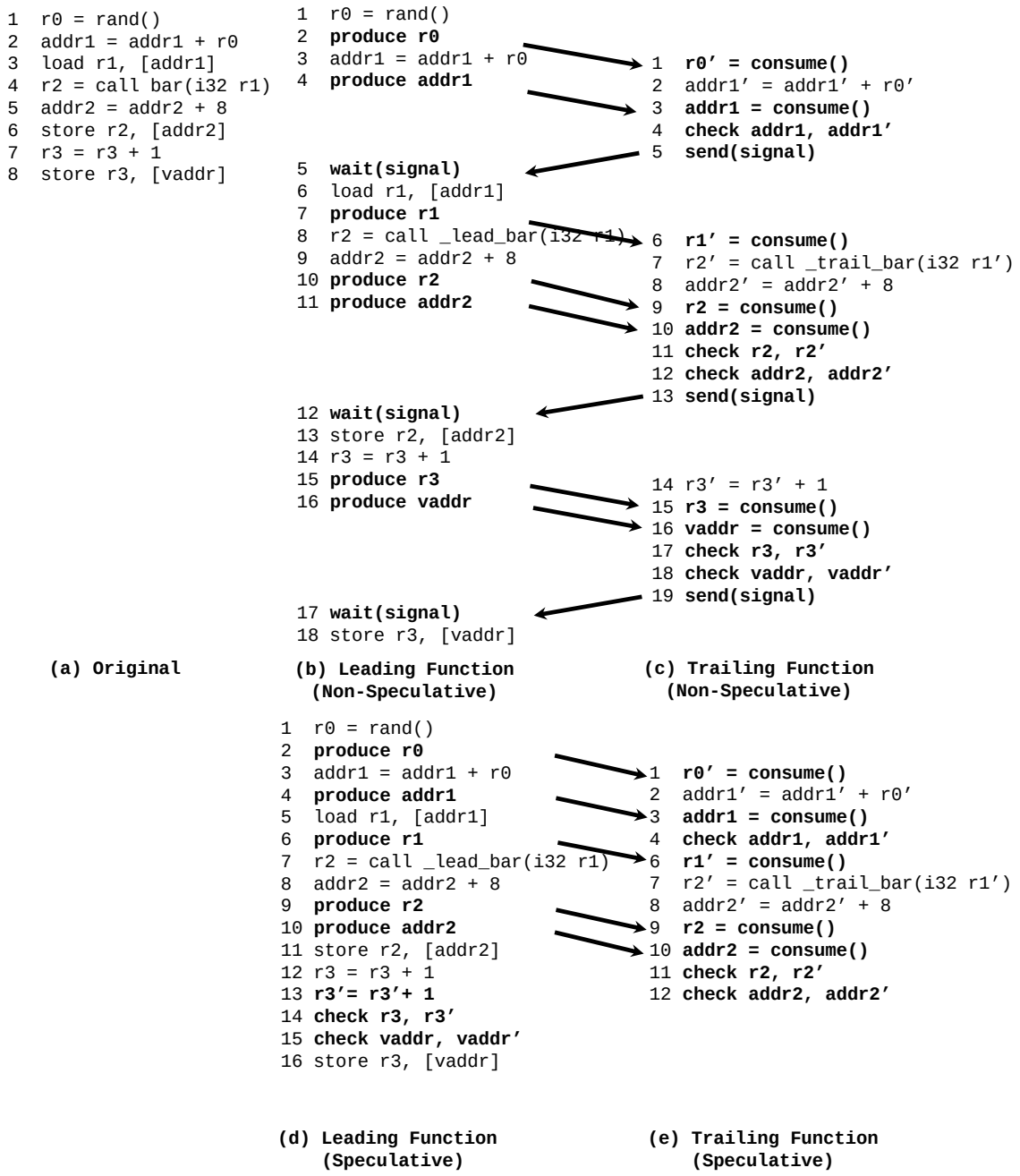

Figure 1: Program transformation with and without DAFT

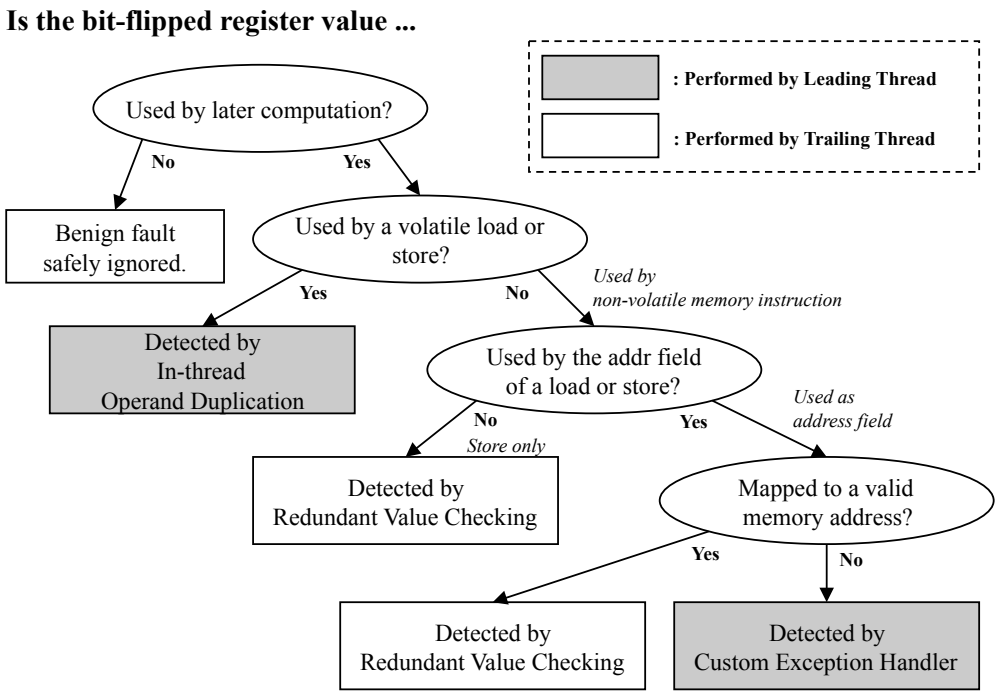

Figure 2: Classification of possible usage scenarios of a bit-flipped register value and fault detection mechanisms in DAFT

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 Figure 1(d) and (e). A code generation algorithm to handle this case is described in Section 4.

Redundant Value Checking

If a transient fault flips a bit in a register to be stored to a non-volatile memory variable later, the fault is detected through 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 of `r2` and traps the operating system if the values do not match.

Custom Exception Handler

The last scenario of a faulty value is usage as a non-volatile load or store address. Depending on whether the faulty value maps to a valid memory address or not, the fault is detected either by the redundant value checking mechanism previously discussed, or by DAFT's custom exception handler. 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 [19] relies on a system exception handler to abort the program. Unfortunately, this is not a safe solution. It changes the program behavior and cannot tell the difference between a normal program exception and a transient fault. The custom exception handler in DAFT catches all segmentation faults as shown in Figure 3. 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 a normal program exception, and the original exception 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 exception handlers.

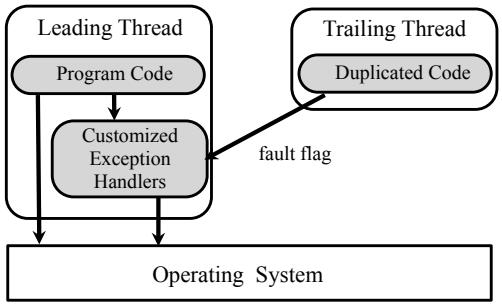

Figure 3: The exception handling diagram in DAFT

3.4 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 in whole program performance. To further 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.

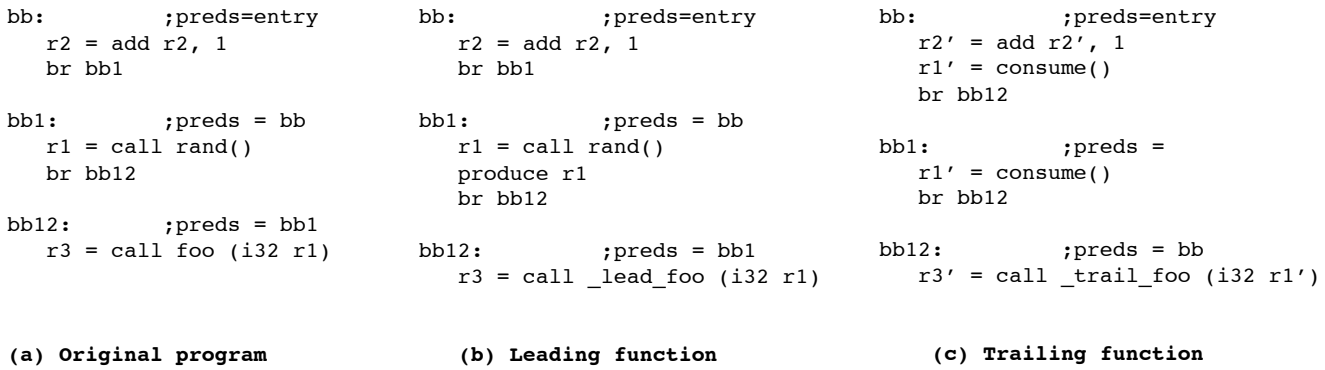

Figure 4: Branch removal after DAFT code generation

Branch Removal

Since the trailing thread does not duplicate all instructions in the original program, it may sometimes contain basic blocks which 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 work on trailing thread. In Figure 4(a), basic block `bb1` contains only one library function call and an unconditional branch to basic block `bb12`. DAFT transformation in Figures 4(b) and (c) creates a basic block `bb1` in trailing function containing only a `consume` and an unconditional branch. It is preferable to remove basic block `bb1` entirely and move the communication to basic block `bb` to avoid one unnecessary branch.

Loop Invariants 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. Since our exception handler captures segmentation faults caused by transient faults, it is safe to move the memory address check out of the loop, removing one or more communications per iteration.

Software Communication Queue

In DAFT, an unbalanced lock-free ring buffer software queue library is used for inter-thread communication. 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.

Streaming store and prefetching are enabled in the queue implementation for DAFT 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. The consumers of that store see the value in memory as soon as it is required. 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 used.

4. AUTOMATIC CODE GENERATION

DAFT is a fully automatic compiler transformation. For a given input IR, it identifies the replicable instructions, generates code for redundant computation, inserts inter-thread communication for critical values, and inserts value checking and exception handlers for fault detection. A high-level view of the algorithm is presented in Algorithm 1. The following sections explore the phases of the algorithm.

4.1 Replicable Instruction Sets

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

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

Non-replicable instructions are those which directly load from or store to memory, or are library function calls. Redundant replicable instructions are those which do not access memory, or are not calls to library functions. In-thread replicable instructions are those which compute the address or value of a volatile store. These must be handled differently than other redundant replicable instructions, since volatile stores cannot be re-ordered. DAFT replicates in-thread redundant instructions into the leading thread, whereas redundant replicable instructions are replicated into the trailing thread.

4.2 Building Redundant Program Structure

Next, we construct a new, empty function to serve as the trailing thread. Both threads must follow the same control flow pattern. However, not every basic block will perform work in the trailing thread. For efficiency, we will 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, (ii) any instruction from bb is in the non-replicable set, or (iii) there is another block bb' such that bb' is relevant to the trailing thread, and bb is transitively control dependent on bb'.

We create an empty copy of every relevant basic block in the trailing thread. Additionally, we duplicate the control-flow instruction (branch, switch, return, etc) at the end of each basic block. 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.

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

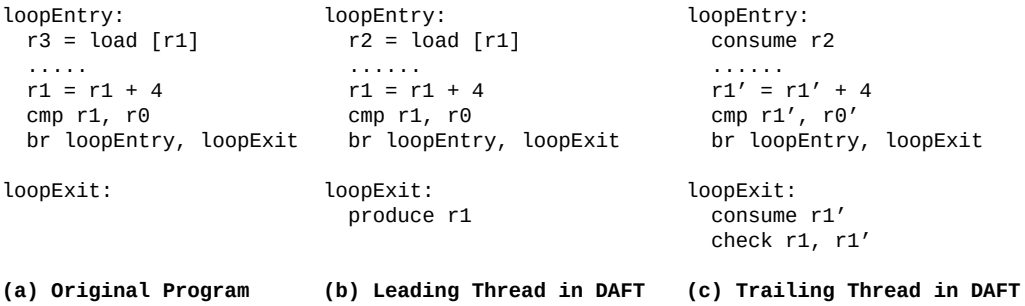

Figure 5: Communication lifting after DAFT code generation

Algorithm 1 Automatic DAFT transformation

for all Function $func \in program$ **do**
  // Building Replicable Instruction Set
  $RedundantReplicableSet = InThreadReplicableSet = NonReplicableSet = \emptyset$
  for all Instruction $inst \in func$ **do**
    if $inst$ is load or store **then**
      $NonReplicableSet = NonReplicableSet \cup \{inst\}$
      if $inst$ stores to volatile variable address **then**
        for all Instruction $prev\_inst \in DefinitionChain(inst)$ **do**
          $InThreadReplicableSet = InThreadReplicableSet \cup \{prev\_inst\}$
        end for
      end if
    else if $inst$ is register computation **then**
      $RedundantReplicableSet = RedundantReplicableSet \cup \{inst\}$
    else if $inst$ is non-library function call or indirect function call **then**
      $RedundantReplicableSet = RedundantReplicableSet \cup \{inst\}$
    else
      $NonReplicableSet = NonReplicableSet \cup \{inst\}$
    end if
  end for
  $RedundantReplicableSet = RedundantReplicableSet - InThreadReplicableSet$

  // Identify and Clone Relevant Basic Blocks
  $RelevantBasicBlocks = \emptyset$
  for all Basic block $bb \in func$ **do**
    if bb is relevant to Trailing thread **then**
      $RelevantBasicBlocks = RelevantBasicBlocks \cup \{bb\}$
    end if
  end for

  // Replication and Communications
  for all Instruction $inst \in func$ **do**
    if $inst \in InThreadReplicableSet$ **then**
      $inst.clone() \rightarrow leadingFunction$
      insertFaultChecking($inst$) $\rightarrow leadingFunction$
    else if $inst \in RedundantReplicableSet$ **then**
      $inst.clone() \rightarrow trailingFunction$
    else if $inst \in NonReplicableSet$ **then**
      insertCommunication($inst$)
      insertFaultChecking($inst$) $\rightarrow trailingFunction$
    end if
  end for
end for

conditions, that value must be communicated to the trailing thread with an additional produce-consume pair.

4.3 Code Replication and Communication Insertion

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 location.

For a memory load instruction, a produce-consume pair is created for the memory address. Similarly, two produce-consume pairs are created for `store` instruction for communicating value and address, respectively. Before each binary function call, each argument which is passed via register is produced to the trailing thread for correctness checking. If a binary 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.

4.4 Customized Signal Handlers

It is possible that a misspeculated fault can lead to an exception before the trailing thread detects the fault through redundant checking. However, it is also possible that the original program to throw the same exception during its execution, with or without a transient fault. We must distinguish between these two cases. DAFT achieves this by creating custom signal handlers for each application.

Specialized exceptions handlers are added in order to distinguish between transient faults and normal faults. These signal handlers are registered (via `sigaction()`) at the beginning of program execution.

When a signal is caught, the custom signal handler is invoked. The signal handler waits for the trailing thread to raise a red flag on fault. If no fault is reported before timeout occurs, the signal handler assumes that this is a normal exception and calls the corresponding system exception handler.

4.5 Indirect Function Call

For indirect function calls through function pointers, the compiler cannot tell which function will be called. Therefore, the leading version of the callee function is always invoked. Such calls originate either in the leading thread or the trailing thread; an extra flag is added to distinguish these cases. If the function is called from the trailing thread, the leading thread function will serve as a trampoline and invoke the corresponding trail version of the function. For the leading thread, that is not very computationally expensive: only one long jump is made at each call. For the trailing thread, the trampoline in the leading thread is used to invoke the correct trailing function. If a wrapper function is used for indirect calls, each function call in both threads will have to go through two calls which increase runtime overhead. Function calls from libraries are also possible if the compiler understands the calling convention.

4.6 Example Walk-through

A sample program 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*.

Table 2: Replicability of instructions in Figure 1(a)

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

A tricky case is volatile memory access. In this example, at first instruction 7 in Figure 1(a) is a regular computation and therefore redundantly replicable. But as soon as instruction 8 is scanned, we realize that `r3` is stored as a volatile variable. At this point, the def-use chain of `r3` is computed. Instruction 7 is then removed from redundant replicable set and inserted into the in-thread replicable set. This is why replicable instruction sets must built before code duplication and communication insertion. Such information is stored in a data structure similar to Table 2.

Once the instructions are classified, the code is scanned for code duplication. All redundant thread replicable and in-thread replicable instructions are copied to the trailing and leading functions, respectively. Since branch and function call instructions are all redundant thread 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 come into or escape from the SoR. In this example, `load` instruction 3 in Figure 1(a) escapes 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 `store` instruction 6, 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 binary function call, 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. On the contrary, non-binary function call such as instruction 4 needs 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 [5]. DAFT uses fast, lock-free software queues with streaming write and prefetching for inter-thread communication.

5.1 Reliability Analysis

To measure the fault coverage of DAFT, Intel's PIN instrumentation tool [6] is used to inject single bit flip faults. We first perform a profile run of the program to count the number of dynamic instructions. In the simulations, no fault is injected into the standard

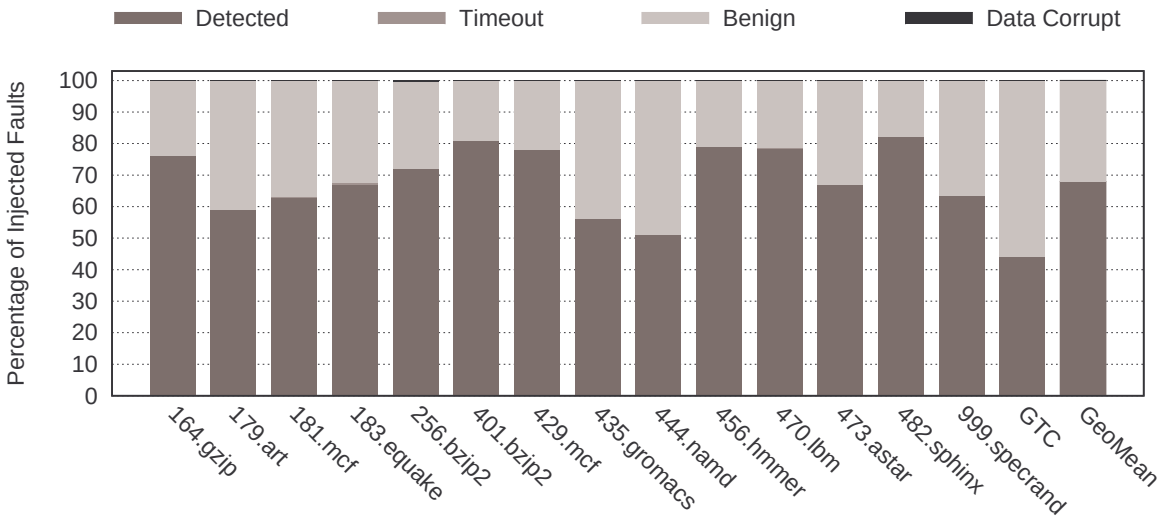

Figure 6: Fault detection distribution

C libraries, since they are not compiled with DAFT and therefore lack transient fault protection. One dynamic instruction is selected randomly. One register is again selected randomly among general-purpose registers, floating point registers, and predicate registers. PIN uses IARG_RETURN_REGS to flip a random bit of the selected register after the selected instruction. Execution is compared against the reference output to ensure that externally visible behavior is unchanged. Single-event-upset (SEU) model is assumed in the evaluation [11, 13, 18]. Each benchmark program is executed 5000 times, with one transient fault injected in each run.

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 a 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 out 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) 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 (value replication), DAFT is not able to detect the fault (represented as *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.

5.2 Performance

DAFT is evaluated on a real multi-core processor. Performance results are shown in Figure 7. The runtime overhead is normalized to the original sequential program without any fault protection. We compare our performance with a software redundant multithreading implementation that does not employ speculation.

The performance overhead of DAFT is 38% (or 1.38×) on average. Compared with redundant multithreading without speculation, DAFT is 2.17× faster. Previous software solutions, such as SRMT [19], 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.

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 load and stores, but the number of memory operations is higher than other benchmark programs. More memory operations means more communications and redundant value checking, which translates 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 one for the memory address. Figure 8 indicates the number of communications (linear in the number of values we pass through software queue) normalized to the number of total instructions in a program.

Figure 9 shows the static size of the binary generated by DAFT normalized to the original program without protection. The code size of DAFT is about 2.4× larger than the baseline program. This is because every register computation instruction is duplicated into two. One produce-consume pair is created for each load, and two pairs are added for each store. Compared with previous work,

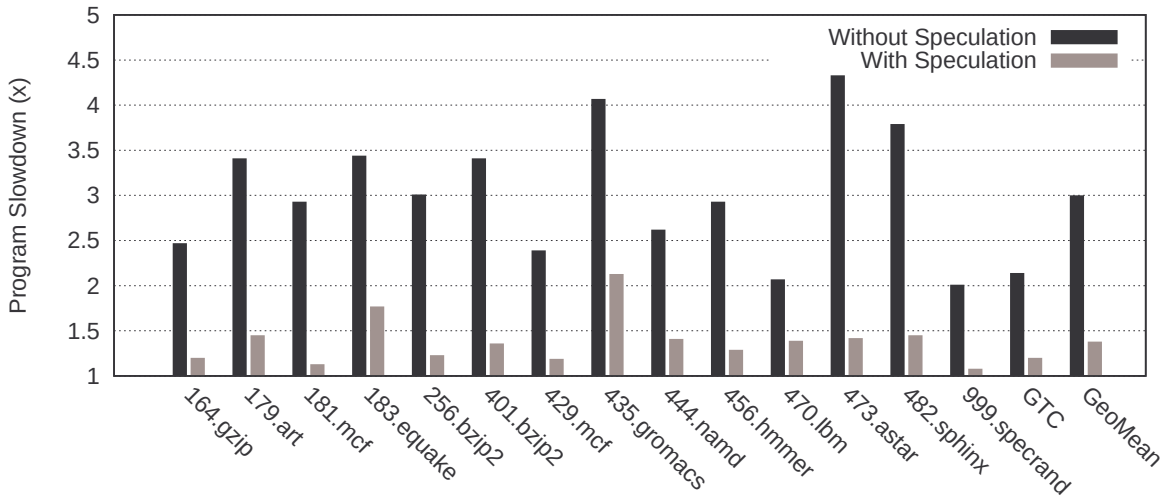

Figure 7: Performance overhead of redundant multithreading with and without speculation

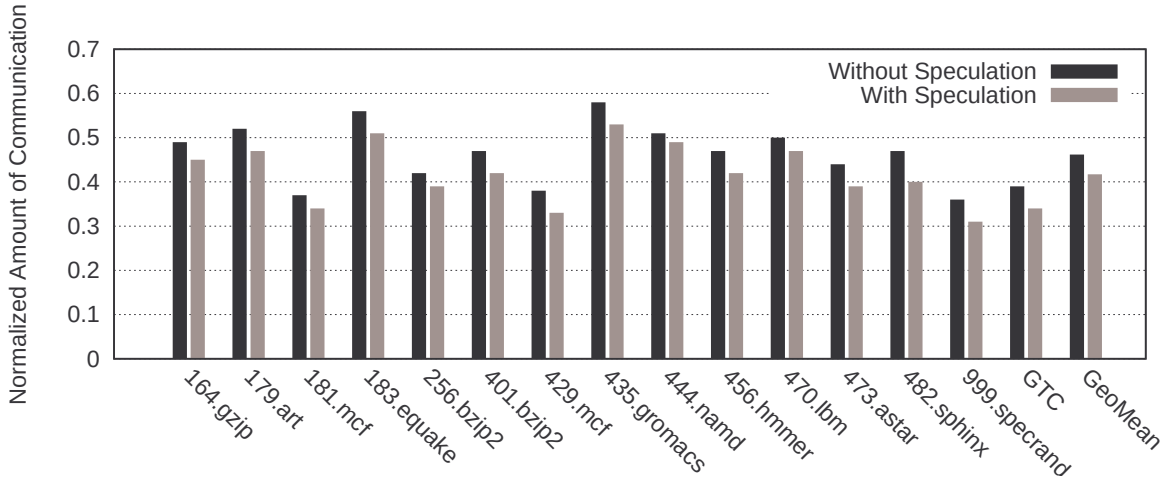

Figure 8: Number of communication instructions (produce/consume) normalized to the total number of instructions executed

DAFT has a similar increase in binary size, yet lower runtime overhead, due to the fact that DAFT utilizes multi-core for execution.

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 redundancy for fault detection.

This paper presents a fast, safe and memory-efficient redundant multithreading technique. By combining speculation, customized fault handlers, and intelligent communication schemes, DAFT provides advanced fault detection on off-the-shelf commodity hardware. It features minimal runtime overhead and no memory bloat. Unlike some of previous software solutions, DAFT correctly handles exceptions and differentiates program exceptions from transient faults. DAFT can provide reliability for mission-critical systems without any specialized hardware or memory space usage explosion.

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.

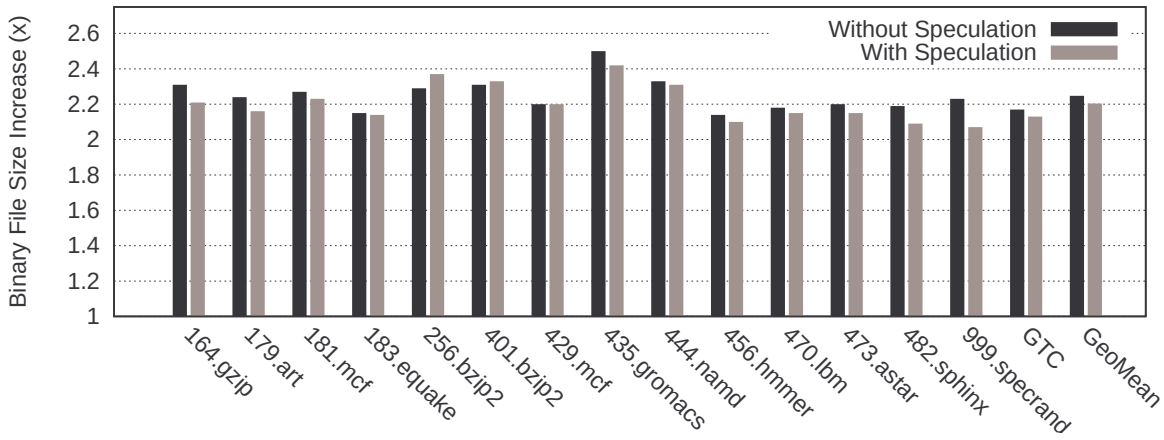

Figure 9: DAFT-generated binary size normalized to the original binary size

7. REFERENCES

[1] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN '06 Conference on Programming Language Design and Implementation*, June 2006.

[2] 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.

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

[4] R. W. Horst, R. L. Harris, and R. L. Jardine. Multiple instruction issue in the NonStop Cyclone processor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 216–226, May 1990.

[5] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[6] 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 *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[7] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.

[8] 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.

[9] 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.

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

[11] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36. ACM Press, 2000.

[12] 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.

[13] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, pages 148–159, June 2005.

[14] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 84. IEEE Computer Society, 1999.

[15] A. Shye, T. Moseley, V. J. Reddi, J. B. t, and D. A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. *Dependable Systems and Networks, International Conference on*, 0:297–306, 2007.

[16] 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. In *IEEE Micro*, volume 19, pages 12–23, March 1999.

[17] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[18] 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.

[19] C. Wang, H.-S. Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society.

[20] Y. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, volume 1, pages 293–307, February 1996.

[21] Y. Yeh. Design considerations in Boeing 777 fly-by-wire computers. In *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium*, pages 64 – 72, November 1998.