COMET: Communication-Optimised Multi-threaded Error-detection Technique

Konstantina Mitropoulou[†], Vasileios Porpodas[‡1] and Timothy M. Jones[†]
[†]Computer Laboratory, University of Cambridge, UK
[‡]Intel, USA
konstantina.mitropoulou@cl.cam.ac.uk

ABSTRACT

Relentless technology scaling has made transistors more vulnerable to soft, or transient, errors. To keep systems robust against these, current error detection techniques use different types of redundancy at the hardware or the software level. A consequence of these additional protection mechanisms is that these systems tend to become slower. In particular, software error-detection techniques degrade performance considerably, limiting their uptake.

This paper focuses on software redundant multi-threading error detection, a compiler-based technique that makes use of redundant cores within a multi-core system to perform error checking. Implementations of this scheme feature two threads that execute almost the same code: the main thread runs the original code and the checker thread executes code to verify the correctness of the original. The main thread communicates the values that require checking to the checker thread to use in its comparisons.

We identify a major performance bottleneck in existing schemes: poorly performing inter-core communication and the generated code associated with it. Our study shows this is a major performance impediment within existing techniques since the two threads require extremely fine-grained communication, on the order of every few instructions. We alleviate this bottleneck with a series of code generation optimisations at the compiler level. We propose COMET (Communication-Optimised Multi-threaded Error-detection Technique), which improves performance across the NAS parallel benchmarks by 31.4% (on average) compared to the state-of-the-art, without affecting fault-coverage.

CCS Concepts

•Software and its engineering → Source code generation; •Computer systems organization → Reliability;

Keywords

Error Detection, Soft Errors, Communication Optimisations, Code Generation

[1]Work performed whilst at the University of Cambridge.

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

1. INTRODUCTION

Transient errors [28] are faults that occur once and do not persist. Studies [7, 18, 26, 29] show that these errors become more frequent as we move to smaller transistor technologies and lower voltage thresholds. Transient errors usually cause bit-flips within hardware structures and a common design strategy to detect them is to use redundancy, which can take several forms, e.g., hardware [5, 8], instructions [6, 9, 22, 24, 30, 32], threads [21, 23, 25] or processes [27, 31].

Instruction-level redundancy may be the preferred option for several reasons. First, it is more flexible and cheaper than hardware solutions, since it can be applied after recompilation of the workloads on any existing system. Second, it is less prone to false positives as it only detects errors that affect the application's output. Finally, the designer can choose the program region that needs protection, rather than the whole application.

In this form of redundancy, program instructions are replicated and checks are inserted wherever they are needed. The checks compare the outputs of the original and the replicated instructions. If the outputs are identical, then there is no error, otherwise the execution of the program has to stop and fall back to the last checkpoint, where the execution of the program resumes. It is common practice for the error detection schemes that are based on redundancy to protect only the processor core [21, 23, 24, 30, 32]. The reason is that memory has its own protection systems, such as error correcting code (ECC) or parity checking.

There are two flavors of instruction-level error detection. *Thread-local* error detection [6, 9, 22, 24]: here, the replicated instructions and the checking code are emitted in the same thread as the original instructions. *Redundant multi-threading* error detection [21, 23, 25, 30, 32]: the checker code is placed in a different thread (named the checker thread). The main thread (original code) has to send the values that should be checked over to the checker thread.

With the abundance of multi-core systems and diminishing returns from wide-issue superscalar processors, redundant multi-threaded approaches are generally preferred over thread-local schemes. The reason is that thread-local error detection increases the code size considerably and commonly creates hardware congestion. On the other hand, redundant multi-threading removes the overhead of the checking code off the critical path and places it into a separate thread (the checker thread). Minimising the performance impact of error-detection is very important, considering that transient errors are rare.

This work analyses the overheads of software redundant

multi-threading. A common design paradigm for these techniques relies on very frequent inter-core communication (every few instructions) over a software queue, in order to pass the values produced by the main thread to be checked by the checker thread. Our analysis shows that the performance of these techniques is very sensitive to the quality of the generated code that performs the inter-core communication. The main overhead comes from the fact that the data has to travel from the L1 cache of one core, through some part of the memory hierarchy, all the way to the L1 cache of the other core. Thus, a naïve implementation results in a series of problems, such as cache ping-pong, false-sharing and cache-thrashing. Naïve implementations make use of software queues within the literature that were designed for infrequent communication [10, 13]. More advanced implementations make use of multi-section queues [12, 14, 15, 30] that perform better for such high throughput and frequent inter-core communication.

In this work, we highlight the performance bottlenecks of software redundant multi-threading and, in response, propose COMET, a faster error detection solution based on optimised code generation for inter-core communication. COMET introduces faster inter-core communication than the state-of-the-art by eliminating the overheads of complicated control-flow and by reducing the instructions used for transferring data to be checked down to a single instruction, close to a theoretically optimal implementation. Using COMET, performance overheads are reduced by 31.4%, compared to a state-of-the-art technique.

The rest of the paper is organised as follows. Section 2 describes the different types of software queues, then section 3 analyses the overheads of software redundant multi-threading. Section 4 presents our proposed communication and compiler optimisations. Next, section 5 describes the error detection algorithm. Finally, the performance and fault-coverage results are explained in section 6.

2. SOFTWARE REDUNDANT MULTI-THREADED ERROR DETECTION

Software redundant multi-threading is a compiler-based technique that uses a redundant thread for code replication and checking (examples include DAFT [32] and SRMT [30]). Figure 1 shows how we get from some original pseudocode to the protected dual-threaded code. The original instructions are shown in blue, while the replicated instructions are shown in red. The check instructions are shown in grey. Their job is to compare the values that the checker thread computes against the values computed by the main thread. As the instructions execute on separate threads on the target machine, the data has to be sent through a software queue. The instructions that send or receive instructions through the queue are shown in green. It is worth noting that the data transfers are unidirectional from the main thread to the checker thread (as in existing work [30, 32]). This allows the two threads to operate decoupled from one another, achieving higher performance. For the best performance the threads (main and checker) are mapped to separate physical cores, and not to logical cores within a processor supporting simultaneous multi-threading (SMT), as demonstrated by Wang et al. [30].

It is common design practice to insert the checks before memory instructions [21, 23, 24, 30, 32]. The reason is that

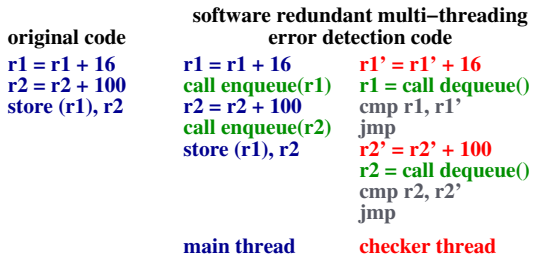

Figure 1: Transformation of the code to apply the software redundant multi-threading error detection scheme.

error detection focuses on detecting transient errors in the processor and not in the memory. Therefore, checks are added before load and store instructions in order to make sure that the memory address and the data is error-free. In figure 1, the address (in r2) and the data (r1) for the store instruction are checked. The main thread sends these values over to the checker thread which checks if the values received match the ones computed locally (r2' and r1' respectively). It is worth noting that all sate-of-the-art redundant multi-threading techniques [30, 32] lack perfect fail-stop functionality, as a trade-off in favour of higher performance. In order to minimise the risk of propagating errors to I/O, these techniques enhance checking before volatile stores by either synchronising with the checker thread [30], or by performing the check in-thread [32].

As expected, performance degrades as we introduce new instructions into the code. The more memory instructions in a program, the more the checks and communication instructions are inserted. If we want to maintain a given level of error detection, we cannot optimise by reducing the number of checking instructions. Thus, the performance overhead of the software redundant multi-threading error detection scheme is largely dependent on the efficiency of the code generated for inter-core communication. This problem has been addressed to some degree by others [30, 32] with the use of a multi-section lock-free single-producer/single-consumer (SP/SC) queue [12] which performs well in frequent-communication scenarios.

In this work we propose a code generation scheme that reduces the overheads incurred by the inter-core communication instructions even . In order, though, to compare against existing solutions ([30, 32]), we briefly describe how the existing inter-core communication works.

The main characteristic of a multi-section queue (MSQ) is that both the producer and the consumer can enqueue / dequeue data to/from the queue at the same time, but within different sections of the queue. This allows queue synchronisation checks (necessary to ensure that there is space for the producer to write to and data for the consumer to read) to execute much less frequently than normal: once per section, instead of at every access. These synchronisation checks ensure that only one thread can access a section at a time. For example, if the enqueue pointer reaches the end of the first section and the other thread has not finished with the second section, then the first thread cannot enter the second section and it waits until the other thread finishes. Development of MSQ realised a big code generation improvement over previous designs. Lamport's queue [13] requires each thread to execute a check upon every queue access. This synchronisation overhead is prohibitive for high-

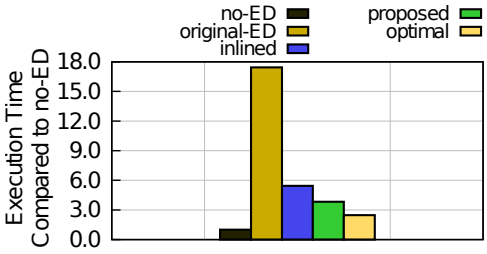

Figure 2: The performance of different optimisations on software redundant multi-threading. The overheads are normalised to code without error detection (no-ED).

throughput fine-grained communication scenarios, such as redundant multi-threading error detection. For this reason, MSQ has been the queue of choice for software error detection techniques.

3. ANALYSIS OF COMMUNICATION OVERHEADS

In the previous section we motivated why multi-threaded error detection needs high performing code for the inter-core communication aspect of the technique. In this section, we show how we may improve the quality of code generation for the communication code even further.

3.1 Function Call Overhead

In section 2, it was shown that enqueue and dequeue operations are performed as frequently as the memory accesses in the original program. Considering that the execution of a call is expensive, frequent execution of enqueue calls can dramatically increase the execution time of the main thread. Therefore, a straight-forward optimisation is to inline the enqueue and dequeue functions. This substantially reduces the overhead of accessing the queue. For example, in matrix, (a matrix multiplication benchmark from the LLVM test-suite [2]), the performance gain due to the inlining of enqueue/dequeue functions is 3.21×, as shown in figure 2. The dark yellow bar shows the overhead of the error detection code with calls (17.44×) and the blue bar shows the overhead with function inlining (5.44×). Matrix is a well-known memory intensive benchmark and for this reason we consider it to be one of the workloads that stresses multi-threaded error detection the most. Even though inlining reduced the overhead considerably, it is still very significant (5.44×) once we compare it against the code without error detection. Therefore more code tuning is required.

3.2 Control-Flow Overhead

Each time the code of a queue function is inlined, a significant number of instructions with non-trivial control-flow are added into the code. Listing 1 shows the source code of enqueue and dequeue functions and figure 3 shows their control-flow. Lines 2 and 3 of listing 1 describe the actual enqueue process which consists of a store of the data into the queue and then an increment of the enqueue pointer. This code is in the first basic-block of figure 3(a). The rest of the code (listing 1, lines 5–9, and figure 3(a), basic-blocks 2–6) show how synchronisation between the main thread and the checker thread works.

In figure 1, the main thread's code is split into new basic-blocks twice: once before the first enqueue call and once

```
1  void enqueue (queue_t q, long data) {
2    *q->enqPtr = data;
3    q->enqPtr = (q->enqPtr + 8) & ROTATE_MASK;
4    /* Synchronisation */
5    if ((q->enqPtr & SECTION_MASK) == 0) {
6      while (q->enqPtr == q->deqLocalPtr) {
7        q->deqLocalPtr = q->deqSharedPtr;
8      }
9      q->enqSharedPtr = q->enqPtr;
10   }
11 }
12
13 long dequeue(queue_t q) {
14   /* Synchronisation */
15   if ((q->deqPtr & SECTION_MASK) == 0) {
16     q->deqSharedPtr = q->deqPtr;
17     while (q->deqPtr == q->enqLocalPtr) {
18       q->enqLocalPtr = q->enqSharedPtr;
19     }
20   }
21   long data = *((long *)q->deqPtr);
22   q->deqPtr = (q->deqPtr + 8) & ROTATE_MASK);
23   return data;
24 }
```

Listing 1: Enqueue and dequeue functions for MSQ [30].

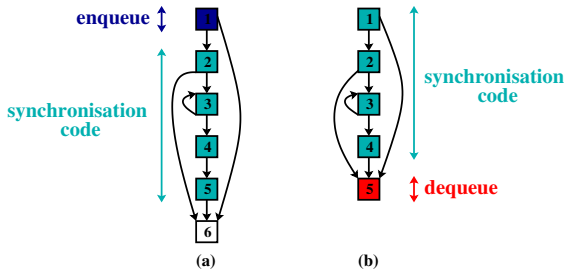

Figure 3: The basic-blocks and the control-flow of (a) enqueue and (b) dequeue functions.

more before the second enqueue call. Following this, inlining each call to a function replaces one basic-block with five new ones (basic-block 6 in figure 3 corresponds to the block after the call). Therefore, after inlining, the structure of the original code changes completely as the basic-blocks are split and new ones added. In fact, the original single basic-block gets turned into code containing 11 basic-blocks.

This increase in control-flow complexity has several problems associated with it. First, it makes instruction scheduling problematic, as basic-blocks act as scheduling barriers even for powerful inter-block schedulers [11, 16, 17]; for example, hoisting instructions with side-effects is prohibited. Second, it makes the live ranges of registers longer and therefore it increases register pressure. As a result, the number of register spills increases significantly. Finally, the job of the out-of-order execution engine also becomes harder as it needs to be able to predict and extract ILP out of many outstanding branches. Figure 4 shows the number of branch predictor misses for the different approaches. These were gathered by running perf [4] while executing matrix on a quad-core Intel Core i5-4570. The branch misses for the inlined code (blue bar) are 12.85× more than the branch misses within the original code.

3.3 Optimality

To sum up, we have shown that the synchronisation code of the enqueue/dequeue operations is a huge overhead as

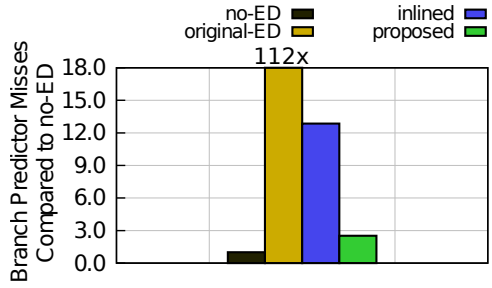

Figure 4: The impact of the different error detection approaches on the processor's branch predictor. The number of branch predictor misses is normalised to the number of misses for the code without error detection (no-ED).

it introduces complex control-flow in regions of code which used to be a lot more simple. In addition, a large number of the added instructions are not executed the vast majority of times that the code is run. An ideal enqueue function with no synchronisation code would consist of only two instructions: the store of the data to the queue and the increment of the enqueue pointer. In theory, this may be achieved with a queue of infinite size. In figure 2 we simulate such an "infinite" queue using a queue that is large enough (several gigabytes) so that the indices do not reach its end during the entire execution of the benchmark. The light yellow bar shows the overhead of this technique (optimal) for matrix. In section 4, we describe how we actually realise this idea to radically improve code generation for error detection.

4. COMMUNICATION OPTIMISATIONS

Our proposed solution for increasing the performance of software multi-threaded error detection, named COMET, avoids generating queue synchronisation instructions altogether. The code generation is inspired by a software-level exception-based lock-free multi-section SP/SC queue [20]. We adapt this idea to the context of compiler-generated error-detection. The design of the COMET queue is considerably simpler compared to the original, due to the fact that we have full control over the generated coded at the compiler level. Furthermore, COMET implements a series of code generation optimisations to achieve peak performance in this context.

4.1 Removal of Synchronisation Instructions

The main novelty of Lynx [20] is in implementing an SP/SC queue using enqueue/dequeue operations with just two instructions' overhead. This is accomplished by exploiting the memory protection system that both commodity processors and operating systems support. Each queue section is followed by a non-readable and non-writable memory-protected zone, referred to as a red-zone (see figure 5(a)), with a further additional red-zone at the end of the queue. The red-zones after each section are called section synchronisation red zones (SSRZ), whereas the final zone at the end of the queue is called a pointer rotation red zone (PRRZ).

Each red-zone is the size of a memory page. The idea is that when the enqueue/dequeue pointer reaches the red-zone at the end of the section (figure 5(b)), a segmentation fault signal (SIGSEGV) is raised. The segmentation fault is captured by a custom exception handler which is where synchronisation takes place, off the critical path of execution.

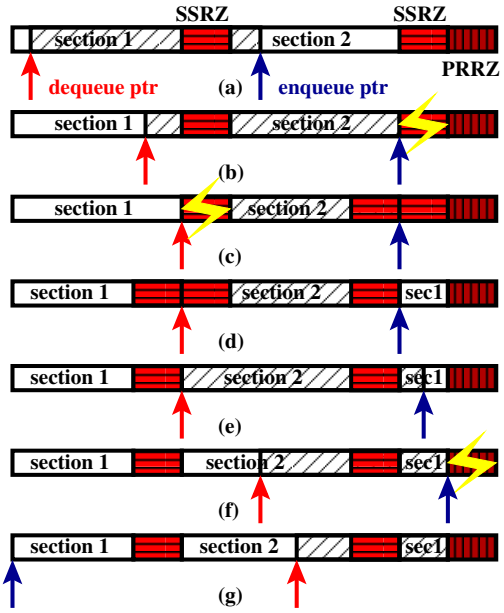

Figure 5: An example of the Lynx queue [20]. Section synchronisation red-zones guarantee only one thread can access a section at a time, and move towards the start of the queue. The pointer rotation red-zone is fixed and rotates the pointer when it reaches the end of the queue.

The handler, therefore, decides whether and when a thread is allowed to proceed to the next section.

In the example, shown in figure 5(b), the enqueue pointer cannot enter the next section because it is still in use by the other thread. Therefore, it has to wait in a spin-loop in the exception handler. In Lynx, the red-zones physically move left to lower addresses in a circular manner, as shown in figures 5(b)–(d). Once the dequeue pointer (figure 5(c)) reaches the end of its section, the enqueue thread is allowed to proceed to the next section (figure 5(d)). Similarly, the dequeue thread enters the other free section (figure 5(e)).

So far, we have described SSRZs, which move backwards across the queue; in contrast the PPRZ remains at the end of the queue in a fixed location. Its role is to trigger the rotation of the enqueue/dequeue pointer back to the beginning of the queue. When the enqueue pointer accesses the PRRZ (figure 5(f)), it is modified within the exception handler to point to the beginning of the queue again (figure 5(g)).

4.2 COMET's Optimised Queue Design

This existing queue design is complicated and sub-optimal due to the fact that it is coded at the source level. COMET, on the other hand, generates communication code within the compiler and has full control over it. Therefore, the queue in COMET is a lot simpler and further optimised, as shown in figure 6. The queue design is still based on the idea of protected memory regions (red-zones), but its operation is simpler and more efficient. This is because there are only two red-zones (A and B in figure 6) and they are fixed (i.e., they do not move). The red-zones no longer need to rotate, saving kernel execution overheads and simplifying the handler's code. Second, there is no need for a pointer rotation red-zone at the end of the queue since red-zone B is used for both synchronisation and pointer rotation.

In our example, when the enqueue thread accesses red-

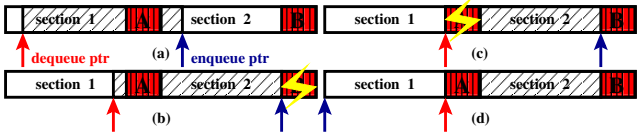

Figure 6: An example of COMET's queue.

zone B, an exception is triggered (figure 6(b)). The dequeue thread still reads data from the first section, therefore the enqueue thread has to wait in a spin-loop in COMET's handler. In figure 6(c), the dequeue thread reaches the first red-zone. At this point the handler identifies that the first section is free and the enqueue pointer is rotated to the beginning of the queue (figure 6(d)).

COMET's handler has to perform three tasks upon trapping a SIGSEGV signal. First, it determines which thread raised the signal, in order to identify whether it is an enqueue or dequeue operation. Second, it takes care of thread synchronisation. Third, if the pointer is at the end of the queue, then the handler rotates it round to the beginning of the queue.

4.3 Pointer Update Using a Fixed Register

As mentioned earlier, one of the signal handler's tasks is to update the queue index pointers to point to the next section. For this to happen, the exception handler needs to update the value of the address register from the instruction that triggered the segmentation fault. The instruction that triggers the exception is either a store into the queue, in the case of an enqueue, or a load from the queue, in the case of a dequeue. The compiler is free to assign any available register to the address operand of these memory instructions, while performing register allocation. Therefore, determining the register that corresponds to the queue pointer is non-trivial within the exception handler, as it requires parsing the instruction's opcode. This is a task that the handler has to perform to enable the optimised queue. More importantly, the compiler may re-use this register for other tasks, such as an induction register within a loop. Therefore an update to the pointer register (for the purposes of the queue) may have side-effects on the execution of other, unrelated parts of the code, breaking the code semantics. This suggests that the queue pointer registers must be assigned to hard registers right at the point where the code gets generated, otherwise the implementation is infeasible.

COMET uses a predetermined hard register as the address register for the store and load instructions of the enqueue and dequeue operations. On the x86_64 architecture, we use hard register r15, the last general purpose register of this architecture, for both threads. This simplifies the whole design, allowing the handler to have full access to the register, without having to consider cases where that would break program execution. This is exactly the reason why a simple queue design, such as that shown in figure 6, is feasible in COMET but not general-purpose enough for Lynx [20].

Upon receiving a SIGSEGV signal, COMET's exception handler determines which thread raised the exception (the main or the checker) and then reads the value of r15 to determine the location of the enqueue or dequeue pointer, whichever of the two triggered the exception. If the value of r15 is the address at the beginning of red-zone A and section 2 is free, then the handler will update r15 to point to the beginning of section 2. Similarly, if the exception is generated by accessing red-zone B and section 1 is free, then the handler will move the r15 pointer to the beginning of the queue (figure 6(d)).

To conclude, the fact that COMET generates the queue code at compile-time allows us to optimise the queue design as compared to Lynx. It also allows for several further optimisations at the compiler-level, which are discussed in the following sections.

4.4 Register Promotion Optimisation

The code of the enqueue function for the optimised queue design consists of four instructions: One to load the value of r15 from memory (obtain the enqueue pointer), another to store the data to the address in r15 (perform the enqueue). A third instruction increments r15 (moves the enqueue pointer to the next position in the queue), and finally, a fourth to store r15 back to memory (save the value of the enqueue pointer).

Therefore, if we naïvely inline the code of the enqueue function, then we will emit four instructions to replace each function call. Even though inlining these four instructions is faster compared to a function call, we can do better. The loading and re-storing of r15 from and to memory is redundant across the body of a function. We optimise the code by loading r15 from memory only at the beginning the function and storing its value back to memory at the end of the function. Since function calls may clobber r15, we have to make sure that we maintain its value by saving it to a dedicated memory location before any other function calls within this function, and restoring it right after returning. In this way, we guarantee correctness. The original code with error detection is shown in figure 7(a) and the optimised inlined COMET code is shown in figure 7(b).

4.5 Address Offset Fusion Optimisation

The code generated by COMET for consecutive enqueue (or dequeue) operations is a sequence of `store(r15)` (or `load(r15)`) and `r15 = r15 + 8`, as shown in figure 7(b). This can be optimised further by embedding the increment of the index within the memory instruction itself. For example consecutive stores can become `store(r15)`, `store(r15+8)`, as shown in figure 7(c). This reduces the instruction overhead by half, from two instructions per enqueue or dequeue down to just one.

This optimisation is implemented at the basic-block level and is applied in a peephole manner. Similar to the register promotion optimisation, there are some corner cases that need special care in order to maintain correctness. The value of r15 should be updated with its current value (`r15 = r15 + offset`) before functions calls and at the end of each basic-block. This is important because the value of r15 does not get updated throughout the basic-block body as the increment of the address is performed within the memory instruction (i.e., the assignment `r15 = r15 + 8` is no longer executed).

4.6 Packed Checking

As mentioned in section 2, there are two values to check for each store instruction: the data and the address. In redundant multi-threading both values are sent across cores to be checked by the checker thread. These two values can be packed together into a single value in order to reduce accesses to the queue. We pack the data and the address

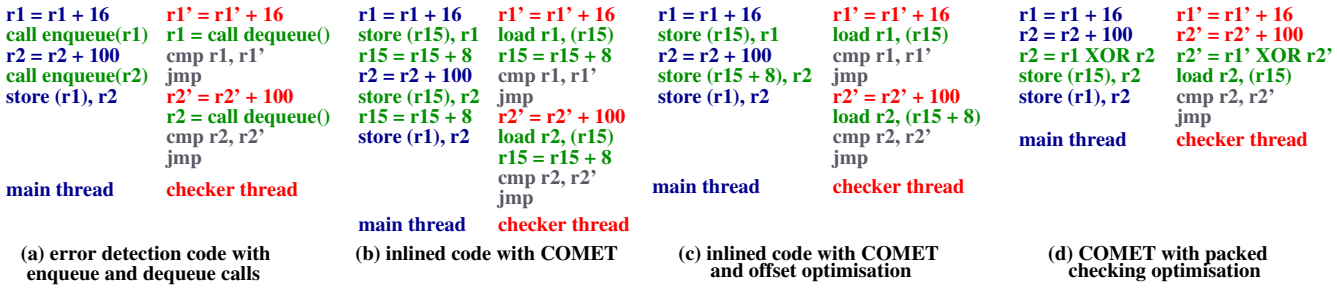

Figure 7: The transformation of the error detection code after inlining the enqueue/dequeue functions of COMET

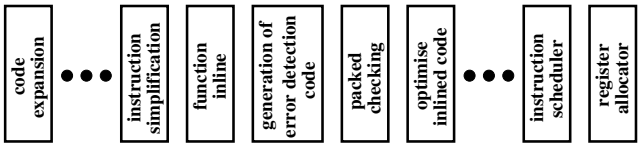

Figure 8: COMET optimisations.

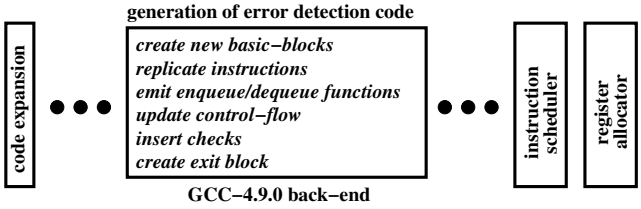

Figure 9: Code generation.

into a single value by calculating their logical XOR. The output of the XOR is the only value sent over to the checker thread. The idea is that if a bit-flip occurs in either the data or the address, the bit-flip will still be present in the packed value. The checker thread performs the same XOR calculation locally and compares the value produced with the value received from the main thread, as shown in figure 7(d). This optimisation does not necessarily reduce the number of instructions (as COMET's enqueue and dequeue require only two instructions in the worst case).

The data and the address to be XORed together may have different types. In this case, an instruction that does type conversion should be emitted before the packing of the checks. For this reason, the proposed optimisation may slightly increase the number of instructions required for the enqueue or dequeue operations, however it also reduces the number of accesses to the queue and the overall stress on the memory system.

4.7 Summary

We have shown that COMET is very close to the oracle queue (optimal in section 3) in terms of its instruction overhead. The code generated by COMET maintains the original program's control flow and does not fragment the basic-blocks. Each enqueue and dequeue operation has only two instructions' overhead: one store / load instruction and one addition that increases the address of the enqueue / dequeue pointers. In certain cases, this can be further reduced down to one instruction per enqueue / dequeue operation with the help of an address offset fusion optimisation and the packed checking optimisation.

The error detection code in figure 1 will be transformed as shown in figure 7. All of the above improve performance. In the case of matrix, figure 2 shows that the overhead of error detection is reduced by 4.55×, which is very close to the overhead of the optimal. In addition, figure 4 shows that the proposed technique has 5.09× fewer branch predictor misses compared to the inlined code of MSQ. The branch predictor misses in the proposed technique are mainly due to the checks within the checker thread, which are reduced through the packed checking optimisation. As a result, fewer branches are executed (a check is a compare instruction and a jump); in comparison, MSQ's inlined code has three branches. Figure 4 suggests that the branches within the inlined code of MSQ, in combination with the checks, prove to be a bottleneck for the branch predictor.

5. COMET IMPLEMENTATION

We implemented both the baseline and our proposed technique, COMET, as RTL passes in the back-end of GCC-4.9 [1]. The part of the compilation pipeline related to the approaches is shown in figure 8. The "generation of error detection code" pass is where the main code generation is performed, while everything else between "instruction simplification" and "optimise inlined code" are support passes. A list of actions performed in the code generation pass is shown in figure 9.

5.1 Low Level Details

In our implementation, each function has two execution paths: one for the main thread and one for the checker thread. Generation of the error detection code consists of the following steps:

1. *Thread selection basic-block:* At the beginning of each function, we emit a basic-block which decides whether the execution should be diverted to the main thread or the checker thread (the thread selection basic-block in figure 10).

2. *Copy basic-blocks:* The checker thread should be created from scratch. Therefore, we start by emitting basic-blocks for the checker code, which are duplicates of the main thread's blocks.

3. *Instruction replication:* For each basic-block, we copy the original instructions and we emit them in the relevant basic-block within the checker code. It is com-

mon design practice [21, 23, 24, 30, 32] not to replicate memory instructions. This convention works well because, if the opcode of the store changes, then the checks will capture the error. In addition, if the address of the store changes, then an exception may be generated. As a result, only the main thread is allowed to read / write from / to memory. To realise this, the main thread sends the values that it loads from memory to the checker thread through the software queue. Thus, an enqueue operation is emitted after the load instruction in the main thread. In the checker thread, a dequeue operation is emitted where load instruction should have been,

4. *Checks insertion:* To communicate the values that need verification, enqueue operations are emitted before memory instructions in the main thread. In the checker thread, the dequeue operations are emitted in the place of the memory instructions. The checks are emitted after the dequeue operations. At this point, we just emit the calls to the queue functions. For each store, we have to send the data value and the address to the checker thread in order to check their correctness (packed checking occurs in a later pass). For each load, we send the address to the checker thread where its correctness is verified.

5. *Update the control-flow:* This has to be performed within the checker thread's basic-blocks. It is done in two stages:

   - For each original basic-block, we find its edges and its successors. Then, for the replicated basic-block, we emit the same number and the same type of edge (i.e., fall-through, fall-back) as those in the original basic-block. If an error occurs in a control-flow instruction, then the main and checker threads will execute different paths, so the error will be detected (figure 10, red edges).
   - In the case of an error, execution is diverted to a basic-block (named exit block). To realise this, an edge that jumps to the exit block is added at each check (figure 10, grey edges).
   - The checker thread executes the same function calls as the main thread. In the case of indirect calls it either receives the call target from the main thread or it calculates it locally.

6. *Inline the queue function:* The code of the queue functions is emitted in the error detection code.

It was shown earlier that the algorithm for the generation of the error detection code separates the memory instructions from the rest of the instructions. However, the code that the compiler generates for x86_64 architectures mainly includes CISC instructions. This means that the computation and the memory accesses are usually in one instruction. In order to overcome this problem, we have added the "instruction simplification" pass before the pass that generates the error detection code (figure 8). This pass simplifies the CISC instructions. In other words, it separates the memory accesses from the computation and for each CISC-style instruction it generates a sequence of new RISC-style instructions (still from the x86_64 ISA) with the memory instructions separate from the computation instructions. This

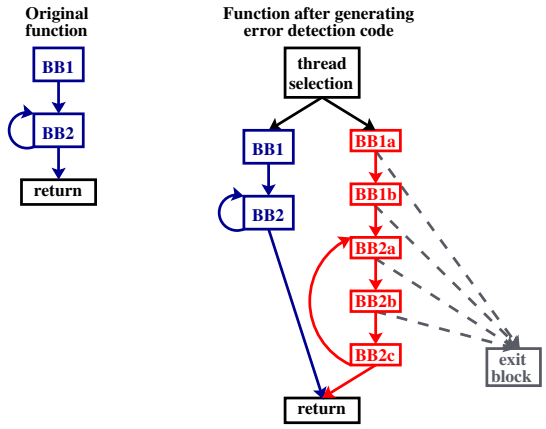

Figure 10: The structure of a function after generating the error detection code.

optimisation does not change the program semantics and it does not have any impact on the sequence of memory instructions within the program.

In addition, we had to implement our own function inliner since GCC's operates only in the middle-end (on the Gimple IR). Our function inliner is simple and performs inlining of the queue functions. It works by copying the RTL code from the queue function bodies and, in step 6 of the generation pass, it emits it into the code as part of the error detection code. In order to do this, we have to change the call graph and make sure that the queue functions will be processed before the functions of the program.

Finally, the passes "packed checking" and "optimise inlined code" perform the optimisations described in subsections 4.6, 4.4 and 4.5.

5.2 Hardware Platform Compatibility

By design, COMET can support any hardware platform. However, we implemented and tested it on an x86_64 machine running Linux. Although x86_64 is sequentially consistent, following the total store order (TSO) memory consistency model, the optimised queue works even on architectures with more relaxed consistency models. Such hardware requires the use of memory fences within the exception handler, which, according to prior work [20], has negligible performance overhead.

6. RESULTS

We first describe the runtime of COMET, showing how it improves performance compared to state-of-the-art. Next, we analyse its execution and fault coverage results.

6.1 Performance Evaluation

We measured the performance of COMET and its constituent parts on a real desktop machine containing a quad-core Core i5-4570 at 3.2GHz with 16GB of DDR3 DRAM, running Linux 3.13.0. We compiled benchmarks from the NAS parallel benchmark suite [3]. We have two versions of the proposed technique: COMET-unpacked (COMET-U) which consists of instruction simplification, the function-inline optimisation, the communication optimisation (COMET queue) and the address offset fusion optimisation, and COMET which is COMET-U with the packed checking optimisation enabled. We compare COMET

against a state-of-the-art redundant multi-threading error detection technique, similar to DAFT [32] and SRMT [30], which we refer to as MTED (multi-threaded error detection). MTED uses the MSQ with inlined enqueue and dequeue operations.

Figure 11 presents the execution time of COMET-U (green bar), COMET (light yellow bar) and MTED (blue bar) normalised to the original unmodified code (no-ED, black bar). For completeness we also show the overhead introduced by the instruction simplification pass (section 5.1) with the dark yellow bar. BT, CG, EP, IS, and SP are the NAS benchmarks that we managed to compile and execute with no errors using our prototype compiler. The last column shows the geometric mean. We ran all NAS benchmarks with the large problem size.

The first thing to notice is that the impact of error detection on performance varies considerably across the various applications. Nevertheless, COMET reduces the overhead of the error detection code down to 2.85× on average, which is an improvement of 31.4% over the state-of-the-art.

6.2 Analysis

To provide more insights into the performance results shown in figure 11, we provide several other metrics in figures 12 and 13. Figure 12 shows the number of dynamic memory instructions (loads and stores) from the original code (without error detection (no-ED)) as a percentage of total instructions. We can use this figure as an indication of the dynamic number of enqueue and dequeue instructions. As mentioned earlier in section 5, for each memory instruction, regardless of whether it is a load or a store, we emit two enqueue instructions in the main thread and, symmetrically, two dequeue instructions in the checker thread. In the case of a load, the first enqueue instruction sends the address to the checker thread, which validates the correctness of the address, while the second enqueue instruction sends the loaded value to the checker thread. In the case of a store, the first enqueue sends over the address, while the second sends the value to be stored.

Figure 13 shows how the error detection schemes increase the number of dynamic instructions. Both error detection schemes increase the dynamic instruction count considerably, but COMET is significantly better than MTED. In SP in particular, MTED increases the dynamic instruction count by more than 34×, while COMET increases it by 5.6×. Overall, COMET reduces the dynamic instruction count over the state-of-the-art by 2.85×. The numbers reported in figures 12 and 13 are collected using the Linux perf tool [4].

BT and SP benefit the most from the proposed technique, both in terms of performance (figure 11) and in terms of dynamic instruction count (figure 13). COMET reduces the overhead of the error detection code by 44.9% and 60.9% for BT and SP respectively. This is expected, as the dynamic memory instruction count (figure 12) of these two benchmarks is the highest, with BT having a ratio of 21% dynamic memory instructions, while SP has 18% for the original code.

The code associated with COMET's enqueue and dequeue operations is so optimised that COMET with the packing optimisation actually increases the count of the dynamic instructions by 6.2% (figure 13). The additional instructions come from type casts required when XORing data of

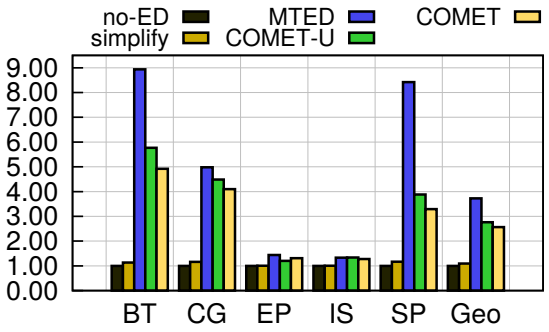

Figure 11: Performance overhead of the different schemes compared with the original code (no-ED).

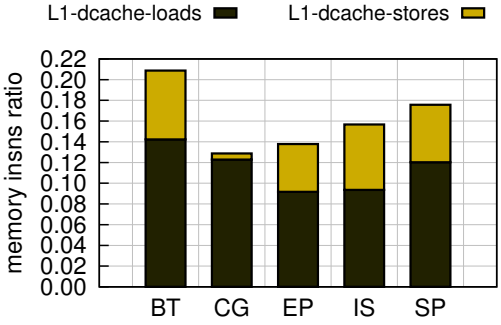

Figure 12: The ratio of memory instructions over the total number of instructions for no-ED code.

different types. For example, given that the address value is always 64-bits, any non 64-bit values require casting before the XOR operation. Nevertheless, figure 11 shows that COMET usually achieves better performance than COMET-U by 7.2% as it reduces the inter-core communication. In EP, however, the overhead of the additional instructions is so significant that COMET performs worse than COMET-U.

The instruction simplification pass (section 5.1), required for accurate error detection in x86_64, introduces a rather minor performance degradation. Figure 11 shows it is approximately 9% on average. Although instruction simplification has a relatively small impact on performance, it increases the number of dynamic instructions by 33.7% on average. The overhead of instruction simplification is related to the increased instruction count and the increased pressure on the processor's front-end (instruction caches, instruction decoding, etc.). As described in section 5, instruction simplification affects CISC instructions with memory operations inside them. Therefore, as expected, the performance impact of instruction simplification is higher on benchmarks with more memory instructions (e.g., BT, SP).

6.3 Fault Coverage

The proposed error detection technique is focused on reducing the performance overheads related to code generation. The quantity and quality of error checking is identical to the existing multi-threaded techniques. Therefore the fault coverage of the proposed scheme remains the same. Nevertheless we provide fault coverage results for completeness.

For the evaluation of fault coverage, it is common practice [6, 9, 21, 22, 24, 23, 25, 30, 32] to use a single-event

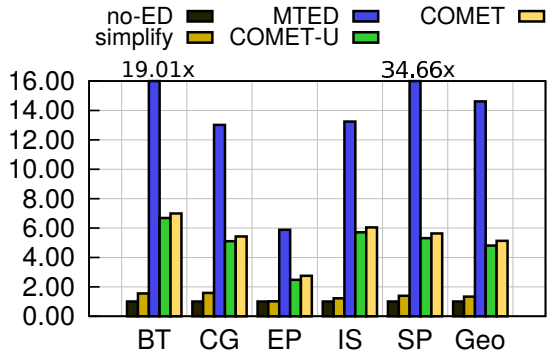

Figure 13: The number of dynamic instructions within the original code with simplified instructions (simplify), MTED, COMET-U (COMET-unpacked) and COMET, normalised to the original code (no-ED).

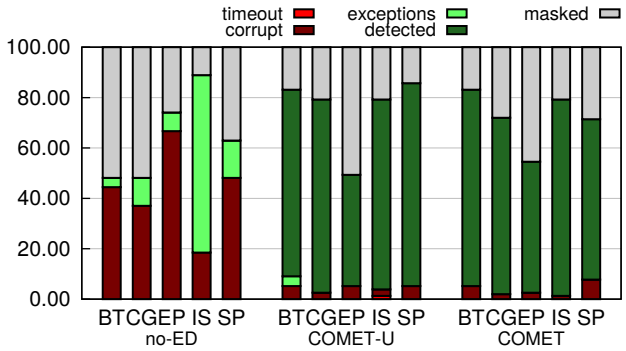

Figure 14: Fault coverage of the original code and both COMET versions.

upset (SEU) fault model. This means that only a single bit flip can happen on any given run. The resulting errors have varying impact on the execution of the program. Errors in the data and the addresses of the instructions are usually detected by the checks. According to SEU model, an error can only manifest in one of the threads (main or checker) at a time and never to both of them at the same time. Therefore, only one of the values of the checks will be wrong and the check will fail. In addition, an error to the address of a memory instruction might raise an exception if the new address is not part of program's memory space. Opcode errors also create exceptions if the new instruction does not belong to the ISA. These exceptions are captured by COMET's custom exception handler. Finally, if an opcode error leads to a new valid instruction, then the new instruction will produce a wrong output and the check will capture the error. In this way, COMET detects the majority of the errors in a program's execution and the queue. The queue is also protected by ECC since it is part of the memory hierarchy. However, COMET cannot detect errors in the status variables of the queue; these errors may lead to dead-lock.

The fault-coverage evaluation was performed similarly to existing work [6, 24, 30, 32], using Monte Carlo simulation. We used an in-house tool that performs the following actions: 1. selects a random thread (either the main or the checker), 2. selects a random dynamic instruction, 3. selects a random register output, 4. selects a random bit from that register and 5. flips the value of this bit.

After fault injection, we check the program's behaviour and classify the outcome. First, if the program execution triggers an exception, then COMET's exception handler captures it and reports the error as detected. Second, if the program finishes in time and its output is correct, then we consider the error as masked (i.e., a benign error). Third, if the program finishes in time, but the output is incorrect, then we have data corruption. Fourth, if the program does not finish in time, then we consider it as a time-out error. Finally, errors that are detected by the algorithm are classified as detected.

We repeated this process three hundred times for each benchmark. The results are shown in figure 14. Both COMET-U and COMET have similar fault coverage. As expected, the unprotected code suffers from very high data corruption ratios, of up to 65% for EP. Depending on the type of computation performed by each benchmark, it may be more or less vulnerable to errors. Benchmarks like EP are very vulnerable, while others, like IS, tend to trigger more exceptions.

COMET is able to provide very high detection rates against transient faults. The output gets corrupted in the small minority of cases, while the majority of faults get detected by either the exception handler or the checker code within the checker thread. The majority of the corruptions and exceptions in COMET are due to the injection of errors into unprotected code (like linked libraries and the handler code of the communication queue). Both of these can be addressed with recompilation of the unprotected code using COMET.

7. RELATED WORK

Code redundancy can take various forms: thread, instruction, process and hardware redundancy.

Redundant multi-threading (RMT) was introduced by Rotenberg in AR-SMT [25]. The main idea is that an exact replica of the original thread is created. The replicated (trailing) thread lags behind the original (leading) thread. The leading thread pushes the output of each instruction into a buffer. The trailing thread checks the values from the buffer with those that it produces.

Reinhardt and Mukherjee [23] introduce the concept of the sphere of replication. The sphere of replication determines the part of the system that is protected by a given technique. To reduce the overhead of RMT, Mukherjee et al. proposed chip-level redundant multi-threading (CRT) [21]. In this approach, the leading and the trailing threads run on different cores. The main disadvantage of redundant multi-threading is that it reduces the system's total throughput since it requires more threads and hardware resources. Additionally, compared to instruction-level approaches (where software queues are used for the communication between the threads), most of the redundant multi-threading schemes require custom hardware.

Instruction-level redundancy SRMT [30], inspired by redundant multi-threading error detection, proposes a multi-threading technique that uses software checks instead of hardware ones. DAFT [32] improves this technique further by decoupling the execution of the original and the checker thread. In thread-local instruction-level error detection, the original instructions, the replicated instructions, and the checks are in the same thread. Thread-local error detection was first introduced in EDDI [22]. Next, SWIFT

[24] improved performance by reducing the memory overhead. DRIFT [19] shows that basic-block fragmentation is a major performance bottleneck for SWIFT. Basic-block fragmentation is due to frequent checking and it prevents the compiler from applying aggressive optimisations. Chang et al. [6] present triple-modular redundancy at the instruction level. An improvement of thread-local error detection is Shoestring [9]. The main idea is that transient errors generate symptoms like memory exceptions, cache misses, and branch mispredictions. The appearance of these symptoms implies the existence of transient errors. Therefore, they propose an algorithm which identifies the instructions that can generate the symptoms and they do not replicate them.

Process-level redundancy (PLR) Shye et al. [27] replicate the processes of the application and compare their outputs to ensure correct execution. The processes synchronise to compare their outputs when the value escapes user space to the kernel. RAFT [31] improves this scheme by removing the synchronisation barriers. PLR has a small overhead since it checks fewer values than other approaches, but this comes at the cost of maintaining multiple memory states.

Hardware-based redundancy replicates hardware units. Hence, the whole system must be custom designed for fault-tolerance. Although this process is very expensive and less flexible than those described previously in this section, hardware-based approaches often suffer less performance degradation from fault tolerance. Typical examples are the HP NonStop Advanced Architecture (NSAA) [5] and IBM's z series [8].

8. CONCLUSION

Software error detection techniques provide a flexible and easily deployed alternative to hardware error detection for transient faults. Multi-threaded error detection schemes make use of dedicated cores within a multi-core system to execute the redundant code and to perform error checking. The major issue of such software techniques, however, is that they degrade performance considerably. In this work we focus on a specific code generation problem, common to these techniques: that of poorly performing generated code responsible for frequent inter-core communication. We introduce COMET, a novel technique that optimises this performance critical code. The performance achieved by COMET is 31.4% higher on average than the state-of-the-art, while significantly reducing the number of instructions executed.

9. ACKNOWLEDGEMENTS

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant references EP/K026399/1 and EP/J016284/1. Additional data related to this publication is available in the data repository at http://dx.doi.org/10.17863/CAM.590.

10. REFERENCES

[1] GCC: GNU Compiler Collection. *http://gcc.gnu.org*.
[2] The LLVM Compiler Infrastructure. *http://llvm.org*.
[3] NAS Parallel Benchmarks. *http://www.nas.nasa.gov/publications/npb.html*.
[4] PERF: Linux Profiling With Performance Counters. *https://perf.wiki.kernel.org*.
[5] D. Bernick, B. Bruckert, P. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop Advanced Architecture. In *DSN 2005*.
[6] J. Chang, G. Reis, and D. August. Automatic Instruction-Level Software-Only Recovery. In *DSN 2006*.
[7] C. Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro 2003*.
[8] M. L. Fair, C. R. Conklin, S. Swaney, P. Meaney, W. Clarke, L. Alves, I. N. Modi, F. Freier, W. Fischer, and N. E. Weber. Reliability, Availability, and Serviceability (RAS) of the IBM eServer Z990. *IBM Journal of Research and Development 2004*.
[9] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *ASPLOS 2010*.
[10] K. Gharachorloo and P. B. Gibbons. Detecting Violations of Sequential Consistency. In *Proceedings of SPAA 1991*.
[11] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, et al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *the Journal of Supercomputing 1993*.
[12] T. B. Jablin, Y. Zhang, J. A. Jablin, J. Huang, H. Kim, and D. I. August. Liberty Queues for EPIC Architectures. In *Proceedings of EPIC Workshop 2010*.
[13] L. Lamport. Specifying Concurrent Program Modules. *TOPLAS 1983*.
[14] P. P. Lee, T. Bu, and G. Chandranmenon. A Lock-Free, Cache-Efficient Shared Ring Buffer for Multi-Core Architectures. In *ANCS 2009*.
[15] P. P. Lee, T. Bu, and G. Chandranmenon. A Lock-Free, Cache-Efficient Multi-Core Synchronization Mechanism for Line-Rate Network Traffic Monitoring. In *IPDPS 2010*.
[16] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. Lichtenstein, R. P. Nix, J. S. O'donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling Compiler. *The journal of Supercomputing*,1993.
[17] S. A. Mahlke, W. Y. Chen, W.-m. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel Scheduling for VLIW and Superscalar Processors. In *ASPLOS 1992*.
[18] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender. Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer. *IEEE Transactions on Device and Materials Reliability 2005*.
[19] K. Mitropoulou, V. Porpodas, and M. Cintra. DRIFT: Decoupled compileR-based Instruction-level Fault-Tolerance. In *LCPC 2013*.
[20] K. Mitropoulou, V. Porpodas, X. Zhang, and T. M. Jones. Lynx: Using OS and Hardware Support for Fast Fine-Grained Inter-Core Communication. In *ICS 2016*.
[21] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *ISCA 2002*.
[22] N. Oh, P. Shirvani, and E. McCluskey. Error Detection by Duplicated Instructions in Super-scalar Processors. *IEEE Transactions on Reliability 2002*.
[23] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *ISCA 2000*.
[24] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. SWIFT: Software Implemented Fault Tolerance. In *CGO 2005*.
[25] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *International Symposium on Fault-Tolerant Computing 1999*.
[26] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *DSN 2002*.
[27] A. Shye, T. Moseley, V. Reddi, J. Blomstedt, and D. Connors. Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance. In *DSN 2007*.
[28] D. J. Sorin. Fault Tolerant Computer Architecture. *Synthesis Lectures on Computer Architecture*,2009.
[29] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *DSN 2004*.
[30] C. Wang, H.-S. Kim, Y. Wu, and V. Ying. Compiler-Managed Software-Based Redundant Multi-Threading for Transient Fault Detection. In *CGO 2007*.
[31] Y. Zhang, S. Ghosh, J. Huang, J. W. Lee, S. A. Mahlke, and D. I. August. Runtime Asynchronous Fault Tolerance via Speculation. In *CGO 2012*.
[32] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August. DAFT: Decoupled Acyclic Fault Tolerance. In *PACT 2010*.