Fault Tolerance in Multi-Core Systems

Stefan Reif
Friedrich-Alexander-Universität Erlangen-Nürnberg
ke42caxa@cip.cs.fau.de

ABSTRACT

Modern processors provide multiple cores for parallel computing. This paper describes how parallel processing on multiple cores can provide efficient fault tolerance. In general, multi-core processors provide structural redundancy which can be exploited for efficient replication. Furthermore, hardware features can improve performance of redundant execution by exchanging information between replicas. However, multicore performance can also be exploited for fault toperance without dedicated hardware support. This seminar paper introduces four fault-tolerance approaches, which each exploit multi-core processors. Furthermore, a comparison between the techniques shows both advantages and disadvantages for each technique.

1. INTRODUCTION

As transistors become smaller, vulnerability of modern processors regarding transient hardware fault increases. In general, hardware errors can lead to software errors, which can lead to system failure. In consequence, software-based systems become less reliable.

One particular class of hardware errors are transient errors. There, a part of the hardware malfunctions only for a small amout of time. One reason for transient hardware errors are physical effects like radiation or cosmic particles. When such an effect inverts the state of a transistor, then the computation output can be wrong.

To avoid wrong computation results, additional error checking mechanisms are required. For instance, redundancy and replication allow error detection, and correction. In general, replication means that one result is computed in more than one way. The benefit of replication is that the computation output can be verified by comparing it to the output of other replicas. On mismatch, one of the results is wrong. Unfortunately, redundancy lead to a computation overhead. Hence, this paper focuses on multi-core processors, because the provide an inherent structural redundancy, which can be exploited for efficient fault detection.

This paper introduces four techniques that use multi-core processors for fault detection. First, Section 2 introduces to the basic terms of fault tolerance. Then, Sections 3, 4, 5, and 6 each introduce one fault tolerance technique. While the first two solutions base on special-purpose hardware, the latter two are implemented in software. Last, Section 7 contains compares all techniques regarding various aspects.

2. BACKGROUND

This section contains a brief introduction to fault tolerance. A basic concept of fault tolerance is replication. In general, replication means that a computation of the system is derived from more than one source. As long as no error occurs, all replicated computations lead to the same result. In contrast, the results are different when an error occurs in one of the replicas.

One key aspect regarding replication techniques is the sphere of replication (SOR). Basically, the SOR describes which resources are replicated for fault tolerance. To ensure that all replicas have the same output values, all of them need equivalent input data. Consequently, input replication is required. In general, all information that enters the SOR needs to be replicated so that all replicas use the same values. Then, output comparison can detect errors, and activate an error handling mechanism. Therefore, all information that leaves the SOR needs verification by comparing it to the output of other replicas.

This thesis focuses on multi-core processors, which provide inherent structural redundancy. As it provides multiple cores, the computation power can be exploited for redundancy with only little time overhead compared to a single-threaded implementation. When all cores of the processor compute the same function, then their results have to be equal. Thus, an important benefit of the multi-core processor is that redundant code execution has only little impact on the execution time.

Another key reason to exploit multi-core processors for fault tolerance is that this feature is common to off-the-shelf hardware. Some of the approaches presented in this paper do not require custom hardware, which generally tends to be expensive. Additionally, the multi-core feature can provide a tradeoff between performance and fault tolerance. When the application does not need the whole computation power

of the processor, then the remaining cores could offer fault detection and correction mechanisms. Unfortunately, not all techniques in this paper provide this flexibility.

The following sections introduce four concepts, which provide fault tolerance mechanisms efficiently on mult-core processors. While the first two approaches are implemented entirely in hardware, the latter are implemented in software and therefore more flexible. Furthermore, each approach has some advantages and some drawbacks compared to the others.

3. LOCKSTEPPING

This section describes Lockstepping, which is a hardware-based redundancy technique for fault detection[2]. It has been commercially available for decades[6]. Here, the physical processor core constitutes the sphere of replication. Consequently, multiple physical cores compute the same function.

An additional hardware module duplicates all hardware information that enter the processor, and compares all output information in order to detect differences. On error, the output comparison module raises a trap signal to the processors.

The need for identical output implies that the incolved processor cores must be deterministic. Furthermore, time plays an important role regarding determinism: All cores must provide the same output in the same time. Otherwise, the time difference could lead to an output mismatch, which then leads to the error detection mode. Therefore, the duration of every instruction must be deterministic.

Synchronization is also required for all signals that enter the CPU. For instance, each interrupt must occur at the same cycle for each processor core. Otherwise, output mismatch can occur.

The main advantage of Lockstepping is that it is transparent to the software, except for the trap that signals output mismatch. Then, a trap handler is required to reset both processors to a valid state, so that computation can continue.

However, Lockstepping has some disadvantages regarding the involved processor cores. Basically, the need for low-level hardware determinism leads to restrictions in hardware design.

First, Lockstepping leads to a huge hardware overhead. For fault detection, two entire processor cores are required, which duplicates the total hardware costs. Second, nondeterminism can occur in modern off-the shelf hardware. For instance, differences in hardware bits lead to a Lockstep error, even if they have no effect on code execution. Third, clock domain crossing leads to asynchronous events, which would subverts hardware determinism. Additionally, deterministic execution over the entire hardware life time cannot be tested.

The following sections provide multi-core based redundancy techniques with, in contrast to Lockstepping, relaxed need for syncronization and hardware determinism.

4. SIMULTANEOUS AND REDUNDANT MULTITHREADING

This section describes hardware-based Simultaneous and Redundant Threading[4] (SRT). This approach uses a hardware feature called Simultaneous Multi-Threading (SMT). There, two physical cores share some resources in a fine-grained manner[9]. The basic idea of SRT is that two threads execute the same function, and they run on two cores on the same chip. At SMT, the fine-grained resource sharing reduces the hardware overhead compared to Lockstepping because, in total, less then two entire processor cores are required[3].

4.1 Implementation

The sphere of replication contains the processor state. In consequence, load operations need duplication, and store operations need consistency checks.

However, the resource sharing implies a problem. As both threads compute the same function, both need the same resources, when they execute the same instruction. The consequence is a permanent conflict between the redundant threads. In order to reduce this conflict, the redundant threads run asynchronously. This is a difference to Lockstepping, which requires that both replicas execute the same instruction at the same time. In consequence, two thread roles exist: One thread is called the leading thread. The second thread, also named the trailing thread, executes the same instructions with a short delay. The time difference reduces the conflict for hardware resources.

For input duplication, the processor typically provides a load-value queue (LVQ). This hardware queue records all load operations of the leading thread, storing both the load address and the value. When the trailing thread later performs the same load operation, the hardware compares the address to the one in the LVQ, which also provides the load value.

To detect output mismatch, a Store Buffer keeps track of store operations of the leading thread. When the trailing thread later executes the same store instruction, then the hardware automatically verifies the value before it leaves the SOR.

Compared with Lockstepping, the main advantage of SRT is the asynchronity of the redundant threads. Therefore, SRT avoids the overhead of strict hardware synchronization. Additionally, the performance of SRT can be improved further using hardware features, as the next subsections will show.

4.2 Slack Fetch

One key optimization approach targets cache efficiency of the redundant threads. Slack Fetch is such a hardware-based optimization technique. Here, the time delay between the two redundant threads helps to avoid cache misses. In general, the leading threads performance can suffer from cache misses. In contrast, the trailing thread can later uses the same cache lines as the leading thread, without reloading it from main memory. This leads to the basic idea of Slack

Fetch: The leading thread warms up the cache lines so that the trailing thread hardly encounters any cache miss.

Technically, both threads manipulate the cache control hardware for Slack Fetch, so that it prefers fetching data for the leading thread, until the predefined slack is reached. Later, the hardware automatically maintains the desired slack. Reinhard et. al. have shown that slack fetch improves the overall performance by 10%.

4.3 Branch Outcome Queue

Similarly to Slack Fetch, which targets cache efficiency, the leading thread can improve the performance of the trailing thread by providing information to the branch prediction hardware. In general, modern processors make use of an instruction pipeline to speed up execution, by executing multiple instructions in different stages in parallel. However, conditional branch instructions leads to inefficient program execution, because the instruction pipeline cannot reliably predict which is the next instruction to load.

At this point, the Branch Outcome Queue can improve the performance of the trailing thread. As both threads execute the same instructions, the leading thread can pass its branch information to the trailing thread. Then, the branch prediction hardware can choose the correct control flow based on the information provided by the leading thread. According to Reinhard et. al, the Branch Outcome Queue can improve performance by 14%.

4.4 Evaluation

In average, SRT provides a performance improvement of 16%, with a maximum of 26%, compared to Lockstepping. Performance can be enhanced even further by hardware features like Slack Fetch and a Branch Outcome Queue. Furthermore, SRT requires less hardware than Lockstepping because the two threads share hardware resources.

Similar to Lockstepping, SRT is transparent to the application, except for error detection. Consequently, it can be applied to any application.

However, SRT needs hardware support like Lockstepping. In contrast to SRT, the following section describes a multi-threading approach that does not require hardware support.

5. REDUNDANT MULTI-THREADING FOR TRANSIENT FAULT DETECTION

Software-based Redundant Multi-Threading (SRMT) uses the compiler to automatically create redundant threads[8]. Similarly to SRT, each computation is handled by two threads. Again, a leading thread is backed up by a tailing thread for error detection.

The SOR of SRMT is the thread state. In general, two scenarios exist where data enters the SOR. First, load operations on shared memory bring data into the SOR. Second, the return values of system calls come from outside of the SOR. Both times, only the leading thread perform the related operation. Then, it sends the value to the trailing thread, which uses the received value instead of performing the operation itself. In consequence, every operation that brings data into the SOR is performed only once.

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

Figure 1: The leading thread performs all load operations and communicates the value to the trailing thread.

Similarly to the load operations, only the leading thread performs operations where data leaves the SOR. For instance, system call parameters and store operations to global memory result in data leaving the SOR. Here, the leading thread first communicates the corresponding value to the trailing thread. When the trailing thread has the same data in his state, then it acknowledges the value. The trailing thread detects errors, when the received data is not equal to the copy in its own thread state.

This communication pattern can be created by the compiler: A compiler can detect load and store operations to shared memory. Using this information, it can automatically create the leading and the trailing version of the thread function. The resulting fault detection feature is transparent to the programmer. Thus, SRMT shifts the responsibility of fault detection to the compiler.

5.1 Compiler Transformations

In total, the compiler performs multiple code transformations. First, functions need to exist multiple times in different versions. Second, the compiler needs to insert code for fault detection. Additionally, memory operation reordering can reduce communication overhead and therefore improve performance.

5.1.1 Function Duplication

For each function, the compiler generates a leading and a trailing version, for the leading and the trailing thread, respectively. In general, both threads have to execute slightly different code for fault detection. In the following, several code transformations are described, which are applied to both function versions.

5.1.2 Input Duplication

Whenever data enters the SOR, the code needs modifications. In general, only the leading thread performs operations which bring data into the SOR. Therefore, the compiler creates code using the schema shown in Figure 1.

As Figure 1 shows, all load operations are performed by the leading thread. Then, it communicates the value to the second thread, which uses the received value instead of performing its own load operation. In consequence, both threads operate on the same data.

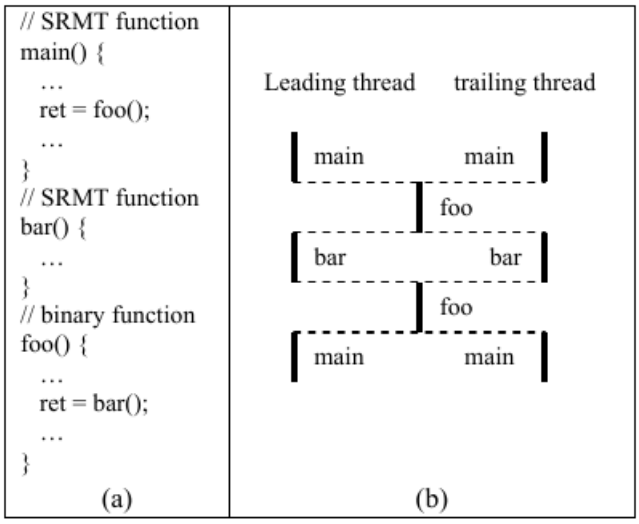

Figure 2: The extern version of a SRMT function provides fault detection by executing both leading and trailing function

5.1.3 *Externally Visible Functions*

For all globally visible functions, the compiler generates three versions of the code. In addition to the leading and the trailing version of the functions, an extern version of the function acts as a callback for library functions. When a function is only accessible in binary code, then it does most likely not support SRMT. In consequence, a callback to SRMT is called only once.

This non-SRMT function is called binary, because it accessible only as binary code. Figure 2 shows a call graph including a binary function. However, it cannot call the leading or the trailing version of the function, because none of them is aware that it is called from a non-SRMT function. Therefore, a third version of the SRMT function is required, which is only called from outside of the compilation unit. Here, the binary function `foo` calls the SRMT function `bar`, which is visible to the library function. To achieve fault detection, the external version executes both leading and trailing function instances in two threads. Consequently, the external version provides fault detection as well.

5.1.4 *Output Comparison*

When data leaves the SOR, the compiler emits instructions that verify that the data is correct. In detail, the leading thread first sends the value to the trailing thread. Then, the trailing thread compares the value to its local copy and acknowledges the value.

When the values differ, the trailing thread raises an error. This provides fail-stop semantics: On error, the program reliably aborts computation. Unfortunately, fault detection leads to an overhead because of the inter-thread communication. In the following, two approaches are described to reduce communication costs.

5.2 Efficient Inter-Core Communication

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

Figure 3: Before data leaves the SOR, the trailing thread acknowledges the value to the leading thread.

Wang et. al. proposed two approaches to speed up SRMT. The first of them only considers the software-part of the inter-thread communication. The second one considers hardware support to minimize communication costs.

5.2.1 *Optimized Software Queue*

In order to reduce the amount of inter-core communication, multiple data packets can be sent at once. Wang et. al. call this technique Delayed Buffering. Then, the first message is delayed until enough data is accumulated.

Furthermore, a technique called Lazy Synchronization reduces access to shared data. There, local copies of the head and tail variable of the communication queue reduce communication overhead. In total, both techniques reduce the number of cache misses by 83.2% for the L1 cache and 96% for the L2 cache. Additionally, communication costs could be reduced even further using dedicated hardware support.

5.2.2 *Hardware Message Queues*

In order to minimize message costs, the hardware itself could provide an efficient message queue. This minimizes the cache coherence overhead for inter-thread communication. Wang et. al. reason that other application can profit from efficient inter-core communication queues as well, for instance regarding the producer-consumer design pattern. Unfortunately, the authors could only test this feature in a processor simulator.

5.3 Evaluation

As the SRMT approach produces twice the number of threads, the resulting application can use up to twice the number of processors. Then, additional processor cores provide efficient redundant execution.

One bottleneck of SRMT is the communication between the leading and the trailing thread. Unfortunately, each access

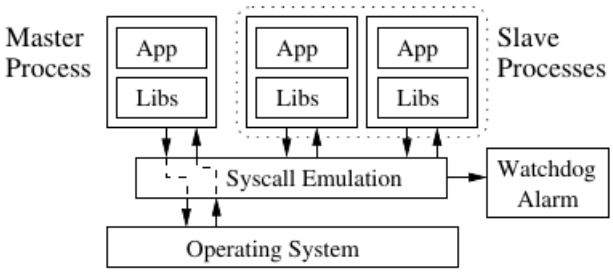

Figure 4: Architecture Overview of PLR

to shared memory requires communication, which limits the multi-core performance.

Optimally, the hardware could reduce the communication overhead by providing message queues. Using hardware-based on-chip queues, SRMT leads to an overhead of 19% compared to non-SRMT programs. Using a shared L2 cache for communication, SRMT has an average slow-down factor of 2.86.

As fault tolerance is generated by the compiler, the user cannot dynamically configure the fault tolerance technique. However, the user can dynamically choose between a SRMT and a non-SRMT version of the program. Another advantage of SRMT is that can even provide redundancy for non-deterministic, multi-threaded applications. However, communication overhead remains a drawback of SRMT. In average, 0.61 Bytes per cycle need to be transferred. Further reduction of communication overhead requires an abstraction on a higher logical level.

6. PROCESS LEVEL REDUNDANCY

For Process Level Redundancy (PLR), a whole process is replicated[5]. The PLR technique is implemented entirely in software. Each process instance runs on an individual core, in its own address space.

Figure 4 shows the architecture of PLR. A system call emulation coordinates all instances of the process. Furthermore, PLR requires a watchdog alarm, for instance to detect errors that lead to an infinite loop.

The system call emulation manages the process instances. For error detection, it assumes that all process instances are equal. Consequently, the replicated application needs to be deterministic[7]. Otherwise, the processes might behave different, but the system call emulation cannot distinguish indeterminism from errors.

6.1 System Call Emulation

When one of the processes emits a system call, then the system call emulation blocks this particular process, and waits for all other processes to do the same system call. Then, the system call emulation compares all arguments. When no error occurred, then all parameters of the system call are necessarily equal among all replicated process instances. Otherwise, the Syscall Emulation Unit terminates the defective process, and spawns a new instance using fork()[1]. The Syscall Emulation Unit then propagates the system call to the operating system, and duplicates the result of the real system call to all process instances.

6.2 System Call Watchdog

Besides system call interception, the Syscall Emulation Unit provides a watchdog alarm. This module starts a timer when any process emits a system call. When the timer expires, the Syscall Emulation Unit assumes that an error occurred. Two scenarios exist where the timer expires. First, a faulty process can emit a wrong system call. In this case, none of the other instances emits the system call, or maybe a different one. Then, the Syscall Emulation Unit terminates the faulty process instance which performed the wrong system call. Second, an error can lead to an infinite loop. Then, the defective process does not emit any system call. Consequently, the Syscall Emulation Unit terminates the faulty process when the timer expires.

6.3 Application Constraints

In general, PLR can replicate any process, because the replication is opaque to both the operating system and the replicated process instances. However, the Syscall Emulation Unit requires that the application is deterministic. Otherwise, multiple legal computations exist which then lead to different output. Then, the Syscall Emulation Unit cannot distinguish these legal—but unequal—results from erroneous computations. Therefore, nondeterminism in application leads to false-positive error checks. It is the responsibility of the user to not apply PLR to nondeterministic applications.

As parallel processing implies nondeterminism, the replicated application itself cannot be parallel. In consequence, PLR cannot be applied to applications that can run on multiple cores. Therefore, the overhead of PLR can be considered very low because it only uses resources that would otherwise be idle. Then, only the Syscall Emulation leads to an overhead, because of input data duplication, output data comparison and synchronization.

6.4 Evaluation

As a software-based redundancy technique, PLR is very flexible. It provides a mechanism for error detection and correction that is transparent for both the operating system and the replicated application, except for the need for determinism. In consequence, the technique can be applied to existing programs on arbitrary hardware.

According to Shye et. al, PLR leads to an overhead of 16.9% or 41.1% for fault detection, or fault correction, respectively. Both values indicate the overhead compared to single process execution, which does not provide fault tolerance.

However, the replicated process instances of PLR are independent as long as no system call occurs. Therefore, all redundant processes can be computed in parallel. In consequence, the application is able to compute on multiple processor cores.

An additional benefit of PLR is its scalability. As the replication is entirely transparent to the application, the amount of redundancy can be adjusted dynamically. Therefore, PLR

allows the user to dynamically enable or increase fault tolerance, for instance when a processor core is idle.

Furthermore, PLR compares only values that leave the address space of the process instances. Consequently, logically benign faults can be ignored and do not activate the error correction mechanism.

7. DISCUSSION

This section contains a discussion of all fault-tolerance techniques introduced in the previous sections. All of them efficiently exploit multi-core hardware to provide fault detection. However, they differ in multiple aspects, which are discussed in the following.

7.1 Hardware-Support

One key aspect separating the individual fault detection techniques is the amount of hardware support they require. For instance, Lockstepping is a purely hardware-based solution. For instance, both input duplication and output comparison are implemented in hardware. In consequence, this approach cannot be used on off-the-shelf processors which does not provide these hardware modules.

Here, hardware-based SRT is similar to Lockstepping, because it does not run on arbitrary hardware. Furthermore, special hardware features like Slack Fetch and the Branch Outcome Queue can improve performance.

Similarly, compiler-based SRMT can use hardware features for performance optimization, but it can also run on any multi-processor hardware.

In contrast to the other techniques, PLR is implemented entirely in software. Therefore, PLR can run on all processors. However, this implies that hardware features cannot improve PLR performance.

7.2 Tolerance of Benign Faults

The fault detection techniques differ in the handling of benign faults. For instance, at Lockstepping, all information that leaves the processor has to be equal. In consequence, the hardware detects all faults, and it cannot ignore logically benign faults. In contrast, PLR only detects faults that lead to a wrong output.

For Lockstepping, benign faults cannot be ignored because the hardware cannot know that a particular information is ignored later. Therefore, every benign fault leads to error handling, which then leads to an overhead compared to ignoring the fault.

Similar to Lockstepping, the hardware-based SRT technique compares all information leaving the processor cores. Again, it cannot ignore benign faults. Therefore, the overhead of error correction can be considered rather high due to false-positives in error checks.

In contrast to the two hardware techniques, the compiler-based SRMT technique compares fewer values, which then leads to fewer consistency checks, and less false-positive fault detections.

7.3 Application Constraints

A huge benefit of hardware-based fault detection is transparency for the software. Except for the error-recovery function, fault tolerance is transparent to the application. Therefore, both lockstepping and SRT can be used for any application.

Similarly, SRMT is transparent to the application. The only requirement is that the source code is available, because this approach bases on compiler code transformations.

However, PLR requires that the redundant application is deterministic. In consequence, the application itself cannot make use of multiple cores. Here, the PLR technique allows the application to use otherwise idle ressources.

7.4 Flexibility

One key benefit of PLR is its flexibility regarding the tradeoff between performance and fault detection. Here, the user can dynamically decide whether to use the multi-core processor to improve the application performance or error detection.

Similar to PLR, SRMT allows the user to dynamically choose either a performance boost or fault detection. Furthermore, SRMT supports multi-threaded applications, so that the non-SRMT version of the program can actually also exploit the multi-core processor for performance.

In contrast, both Lockstepping and the hardware-based SRT approach do not necessarily allow the user to use the second core for performance improvements.

8. CONCLUSION

In summary, multi-core processors provide structural redundancy which can be exploited for efficient fault tolerance. Multiple approaches exist to implement fault detection mechanisms on multiple processor cores. They differ in various aspects, like the amount of hardware support required to implement the technique. Some techniques like Lockstepping require special hardware for fault detection. In contrast, other techniques like PLR can be applied to any processor.

Software-based techniques like PLR and the compiler-based SRMT additionally provide a flexible tradeoff between performance and fault detection. There, the user can dynamically choose either fault tolerance or application performance.

9. REFERENCES

[1] The GNU C Library. https://www.gnu.org/software/libc/manual/, 2014. [Online; accessed 2015-01-06].

[2] S. Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufman Publishers, 2008.

[3] S. S. Mukherjee. Detailed design and evaluation of redundant multithreading alternatives. *Ann Arbor*, 1001:48109–2122.

[4] S. K. Reinhardt and S. S. Mukherjee. *Transient fault detection via simultaneous multithreading*, volume 28. ACM, 2000.

[5] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 297–306. IEEE, 2007.

[6] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 223–234. IEEE Computer Society, 2006.

[7] D.-I. P. Ulbrich. Redundante ausführung. https://www4.cs.fau.de/Lehre/SS14/V%5FVEZS/Skript/04-Redundanz%5FhoA4.pdf, 2014. [Online; accessed 2015-01-06].

[8] C. Wang, H.-s. Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 244–258. IEEE, 2007.

[9] Wikipedia. Simultaneous multithreading — wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Simultaneous%5Fmultithreading, 2014. [Online; accessed 2015-01-06].