A Technical Report on Non-Intrusive Dynamic Application Soft Error Detection

Karthik Shankar, Roman Lysecky
Department of Electrical and Computer Engineering
University of Arizona, Tucson, AZ
{karthik1, rlysecky}@ece.arizona.edu

ABSTRACT

Advances in integrated circuits present several key challenges in system reliability as transient and permanent soft errors are expected to increase with successive technology generations. Computing systems must be able to continue functioning in spite of these soft errors, necessitating the development of new methods for self-healing circuits that can detect and recover from these errors. We present a dynamic application soft error detector (DASED) capable of non-intrusively detecting soft errors within the execution of a software application without requiring modifications to the application or the target processor hardware. The DASED design is capable of detecting 92% of all control errors and 85% of all unmasked errors while incurring only a 6% area overhead.

Keywords

Soft error detection, dynamic profiling, multitasking, real-time embedded systems, multicore systems.

1. INTRODUCTION

Exponential advances in process technologies and circuit capacities have enabled the current success and future promise of high-performance computing enabled by multicore computing. At the same time, such advances in integrated circuits present several key challenges in system reliability in that transient and permanent errors, such as electro migration, negative bias temperature instability (NBTI), time dependent dielectric breakdown (TDDB), and thermal cycling (TC), are expected to increase [9][17]. To be successful, computing systems must be able to continue functioning in spite of these soft errors, necessitating the development of new methods for self-healing circuits that can detect and recover from these soft errors.

Within multicore computing systems, soft errors can affect both the functionality and performance of software executing on each processor. But as software can be executed on any processor core, multicore systems offer a significant advantage over single core alternatives. If a soft error occurs within a specific core – either transient or permanent – the software tasks executing on that core can be relocated to another processor core. Such a self-healing system would need to monitor the execution of the executing software tasks to identify soft errors and recover from those errors by re-executing the affected software tasks or by migrating the affected software tasks to another processor core. As many soft errors may be transient, it is further necessary to re-evaluate the error state of previously affected cores to determine if those cores can be safely utilized again, and under what conditions.

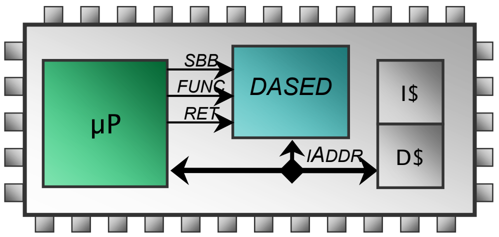
A self-healing system is capable of monitoring its own execution to isolate the source of soft errors and recover from those errors by reconfiguring the system to perform the same computation. For multicore software implementations, it is minimally necessary to detect soft errors that affect the correctness and/or performance of software execution and recover from these faults by executing the affected software regions using other available processing resources.

Most existing approaches for detecting soft errors within the execution of a software application are intrusive in nature, either requiring additional instructions to be inserted within the software binary or requiring additional hardware components within the microprocessor itself. For embedded systems, such methods potentially change the behavior of the application and incur significant runtime overhead. In the case of real-time systems, which are usually designed with tight timing constraints, the slightest runtime overhead can lead to missed deadlines and potential system failure [6]. Alternatively, most hardware-based approaches directly incorporate redundant computational units or specialized components for detecting errors that can affect the critical path of the microprocessor itself. In addition, these approaches assume a system developer has access to the microprocessor design itself. For many embedded systems, processor cores are provided as hard macros or encrypted sources, thus limiting developers' ability to incorporate hardware based soft error detection methods directly within the processor.

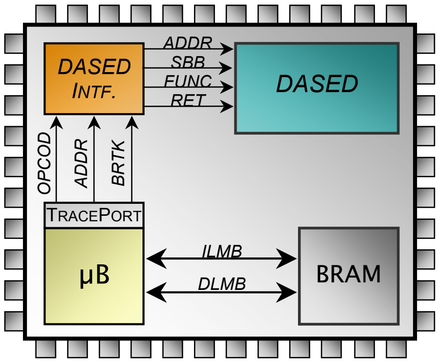
In this paper, we present a dynamic application soft error detector (*DASED*) capable of non-intrusively detecting soft errors within the execution of a software application without requiring modification to the software application or microprocessor hardware. The *DASED* design utilizes the non-intrusive profiling methodology – originally presented in [1][2] – to monitor the control flow of a software application given a statically determined profiling based on known execution characteristics of an application's tasks.

2. PREVIOUS WORK

A popular method for detecting and recovering from errors is fault tolerant computing. Fault tolerant computing utilizes redundancy – such as triple modular redundancy [15] – within the computing resources to reduce the chance that a soft error will affect the final outcome of the application or circuit. A fault tolerant software system might utilize multiple processor cores where all cores execute that same task. The results from the redundant executions are compared to determine if a consensus of the final result can be reached. For example, if three or more out of five redundant computations match, that result is likely correct. By utilizing redundancy, the probability of an error affecting more than one processor is significantly reduced. As such, the system can continue to execute correctly as long as a specified number of processors produce the same result. While effective, fault tolerant computing requires significant duplication of resources as redundancy is required at every level of the computation, i.e., a fault tolerant computing system would need redundant processors, caches, and memories. In addition, concurrent execution of redundant computations increases power consumption in proportion to the level of redundancy. As cost and power consumption are critical design constraints, fault tolerant computing is prohibitive for many computing systems.



(a)



(b)

Figure 1. (a) Overview of the Dynamic Application Soft Error Detector integration with microprocessor system utilizing signals for detecting short backwards branches (*sbb*), function calls (*func*), function returns (*ret*), and (b) an example system integration with a Xilinx MicroBlaze system in which the processor's trace port provides the required signals needed by the *DASED* design.

DIVA [4] is a dynamic error detection and recovery technique with integrated functional redundancy within the retirement stage of an out-of-order microprocessor. The *checker* components incorporated within the retirement stage verify the correctness of all computations and communication operations be re-executing the retired instructions in order. If a computation or memory address was incorrectly computed, the entire pipeline of the processor will be flushed and the errant instruction is re-executed. Such an approach requires that the DIVA checker components be functionally and electrically reliable to avoid errors within the checker components themselves. By re-executing all computations, DIVA is capable of detecting and recovering from most soft errors, but comes at the unavoidable tradeoff of significantly increased area requirements and performance overhead.

Argus [12] utilizes a combination of computational redundancy, parity bits within the memory subsystems, and dynamic data flow verification [13] to verify both the control and data flow of software execution for simple processor cores. To verify the control and data flow of the application, Argus inserts signatures within the basic blocks of the application to indicate the set of valid successor blocks. At runtime, the control and data flow checkers can then verify the correct execution sequence by verifying that each subsequently executed basic block adheres to the statically determined execution order. Overall, Argus achieves a 98% error detection rate for unmasked errors and requires only 10% area overhead, but incurs a performance penalty of 4% on average and as high as 19%.

[8] proposes a compile time approach to detecting soft errors by inserting duplicate instructions within the compiled software application. In this approach, the output of the original instruction execution is stored in a hardware queue, such that when the duplicate instruction is executed the result can be compared to the original value – asserting an error if the two results do not match. Interestingly, this approach allows designers to tradeoff reliability – i.e. error detection rate – with the resulting performance and energy overhead by controlling the amount of the duplicated instructions.

Other approaches, such as [10][16] – originally targeted at detecting malicious code execution – can be utilized to detect soft errors that affect the control flow of a software application. These approaches either rely on verifying the function call execution sequence or verifying the target address for branch instructions by tracking which instruction addresses are valid sources for reaching a particular instruction destination.

3. DYNAMIC APPLICATION SOFT ERROR DETECTOR

Figure 1 (a) presents an overview of *DASED's* integration within a microprocessor system. *DASED* non-intrusively monitors a microprocessor's instruction bus to detect short backwards branches, function calls, function returns, and context switches. It considers a short backwards branch as any branch instruction whose target address is a negative offset of less than 1024, which corresponds to small loops containing less than 256 instructions. While the *DASED* design could directly decode the instructions on the instruction bus, we currently assume the microprocessor provides a one-bit output, *sbb*, indicating a short backwards branch has been executed, a one-bit output *func* signal indicating a function call is being executed, and a one-bit output *ret* indicating a function has returned. To detect function calls and function returns, *DASED* requires the address from which a function was called and the address to which the function returned. Such support is often already provided by many microprocessors [3][11][18], typically through a trace port used for debugging as illustrated in Figure 1 (b) for a Xilinx MicroBlaze system.

DASED non-intrusively monitors the address bus of the processor along with other signals to detect any soft errors in the system. Given the initial software application, an initial profiling or static analysis is utilized to determine the execution statistics for most critical – i.e. frequently executed – loops within the application. These execution statistics include the instruction address identifying the short backwards branch *(Tag)*, the size of the loop *(Offset)*, the minimum number of loop iterations per execution *(Minimum Iterations)*, and the maximum number of loop iterations per execution *(Maximum Iterations)*. This initial profiling or analysis step can either be executed statically at compile time using static analysis or simulation or dynamically at runtime during an initial execution phase in which the absence of errors can be guaranteed or verified.

During application execution, *DASED* will dynamically detect errors in the control flow of the application – i.e. execution behavior directly related to branches, function calls, and context switches – by comparing the dynamic application execution behavior with the statically determined loop execution statistics to detect soft errors that ultimately manifest in the change in execution behavior of the loops being profiled.

Figure 2 presents the internal architecture of the *DASED* design. *DASED* consists of a task filter for detecting context switches and instruction address errors, a FIFO to synchronize between the microprocessor and profile cache, a profile cache that stores the required profile statistics for those loops being

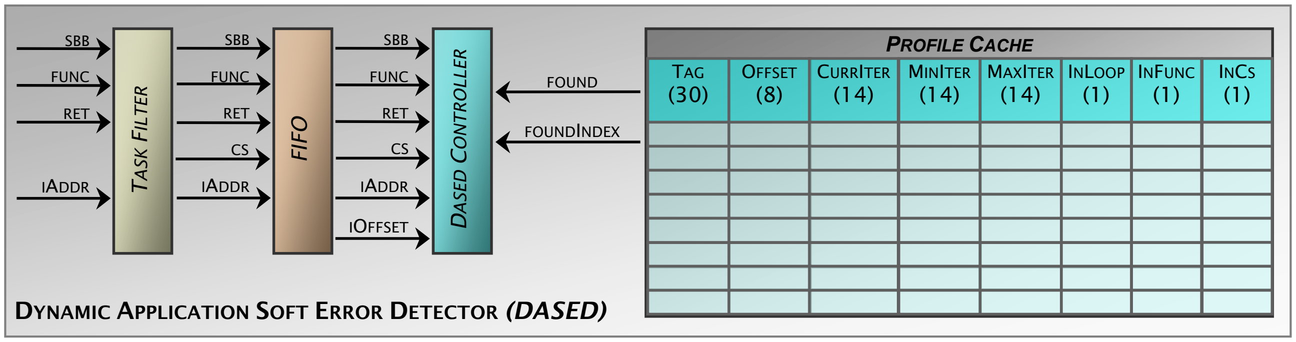


Figure 2. Architectural overview of Dynamic Application Soft Error Detector (*DASED*) consisting of a Task Filter, FIFO, *DASED* Controller, and Profile Cache *(bit widths for profile cache entries shown in parentheses)*.

monitored, and a *DASED* controller that analyzes the short backwards branches, function calls, function returns, and context switches to monitor loops' dynamic execution behavior to detect control errors.

3.1 Task Filter

The task filter is primarily utilized to non-intrusively detect context switches between the tasks being monitored. It is implemented as a programmable array storing the starting and ending address of each task, or any region of code, to be monitored. To detect context switches, the task filter monitors the processor's instruction bus to determine which task is currently executing or that no monitored tasks are executing. Whenever a change in context is detected – either from one monitored task to another or between a monitored task and a non- monitored task – the task filter will assert the *cs* output along with outputting the current address at which the change was detected. The task filter also filters the *sbb*, *func*, *ret*, and *iAddr* from the processor for all non- monitored regions of code. In other words, if the currently executing instruction does not fall within a monitored task, or code region, then the *sbb*, *func*, and *ret* inputs from the processor will be ignored.

The task filter also monitors the instruction addresses to detect errors related to the execution of instructions that are incorrectly aligned or do not correspond to valid instruction addresses within the application's address space. In addition to storing the address bounds for each task to be monitored, the task filter also stores the entire range of valid addresses for all code within the application, including any library or operating system code. If the instruction address does not fall within the valid address bounds or is incorrectly aligned, the task filter will directly assert an error.

3.2 FIFO

The FIFO monitors the *sbb*, *func*, *ret*, and *cs* signals from the task filter. Whenever a profile event is detected, the FIFO stores the *Tag* and *Offset* for short backwards branch, the originating address of function calls, the return address of function returns, or the instruction address immediately after a context switch, which are provided by the task filter. The *DASED* FIFO includes a small FIFO that stores the address of interest, short backwards branch offset when needed, and an encoding indicating if the entry is a short backwards branch, function call, function return, or context switch.

In addition, the FIFO is used to synchronize between the operating frequency of the microprocessor and task filter and the operating frequency of the internal *DASED* design because the microprocessor and task filter may operate at a higher clock frequency. As short backwards branches do not occur on every clock cycle, the internal *DASED* design need not operate at the same frequency of the microprocessor. A typical loop of interest within a software application consists of at least two to three instructions in addition to the short backwards branch at the end of the loop. We experimentally determined that the smallest profiled loop within the applications considered consists of 4 instructions. Hence, it should be sufficient to assume that short backwards branches on average occur no more than once every four instructions, implying the internal *DASED* design can efficiently operate at one fourth the operating frequency of the microprocessor. However, the FIFO should be large enough to accommodate bursts of short backwards branch activity that may occur periodically as an application executes.

In addition, *DASED* needs to monitor function calls, function returns, and context switches. The combined frequency of all events is not expected to increase the maximum expected frequency of such events. This is evident in the fact that both function calls and function returns require at least several instructions for maintaining the application's execution stack that limits their overall frequency. Similarly, context switches require dozens of instructions to store and restore task contexts.

3.3 Profile Cache

The profile cache is a small memory that maintains the current profiling results, intermediate state information needed for loop identification and monitoring, and the statically determined minimum and maximum iteration statistics for each loop. We currently consider a profile cache with 32 entries, which is sufficiently large to profile and monitor the embedded software applications considered within this paper – although a larger profile cache may be needed for significantly larger applications.

3.3.1 Loop Identification

Profiled loops are identified within the profile cache by the address of the loop's short backwards branch, which serves as the *Tag* entry for the cache, and by the loop's *Offset* determined by the FIFO. Considering a 32-bit ARM processor and a byte addressable memory, the lower two bits for all instruction addresses will be identical. Hence, the profile cache's *Tag* entry is a 30-bit entry that stores the most significant 30 bits of a loop's short backwards branch address. The profile cache's *Offset* entry is an 8-bit entry that corresponds to the size of the loop in number of instructions. As described earlier, both the *Tag* and *Offset* for a loop are calculated by the FIFO and provide the mechanism for identifying loop bounds.

3.3.2 Iteration Statistics

The main profiling information stored within the profile cache includes the statically determined minimum and maximum loop iterations along with the dynamic loop iterations for a loop's current execution. The *Current Iterations* provides the number of times a loop has iterated for the current loop execution and is stored within the profile cache as a 14-bit entry. As a 14-bit entry, *DASED* can accurately profile loops with a maximum of 16384 iterations per execution, which is very well suited for most applications. The *Minimum Iterations* and *Maximum Iterations* store the minimum and maximum number of times a loop iterates per execution in the absence of errors.

3.3.3 Loop/Function/Context Switch Monitoring

The profile cache contains a 1-bit *InLoop* flag utilized to indicate a loop is currently being executed. The *InLoop* flag is essential in determining if the execution of a short backwards branch corresponds to a new execution or an additional iteration for the current execution. A 1-bit *InFunc* flag is utilized to indicate a loop has called a function that is currently being executed. In addition, a 1-bit *InCS* flag is utilized to indicate a loop's execution has been interrupted due to a context switch. The *InFunc* and *InCS* flags are essential in ensuring that the *InLoop* flag for a loop that has called a function or whose execution has been interrupted due to a context switch is not incorrectly reset.

3.4 DASED Controller

Figure 3 presents pseudocode for *DASED* controller's operation. The controller interfaces with the FIFO and updates the dynamic profiling information for the loops within the profile cache. The *DASED* controller either receives the *sbb* signal along with the calculated branch offset, *iOffset* or one of *func*, *ret*, or *cs* signals from the FIFO in addition to a *found* and *foundIndex* signals from the profile cache. The *found* and *foundIndex* signals indicate if the current short backwards branch is found within the profile cache and at what location. In all cases the address of the instruction of interest is provided by the *iAddr* signal from the FIFO.

Whenever a short backwards branch is detected, the *DASED* controller will determine if the loop is found within the cache. If the loop is found and the loop is currently executing – as indicated by the loop's *InLoop* flag – the short backwards branch execution indicates a loop iteration has been detected and the loop's current iterations are incremented. Otherwise, if the loop is not currently being executed, a new loop execution is detected. For new loop executions, the controller sets the *InLoop* flag and sets the current iterations to one.

Whenever a context switch is detected, the controller first sets the *InCS* flags for all currently executing loops, i.e., those loops whose *InLoop* flag is still set. This can be efficiently implemented simply by copying all *InLoop* entries to the corresponding *InCS* entries within the profile cache. The controller then determines which loops, if any, will resume execution as the result of the context switch. Thus, if the address after a context switch falls within the bounds of any loops whose *InLoop* flag is set, the *InCS* flag for those loops is reset.

Whenever a function call is detected, the controller sets the *InFunc* flags for all currently executing loops, i.e., those loops whose *InLoop* flag is still set. This can be efficiently implemented simply by copying all *InLoop* entries to the corresponding *InFunc* entries within the profile cache. Whenever a function return is

```
DASED (iAddr, iOffset, sbb, func, ret, cs, found, foundIndex):

if ( cs ) {
   for all i, InCS[i] = InLoop[i]
   for all i, if ( InLoop[i] && (iAddr <= Tag[i] &&
                                  iAddr >= Tag[i]-Offset[i])
      InCS[i] = 0
}
if ( func ) {
   for all i, InFunc[i] = InLoop[i]
}
else if ( ret )
   for all i, if ( (InFunc[i] || InCS[i] ) &&
                   (iAddr <= Tag[i]  &&
                    iAddr >= Tag[i]-Offset[i]) ) {
      InFunc[i] = 0
      InCS[i] = 0
   }
}
else if ( sbb )  {
   if ( found ) {
      if ( InLoop[foundIndex] ){
         CurrIter[foundIndex] = CurrIter[foundIndex] + 1
         if (  CurrIter[foundIndex] > MaxIter[foundIndex] )
          return ERROR
      }
      else {
         CurrIter[foundIndex]            = 1
         InLoop[foundIndex]              = 1
      }
   }
}
if ( sbb || func || cs ){
   for all i, if ( InLoop[i] && !InFunc[i] && !InCS[i] &&
                !(iAddr <= Tag[i] && iAddr >= Tag[i]-Offset[i]) ) {
      InLoop[i] = 0
      if (CurrIter[foundIndex] < MinIter[foundIndex]  ||
         CurrIter[foundIndex] > MaxIter[foundIndex] )
           return ERROR
   }
}
```

Figure 3. Pseudocode for *DASED* controller

detected, the *DASED* controller resets the *InFunc* and *InCS* flags for those loops that contain the address of the function return's destination, i.e., the loops from which the corresponding function was called. We note that if a function call is executed from the innermost loop of a nested loop, the *InFunc* flag for all loops within the nested loop structure will be set. On return from that function call, the controller must reset the *InFunc* flags for all loops within the nested loops.

For all profiling events, the controller checks all entries of the profile cache whose *InLoop* flag is set to determine if the application is still executing within those loops. The controller also utilizes the *InFunc* and *InCS* flags to ensure that the *InLoop* flag is not incorrectly reset during a function call or context switch. For all detected short backwards branches, function calls, function returns, and context switches, the *DASED* controller checks all entries of the profile cache whose *InLoop* flag is set and whose *InFunc* and *InCS* flags are not set to determine if the application is still executing within those loops. If a loop is no longer being executed, the profile controller resets the *InLoop* flag.

The *DASED* controller detects control errors by monitoring the loop iterations per loop execution to verify that the current iterations for the past loop execution are within the statically determined minimum and maximum iteration bounds. As soon as the current loop iterations exceed the maximum iterations, the *DASED* controller will assert an error. Additionally, when the end of a loop execution is detected, the current iterations are compared with the minimum iterations, asserting an error if the current iterations are less than the minimum.

4. EXPERIMENTAL RESULTS

The *DASED* design was implemented in Verilog and synthesized using Synopsys Design Compiler targeting both a UMC 180nm and TSMC 90nm technology. With the 180nm technology, the *DASED* design requires 0.81 mm^2 with a maximum operating frequency of 835 MHz. The area required for the *DASED* design is approximately 7% of the area of an ARM9 with 32KB cache implemented within the same 180nm technology. Using the TSMC 90nm technology, the *DASED* design requires 0.14mm^2 with a maximum operating frequency of 1.36 GHz. The area required for the 90nm *DASED* design is approximately 6% of the area of an ARM1156T implemented within the same technology. Overall the *DASED* design can achieve operating frequencies suitable for high-end embedded systems – e.g. 1.2 GHz Marvell Kirkwood processor – while incurring a small area overhead. We reiterate that the *DASED* design is a non-intrusive design that does not affect the critical path of the target microprocessor or execution of the application being monitored. We further note that *DASED's* profile cache is currently implemented using registers. By re-implementing the profile cache using SRAM, we anticipate that a more area efficient design can be created, although we leave this as future work.

We have developed a taxonomy for soft errors that defines three types of soft errors: control errors, data errors, and masked errors. This taxonomy is specifically related to single event upsets and is useful in categorizing the types of errors and how those errors ultimately affect an application's execution. Control errors are those errors that affect the control flow of an application leading to noticeable changes in the application execution or even catastrophic failure – e.g. different instruction executed, fetching instruction from memory outside of the valid application address bounds. Data errors are those errors that only affect the data outputs from an application – e.g. producing incorrect pixel values in a video stream – but do not affect the control flow of the application. Masked errors are those errors that neither affect the control flow of an application nor affect the data outputs from that application – e.g. a soft error may affect the value used within comparison for which the resulting comparison yields the same result. Conversely, unmasked errors are defined as any error that either affects the control flow or data outputs of an application. In other words, unmasked errors correspond to the set of all control and data errors.

In order to distinguish between control, data, and masked errors, we developed an automated simulation environment capable of determining if the inserted error is a control error, data error, or masked error. Initially, we simulated each application using the SimpleScalar simulator [5] without errors to determine an error-free instruction trace along with tracing all data outputs from the application execution. This error-free trace can then be utilized later to determine if an inserted error within the application execution is manifested as a control error, data error, or masked error. Applications are then simulated again with a soft error randomly inserted within the application execution, and the resulting instruction and data traces are generated. If the two instruction traces do not match, the inserted error is considered a control error as it results in a change in the instruction execution sequence. Alternatively, if only the data outputs from the two executions do not match, then the inserted error is considered a data error. If neither the instruction traces nor data traces differ, the inserted error is considered a masked error.

Table 1. Overview of single-tasked and multitasked applications composed of applications from the MiBench benchmark suite.

	CJPEG	*DJPEG*	*FFT*	*TIFF2BW*	*TIFF2RGBA*	*SUSAN*	*DIJKSTRA*	*BIT COUNT*	*STRINGSEARCH*	*QSORT*	*RAWCAUDIO*	*RAWDAUDIO*
MT2.1			✓		✓							
MT2.2	✓									✓		
MT2.3						✓	✓					
MT2.4		✓		✓								
MT2.5								✓	✓			
MT2.6	✓	✓										
MT2.7				✓	✓							
MT3.1								✓	✓			✓
MT3.2			✓		✓						✓	
MT3.3	✓					✓				✓		
MT3.4		✓		✓			✓					
MT4.1	✓					✓	✓			✓		
MT4.2		✓		✓					✓		✓	
MT5.1	✓		✓			✓	✓			✓		

We performed extensive experiments using our automated simulation and soft error injection environment for 12 single task applications and 14 multitasked applications consisting of two to five tasks, where each task corresponds to an application from the MiBench benchmark suite [7]. Table 1 presents an overview of the various single-tasked and multitasked applications considered. All applications were executed within the RTEMS operating system [14].

Each application was executed 100 times during which a single event upset was randomly inserted within the application execution. **Table 2** presents a breakdown of the percentage of data errors (%DE), percentage of control errors (%CE), and percentage of masked errors (%ME) within each application, and highlights *DASED's* percentage of detected control errors (%DCE), percentage of detected unmasked errors (%DUME), mean time to detection with 90% confidence interval, minimum time to detection, and maximum time to detection for all applications considered. For evaluating the mean, minimum, and maximum time to detection, we assume a microprocessor operating frequency of 1 GHz.

On average, the *DASED* design is able to detect 92% of all control errors and 85% of all unmasked errors across the various applications considered. However, for a few applications, specifically *MT1.fft*, *MT1.rawcaudio*, and *MT1.rawdaudio*, the unmasked error detection rate is less than 50%. This low error detection rate can be partially attributed to a higher prevalence of data errors that affect the data output without affecting the application control flow, which cannot be detected using our *DASED* design.

For those errors that are detected, the mean time to detection is on average 9869±1809 µs. The variation in detection time between applications, as well as cross multiple simulations of the same application, is a result of both the type of error and the method by which errors are detected using *DASED*. For example, if a soft error results in the invalid execution of an instruction at an address outside of the application bounds, then the error can be immediately detected within a few cycles – and in many cases 0 cycles. However, if the soft error results in a change in the execution behavior of a loop – e.g. a loop's iterations exceeds the statically determined maximum iterations – the detection time is affected by both the size of the loop and the maximum iteration bound for that loop. The largest time to detection across all applications is 229513 µs (or 229 ms). This implies that while a soft error may have little impact on the application execution at the time of occurrence, the effects of that error can have lasting long-term impact on the application execution.

5. CONCLUSIONS AND FUTURE WORK

DASED is capable of non-intrusively detecting soft errors within the execution of a software application without requiring modifications to the application or the target processor hardware. For the applications considered, *DASED* achieves an error detection rate of 92% for control errors and 85% for all unmasked errors with a mean time to detection of less than 10 ms. Furthermore, the *DASED* design is an area efficient design that requires only 6-7% area overhead and can achieve operating frequencies as high as 1.36 GHz. Future work includes integrating *DASED* within a prototype embedded systems using a microprocessor's trace port to provide the required runtime signals. Furthermore, we are currently investigating additional non-intrusive methods that leverage existing data profiling methods to monitoring the data access behavior in order to detect data errors within *DASED*.

6. ACKNOWLEDGMENTS

7. REFERENCES

[1] Nair, A., R. Lysecky. Non-Intrusive Dynamic Application Profiler for Detailed Loop Execution Characterization. International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES), pp. 23-30, 2008..

[2] Shankar, K., R. Lysecky. Non-Intrusive Dynamic Application Profiling for Multitasked Applications. Design Automation Conference (DAC), pp. 130-135, 2009.

[3] ARM, Ltd. CoreSight On-chip Debug and Real-time Trace Technology. http://www.arm.com/products/solutions/CoreSight.html, 2009.

[4] Austin, T. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. International Symposium on Microarchitecture (MICRO), 1999.

[5] Burger, D., T. Austin. The SimpleScalar Tool Set, Version 2.0. University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June 1997.

[6] Buttazzo, G. Achieving Scalability in Real-Time Systems, Computer, Vol. 39, No. 5, pp. 54-59, 2006.

Table 2. Breakdown of the percentage of data errors (%DE), percentage of control errors (%CE), and percentage of masked errors (%ME) within each application, along with *DASED*'s percentage of detected control errors (%DCE), percentage of detected unmasked errors (%DUME), mean time to detection with 90% confidence internal, minimum time to detection, and maximum time to detection for all applications considered.

						Time to Detection (µs)		
Application	%CE	%DE	%ME	%DCE	%DUME	Mean	Min	Max
MT1.cjpeg	85	0	15	98	98	4757±1106	0	12880
MT1.djpeg	100	0	0	100	100	6521±1039	0	12909
MT1.fft	66	17	17	61	48	19606±7061	0	82818
MT1.tiff2bw	58	12	30	91	76	30±34	0	1080
MT1.tiff2rgba	64	3	33	100	96	181±114	0	3073
MT1.susan	34	0	66	100	100	2135±846	0	7899
MT1.dijkstra	64	20	16	88	66	5391±2237	0	30089
MT1.bitcount	74	19	7	80	63	39±14	0	337
MT1.stringsearch	73	0	27	96	96	265±139	0	3100
MT1.qsort	55	0	45	89	89	3872±2002	0	37300
MT1.rawcaudio	76	10	14	41	37	2552±535	0	5416
MT1.rawdaudio	60	30	10	57	40	2917±2050	0	25839
MT2.1	68	7	25	96	87	31298±7895	0	127950
MT2.2	100	0	0	100	100	6675±1206	0	16736
MT2.3	63	1	36	89	88	3428±1444	0	26412
MT2.4	100	0	0	100	100	10797±772	0	12835
MT2.5	75	15	10	97	81	5024±1174	0	19833
MT2.6	100	0	0	100	100	2866±818	0	14409
MT2.7	70	7	23	100	91	335±101	0	2026
MT3.1	69	14	17	97	81	3055±1114	0	34954
MT3.3	76	18	6	92	74	48781±7283	0	100988
MT3.3	100	0	0	100	100	13455±524	0	15112
MT3.4	100	0	0	100	100	15955±898	0	18134
MT4.1	100	0	0	100	100	23098±817	180	24710
MT4.2	100	0	0	100	100	13482±1528	0	20560
MT5.1	100	0	0	100	100	30068±4294	0	229513
Average:	78.1	6.6	15.3	92	85	9869±1809	7	34112

[7] Guthaus, M., J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. Workshop on Workload Characterization, 2001.

[8] Hu, J., F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, M. J. Irwin. Compiler Assisted Soft Error Detection Under Performance and Energy Constraints in Embedded Systems. ACM Transactions on Embedded Computing Systems, Vol. 8, No. 4, Article 27, 2009.

[9] International Technology Roadmap for Semiconductors. http://www.itrs.net/, 2008.

[10] Kiriansky, V., D. Bruening, S. Amarasinghe. Secure Execution via Program Shepherding, USENIX Security Symposium, pp. 191-206, 2002.

[11] Malik, A., B. Moyer, D. Cermak. A Low Power Unified Cache Architecture Providing Power and Performance Flexibility. International Symposium on Low Power Electronics and Design (ISLPED), pp. 241-243, 2000.

[12] Meixner, M., M. Bauer, D. Sorin. Argus: Low-cost, Comprehensive Error Detection in Simple Cores. IEEE Micro, Vol. 28, No. 1, pp. 52-59, 2008.

[13] Meixner, M., D. Sorin. Error Detection Using Dynamic Data flow Verification. International Conference on Parallel Architectures and Compilation Techniques, 2007.

[14] Real-Time Operating System for Multiprocessor Systems (RTEMS), http://www.rtems.org, 2008.

[15] Siewiorek, D., R. Swarz. The Theory and Practice of Reliable System Design. Digital Press, Bedford, Massachusetts, 1991.

[16] Shi, W., H.-H. Lee , L. Falk , M. Ghosh. An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processors. International Symposium on Computer Architecture (ISCA), pp.102-113, 2006.

[17] Srinivasan, J. S. Adve, P. Bose, J. Rivers. The Impact of Technology Scaling on Lifetime Reliability. International Conference on Dependable Systems and Networks (DSN), 2004.

[18] Xilinx, Inc. Setup of a MicroBlaze Processor Design for Off-Chip Trace. http://www.xilinx.com/support/documentation/application_notes/xapp1029.pdf, 2008.