Prototype Test Insertion Co-processor for Agile Development in Multi-threaded Embedded Environments

Dongcheng Deng[1], Michael Smith[1], Syed Islam[2] and James Miller[3]

[1] *Department of Electrical and Computer Engineering*

University of Calgary, Calgary, Alberta, Canada, T2N 1N4

[2] *Department of Electrical Engineering and Computer Science*

York University, North York, Ontario, Canada, M3J 1P3

[3] *Department of Electrical and Computer Engineering*

University of Alberta, Edmonton, Alberta, Canada, T6G 2R3

E-mail: (ddeng, smithmr)@ucalgary.ca, sislam@cse.yorku.ca, jimm@ualberta.ca

Abstract − Agile methodologies have been shown useful in constructing Enterprise applications with a reduced level of defects in the released product. Movement of Agile processes into the embedded world is hindered by the lack of suitable tool support. For example, software instrumented test insertion methods to detect race condition in multi-threaded programs have the potential to increase code size beyond the limited embedded system memory, and degrade performance to an extent that would impair the real-time characteristics of the system. We propose a FPGA-based, hardware assisted, test insertion co-processor for embedded systems which introduces low additional system overhead and incurs minimal code size increase. In this preliminary study, we compare the ideal characteristics of a FPGA-based test insertion co-processor with our initial prototype and other proposed hardware assisted test insertion approaches.

Keywords − Embedded systems development, multi-thread program, race condition detection, agile testing, FPGA-based co-processor, hardware-assisted test insertion.

I INTRODUCTION

Economic losses associated with inadequate software testing are stated as being $59 billion / year in the US alone [1]. Hence, it is not surprizing that considerable effort has been made to mitigate these losses with technically-feasible cost reductions expected to be in the range of $22 billion / year. With Agile methodologies shown as successful in desktop software application development [2], attempts were made to transfer this defect-reducing approach into the embedded software development world [3], [4] . EXtreme Programming Inspired (XPI) embedded software development [5] and Embedded Test Driven Development (Embedded TDD) [6] lifecycles approaches are, to re-phrase Beck [2], "*intended to infect the whole embedded development process with tests in an attempt to successfully generate products with low defect rates*".

In the embedded world, we envision two general testing environments

(A) Development of new code; and

(B) Merging new code with existing customer legacy code or recently acquired third party software.

In this paper we will consider an application involving multi-threads, operating in either single or multi-core environments. Such systems require the developer / tester to:

(1) Ensure that shared memory or devices are accessed by two or more threads with proper synchronization; and

(2) Quickly identify, during testing, that suspended threads are waiting for a resource beyond the system's hard time requirements.

A key question is how to modify automated test development and testing tools supporting Agile methodologies to encourage and enable the embedded developer to build and frequently run tests. Movement of enterprise system software instrumentation approaches to race detection into the embedded environment can be frustrated by limited system memory; and the need to avoid introducing false positive or negative results into the more stringent real-time characteristics of the embedded systems.

To become more Agile in testing for possible data races in a multi-threaded embedded environment, we have previously suggested adapting the extensive debug hardware present in modern embedded processors for use as dynamic testing mechanisms [7], [8]. For example, a processor's instruction address bus watch-unit normally assists in setting break-points during debugging of a live system-under-test. However, we suggest it can be used to generate a zero overhead testing task to detect threads blocking beyond design requirements as follows. A hardware timer interrupt of a specified duration could be activated prior to a thread being suspended. A hard time constraint error would be reported unless this interrupt was de-activated by the watch unit upon detecting the execution of the first instruction of the reactivated thread.

In this paper, we present the concept of using a FPGA-based test insertion co-processor to provide any processor with the ability to watch (recognize) specific instruction or data memory activity. The intention is that the co-processor will provide a developer working within Agile (test first) or the conventional Waterfall and V-shaped (test-last) processes a low-overhead approach capable of

(A) Identifying that no new data races have been introduced into a newly developed multi-thread systems; and

(B) Providing the capability to identify the possible data races in legacy or third party systems caused by complex interactions of existing threads without detailed code knowledge, i.e. working with compiled object code rather than source code,

The remainder of this paper is organized as follows: related works showing the advantages and limitations of the existing data races detection methods in an embedded context are presented in Section 2. Section 3 discusses the ideal characteristics required for a FPGA-based test insertion co-processor. Section 4 presents our concept for implementing a prototype FPGA-based test insertion co-processor. In Section 5 we compare the co-processor's expected performance with the hardware instruction watch and data watch hardware unit found on the Analog Devices BF5XX (Blackfin) family of processors in the context of two case studies. Finally, Section 6 summarizes this paper and proposes directions for future work.

II Related Work

Previous research has reported on software methods to insert testing code into existing source code in order to detect race conditions. However, the overheads of software instrumentation have limited their applicability. One comment [9] was

It is NOT UNUSUAL for the instrumented code to be 20 times larger in size than the non-instrumented code and suffer from a 300 fold (300x) performance loss.

These drawbacks limit software instrumentation methods from being practical in resource-stringent embedded systems.

To reduce overhead, architectural changes to the processor's silicon were suggested in [10] to provide test insertion capability. As a low cost alternative to such custom design approaches, Smith *et al.* [7] and Huang *et al.* [8] proposed the use of existing hardware debugging features found on many common embedded processors to provide a test insertion mechanism.

They examined the Analog Devices Blackfin ADSP BF-5XX family of processors which had an on-chip debug architecture containing both instruction watch (IW) and data watch (DW) hardware units [11] . The existing instruction watch unit had capabilities close to what might be considered as ideal hard-ware assisted test insertion capabilities; i.e. zero changes were required to existing code to insert a test and no overhead was incurred while trying to recognize whether a particular instruction was being blocked. Huang *et al.* [8] showed that an overhead of less than 20 cycles, roughly twice the Blackfin processor's pipeline length, was needed to insert a test that formed part of an exception interrupt routine triggered by the instruction watch unit.

This processor's data watch unit also required zero code changes and no overhead to detect multi-threads accessing a specific shared memory location. However, in contrast to the instruction watch unit action, the data watch unit activated

emulation interrupt routines. These ran in the external development environment and required tenths of seconds, 200,000+ cycles, to insert a required test.

Given these issues, and the fact that other processors also have limited hardware debugging capability, we propose the development of a FPGA-based test insertion co-processor that could, in principle, be added to the busses of any processor to provide low overhead, hardware assisted, test insertion support.

III IDEAL CHARACTERISTICS

To assist in identifying the ideal characteristics of a FPGA test-insertion co-processor, we imagine that it is used in the context of testing to identify the presence of possible data races. Test insertion methods provided by the system should not change the result of data race analysis; i.e. neither introducing new data races (false positives) nor masking existing data races (false negatives).

Extending the ideas of Huang *et al.* [8], the co-processor should be able to insert a test without the need for the developers to modify the original code. We suggest that this requirement can be met by giving the co-processor the capability of monitoring and capturing instruction, data and control bus activities from the processor. To achieve best performance, test insertion mechanisms should work concurrently with, and require minimal participation from, the processor.

The performance impact of the co-processor on the running thread can be estimated using the analysis approach suggested in [7]

- Since the co-processor works concurrently with the processor and unobtrusively in the background, there is no cost until a specified data / instruction activity is recognized and the processor has to respond to the activity.
- An overhead of $C_{RECOGNIZE}$ clock cycles are required for the processor to respond to the co-processor's request to insert a test; and
- C_{TEST} clock cycles are required for the processor to execute the test itself.

Assume that the program contains N instructions of which a fraction of k of these N instructions require test insertion. The performance ratio, PR, comparing the execution time of the instrumented code and the original un-instrumented code is:

$$PR = \frac{Instrumented\ Code\ Execution\ Time}{Original\ Code\ Execution\ Time} = 1 + k(C_{TEST} + C_{RECOGNIZE} + 1) \quad (1)$$

where C_{TEST} cycles are required to perform the actual test.

We have considered a number of ways to minimize $C_{RECOGNIZE}$ and C_{TEST} to reduce the overhead of test insertion. For example the C_{TEST} overhead can be made smaller if the co-processor testing environment can be configured to gather test information for later off-line analysis. This approach was proposed for the hardware-assisted test coverage tool *E-COVER* [12]

An interesting feature of the embedded system debug hardware on the Analog Devices Blackfin processor was the 'algorithmic awareness' of the hardware trace unit. The debug program flow trace unit can be configured to only respond the first time around a tight loop associated with a signal processing algorithm commonly used in embedded applications [12]

The total overhead for the co-processor to check for valid locks within long loops accessing a common block of shared memory would be reduced if a similar capability was provided to the co-processor. A small additional, once a loop, overhead would be incurred as the processor informed the co-processor to stop checking for valid data locks while the loop continued executing. The co-processor instruction watch unit would be configured to reactivate the test insertion when the co-processor's instruction watch unit recognized that the first instruction after the loop was being executed.

Since the Blackfin processor data and instruction watch unit are on chip, they are capable of responding fast enough to activate an exception handler that blocked (interrupted) the instruction or data operation being watched. This implies that the $C_{RECOGNIZE}$ for these units will have two components

- The cycles for entering and exiting the exception handler, and
- The additional cycles required reconfiguring the watch unit's *WATCH_COUNTER* register to allow the watched activity to complete normally as the interrupted instruction or data watched stream is re-activated upon exiting the exception handler. Otherwise an infinite loop of exceptions will be generated.

As an external co-processor can't, or rather should not, cause an exception, we propose the use of the non-maskable interrupt (NMI) input line to cause the processor to insert a test. Entering and exiting the NMI handler will incur approximately the same overhead as for the exception handler on most

processors. As explained above, the co-processor architecture must include a *WATCH_COUNTER* register if FPGA's clock speed is fast enough to cause the instruction generating the watched activity to be interrupted. For slower FPGA implementations, the watched activity will produce the NMI signal after the activity has completed; i.e. an instruction not involving a watched activity will be interrupted. This can be resumed without producing unintentional additional instruction or data watch interrupts.

IV Implementation of FPGA-based Test Insertion Co-processor

a) Proposed architecture and use

Fig. 1 illustrates the proposed use of our FPGA-based test insertion co-processor for inserting tests in a multi-threaded test environment. In Step 1, a run-once function is used to configure the FPGA-based test insertion co-processor's *Memory-Mapped Watch Unit Registers* to watch instructions within a specific thread component or any access to a shared memory location via *SPI* or standard memory write instructions. The specific locations of the instruction or data can be specified as part of the project's loader description file (*.ldf*) or determined from the linker map. Step 2 is to allow the original, unmodified, threads to execute while (Step 3) the processor's instruction and data address busses are monitored by the co-processor.

No processor overhead is incurred (Step 4) until the co-processor recognizes a defined watched activity. At this time (Step 5) the co-processor pulls the processor's NMI line high. On receiving the *NMI* signal (Step 6), the processor switches to a *NMI* handler which (Step 7) inserts the test which immediately analyses the watched condition or stores test information for later analysis. Finally (Step 8) the interrupted watched activity is allowed to complete.

If the FPGA implementation has a slow clock speed, the co-processor will only need to reconfigure itself to immediately recognize further watch activity during Step 5. However this approach is not suitable if (A) the FPGA clock is as fast as the FPGA test-insertion co-processor, or (B) the co-processor is watching a loop involving access to a block of shared memory. In this case the co-processor will configure itself to ignore the next *Q* watched activities to (A) avoid setting up an infinite stream of *NMI* signals or (B) unnecessarily insert the same test to validate the lock of a shared memory blocks.

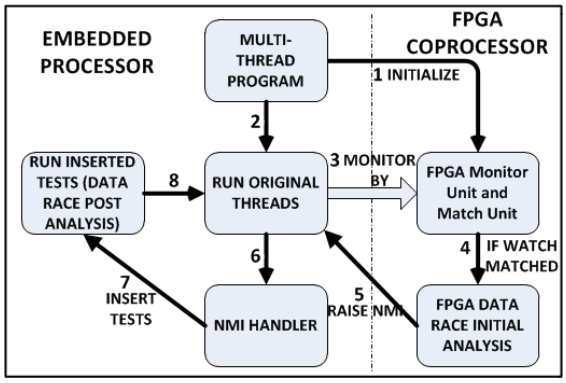

Fig. 1: Program flow of the FPGA-based test insertion system. (1) The multi-thread program running on the embedded processor first initializes the co-processor via an *SPI* or standard memory write instructions. (2) Then the original threads are run with (3) co-processor monitoring the processor's data and address bus activities. On finding a watched activity (4), an NMI is generated (5) causing the processor to (6) run an NMI handler which (7) inserts a test before (8) allowing the watched activity to execute without further interruptions.

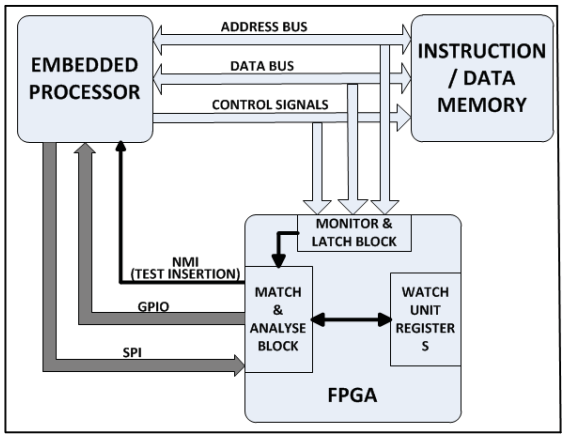

Fig. 2: FPGA-based co-processor monitors the buses activities and latches the values. The co-processor's *Match & Analyse Block* compares the latched value to the watched value stored in *Watch Unit Registers*. On recognizing the desired watched activity, the processor's *NMI* pin is asserted. Co-processor configuration information is sent by the processor over the *SPI* interface. Information, such as which watched activity has been recognized, can be send back to the processor over the faster, parallel general purpose input / output (GPIO) interface when the NMI is asserted. By sending configuration message via GPIO, *SPI* or Address and Data buses, the *Watch Unit Registers* of FPGA can be configured by the embedded processor.

Fig. 2 shows how we propose to have the FPGA-based co-processor recognize specific watched activities. If the processor has a von-Neuman architecture then co-processor only needs to monitor the common address bus used to fetch instructions or data. For a Harvard architecture processor, the co-processor must sniff the address buses of the

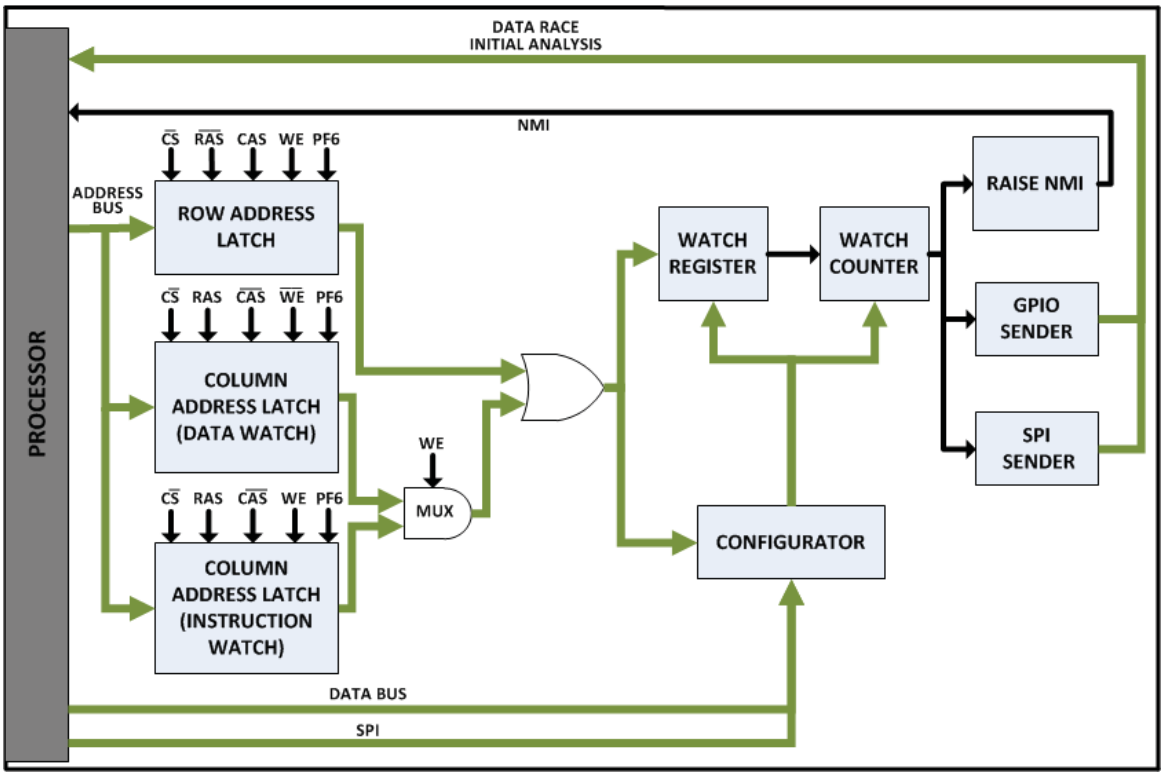

Fig. 3: Schematic of the proposed FPGA-based test insertion co-processor hardware logic as implemented on the Blackfin *FPGA* daughter board [13] . The green lines are buses while the black lines denoted single signal line. PF6 line from Blackfin acts as the FPGA-based test insertion co-processor watch-enable signal.

separate instruction (program) and data memory blocks. The processor's control bus can be used to allow the co-processor to distinguish between writes to and reads from a shared data memory location or resource. Activating reads without a valid lock is often considered an acceptable programming practice. This approach would speed the testing up by avoiding the insertion of what the developer considers as unnecessary tests.

b) Practical Implementation

We implemented our prototype co-processor on a *FPGA* evaluation board [13] designed to interface directly to the address, data and control busses of the Analog Devices ADSP-BF533 Blackfin processor. This provided an easy route to compare the speed of our co-processor insertion approach to the existing Blackfin instruction watch and data watch debug hardware which respectively provide viable and non-viable test insertion mechanisms.

The co-processor would be able to snoop the internal buses of the ADSP-BF533 if the FPGA is directly connected to the embedded core of the embedded processor. For this prototype we were limited to monitoring the address and data buses to the external SDRAM memory blocks of the Blackfin evaluation board as shown in Fig. 3.

When the SDRAM chip-select (CS) and write-enable (WE) became active, the row access select line (RAS) was used to latch the upper bits of the processor's address bus before the column access select line (CAS) latched the lower bits of the address[14]. The latched components of the address bus were then combined and presented to a bank of watch registers to compare to known data address bus or program address bus values stored during the co-processor initialization phase.

On recognizing the specific watched activity, the watch register block raised the processor's NMI signal high. If the processor contains multiple watch blocks, the watch block must send an identification byte to the processor over the *SPI* interface. Alternatively an identification byte can be placed on the processor's GPIO pins if these are not in use.

The configuration commands can be sent via standard write instructions to SDRAM, which has performance advantage of configuring via SPI. However, co-processor initialization speed is not a concern as this occurs prior to the time critical execution of the threads. Therefore, it is probably easier to use the *SPI* lines – with handshaking mechanisms – to pass initialization information between processor and co-processor given the disparity between the processor and the FPGA clock rates.

The co-processor architecture must support *WATCH_COUNTER* and *WATCH_PERIOD* registers if the FPGA clock is fast enough to interrupt the watched activity. In this situation, the co-processor must be de-activated to avoid generating a second, invalid, NMI signal when the watched activity is allowed to complete after test insertion. These registers are also needed to make the co-processor algorithm aware, e.g. if the co-processor is to be de-activated during a loop to increase the test performance ratio.

During initialization, or in principle during run-time as performed with the *E-COVER* test coverage unit [12] , the *WATCH_PERIOD* register of a *WATCHUNIT* block is set to the number of watched activities that are to be ignored. When the test is activated the *WATCH_PERIOD* value is loaded into the *WATCH_COUNTER* register. As each watched activity is recognized, the *WATCH_COUNTER* is decremented. The *WATCHUNIT* sends the NMI signal to the processor when a zero count is reached. The *WATCH_COUNTER* is reloaded with the *WATCH_PERIOD* value to prepare for the next watch operation.

V Performance Analysis of FPGA-based Test Insertion Co-processor

a) Ideal Blackfin test insertion performance

From Eqn. (1) we have that the performance ratio for the FPGA-based test insertion co-processor is

$$PR = \frac{Instrumented\ Code\ Execution\ Time}{Original\ Code\ Execution\ Time} = 1 + k(C_{TEST} + C_{RECOGNIZE} + 1)$$

Information on the cost on entering and exiting a non-maskable interrupt service routine on the Blackfin BF5XX can be found in [15]. For this processor it requires 10 cycles to jump into the NMI routine and 5 cycles to jump out, leading to $C_{RECOGNIZE} = 15$. If we assume that 1% of the data or instruction activity requires watching, $k = 0.01$, then the overhead of recognizing when to insert a tests gives a performance ratio of *1.15*

b) Comparison with the hardware test insertion mechanisms

The actual performance loss of FPGA-based test insertion co-processor was 27 cycles. This is higher than the theoretical 15 cycles simply because of an additional 12 instruction overhead of a software patch required ensuring that the NMI handler was not impacted by a hardware anomaly in Blackfin silicon version 0.5 or lower [16].

The reported four test insertion techniques using existing architectural debugging hardware features presented in embedded processors are provided here so as to compare the performance result of the FPGA-based test insertion co-processor:

- Existing Blackfin Instruction Watch Hardware [8]: 19 cycles / test insertion (ideal) and 22 cycles / test insertion (actual). The additional cycles over the co-processor performance are produced by the instructions required to re-configure the instruction watch count register in order to allow the Instruction Watch Hardware to re-activate the watching. This is automatically handled by the FPGA-based test insertion co-processor's *WATCH_COUNT* and *WATCH_PERIOD* register functionality. Further overhead is incurred by pipeline issues associated with storing and recovering registers used to reconfigure the Blackfin's watch count register.

- Existing Blackfin Data Watch Hardware [8]: 19 cycles / test insertion (ideal) and 200,000+ cycles / test insertion (actual) as the processor switches to an emulation mode that interacts with the external development system.

- Dual Instruction Watch and Data Watch method [8]: 350 cycles / test insertion (ideal) and 480 cycles / test insertion (actual). This approach avoids the emulation problem by checking the Blackfin data watch register every few instructions via exceptions caused by the instruction watch unit rather than allowing the data watch unit to activate the non-ideal emulation handler.

- Dual data cache and instruction watch methods [7]: 60 cycles / test insertion (ideal) and 84 cycles / test insertion (actual). To recognize the watched data activity, the watched data address is placed in into a data cache which is then invalidated. On attempting to access a data value stored in the invalidated cache, the processor activates a fill cache interrupt which can be used to insert the test. In principle, this test insertion approach is available on all processors with very low overhead. However, a mechanism, such as an instruction watch register, must also be present to re-invalidate the cache as otherwise only one test can be inserted.

Comparing to the hardware-assisted test insertion methods using the existing debugging hardware, the FPGA-based test insertion co-processor provides a

smaller performance loss and more applicability to all embedded processors.

VII Conclusion & Future Work

Agile methods are proven to be successful in the desktop application development world to reduce the defects in released code. Efforts have been made to transfer this success to embedded software development. The major obstacle, however, is the lack of tools to support applying Agile methodologies onto the embedded world. We proposed a FPGA-based test insertion co-processor test insertion system and demonstrate its possible use to detect data races and blocked resource activity in multi-threaded embedded systems without requiring the embedded processor having specific hardware debugging features. A performance analysis shows that the FPGA-based test insertion co-processor outperforms other methods that attempt to use existing embedded system debugging hardware to implement the test insertion. The next step is to port code benchmarks used to evaluate software instrumentation approaches to data-races to the embedded environment to allow a practical comparison of hardware and software test insertion mechanisms.

Acknowledgements

Financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC), Analog Devices (US), and the University of Calgary.

References

[1] G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing," *National Institute of Standards and Technology,* 2002.

[2] K. Beck. "Extreme Programming Explained," *Embrace change,* 2006.

[3] B. Greene, "Agile methods applied to embedded firmware development," *Agile Development Conference 2004, IEEE*, 2004, pp. 71-77.

[4] D. Dahlby, "Applying Agile Methods to Embedded Systems Development," *Embedded Software Design Resources*, *41*, 2004, pp. 101-123.

[5] M. R. Smith, J. Miller, L. Huang, A. Tran, "A More Agile Approach to Embedded System Development," *IEEE Software (Special Issue on Embedded Software),* 2009, pp. 50-57.

[6] J. Grenning, "Applying test driven development to embedded software," *Instrumentation & Measurement Magazine, IEEE*, volume 10, issue 6, 2007, pp. 20-25.

[7] M. R. Smith, M. Helmi, J. Miller, "Comparison of Approaches to Use Existing Architectural Features in Embedded Processors to Achieve Hardware-Assisted Test Insertion," *Proceedings Work-in-Progress Session of the 22nd Euromicro Conference on Real-Time Systems (ECRTS'10),* Brussels, Belgium, 2010.

[8] F. Huang, M.R. Smith, A. Tran, J. Miller, "E-RACE, A Hardware-Assisted Approach to Lockset-Based Data Race Detection for Embedded products," *19th International Symposium on Software Reliability Engineering, ISSRE 2008,* Seattle, USA, 2008, pp. 277-278.

[9] Intel. "Intel Corporation, Intel Thread Checker," [Online]. Available: www.intel.com/support/performancetools/threadchecker/windows/, Accessed: May, 2007.

[10] P. Zhou, R. Teodorescu, Y. Zhou. "HARD: Hardware-Assisted Lockset-based Race Detection," *IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*, 2007, pp. 121-132.

[11] Analog Devices, (2013, February), "Blackfin ® Processor Programming Reference, Revision 2.2, February 2013," [Online]. Available: http://www.analog.com/static/imported-files/processor_manuals/Blackfin_pgr_rev2.2.pdf/, Accessed: April, 2013.

[12] A. Tran, M. R. Smith, J. Miller, "A hardware-assisted tool for fast, full code coverage analysis", *19th International Symposium on Software Reliability Engineering 2008. ISSRE 2008,* IEEE, 2008, pp. 321-322.

[13] Analog Devices, (2012, July), "Blackfin FPGA EZ-Extender Manual, Revision 2.1, July 2012". [Online]. Available: http://www.analog.com/static/imported-files/eval_kit_manuals/Blackfin_FPGA_ext_man_rev.2.1.pdf, Accessed: April, 2013.

[14] Micron, (2013, March), "512Mb SDRAM Component Data Sheet," [Online]. Available: http://download.micron.com/pdf/datasheets/dram/sdram/512MbSDRAM.pdf, Accessed: April, 2013.

[15] Analog Devices, (2003, September), "ADSP-BF531/532/533 Blackfin® Processor Multi-cycle Instruction and Latencies," [Online].

Available: http://www.analog.com/static/imported-files/application_notes/EE-197.pdf, Accessed: April, 2013.

[16] Analog Devices, (2011, May), "ADSP-BF531/BF532/BF533 Blackfin Anomaly List for Revisions 0.3, 0.4, 0.5, 0.6 (Rev G, 05-23-2011)," [Online]. Available: http://www.analog.com/static/imported-files/ic_anom/ADSP-BF531_BF532_BF533_anomaly_RevG.pdf, Accessed: April, 2013.