Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 406857, 17 pages
doi:10.1155/2011/406857

Research Article

High-Level Synthesis of In-Circuit Assertions for Verification, Debugging, and Timing Analysis

John Curreri, Greg Stitt, and Alan D. George

NSF Center for High-Performance Reconfigurable Computing (CHREC), ECE Department, University of Florida, Gainesville, FL 32611-6200, USA

Correspondence should be addressed to John Curreri, curreri@hcs.ufl.edu

Received 13 August 2010; Accepted 14 December 2010

Academic Editor: J. M. P. Cardoso

Despite significant performance and power advantages compared to microprocessors, widespread usage of FPGAs has been limited by increased design complexity. High-level synthesis (HLS) tools have reduced design complexity but provide limited support for verification, debugging, and timing analysis. Such tools generally rely on inaccurate software simulation or lengthy register-transfer-level simulations, which are unattractive to software developers. In this paper, we introduce HLS techniques that allow application designers to efficiently synthesize commonly used ANSI-C assertions into FPGA circuits, enabling verification and debugging of circuits generated from HLS tools, while executing in the actual FPGA environment. To verify that HLS-generated circuits meet execution timing constraints, we extend the in-circuit assertion support for testing of elapsed time for arbitrary regions of code. Furthermore, we generalize timing assertions to transparently provide hang detection that back annotates hang occurrences to source code. The presented techniques enable software developers to rapidly verify, debug, and analyze timing for FPGA applications, while reducing frequency by less than 3% and increasing FPGA resource utilization by 0.7% or less for several application case studies on the Altera Stratix-II EP2S180 and Stratix-III EP3SE260 using Impulse-C. The presented techniques reduced area overhead by as much as 3x and improved assertion performance by as much as 100% compared to unoptimized in-circuit assertions.

1. Introduction

Field-programmable gate arrays (FPGAs) show significant power and performance advantages as compared to microprocessors [1], but have not gained widespread acceptance largely due to prohibitive application design complexity. High-level synthesis (HLS) significantly reduces application design complexity by enabling applications written in a high-level language (HLL) such as C to be executed on FPGAs. However, limited HLS support for verification, debugging, and timing analysis has contributed to limited usage of such tools.

For verification, designers using HLS can use assertion-based verification (ABV), a widely used technique in electronic design automation (EDA) tools [2], to verify runtime behavior by executing an application that contains assertions against a testbench. However, assertion-based verification of programs written in C using HLS tools, such as Impulse-C [3] and Carte [4], is often limited to software simulation of the FPGA's portion of the code, which can be problematic due to common inconsistencies between simulated behavior and actual circuit behavior. Such inconsistencies most commonly result from timing differences between the software thread-based simulation of the circuit and the actual FPGA execution [5]. In some cases, these inconsistencies may cause an application that behaves normally in software simulation to never complete (i.e., hang) when executing on the FPGA. Debugging an HLS-generated circuit to identify the cause of such hangs is a significant challenge that currently requires excessive designer effort.

Timing analysis, a procedure which determines if performance constraints are met, is an additional limitation of many HLS tools. Although timing analysis is widely used in physical design tools, in many cases, HLS tools do not consider timing constraints. Even worse, designers are unaware of the performance of different regions of an HLS-generated

circuit, which makes optimization more difficult. Although timing measurements can be taken during high-level simulation, such measurements are based on software simulation and do not reflect actual circuit performance [6].

One potential solution to these verification, debugging, and timing-analysis problems is for designers using HLS to use postsynthesis register-transfer-level (RTL) simulation. However, such an approach requires a designer to manually add assertions to HLS-generated hardware-description-language (HDL) code, which is a cumbersome process (as compared to adding assertions at the source level) and there are numerous situations where such simulations may be infeasible or undesirable. For example, a designer may use HLS to create a custom core that is part of a larger multiprocessor system that may be too complex to model with cycle accuracy. Even if such modeling was realized, slow simulation speeds can make such verification prohibitive to many designers.

Ideally, designers could overcome these limitations by specifying assertions in high-level code, which the HLS tool could integrate into generated circuits to verify behavior and timing, while also assisting with debugging. To achieve this goal, we present HLS techniques to efficiently support in-circuit assertions. These techniques enable a designer to use assertions at the source level while checking the behavior and timing of the application. Furthermore, we leverage such assertions to enable a debugging technique referred to as hang detection that reports the specific high-level regions of code where a hang occurs. To realize these in-circuit assertion-based techniques, this paper addresses several key challenges: scalability, transparency, and portability. Scalability (large numbers of assertions) and transparency (low overhead) are interrelated challenges that are necessary to enable thorough in-circuit assertions while minimizing effects on program behavior. We address these challenges by introducing optimizations to minimize performance and area overhead, which could potentially be integrated into any HLS tool. Portability of in-circuit assertion synthesis, for verification or timing analysis, is critical because HLS tools can target numerous platforms and must, therefore, avoid platform-specific implementations. The presented techniques achieve portability by communicating all assertion failures over the HLS-provided communication channels. Using a semiautomated framework that implements the presented HLS techniques, we show that in-circuit assertions can be used to rapidly identify bugs and violations of timing constraints that do not occur during software simulation, while only introducing a small overhead (e.g., reduction in frequency on the order of less than 3% and increase in FPGA resource utilization of 0.7% or less have been observed with several application case studies on an Altera Stratix-II EP2S180 and Stratix-III EP3SE260). Various case studies with optimized assertions have shown a 3x reduction in resource usage and improved assertion performance by as much as 100% compared to unoptimized assertion synthesis. Such work has the potential to improve designer productivity and to enable the use of FPGAs by nonexperts who may otherwise lack the skills required to verify and optimize HLS-generated circuits.

This paper is presented as follows. Section 2 discusses related work. Assertion-synthesis techniques and optimizations are explained in Section 3. Section 4 discusses timing analysis. Hang detection is described in Section 5. Section 6 describes the experimental setup and framework used to evaluate the presented techniques. Section 7 presents experimental results. Section 8 provides conclusions.

2. Related Research

Many languages and libraries enable assertions in HDLs during simulation, such as VHDL assertion statements, SystemVerilog Assertions (SVA) [7], the Open Verification Library (OVL) [8], and the Property Specification Language (PSL) [9]. Previous work has also introduced in-circuit assertions via hardware assertion checkers for each assertion in a design. Tools targeted at ASIC design provide assertion checkers using SVA [10], PSL [11], and OVL [12]. Academic tools such as Camera's debugging environment [13] and commercial tools such as Temento's DiaLite also provide assertion checkers for HDL. Kakoee et al. show that in-circuit assertions [12] can also improve reliability, with a higher fault coverage than Triple Modular Redundancy (TMR) for a FIR filter and a Discrete Cosine Transform (DCT).

Logic analyzers such as Xilinx's ChipScope [14] and Altera's SignalTap [15] can also be used for in-circuit debugging. These tools can capture the values of HDL signals and extract the data using a JTAG cable. However, the results presented by these tools are not at the source level of HLS tools. A source-level debugger has been built for the Sea Cucumber synthesizing compiler [16] that enables breakpoints and monitoring of variables in FPGAs. Our work is complementary by enabling HLL assertions and can be potentially be used with any HLS tool.

Checking timing constraints of HDL applications can be performed with many of the methods mentioned above. SVA, PSL, and OVL assertions can be used to check the timing relationship between expected values of signals in an HDL application [17]. A timed C-like language, TC (timed C), has been developed for checking OVL assertions inserted as C comments for use during modeling and simulation [18]. In-circuit logic analyzers such as ChipScope [14] and SignalTap [15] can also be used to trace application signals and check timing constraints for signal values. The HLS tool, Carte, provides timing macros [6] which return the value of a 64-bit counter that is set to zero upon FPGA reset. However, most HLS tools (including Impulse C) do not provide this functionality. In-circuit implementation of high-level assertions is a more general approach that potentially supports any HLS tool and enables designers to use ANSI-C assertions.

After a comprehensive literature search, we found no previous work related to hang detection of HLS applications (except for the initial work [19] being extended by this paper). Hang detection for microprocessors has been implemented on FPGAs [20]. Nakka et al. [21] separate hang detection for microprocessors into three categories. First, Instruction-Count Heartbeat (ICH) detects a hung

process not executing any instructions. Second, Infinite-Loop Hang Detector (ILHD) detects a process which never exits a loop. Finally, Sequential-Code Hang Detector (SCHD) detects a process that never exits a loop because the target address for the completion of a loop is corrupted. Although similar detection categories could be used for hardware processes generated by HLS tools, the methods needed for hang detection are different; hardware processes typically use state machines for control flow rather than using instructions. The related work found for microprocessor hang detection is typically used to increase reliability of the system by terminating the hung process rather than to help an application developer find the problematic line of code.

Although HDL assertions could be integrated into HLS-generated HDL, such an approach has several disadvantages. Any changes to the HLL source or a different version of the HLS tool could cause changes to the generated HDL (e.g., reorganization of code or renaming of signals), which requires the developer to manually reinsert the assertions into the new HDL. It is also possible that the developer may not be able to program in HDL or the HLS tool may encrypt or obfuscate generated HDL (e.g., Labview-FPGA). HLL assertions for HLS avoid these problems by adding assertions at the source level. Specifically, ANSI-C [22] assertions were chosen to be synthesized to hardware, since they are a standard assertion widely used by software programmers. Synthesizing ANSI-C assertions would allow existing assertions already written for software programs to be checked while running in circuit.

HLS optimizations for assertions were originally introduced in [19]. In this paper, we extend that work with techniques for timing analysis and hang detection.

3. Assertion Synthesis and Optimizations

ANSI-C assertions, when combined with a testbench, can be used as a verification methodology to define and test the behavior of an application. Each individual assertion is used to check a specific run-time Boolean expression that should evaluate to true for a properly functioning application. If the expression evaluates to false, the assertion prints failure information to the standard error stream including the file name, line number, function name, and expression that failed; after this information is displayed, the program aborts.

The presented HLS optimizations for in-circuit assertions assume a system architecture consisting of at least one microprocessor and FPGA and an application modeled as a task graph. These assumptions are common to existing HLS approaches [3]; therefore, the discussed techniques are potentially widely applicable with minor changes for different languages or tools.

In-circuit assertions are integrated into the application by generating a single *assertion checker* for each assertion and an *assertion notification function*, as shown in the top right hand side of Figure 1. The assertion checker implements the corresponding Boolean assertion condition by fetching all data, computing all intermediate values, and signaling the assertion notification function upon failure. The assertion notification function is responsible for printing information regarding all assertion failures and halting the application.

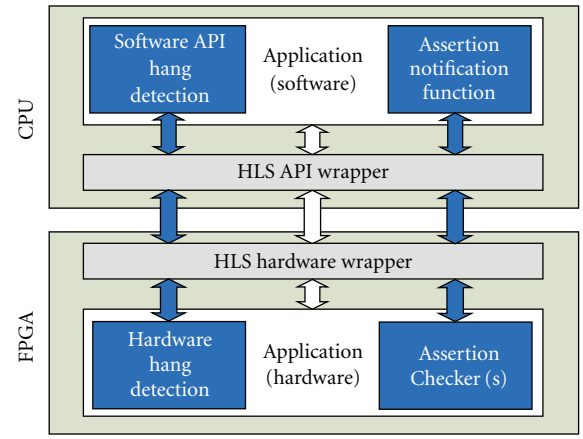


Figure 1: Assertion framework.

The assertion notification function can run simultaneously with the application as a task waiting for failure messages from the assertion checkers. The task is defined essentially as a large *switch* statement per communication channel that implements one case for each hardware-mapped assertion. Although a hardware/software partitioning algorithm could potentially map the assertion notification function task to either hardware or software, typically, the assertion notification function will be implemented in software due to the need to communicate with standard error. Although the added HLS communication channels in the task graph could greatly increase the I/O requirements for hardware/software communication, such a situation is avoided by time multiplexing all communication over a single physical I/O channel (e.g., PCIe bus, single pin). Performance overhead due to this time multiplexing should be minimal or even nonexistent (depending on the HLS tool) since ANSI-C assertions only send messages upon failure and halt the program after the first failed assertion.

One potential method to synthesize assertion checkers into circuits is described as follows. Semantically, an assert is similar to an *if* statement. Thus, assertions could be synthesized by converting each assertion into an *if* statement, where the condition for the *if* statement is the complemented assertion condition, and the body of the *if* statement transfers all failure information to the assertion notification function. Although such a straightforward conversion of *assert* statements may be appropriate for some applications, in general, this conversion will result in significant area and performance overhead. To deal with this overhead, we present three categories of optimizations that improve the scalability and transparency of in-circuit assertions, which are described in the following sections.

3.1. Assertion Parallelization. To maximize transparency of in-circuit assertions, the circuit for the assertion checker should have a minimal effect on the performance of the

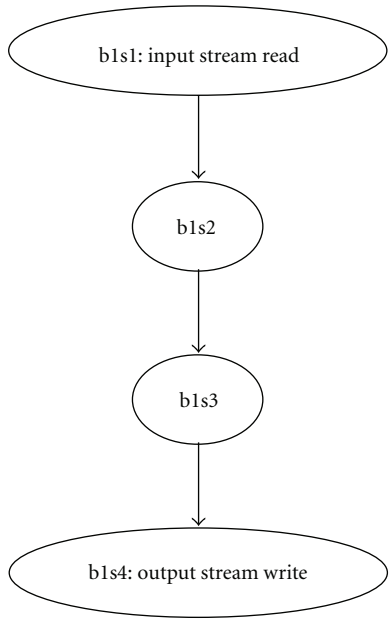


Figure 2: Application's state machine without assertion.

original application. However, by synthesizing assertions via direct conversion to *if* statements, the synthesis tool modifies the application's control-flow graph and resulting state machine, which adds an arbitrarily long delay depending on the complexity of the assertion statement. For Impulse-C, the delay of the assertion $assert((j <= 0 \mid\mid a[0] == i)\&\&(b[0] == 2 \mid\mid i > 0))$ can be shown by comparing the corresponding subset of the application's state machine before (Figure 2) and after (Figure 3) the assertion is added. For this example, the assertion can add up to seven cycles of delay to the original application for each execution of the assertion. While seven cycles may be acceptable for some applications, if this assertion occurred in a performance-critical loop, the assertion could potentially reduce the loop's rate (i.e., the reciprocal of throughput) to 12.5% of its original single-cycle performance, which could significantly affect how application components interact with each other.

HLS tools can minimize the effect of assertions on the application's control-flow graph by executing the assertions in parallel with the original application. To perform this optimization, HLS can convert each assertion statement into a separate task (e.g., a process in Impulse-C) that enables the original application task to continue execution while the assertion is evaluated. Instead of waiting for the assertion, the application simply transfers data needed by the assertion task and then proceeds.

For the previous assertion example, the optimization reduced the overhead from seven cycles to a single cycle as shown in Figure 4. The optimization was unable to completely eliminate overhead due to resource contention for shared block RAMs. Such overhead is incurred when the assertion task and the application task simultaneously require access to a shared resource.

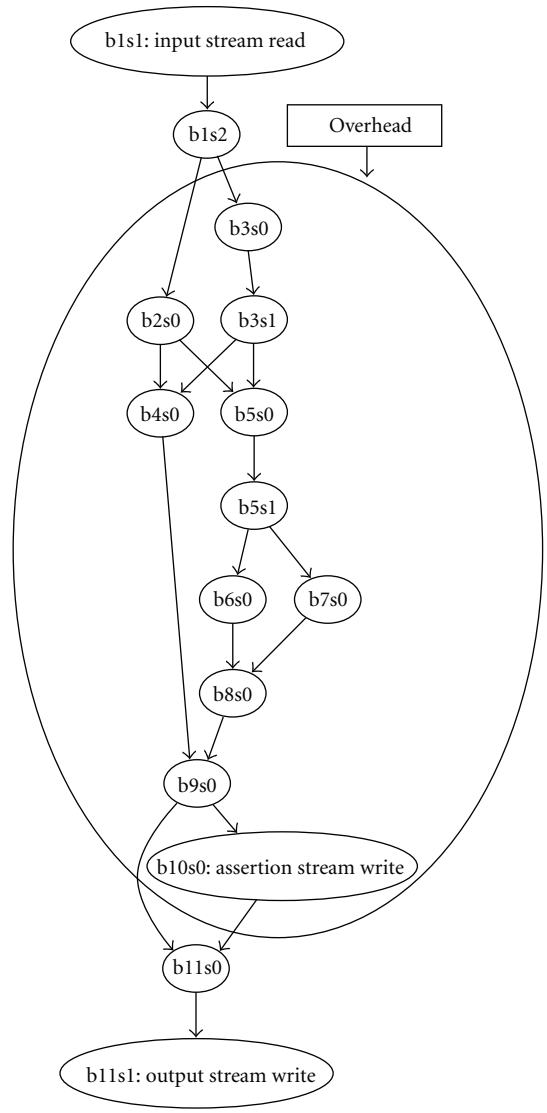


Figure 3: Application's state machine with unoptimized, serial assertion.

3.2. Resource Replication. As mentioned in the previous section, resource contention between assertions and the application can lead to performance overhead even when assertions are executed in parallel. To minimize this overhead, HLS can perform resource replication by duplicating shared resources.

For example, arrays in C can be synthesized into block RAMs. A common source of overhead is due to the limited number of ports on block RAMs that are simultaneously used by both the application tasks and assertion tasks. When accessing different locations of the block RAM, the circuit must time-multiplex the data to appropriate tasks, which causes performance overhead. HLS can effectively increase the number of ports by replicating the shared block RAMs, such that all replicated instances are updated simultaneously by a single task. This optimization ensures that all replicated instances contain the same data, while enabling an arbitrary

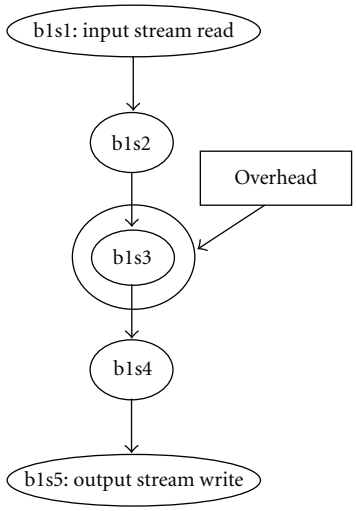


Figure 4: State machine with parallel assertion.

number of tasks to access data from the shared resource without delay.

Resource replication provides the ability to reduce performance overhead at the cost of increased area overhead. Such tradeoffs are common to HLS optimizations and are typically enabled by user-specified optimization strategies (i.e., optimize for performance as opposed to area). One potential limitation of resource replication is that for a large number of replicated resources, the increased area overhead could eventually reduce the clock speed, which may outweigh the reduced cycle delays. However, for the case study in Section 7.2.3, resource replication improved performance by 33% allowing the application's pipeline rate to remain the same.

3.3. Resource Sharing. Whereas the previous two optimizations dealt with performance overhead, in-circuit assertions can also have a large area overhead. Although an assertion checker circuit will generally cause some overhead due to the need to evaluate the assertion condition, HLS can minimize the overhead by sharing resources between assertions. For example, if a particular task has ten assertions with a multiplication in the condition, resource sharing could potentially share a single multiplier among all the assertions.

Although resource sharing is a common HLS optimization [23] for individual tasks, sharing resources across assertions adds several challenges due to the requirement that all statements sharing resources must be guaranteed to not require the resources at the same time. For task-graph-based applications, assertions may occur in different tasks at different times, which prevents a HLS tool from statically detecting mutually exclusive execution of all assertions.

Due to this limitation, HLS can potentially apply existing resource-sharing techniques to assertions within nonpipelined regions of individual tasks, because those assertions are guaranteed to not start at the same time. However, due to the assertion parallelization optimization, different starting times for two assertions do not guarantee that their execution does not overlap. For example, an assertion with a complex condition may not complete execution before a later assertion requires a shared resource. To deal with this situation, HLS can implement all assertions that share resources as a pipeline that can start a new assertion every cycle. Although this pipeline will add latency to all assertions in the same task that require access to the shared resources, such latency does not affect the application and only delays the notification of program failure. This technique of pipeline assertion checking is evaluated in Section 7.2.1.

Resource sharing could potentially be extended to support an arbitrary number of simultaneous assertions in multiple tasks by synthesizing a pipelined assertion checker circuit that implements a group of simultaneous assertions. To prevent simultaneous access to shared resources, the circuit could buffer data from different assertions using FIFOs (e.g., one buffer per assertion) and then process the data from the FIFOs in a round-robin manner. This extension requires additional consideration of appropriate buffer sizes to avoid having to stall the application tasks and an appropriate partitioning of assertions into assertion checker circuits, which we leave as future work.

In some cases, resource sharing may improve performance in addition to reducing area overhead by enabling placement and routing to achieve a faster clock due to fewer resources. However, resource sharing will at some point experience diminishing returns and may eventually increase clock frequency due to a large increase in multiplexers and other steering logic.

4. In-Circuit Timing-Analysis Assertions

For applications with real-time requirements, particularly in embedded systems, verification must guarantee that all timing constraints are met (a process referred to as *timing analysis*) in addition to checking the correctness of application behavior. If an HLS-generated application does not meet timing constraints during execution, then it would be helpful to know the location of the section of code that is violating constraints in order to focus optimization effort. However, determining the performance of an HLS-generated application can be difficult. HLS tools, such as Impulse-C and Carte, provide some compile-time feedback about the rate and latency of a pipelined loop, but it is largely unknown how many cycles a particular line of code will require. While it is possible to determine the number of cycles a line (or lines) of code will take by examining the HDL generated by the tool, delay can be data dependent, as shown in the possible traversals of the state machine generated by the evaluation of the conditional statement $if((j <= 0 \;||\; a[0] == i)\&\&(b[0] == 2 \;||\; i > 0))$ in Figure 3). However, such a process requires significant designer effort and requires the designer to have knowledge of the HLS-generated code. While a delay range for the computation in each line of code could be provided by the HLS tool via static analysis, the delay of communication calls cannot be determined by static analysis. Software

simulation cannot provide accurate timing due to timing differences between thread execution on the microprocessor and execution on the FPGA. In this section, we describe the additional concepts and methods needed to extend in-circuit assertions to perform timing analysis for applications built with HLS tools.

Figure 5 illustrates usage of timing-analysis assertions for an audio filtering application designed with a HLS tool. In this example, the application designer has determined that the filter takes too long to execute on the FPGA by measuring the time to run the application on the FPGA. However, the application designer is unsure of which part of the application in the FPGA is not meeting timing constraints. Using timing-analysis assertions, the application designer can check the timing of different application regions in the FPGA, as shown in the figure in addition to the case study in Section 7.5. Data-dependent delays can be checked to see if they are within bounds for each loop iteration. Although not shown in the figure, the same method can be used to check streaming communication calls for delays caused by buffers becoming full or empty.

In order to enable ANSI-C assertions to check the timing of an application, time must be accessible via a variable. In C, time is typically determined via a function call. In Figure 5, the ANSI-C function, *clock*, is used to return the current time in cycles. To measure the time of a section of code, the *clock* function should be called before and after that section of code, with the difference between the two times providing the execution time (in cycles). To perform timing analysis, an assertion can be used to check a comparison between the expected time and the measured time. For example, in Figure 5, the code in the loop for each filter is expected to take less than 100 cycles.

For timing-analysis assertions, time can potentially be represented in many different formats. However, returning time in terms of cycles will require the least amount of overhead. The ANSI-C library provides the *clock* timing function that returns the number of clock ticks that have elapsed since the program started. However, for C programmers who may want to express time in terms of seconds rather than cycles, the ANSI-C constant expression *CLOCKS_PER_SEC* can be used to convert clock ticks to time in seconds. The clock frequency of the FPGA could be determined by comparison with timestamps sent from the CPU. However, an assertion may need to be checked on the first cycle after an FPGA restart. Since determining the frequency of the FPGA automatically could take too long, a preprocessor constant *FPGA_FREQ* is used to define the FPGA frequency in Hz.

The type defined for representing clock ticks in ANSI-C is *clock_t* that typically corresponds to a long integer. For added flexibility when used in hardware, time can be returned and stored as a 32-bit or 64-bit value. A 64-bit value is used by default. To select a 32-bit value, the preprocessor constant *CLOCK_T_32* must be defined in the code. A 32-bit value can be used to reduce overhead but will overflow after 43 seconds for a clock speed of 100 MHz. During software simulation, the assertions using timing information are ignored, which allows simulation to check correctness of the application while ignoring the timing of the microprocessor.

To enable synthesis of timing assertions, a counter, which is set to zero upon reset, is added in each hardware process that contains a *clock* statement. The value returned by the *clock* statement is generated by latching the counter signal for each transition of the state machine. Use of a latched counter signal ensures that the timer value is consistently taken at the beginning of each state transition for states that execute more than one cycle.

One potential problem with this approach is that HLS tools often reorder statements to maximize parallelism. Therefore, *clock* statements could potentially be reordered leading to incorrect timing results. However, such a problem is easily addressed by making a synthesis tool aware of *clock* statements. In this paper, we alternatively evaluated the techniques using instrumentation due to the inability to modify commercial HLS tools. Although instrumentation could experience reordering problems, for the evaluated examples, reordering of *clock* statements did not occur.

5. Hang-Detection Assertions

A common problem with FPGA applications is a failure to finish execution, which is often referred to as *hanging*. Common causes of hanging include infinite loops, synchronization deadlock, blocking communication calls that wait indefinitely to send or receive data, and so forth. Determining the cause of a hanging application, referred to as *hang detection*, is difficult for HLS-generated FPGA designs. While a debugger could be used to trace down the problem during software simulation, the inaccuracies of software simulation can miss hangs that occur during FPGA execution. To deal with this problem, we extend in-circuit assertions to enable hang detection for HLS-generated application.

One challenge of hang detection using assertions is that it is assumed that the assertion will eventually be checked. If the application waits indefinitely for a line of code to finish (e.g., an infinitely blocking communication call) then a different detection method is needed, since the assertion after the hung line will never be executed as shown in Figure 6(a). Without some mechanism to alert the developer to the current state of the program, it will be difficult to pinpoint the problem. For example, in the filter application (see Figure 7), the source of the problem that is causing the application to hang could be in any of the software or hardware processes.

One potential solution is to use assertions in a counter-intuitive way by adding assertions periodically throughout the code that are designed to fail (i.e., *assert*(0)). By also defining the *NABORT* flag, failed assertions will not cause the application to abort, which allows the developer to manually create an application heartbeat (i.e., a signal sent as a notification that the process is alive) that traces the execution of the application on the FPGA as shown in Figure 6(b). In the filter application example, multiple assertions would need to be placed in strategic locations in each FPGA process to determine the events that take place before the application hangs. The resolution (in terms of lines of code) would be determined by how many assertions are used. Unfortunately,

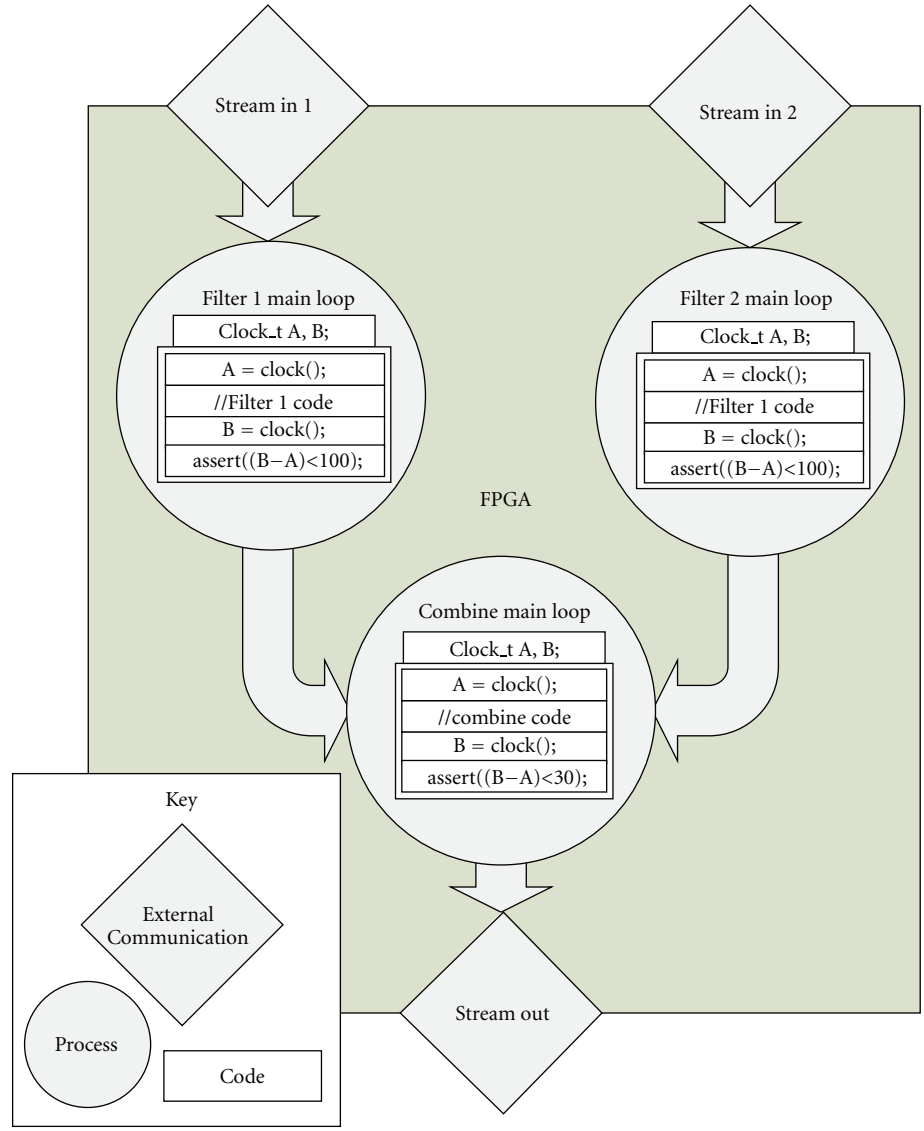
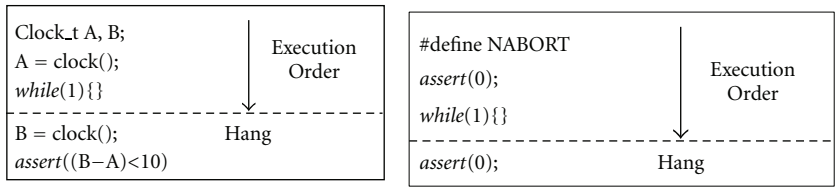


Figure 5: Using timing-analysis assertions with a filter application.



(a) Timing Assertion (b) Assertions set to fail

Figure 6: Manually using ANSI-C assertions for hang detection.

if a large number of assertions are used, then large amounts of communication and FPGA resources could be used by the assertions. Although this approach works, it requires significant designer effort and has large overhead.

To reduce effort and overhead, we present a more automated method of hang detection that does not require user instrumentation and instead uses watchdog timers to monitor the time between changes of the signals that represent the state of the hardware process. The monitoring circuit has software-accessible registers that contain the current state of all hardware process and the state of any hardware process that it has detected as hung. Hang detection is triggered using a watchdog timer for a hardware process that signals when a state takes longer than a user-defined number of cycles;

the assertion pragma, *#pragma assert_FPGA_watch_dog*, sets this timeout period, which is reset anytime a state transition occurs. The watchdog timer is sized to be just large enough to hold the cycle count given in the pragma to reduce FPGA resource and frequency overhead. In software, a separate thread is spawned to monitor the hardware hang detector to check for hung states (i.e., expired watchdog timers). If a hardware process has hung, then the state in the registers is matched to the corresponding line of code via a lookup table generated by parsing an intermediate translation file (both Impulse C and Carte create these files). The state of all other hardware processes are given for reference.

In software, many HLS applications will wait indefinitely at some point in its execution for the FPGA to respond with some form of communication or synchronization. For those applications, hangs caused in the FPGA hardware will also cause the software to hang on the communication or synchronization API call for the FPGA. Although traditional debugging tools can be used to detect these hangs in software, software hang detection is provided to monitor the HLS API calls for convenience. A thread is spawned for all API calls of the HLS tool. The thread will check if the API call finishes within a time period set by the assertion pragma, *#pragma assert_API_watch_dog*. If the API call takes longer than the timeout period, then the current line of code for the API call and all hardware processes will be printed to standard output and the program will abort.

This automated approach simplifies the addition of hang detection to an application, as shown for the filter application in Figure 7 and case study in Section 7.6, compared to manually adding *assert*(0) statements. Two assertion pragmas are added to the application before instrumentation to set the watchdog timeout periods in hardware and software. Although hangs can be caused by the interaction between two or more (hardware or software) processes, providing the state of the hung process along with the current state of all other hardware processes can greatly narrow down the source of problem.

Several improvements can be added to further enhance hang detection of HLS applications. The feedback given to the application developer can be increased by reporting more than the last state of each process in the FPGA. For example, a trace buffer could be added of a user-defined size that would capture the sequence of state that occurred before the hardware process hung. Also, infinite loops in a hardware process will only trigger software API hang detection. Since infinite loops will not stay in a single state to trigger the hang-detection method mentioned above, detection of infinite loops in hardware could also be incorporated by adding a second counter for each process that is dedicated to counting the number of cycles spent in states that are known to be inside one or more loops. The overhead of hang detection could be reduced by allowing the user to select which processes to monitor. The hang detection counters could be removed for some or all processes, while still allowing the current state of the process to be periodically retrieved or retrieved by software API hang detection. This approach would give the user the option to customize hang detection to fit for designs that nearly fill the FPGA.

6. Assertion Framework

To evaluate the assertion-synthesis techniques, we created a prototype tool framework for Impulse-C that implements the techniques via instrumentation of HLL and HDL code. It should be noted that we use instrumentation because we are unable to modify the proprietary Impulse-C tool. All of the techniques are fully automatable and ideally would be directly integrated into an HLS tool.

6.1. Unoptimized Assertion Framework. To implement basic in-circuit assertion functionality, the framework uses HLL instrumentation to convert *assert* statements into HLS-compliant code in three main stages. First, the C code for the FPGA is parsed to find functions containing assertion statements, converting any assertion statements to an equivalent *if* statement. A false evaluation produces a message that will be retrieved from the FPGA by the CPU, uniquely identifying the assertion. Next, communication channels are generated to transfer these messages from the FPGA to the CPU. Finally, the assertion notification function is defined as a software function executing on the CPU to receive, decode, and display failed assertions using the ANSI-C output format. An example of this automated code instrumentation is shown in Figure 8.

To notify the user of an assertion failure, the framework uses an error code that uniquely identifies the failed assertion based on the line number and file name of the assertion. Once the assertion notification function decodes the assertion identifier, the user is notified by printing to the standard error stream by the CPU for the current framework. The framework could be extended to work without a CPU by having the assertion identifier stored to memory, displayed on an LCD, or even flashed as a sequence on an LED by the FPGA. Alternatively, an FPGA could potentially use a softcore processor.

Note that other changes are needed to route the stream to the CPU, such as API calls to create and maintain the stream. The stream must also be added as a parameter to the function. The output of the framework is valid Impulse-C code, allowing further modifications to the source code with no other changes to the Impulse-C tool flow. Once verification of the application is finished, the constant NDEBUG can be used to disable all assertions and reduce the FPGA resource overhead for the final application. An additional nonstandard constant NABORT can be used to allow the application to continue instead of aborting due to an assertion failure.

6.2. Assertion Framework Optimizations. In order to evaluate the optimizations presented in Section 3, a hybrid mix of manual HLL and HDL instrumentation was used. To enable assertion parallelization (Section 3.1), the framework modifies the HLL code to move assertions into a separate Impulse-C process. The framework introduces temporary variables to extract data needed by the assertion. HDL instrumentation then connects the temporary variables and trigger conditions between processes. The results of this optimization can be found in Section 7.2.

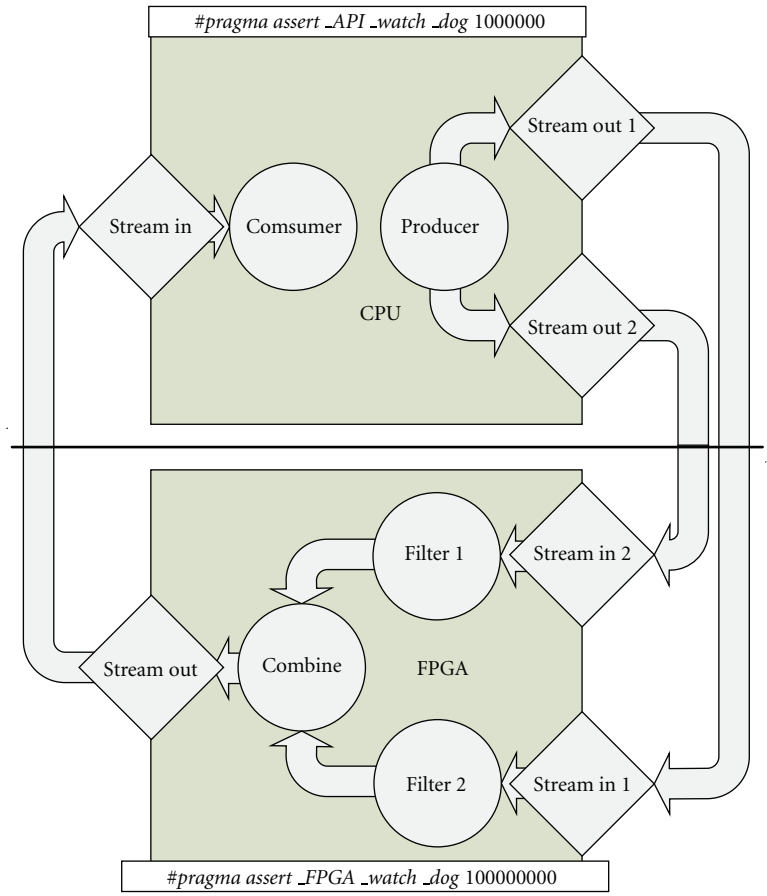


Figure 7: Using hang-detection assertions with a filter application.

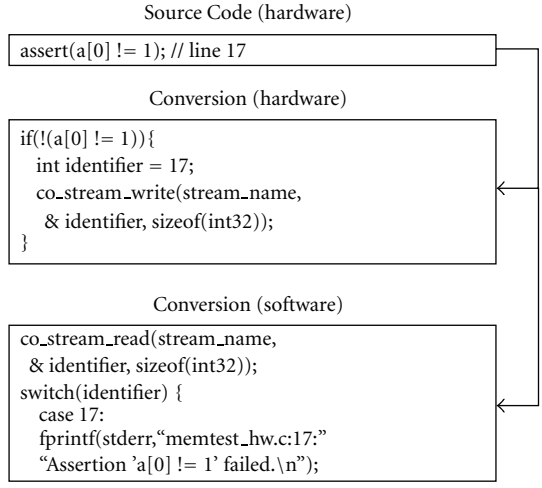


Figure 8: HLL assertion instrumentation.

Resource replication, described in Section 3.2, was performed using manual HLL instrumentation. An extra array was added to the source code that performed the same writes as the original array but reads were only performed by the assertion, as shown in Section 7.3.

The following manual hybrid instrumentation was used to evaluate resource sharing as described in Section 3.3. Although resource sharing could potentially be applied to any shared resource, we evaluate the optimization for shared communication channels, which are common to all

Impulse-C applications. HLL instrumentation creates a streaming communication channel per Impulse-C process and sends the identifier of the assertion upon assertion failure. Creating a streaming communication channel per Impulse-C process can become expensive in terms of resources if a large number of Impulse-C processes contain assertions. To reduce the number of streams created for each process, a single bit of the stream is used per assertion to indicate if an assertion has failed. This technique allows Impulse-C processes to more efficiently utilize the streaming communication channels. When streaming communication resources are shared, a separate process is created via HLL instrumentation that can handle failure signals from up to 32 assertions per process if a 32-bit communication channel is used. For example, if all 32 assertions fail simultaneously, then all 32 bits of the communication channel will simultaneously be asserted. The failure signals are connected to assertions using HDL instrumentation for efficiency. The overhead reduction associated with using this technique is explored in the case study that is presented in Section 7.4.

6.3. Timing-Analysis and Hang-Detection Extensions. Semiautomatic hybrid instrumentation was used to support timing functions presented in Section 4. Impulse-C does not support ANSI-C library calls so the *clock* function calls must be removed. A placeholder variable is declared and used in place of the *clock* statement in the source code. After hardware generation, a Perl script is used to instrument the HDL. A counter is added in each hardware process that contains a *clock* statement, which is set to zero upon reset. A second signal is added to the process that latches the counter signal upon transition of the state machine. The placeholder variable, synthesized into a signal with a similar name in HDL, is replaced with the latched counter signal.

Semiautomatic hybrid instrumentation was used for hang detection in Section 5. For software hang detection, a wrapper was added around each of the Impulse-C library API calls which added the threaded hang detection. The modified software API calls required extra parameters for access to the hardware hang-detection registers. Automatic parsing of the *xhw* file generated by Impulse-C allows states to be converted to line numbers. For hardware hang detection, a hardware process supporting register transfer to software is automatically added to the source code. After Impulse-C generates the HDL, the state machine signals of all other hardware processes are automatically routed into the hang-detection process. The hang-detection circuit is then manually added by overwriting part of the register transfer process.

Although many of the steps for adding timing-analysis and hang-detection instrumentation were manual, all of the steps could be automated via Perl scripts. Ideally, modification to the Impulse-C tool would be made instead of instrumenting source and intermediate code. However, because Impulse-C is proprietary, such modification was not possible for this work.

6.4. HLS Tool and Platform. The framework currently uses Impulse-C. Impulse-C is a high-level synthesis tool to convert a program written in a subset of ANSI-C to hardware in an FPGA. Impulse-C is primarily designed for streaming applications based upon the communicating sequential process model but also supports shared memory communication [5]. Speedups can be achieved in Impulse-C applications by running multiple sequential process in parallel, pipelining loops, and adding custom HDL-coded functions calls.

Quartus 9 was used for synthesis, and implementation of the Impulse-C-generated circuits. The target platforms are the XtremeData XD1000 [24] containing a dual-processor motherboard with an Altera Stratix-II EP2S180 FPGA in one of the Opteron sockets and the Novo-G supercomputer [25] at University of Florida containing 48 GiDEL PROCStar III [26] cards each with four Stratix-III EP3SE260. Impulse C 3.3 is used for the XD1000 while Impulse-C 3.6 with an in-house platform support package is used for Novo-G. Although the XD1000 and Novo-G are high-performance computing platforms, Impulse-C also supports embedded PowerPC and MicroBlaze processors [5]. Furthermore, Novo-G and the XD1000 are representative of FPGA-based embedded systems that combine CPUs with one or more FPGAs. The presented overhead results would likely be similar for other embedded platforms, assuming similar Impulse-C wrapper implementations.

Although we currently evaluate HLS assertions using Impulse-C, the techniques are easily extended to support other languages. For example, in Carte, Impulse-C's streaming transfers would be replaced with DMA transfers. The software-based assertion notification function (see Figure 1) would then need to monitor Carte's FPGA function calls for failed assertions as opposed to monitoring Impulse-C's FPGA processes.

7. Experimental Results

This section presents experimental results that evaluate the utility and overhead of the presented assertion synthesis, timing analysis and hang detection. Section 7.1 motivates the need for in-circuit assertions by illustrating a case study where assertions pass during simulation but fail during FPGA execution. Section 7.2 illustrates the performance and overhead improvements of the assertion parallelization optimization. Section 7.3 evaluates performance benefits of resource replication. Section 7.4 evaluates the scalability of assertions in terms of resource and frequency overhead by applying resource sharing optimizations to the communication channels. Section 7.5 presents the overhead of using assertions for timing analysis. Section 7.6 evaluates two hang-detection methods used on an application that fails to complete.

The designs used in the case studies occupy a relatively small part of the FPGA (24% of logic used in Section 7.5). Designs with higher resource utilization may lead to greater performance degradation and resource overhead of assertions due to increased difficulty in placement and, routing for example. In addition, resource replication might not be applicable for designs that are almost full.

```
1  co_unit64 c2, c1;
2  co_int32 address, array[20], out;
3  c2 = 4294967286; c1 = 4294967296;
4  if (c2 > c1) address = c2 – c1;
5  else address = 0;
6  assert(address >= 0);
7  out = user(address);
8  assert((30 > out) && (out > 20));
9  array[address] = out;
```

Algorithm 1: In-circuit verification example.

7.1. Detecting Simulation Inconsistencies. In this section, we illustrate how assertions can be used for in-circuit verification and debugging to catch inconsistencies between software simulation and FPGA execution of an application. The code in Algorithm 1 shows how assertion statements can be used for in-circuit verification by identifying bugs not found using software simulation. The first assertion is used to detect a translation mistake from source code to hardware (it is possible for a translation mistake to also have an effect on an *assertion*) . The assertion statement (line 6) never fails in simulation but fails when executed on the XD1000 platform. Upon inspection of the generated HDL, it is observed that Impulse-C performs an erroneous 5-bit comparison of c2 and c1 (line 4). The 64-bit comparison of 4294967286 > 4294967296 (which evaluates to false) becomes a 5-bit comparison of 22 > 0 (which evaluates to true), allowing the array address to become negative (line 4). In contrast, the simulator executing the source code on the CPU sets the address to zero (line 5). Impulse C will generate a correct comparison when c1 and c2 are 32-bit variables.

The second assertion (line 8) is used to check the output of an external HDL function (line 7), which is used to gain extra performance over HLS generated HDL. When an external HDL function is used, the developer must provide a C source equivalent for software simulation. However, the behavior and timing of the C source for simulation may differ from the behavior of the external HDL function during hardware execution, again demonstrating a need for in-circuit verification.

For demonstration purposes, this example case is intentionally simplistic and similar conclusions could be drawn using a cycle-accurate HDL simulator. However, in practice, inconsistencies caused by the timing of interaction between the CPU and FPGA would be very difficult to model in a cycle-accurate simulator.

7.2. Assertion Parallelization Optimization. This section provides results for the parallelization optimization of assertions. Section 7.2.1 shows improvements from optimization for Triple-DES encryption. Section 7.2.2 shows optimization improvements for edge-detection. While the applications in the previous sections evaluate frequency overhead, Section 7.2.3 evaluates state machine performance overhead (in terms of additional cycles) and optimization improvements.

Table 1: Triple-DES assertion overhead.

EP2S180	Original	Assert	Difference
Logic used	13677	13851	+174
(out of 143520)	(9.53%)	(9.65%)	(+0.12%)
Comb. ALUT	7929	8025	+96
(out of 143520)	(5.52%)	(5.59%)	(+0.07%)
Registers	10019	10055	+36
(out of 143520)	(6.98%)	(7.01%)	(+0.03%)
Block RAM	222912	223488	+576
(9383040 bits)	(2.37%)	(2.38%)	(+0.01%)
Block interconnect	24657	24878	+221
(out of 536440)	(4.60%)	(4.64%)	(+0.04%)
Frequency (MHz)	145.7	142.0	−3.7 (−2.54%)

7.2.1. DES Case Study. The first application case study shows the area and clock frequency overhead associated with adding performance optimized assertion statements to a Triple-DES [27] application provided by Impulse-C, which sends encrypted text files to the FPGA to be decoded. Two assertion statements were added in a performance critical region of the application to verify that the decrypted characters are within the normal bounds of an ASCII text file. Table 1 shows all sources of overhead, including the streaming communication channels generated by Impulse-C for sending failed assertions back to the CPU. The overhead numbers were found to be quite modest, with resource usage increasing by at most 0.12% of the device and the maximum clock frequency dropping by less than 4 MHz.

For this case study, the optimized assertions were checked in a separate pipeline process to reduce the overhead generated by the assertion comparison. Assertion failures are sent by another process to ensure that assertions can be checked each cycle. The state machine of the application remained unchanged because the optimized assertions were checked in a separate task working in parallel with the application. Since the application's state machine remained the same, the only performance overhead comes from the maximum clock frequency reduction. The resource overhead for optimized assertions actually decreased as compared to unoptimized assertions. The ALUT (Adaptive Look-Up Table) and routing resources needed by Quartus to achieve a maximum frequency of 144.7 MHz for unoptimized assertions was 0.06% greater than the ALUT and routing resources need for optimized assertions that achieved a maximum frequency of 142 MHz.

7.2.2. Edge-Detection Case Study. The following case study integrates performance optimized assertions into an edge-detection application. The edge-detection application, provided by Impulse-C, reads a 16-bit grayscale bitmap file on the microprocessor, processes it with pipelined 5×5 image kernels on the FPGA, and streams the image containing edge-detection information back. Since the FPGA is programmed to process an image of a specific size, two assertions were added to check that the image size (height and width) received by the FPGA matches the hardware configuration.

Table 2: Edge-detection assertion overhead.

EP2S180	Original	Assert	Difference
Logic used	12250	12273	+23
(out of 143520)	(8.54%)	(8.56%)	(+0.02%)
Comb. ALUT	6726	6809	+83
(out of 143520)	(4.69%)	(4.75%)	(+0.06%)
Registers	9371	9417	+46
(out of 143520)	(6.53%)	(6.56%)	(+0.03%)
Block RAM	141120	141696	+576
(9383040 bits)	(1.50%)	(1.51%)	(+0.01%)
Block interconnect	19904	19994	+90
(out of 536440)	(3.71%)	(3.73%)	(+0.02%)
Frequency (MHz)	77.5	79.3	+1.8 (+2.32%)

Table 3: Single-comparison assertion.

	Latency Overhead	
Assertion data structure	Unoptimized	Optimized
Scalar variable	1	0
Array (non-consecutive)	1	0
Array (consecutive)	2	1

Table 4: Pipelined single-comparison assertion.

	Overhead			
	Unoptimized		Optimized	
Assertion data structure	Latency	Rate	Latency	Rate
Scalar variable	1	1	0	0
Array	2	1	1	0

The assertions were added in a region of the application that was not performance critical. As shown in Table 2, the overhead numbers for this case study were also modest, with resource usage increasing by at most 0.06% on the EP2S180.

For the edge-detection case study, the optimized assertions were checked in a separate process to reduce the overhead generated by the assertion comparison. Since the applications state machine remained the same, and maximum clock frequency did not reduce, the application did not incur any performance overhead due to the addition of the assertions. The frequency increase is likely due to randomness in placement and routing results of similar designs. The performance optimization of the assertions increased ALUT resource utilization from 0.03% to 0.06% on the EP2S180.

7.2.3. State Machine Overhead Analysis. This section presents a generalized analysis of performance overhead caused by adding assertions with a single comparison and the performance improvement via optimizations. The results in this section present overhead in terms of cycles and exclude changes to clock frequency, which was discussed in the previous section. We evaluate single-comparison assertions to determine a lower bound on the optimization improvements. To measure the performance overhead of adding assertions, we examine the state machines and pipelines generated by Impulse-C. Impulse-C allows loops (e.g., *for* loops or *while* loops) to be pipelined. Assertions added to a pipeline can modify the pipeline's characteristics. Each pipeline generated by Impulse-C has a latency (time in cycles for one iteration of a loop to complete) and rate (time in cycles needed to finish the next loop iteration). Assertions that are not in a pipelined loop will add latency (i.e., one or more additional states) to the state machine that preserves the control flow of the application. As stated in Section 6.2, assertions can be optimized to reduce or eliminate the overhead of assertions in terms of additional clock cycles required to finish application execution. These optimizations move the comparisons to a separate Impulse-C process so that they can be checked in parallel with the application. Any remaining clock cycle overhead after optimization comes from the data movement needed for assertion checking.

Table 3 shows the latency overhead for nonpipelined, single comparison assertions. In most cases, assertions with these comparisons will increase latency by one cycle. With optimizations, this latency overhead is reduced to zero since extracting data in most cases will not add latency to the application. In the case where an array is consecutively accessed temporally by the application and an assertion, an unoptimized assertion will have a latency overhead of two cycles because of block RAM port limitations. With optimizations, this latency overhead is reduced to one cycle to extract data from the array or block RAM. For more complex assertions, the latency will increase for unoptimized assertions while the latency for optimized assertions will remain the same, as seen when comparing Figures 3 and 4. Even with the multiple array accesses in $assert((j <= 0 \,||\, a[0] == i)\&\&(b[0] == 2 \,||\, i > 0))$, only one cycle is needed to retrieve the array data.

Table 4 shows pipeline latency and rate overhead observed for a single comparison. Adding an unoptimized assertion using a scalar variable to a pipelined loop increased the latency from 2 to 3, resulting in an overhead of one cycle, and degraded the rate from 1 to 2 for the pipeline. Although the rate overhead was a single cycle, this corresponds to a 2x slowdown in performance because the throughput is reduced to half of the original loop. This overhead comes from adding a streaming communication call. For the optimized assertion, the streaming communication call was moved to a separate process that reduced the latency and rate overhead to zero, resulting in a 2x speedup compared to the unoptimized assertions. For assertions using arrays in pipelined loops, adding an assertion caused a 2-cycle latency overhead that increased the latency from 2 to 4. The assertion reduced the rate from 2 to 3, which is a one cycle rate overhead that corresponds to a 50% reduction in performance.

7.3. Resource Replication Optimization. As mentioned in Section 7.2.3, Table 4 shows pipeline latency and rate overhead observed for a single comparison. For assertions used

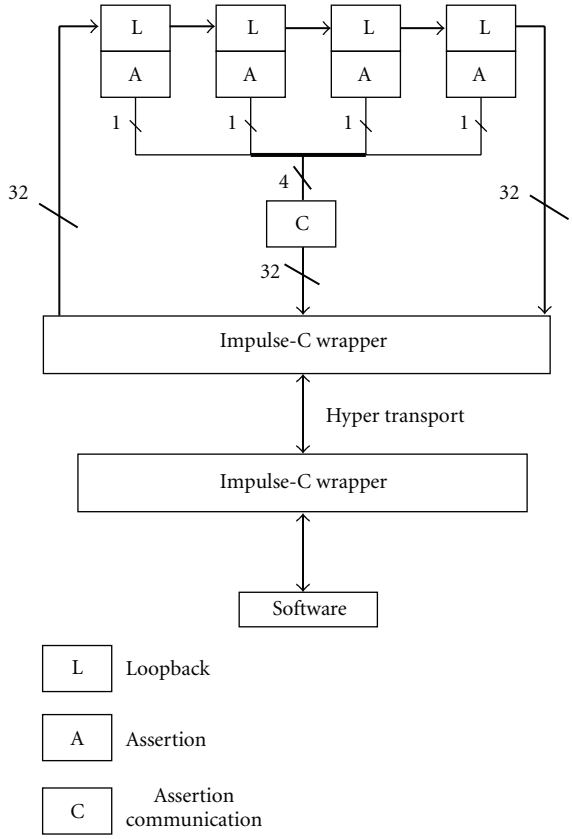


Figure 9: Simple streaming loopback.

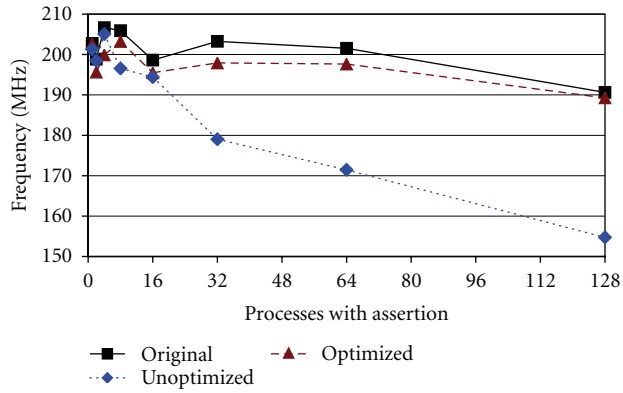


Figure 10: Assertion frequency scalability.

in pipelined loops checking an array data structure, the assertion overhead was reduced via resource replication by adding an additional array to the process dedicated read access to the assertion as described in Section 6.2. With a duplicate array, only the latency increased from 2 to 3 and the rate remained the same which corresponds to a 33% rate improvement over the nonoptimized version. A similar improvement could be gained for a nonpipelined assertion that checks multiple indexes to the same array.

7.4. Resource Sharing Optimization. This section demonstrates the improvement in scalability from resource sharing optimization techniques. We evaluate scalability by measuring the resource and clock frequency overhead incurred by adding assertions to a large number of Impulse-C processes, providing an extremely pessimistic scenario in terms of overhead. A single assertion is added per process which results in a separate streaming communication channel for each process. A single greater than comparison is made per process, generally requiring only minor changes to the process state machine. In this study, the application consists of a simple streaming loopback as shown in Figure 9. The loopback also stores the value and retrieves the value at each stage. Each process added to the application adds an extra stage in the loopback (e.g., for 4 FPGA processes shown as L in Figure 9, incoming data would be passed from the input to the FPGA, passing through each of the processes before being returned to the CPU). The assertion in each process ensures the number being passed is greater than zero. Each process adds overhead in terms of an assertion shown as A in Figure 9 and an extra Impulse-C streaming communication channel shown as C in Figure 9 to notify the CPU of failed assertions. For a 32-bit stream, up to 32 assertions can be connected to the streaming communication channel before a new streaming communication channel is needed.

Using the previously discussed straightforward conversion of *assert* statements to *if* statements, the unoptimized assertions with 128 processes (128 assertions) had a resource overhead on the EP2S180 of 4.07% ALUTs (the highest resource percentage overhead). However, the maximum frequency decreased from 190 MHz for the 128-process original application to 154 MHz or an 18.8% overhead as shown in Figure 10 for the 128-process application with unoptimized assertions.

By applying the resource sharing optimization only to the communication channels so that only a single bit of the stream is used per assertion as described in Section 6.2 (and not the assertion resources), the resource overhead was decreased. The resource overhead on the EP2S180, as shown in Figure 11, was reduced to 1.34% of ALUTs or over a 3x improvement for the 128-process application with assertions. Assertion optimizations increased the maximum frequency for the 128-process application to 189 MHz, as shown in Figure 10, which represents over an 18% improvement. The frequency of the application with assertion optimizations (189.3 MHz) was very close to the original application's frequency of 190.6 MHz. While the resource usage increased consistently for all three tests (original, unoptimized, and optimized) from 1 to 128 processes, the maximum frequencies reported by Quartus did not consistently decrease as the number processes increased until 32 processes were added. The frequency overhead decreased from 32 to 128 processes with optimized assertions because the application added one stream per process, while the assertions only added one stream per 32 processes since 32-bit streaming communication was used. This demonstrates the benefits of the resource sharing optimization for streaming communication channels.

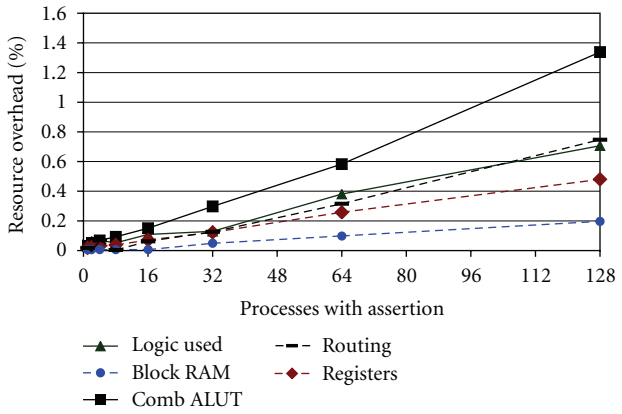


Figure 11: Optimized assertion resource scalability.

7.5. In-Circuit Timing Analysis. This section provides a case study showing the utility and overhead of adding assertions with timing statements to a backprojection application. Backprojection is a DSP algorithm for tomographic reconstruction of data via image transformation. For the backprojection application, instrumentation was added into a nested loop (see Algorithm 2). Two 32-bit timing calls were added around the inner pipelined loop to measure the time required for the pipelined loop to finish generating 512 pixels. After the timing calls, ten assertions were added to find the maximum time required for the pipelined loop to finish for all outer-loop iterations. Since the inner loop has 512 iterations, a minimum of 512 cycles should be needed to complete the loop; however, more cycles could be required for stalls and flushing of the pipeline. To test these assumptions, ten assertions were added to check the timing of the loop with exponentially increasing maximum times and *NABORT* was defined to stop the application from aborting. After execution, only the first assertion passed evaluation, which means that the maximum time for the inner loop is between 640 and 1023 cycles.

This technique allows the application designer to quickly check timing in multiple regions of the application with minimal disturbance to the application in terms of resource and communication overhead. After evaluating the feedback from the assertions, the application designer can modify the application to stream back the exact timing values for problematic regions of code. In addition, the assertion feedback provided before modifying the application can be used to make sure that the timing values streamed back are valid. It is possible that the addition of large data transfers could change the timing of the application.

The backprojection application runs on all four Stratix-III EP3SE260 FPGAs on the GiDEL PROCStar III [26] card. Overhead is only given for one FPGA since the image is split between all four FPGAs. Ideally, a single assertion could check an array of values in a loop for more compact code (see Algorithm 3). However, that approach increases overhead when synthesized with Impulse-C as shown in Table 6 as compared to using individual assertions as shown in Table 5.

```
for(y=0;y<512;y++)
{
   time1 =c1ock();
   for(x=0;x<512;x++)
   {//compute pixel
     . . .
   }
   time2=clock();
   assert((time2-time1)<1024));
   assert((time2-time1)<640));
   assert((time2-time1)<576));
   assert((time2-time1)<544));
   assert((time2-time1)<528));
   assert((time2-time1)<520));
   assert((time2-time1)<516));
   assert((time2-time1)<514));
   assert((time2-time1)<513));
   assert((time2-time1)<512));
   . . .
}
```

Algorithm 2: Adding timing assertions individually to backprojection.

```
int32 constraint[]={1024,640,576,544,528,520,516,514,
513,512};
. . .
for(y=0;y<512;y++)
{
   time1=clock();
   for(x=0;x<512;x++)
   {//compute pixel
     . . .
   }
   time2=clock();
   for (i=0; i<10; i++){
     assert(time2-time1< constraint[i]);
   }
   . . .
}
```

Algorithm 3: Adding timing assertions in a loop to backprojection.

For individual assertions, no additional block RAM was used since assertion failures were transferred via registers rather than using streaming communication on the PROCStar III. The logic overhead of 0.7% is the highest of all the application case studies but is reasonable given that timing calls and multiple assertions were used. The maximum FPGA frequency stayed about the same with an insignificant increase of 0.6 MHz. For a single assertion in a loop, the overhead increased in all categories except for routing. The additional overhead is likely caused by additional complexity of the state machine and the usage of block RAM. The lower routing overhead is probably due to only having to make connections to a single assertion.

Table 5: Individual backprojection timing assertion overhead.

EP3SE260	Original	Assert	Difference
Logic used	48285	49702	+1417
(out of 203520)	(23.72%)	(24.42%)	(+0.70%)
Comb. ALUT	32962	33132	+170
(out of 203520)	(16.20%)	(16.28%)	(+0.08%)
Registers	44098	44595	+497
(out of 203520)	(21.67%)	(21.91%)	(+0.24%)
Block RAM	7114752	7114752	0
(15040512 bits)	(47.30%)	(47.30%)	(0%)
Block interconnect	101317	102740	+1423
(out of 694728)	(14.58%)	(14.79%)	(+0.20%)
Frequency (MHz)	131.9	132.5	+0.6 (+0.45%)

Table 6: Looped backprojection timing assertion overhead.

EP3SE260	Original	Assert	Difference
Logic used	48285	50169	+1884
(out of 203520)	(23.72%)	(24.65%)	(+0.93%)
Comb. ALUT	32962	33459	+497
(out of 203520)	(16.20%)	(16.44%)	(+0.24%)
Registers	44098	44657	+559
(out of 203520)	(21.67%)	(21.94%)	(+0.27%)
Block RAM	7114752	7123968	9216
(15040512 bits)	(47.30%)	(47.37%)	(0.07%)
Block interconnect	101317	102621	+1304
(out of 694728)	(14.58%)	(14.77%)	(+0.19%)
Frequency (MHz)	131.9	131.3	−0.6 (−0.45%)

7.6. Hang Detection. This section shows how in-circuit assertions can be used to detect when an application fails to complete (i.e., hangs), even when software simulation runs to completion. In an effort to speed up a decoder and encoder version of the DES application described in Section 7.2.1, modifications were made that caused the application to complete in software simulation and yet hang on the XD1000. Since Impulse-C does not support *printf* in hardware, assertions were used to provide a heartbeat and "trace" the execution of process on the FPGA. Although this is not a common use of assertions in software, it can be useful to use assertions as a positive indicator rather than a negative indicator when an application is known to crash or hang. *Assert*(0) statements were placed at important points in the code for each FPGA process and *NABORT* was defined to stop the application from aborting. The new code with assertions added was executed via both software simulation and execution on the target platform. After comparing the line numbers of the failed assertions of both runs, it was found that the hang occurred at a memory read, which was causing the process to hang instead of exiting a loop. By identifying the problematic line of code using in-circuit assertions, we were able to debug the application and determined that the memory read should have been a memory write. This correction allowed the process to complete execution.

Table 7: DES hang-detection overhead.

EP2S180	Original	Assertion	Difference
Logic used	21051	21739	+688
(out of 143520)	(14.67%)	(15.15%)	(+0.48%)
Comb. ALUT	12986	13440	+454
(out of 143520)	(9.05%)	(9.36%)	(+0.32%)
Registers	13884	14015	+121
(out of 143520)	(9.67%)	(9.77%)	(+0.09%)
Block RAM	149184	149184	0
(9383040 bits)	(1.59%)	(1.59%)	(0%)
Block interconnect	38924	40241	+1317
(out of 536440)	(7.26%)	(7.50%)	(+0.25%)
Frequency (MHz)	78.8	77.0	−1.80 (−2.28%)

Next, automated hang detection was used on the same problematic DES application. The software hang detector was triggered by the timeout of a communication call. The line number of the software API call was reported back along with the line number (taken before the API call was made) that the hardware process was currently executing. Although hardware hang detection was working correctly in the FPGA, the hardware hang detector was not able to notify the application designer of the problematic line of code since the software API call in conjunction with the erroneous line in the hardware process caused all communication between the CPU and FPGA to stop. To solve this problem, a *sleep* of one second was place above the software API call that was notified as being hung in previous run. The addition of the *sleep* allowed the hardware hang detector to report back the exact line number for the memory read that should have been a memory write. The resource overhead of using automatic hang detection on the Triple-DES application is shown in Table 7. Hang detection had the highest, but still reasonable, percentage of ALUT (0.32%) and routing (0.25%) overhead because of the comparisons and connections made to the state machine of the encoder and decoder hardware process. The assertion pragma, *#pragma assert_FPGA_watch_dog*, was set to a timeout of a hundred million cycles which needed a 30-bit timing register. When using a 64-bit register, the frequency overhead increased to 5.7%. However, such overhead is very pessimistic because even with a 10 GHz clock speed, a 64-bit register supports a maximum timeout of about 58 years. For more typical cases, the frequency overhead should be less than 5.7%.

7.7. Assertion Limitations. The main limitation of in-circuit assertions is that overhead is dependent on the complexity of the assertion statements. For example, a designer could potentially verify a signal processing filter using an assertion statement that performs an FFT and then checks to see if a particular frequency is below a predefined value. In this case, the synthesized assertion would contain a circuit for an FFT, which could have a large overhead. Note that such overhead is not a limitation of the presented synthesis techniques, but rather a fundamental limitation of in-circuit assertions.

To minimize this overhead, we suggest certain coding practices. Whenever possible, designers should use assertion statements that compare precomputed values. Designers should try to avoid consolidating assertions in loops with comparison values stored in arrays because the unnecessary usage of arrays and loops with assertions can increase overhead as shown in Section 7.5. Designers should try to avoid using many logical operators because these operators can cause the HLS tool to create a large state machine to check all combination possibilities of the assertion as shown in Figure 3. By following these guidelines, the assertions will require a minimum amount of resources. Assertion parallelization optimization and resource replication optimization can increase the resource overhead to reduce the performance overhead. Accessing the same array multiple times in an assertion (e.g., $assert(a[i] > a[i-1])$) can be costly either in terms of performance or resource depending if resource replication optimization is used. Even accessing an array only once in an assertion could be costly if the application would normally be using the same array element in the same clock cycle.

8. Conclusions

High-level synthesis tools often rely upon software simulation for verification and debugging executing FPGA processes as threads on the CPU. However, FPGA programming bugs not exposed by software simulation become difficult to remedy once the application is executing on the target platform. Similarly, HLS tools often lack detailed timing-analysis capabilities, making it difficult for an application designer to determine which regions of an application do not meet timing constraints during FPGA execution. The assertion-based verification techniques presented in this paper provide ANSI-C-style verification both for the FPGA and CPU while in simulation and when executing on the target platform. This approach allows assertions to be seamlessly transferred from simulation to execution on the FPGA without requiring the designer to understand HDL or cycle-accurate simulators. The ability of assertions to verify a portion of the application's functionality and debug errors not found during software simulation was demonstrated. ANSI-C timing functions allowed assertions to check application time constraints during execution. Automated hang detection provided source information indicating where software or hardware processes failed to complete in a timely manner. Techniques were shown to enable debugging of errors not found during software simulation that incurred a small area overhead of 0.7% or less and a maximum clock frequency overhead of less than 3% for several application case studies on an EP2S180 and EP3SE260. The presented techniques were shown to be highly scalable, reducing resource overhead of 128 assertions by over 3x, requiring only 1.34% ALUT resources and improving clock frequency by over 18%. The performance overhead of optimized assertions was also demonstrated to be low, with no performance impact observed in the edge-detection case study in terms of frequency degradation or increased cycle usage. A general analysis of performance for single comparison assertions showed that the presented optimizations resulted in a throughput increase ranging from 33% to 100%, when compared to unoptimized assertions, potentially eliminating all throughput overhead. Future work includes further exploration and automation of hang detection.

Acknowledgments

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant no. EEC-0642422. The authors gratefully acknowledge vendor equipment and/or tools provided by Aldec, Altera, GiDEL, Impulse Accelerated Technologies, SRC, and XtremeData, Inc. Special thanks are due to University of Washington ACME Lab for an XD1000 version of the backprojection application that was ported to Novo-G.

References

[1] J. Williams, A. George, J. Richardson, K. Gosrani, and S. Suresh, "Fixedand reconfigurable multi-core device characterization for HPEC," in *Proceedings of High-Performance Embedded Computing (HPEC) Workshop*, Lexington, Mass, USA, September 2008.

[2] Deepchip, "Mindshare vs. marketshare," March 2008, http://www.deepchip.com/items/snug07-01.html.

[3] D. Pellerin and Thibault, *Practical FPGA Programming in C.*, Prentice Hall, Upper Saddle River, NJ, USA, 2005.

[4] D. S. Poznanovic, "Application development on the SRC Computers, Inc. systems," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, p. 78, April 2005.

[5] Impulse Accelerated Technologies, "Codeveloper's users guide," 2008.

[6] SRC Computers, Inc., "SRC-7 Carte v3.2 C programming environment guide," 2009.

[7] Accellera, "SystemVerilog 3.1a language reference manual," May 2004, http://www.eda.org/sv/SystemVerilog_3.1a.pdf.

[8] Accellera, "OVL open verification library manual, ver. 2.4," March 2009, http://www.accellera.org/activities/ovl.

[9] Accellera, "PSL language reference manual, ver. 1.1," June 2004,http://www.eda.org/vfv/docs/PSL-v1.1.pdf.

[10] M. Pellauer, M. Lis, D. Baltus, and R. Nikhil, "Synthesis of synchronousassertions with guarded atomic actions," in *Proceedings of the 3rd ACM and IEEE International Conference on Formal Methods and Modelsfor Co-Design (MEMOCODE '05)*, pp. 15–24, July 2005.

[11] M. Boulé, J. S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *Proceedings of the 8th International Symposium on Quality Electronic Design (ISQED '07)*, pp. 613–618, March 2007.

[12] M. R. Kakoee, M. Riazati, and S. Mohammadi, "Enhancing the testability of RTL designs using efficiently synthesized assertions," in *Proceedings of the 9th International Symposium on Quality Electronic Design (ISQED '08)*, pp. 230–235, March 2008.

[13] K. Camera and R. W. Brodersen, "An integrated debugging environment for FPGA computing platforms," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 311–316, September 2008.

[14] Xilinx, "ChipScope pro 10.1 software and cores user guide," March 2008, http://www.xilinx.com/ise/verification/chipscope_pro_sw_cores_10_1_ug029.pdf.
[15] Altera, "Design debugging using the SignalTap ii embedded logic analyzer," March 2009, http://www.altera.com/literature/hb/qts/qts_qii53009.pdf.
[16] K. Hemmert, J. Tripp, B. Hutchings, and P. Jackson, "Source level debugger for the sea cucumber synthesizing compiler," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03)*, pp. 228–237, April 2003.
[17] H. D. Foster, A. C. Krolnik, and D. J. Lacey, *Assertion-Based Design*, Springer, Berlin, Germany, 2004.
[18] F. Wang and F. Yu, "Assertion-checking of embedded software with dense-time semantics," in *Real-Time and Embedded Computing Systems and Applications*, pp. 254–278, Springer, Berlin, Germany, 2004.
[19] J. Curreri, G. Stitt, and A. D. George, "High-level synthesis techniquesfor in-circuit assertion-based verification," in *Proceedings of the 17th Reconfigurable Architectures Workshop (RAW '10)*, April 2010.
[20] P. Klemperer, R. Farivar, G. P. Saggese, N. Nakka, Z. Kalbarczyk, and R. Iyer, "FPGA implementation of the illinois reliability andsecurity engine," in *Proceedings of the International Conference on Dependable Systemsand Networks (DSN '06)*, pp. 220–221, June 2006.
[21] N. Nakka, G. P. Saggese, Z. Kalbarczyk, and R. K. Iyer, "An Architectural Framework for Detecting Process Hangs/Crashes," in *Dependable Computing - EDCC 2005*, pp. 103–121, Springer, Berlin, Germany, 2005.
[22] GNU, "The GNU C library reference manual," March 2009, http://www.gnu.org/software/libc/manual/.
[23] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, NY, USA, 1994.
[24] XtremeData Inc., "XD1000 FPGA coprocessor module for socket 940," http://www.xtremedatainc.com/pdf/XD1000_Brief.pdf.
[25] CHREC, "CHREC facilities," http://www.chrec.org/facilities.html.
[26] GiDEL, "PROCStar III PCIe x8 computation accelerator," http://www.gidel.com/pdf/PROCStarIII%20Product%20Brief.pdf.
[27] NIST, "Data encryption standard (DES)," October 1999, http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.