High-Level Synthesis Techniques for In-Circuit Assertion-Based Verification

John Curreri, Greg Stitt, Alan D. George
NSF Center for High-Performance Reconfigurable Computing (CHREC)
ECE Department, University of Florida

Abstract—**Field-Programmable Gate Arrays (FPGAs) are increasingly employed in both high-performance computing and embedded systems due to performance and power advantages compared to microprocessors. However, widespread usage of FPGAs has been limited by increased design complexity. High-level synthesis has reduced this complexity but often relies on inaccurate software simulation or lengthy register-transfer-level simulations for verification and debugging, which is unattractive to software developers. In this paper, we present high-level synthesis techniques that allow application designers to efficiently synthesize ANSI-C assertions into FPGA circuits, enabling real-time verification and debugging of circuits generated from high-level languages, while executing in the actual FPGA environment. Although not appropriate for all systems (e.g., safety-critical systems), the proposed techniques enable software developers to rapidly verify and debug FPGA applications, while reducing frequency by less than 3% and increasing FPGA resource utilization by less than 0.13% for several application case studies on an Altera Stratix-II EP2S180 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

High-level synthesis (HLS), also referred to as Electronic System Level (ESL), for FPGAs allows portions of a program written in a High-Level Language (HLL) such as C to be executed on an FPGA. Although HLS can ease circuit design for FPGA-based embedded systems, limited verification and debugging support has resulted in decreased productivity and limited usage of such tools.

Assertion-based verification (ABV), a widely used verification technique in today's Electronic Design Automation (EDA) tools [1], can potentially help application designers using HLS to determine if runtime behavior matches a program's intended 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 [2] and Carte-C [3], 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 [4], but may also occur due to incorrect hardware translation.

Alternatively, a designer could avoid software-simulation inconsistencies by performing ABV using post-synthesis register-transfer-level (RTL) simulation. However, adding assertions to HDL (Hardware Description Language) generated by an HLS tool can be a cumbersome process at best (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 who may be more concerned with productivity (e.g., rapid debugging of non-safety-critical systems).

In order to overcome the limitations of HLS verification, we present HLS techniques to efficiently support in-circuit assertions. These techniques enable a designer to use ANSI-C assertions seamlessly on both the CPU during simulation and the FPGA during hardware execution. By executing in-circuit assertions, designers can avoid the inconsistencies associated with software simulation and the lengthy times required by RTL simulation. Note that although the ANSI-C library [5] supports assertions via the `assert` function, most HLS tools do not support the use of the ANSI-C library on the FPGA, even during simulation.

To realize in-circuit ABV, the presented techniques address several key challenges: scalability, transparency, and portability. Scalability (large number 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 across systems is critical because HLS tools, such as Impulse-C, 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 semi-automated framework that implements the presented HLS techniques, we show that in-circuit assertions can be used to rapidly identify bugs 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 less than 0.13% have been observed with several application case studies on an Altera Stratix-II EP2S180). 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 non-experts who may otherwise lack the skills required to verify HLS-generated circuits.

This paper is presented as follows. Section 2 discusses related work. Section 3 discusses the assertion synthesis techniques. Section 4 describes the experimental setup and framework used to evaluate the presented techniques. Section 5 presents experimental results. Section 6 provides conclusions.

2. Related Research

Many languages and libraries enable assertions in HDLs during simulation, such as VHDL assertion statements, SystemVerilog Assertions (SVA) [6], the Open Verification Library (OVL) [7], and the Property Specification Language (PSL) [8]. 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 [9], PSL [10], and OVL [11]. Academic tools such Camera's debugging environment [12] and commercial tools such as Temento's DiaLite also provide assertion checkers for HDL. Kakoee et al. show that in-circuit assertions [11] 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 [13] and Altera's SignalTap [14] 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 [15] 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.

After a comprehensive literature search, we found no previous work related to high-level synthesis of assertions. 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 [5] 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.

3. Assertion Synthesis and Optimizations

ANSI-C assertions [5], 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 [2]; 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 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.

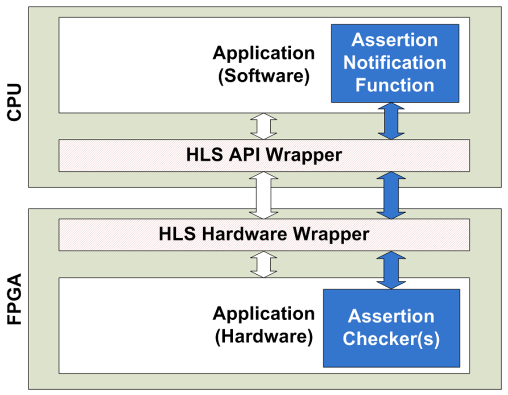

Fig. 1. Assertion framework

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 of 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 way 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 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 $assert((j <= 0 \,\|\, a[0] == i) \,\&\&\, (b[0] == 2 \,\|\, i > 0))$ 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 to 12.5% of its original single-cycle performance which could affect how application components interact with each other.

High-level synthesis 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, high-level synthesis 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. 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.

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, high-level synthesis can perform resource replication by duplicating shared resources.

For example, 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. High-level synthesis 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 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 high-level synthesis 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 5.4, resource replication improved performance by 33% allowing the application's pipeline rate (i.e., throughput) 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, high-level synthesis 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 high-level synthesis optimization [16] 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 high-level synthesis tool from statically detecting mutually exclusive execution of all assertions.

Due to this limitation, high-level synthesis can potentially apply existing resource-sharing techniques to assertions within non-pipelined 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, high-level synthesis 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.

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 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. Assertion Framework

To evaluate the assertion synthesis techniques discussed in the previous section, 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. Ideally, the techniques would be directly integrated into an HLS tool.

4.1. 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, 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 to receive, decode, and display failed assertions using the ANSI-C output format. An example of this automated code instrumentation is shown in Figure 2.

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, which for embedded systems could be stored to memory, displayed on a LCD, or even flashed as a sequence on an LED.

Note that other changes are needed to route the stream to the CPU, such as API calls to create and maintain the stream.

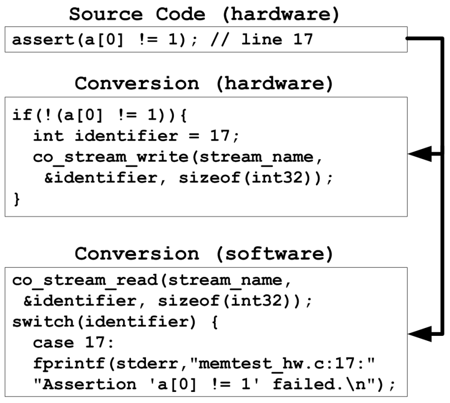
Source Code (hardware)
```
assert(a[0] != 1); // line 17
```
Conversion (hardware)
```
if(!(a[0] != 1)){
  int identifier = 17;
  co_stream_write(stream_name,
   &identifier, sizeof(int32));
}
```
Conversion (software)
```
co_stream_read(stream_name,
 &identifier, sizeof(int32));
switch(identifier) {
  case 17:
  fprintf(stderr,"memtest_hw.c:17:"
  "Assertion 'a[0] != 1' failed.\n");
```

Fig. 2. HLL assertion instrumentation

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.

4.2. 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.

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.

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 which 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. 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 5.3.

4.3. HLS Tool and Platform

The framework currently uses Impulse-C 3.30 and Quartus 9. The target platform is an XtremeData XD1000 [17] containing a dual-processor motherboard with an Altera Stratix-II EP2S180 FPGA in one of the Opteron sockets. Although the XD1000 is a high-performance computing platform, Impulse-C also supports embedded PowerPC and MicroBlaze processors [4]. Furthermore, the XD1000 is representative of FPGA-based embedded systems that combine CPUs with an FPGA. 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-C, 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-C's FPGA function calls for failed assertions as opposed to monitoring Impulse-C's FPGA processes.

5. Experimental Results

This section presents experimental results that evaluate the utility and overhead of the presented assertion synthesis techniques. Section 5.1 motivates the need for in-circuit assertions by illustrating case studies where assertions pass during simulation but fail during FPGA execution. Section 5.2 illustrates area and clock frequency overhead for two application case studies. Section 5.3 evaluates the scalability of assertions in terms of resource and frequency overhead by applying resource sharing optimizations to the communication channels as presented in Section 4.2. Section 5.4 evaluates assertion performance overhead and improvements from optimizations.

5.1. In-Circuit Verification and Debugging

In this section, we present two examples illustrating how assertions can be used for in-circuit verification and debugging. In the first example, the code in Figure 3 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.[1] 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.

[1]It is possible for a translation mistake to also have an effect on an assertion

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.

```
1 co_uint64 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;
```

Fig. 3. In-circuit verification example

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.

The next example illustrates how in-circuit assertions can be used to detect when an application fails to complete (hangs), even when software simulation runs to completion. In an effort to speedup the DES application described in Section 5.2, 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 "trace" the execution of process on the FPGA. Although this is not a common use of asserts in software, it can be useful to use asserts 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. The memory read should have been a memory write and this correction allowed the process to complete execution.

Without assertions, cycle-accurate simulation could be used to determine the line of code at which the hang occurred. However, cycle-accurate simulation would entail a much more complex process of setting up an HDL testbench to find the hung state of the state machine and determine the corresponding line of code linked to that state. The simulation may also have to include the Impulse-C communication wrapper code if the read memory call itself caused the state machine to hang. Using high-level assertions, while not appropriate in all cases, can reduce design time and eliminate the need to understand HDL.

5.2. Application Case Studies

The first application case study shows the area and clock frequency overhead associated with adding performance optimized assertion statements to a Triple-DES [18] application provided by Impulse-C, which sends encrypted text files to the FPGA to be decoded. Two assertion statements were added 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. We suspect this decrease occurred due to the fact that the assertions were in a nested loop of the application. The ALUT 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.

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 $5x5$ 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. 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 performance optimization of the assertions increased ALUT resource utilization from 0.03% to 0.06% on the EP2S180.

TABLE 1
TRIPLE-DES ASSERTION OVERHEAD

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

TABLE 2
EDGE-DETECTION ASSERTION OVERHEAD

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

5.3. Assertion Scalability

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. 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 32 FPGA processes, 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 and an extra Impulse-C streaming communication channel to notify the CPU of failed assertions.

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 4 for the 128-process application with unoptimized assertions.

By applying the resource sharing optimization only to the communication channels as described in Section 4.2 (and not the assertion resources), the resource overhead was decreased. The resource overhead on the EP2S180, as shown in Figure 5, 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 4, which represents over an 18% improvement. The frequency of the application with assertion optimizations (189.3 MHz) was very close to the original applications 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.

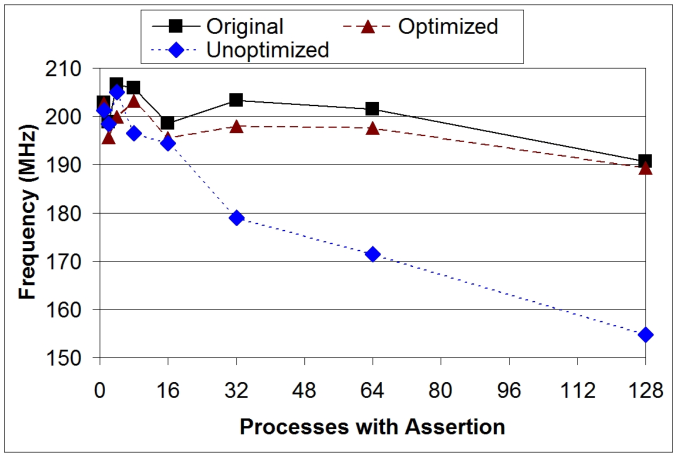

Fig. 4. Assertion frequency scalability

5.4. Assertion Performance Overhead

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 4.2, 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.

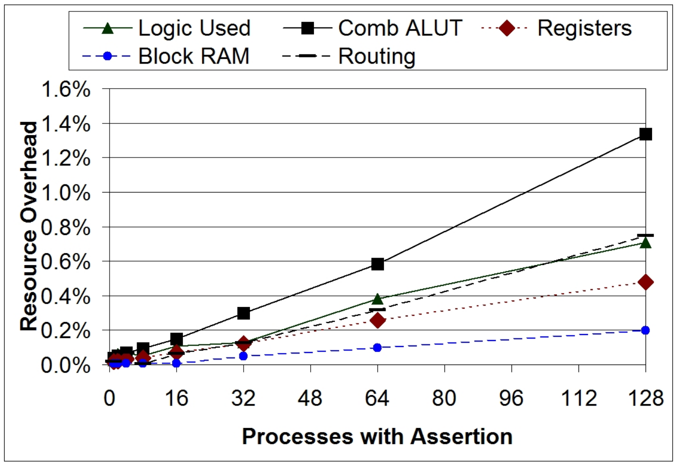

Fig. 5. Optimized assertion resource scalability

Table 3 shows the latency overhead for non-pipelined, 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. 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 3
NON-PIPELINED 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

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 $2x$ slow down 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 which 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. For assertions used 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 4.2. 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 non-optimized version.

6. CONCLUSIONS

High-level synthesis tools often rely upon software simulation for verification. However, FPGA programming bugs not exposed by software simulation become difficult to remedy once the application is executing on the target platform. In this paper, we present high-level synthesis techniques for in-circuit assertion-based verification that enables ANSI-C-style verification on FPGAs. The techniques were shown to enable debugging of errors not found during software simulation with a small area overhead of approximately 0.1% and a maximum clock frequency overhead of approximately 3% for several application case studies on an EP2S180. The presented techniques were shown to be highly scalable, reducing resource overhead of 128 assertions by over $3x$ and improving clock frequency by over 18%. A general analysis of performance for single comparison assertions showed that the presented optimizations resulted in a throughput increase ranging from 33% to 100%, eliminating all throughput overhead. Future work includes adding the ability for assertions to check the timing of the lines of code, which would be useful for verifying timing properties of an application in terms of clock cycles.

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, Impulse Accelerated Technologies, SRC, and XtremeData, Inc.

REFERENCES

[1] Deepchip, "Mindshare vs. marketshare," http://www.deepchip.com/items/snug07-01.html, March 2008.
[2] D. Pellerin and Thibault, *Practical FPGA Programming in C*. Prentice Hall PTR, 2005.
[3] D. Poznanovic, "Application development on the SRC Computers, Inc. systems," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, April 2005, pp. 78a–78a.
[4] Impulse Accelerated Technologies, "Codeveloper's users guide," 2008.
[5] GNU, "The GNU C library reference manual," http://www.gnu.org/software/libc/manual, March 2009.
[6] Accellera, "SystemVerilog 3.1a language reference manual," http://www.eda.org/sv/SystemVerilog_3.1a.pdf, May 2004.
[7] ——, "OVL open verification library manual, ver. 2.4," http://www.accellera.org/activities/ovl, March 2009.
[8] ——, "PSL language reference manual, ver. 1.1," http://www.eda.org/vfv/docs/PSL-v1.1.pdf, June 2004.
[9] M. Pellauer, M. Lis, D. Baltus, and R. Nikhil, "Synthesis of synchronous assertions with guarded atomic actions," in *Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Proceedings. Third ACM and IEEE International Conference on*, July 2005, pp. 15–24.
[10] M. Boule, J.-S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *Quality Electronic Design, 2007. ISQED '07. 8th International Symposium on*, March 2007, pp. 613–620.
[11] M. Kakoee, M. Riazati, and S. Mohammadi, "Enhancing the testability of RTL designs using efficiently synthesized assertions," in *Quality Electronic Design, 2008. ISQED 2008. 9th International Symposium on*, March 2008, pp. 230–235.
[12] K. Camera and R. Brodersen, "An integrated debugging environment for fpga computing platforms," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, Sept. 2008, pp. 311–316.
[13] Xilinx, "ChipScope pro 10.1 software and cores user guide," http://www.xilinx.com/ise/verification/ chipscope_pro_sw_cores_10_1_ug029.pdf, March 2008.
[14] Altera, "Design debugging using the SignalTap ii embedded logic analyzer," http://www.altera.com/literature/hb/qts/qts_qii53009.pdf, March 2009.
[15] K. Hemmert, J. Tripp, B. Hutchings, and P. Jackson, "Source level debugger for the sea cucumber synthesizing compiler," in *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, April 2003, pp. 228–237.
[16] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
[17] XtremeData Inc., "XD1000 FPGA coprocessor module for socket 940," http://www.xtremedatainc.com/pdf/XD1000_Brief.pdf.
[18] NIST, "Data encryption standard (DES)," http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf, October 1999.