A C++ ASIC Design Methodology Facilitated by a C++-Verilog Translator

Dan Joyce, Andreas Nowatzyk[*], and Robert Stets

Non-Stop Hardware Development
Compaq Computer Corporation
Austin, TX 78728-6699
dan.joyce@compaq.com

Western Research Laboratory
Compaq Computer Corporation
Palo Alto, CA 94301
agn@acm.org, robert.stets@compaq.com

Abstract

In this paper, we discuss a C++-based hardware design methodology that employs an automatic C++-Verilog translator. The choice of C++ as the primary RTL representation enables a simulator with speed at least 50 times faster than a comparable Verilog simulator, and the automatic translator enables a single C++ code base to drive a standard Verilog-based ASIC design flow. Our discussion includes details on our mix of custom and commercial tools and on our experiences with this methodology.

1. Introduction

Today's large and complex hardware designs require increasingly large amounts of simulation to validate. Unfortunately, the simulation speed of hardware design languages such as Verilog are relatively slow primarily due to the event-driven nature of their simulation engines. As a result, it is often infeasible to simulate full designs. Instead, designs are typically split into modules, and these modules are driven by bus functional models (BFMs) and tested in isolation or in small groups. Unfortunately, a BFM-based testing approach is not appropriate for large designs with tightly coupled modules that can only be effectively tested in full or near-full configurations.

The Piranha chip multiprocessor [1] is such a design with a large collection of tightly coupled modules. In a simplified description, a Piranha chip includes eight processors, each with their own level one cache, a level two cache shared by all processors, and a mechanism for maintaining cache coherence both inside a chip and then also across multiple chips connected via a high speed multiprocessor interconnect. The cache coherence mechanism operates off of the global state of the system, thereby making it difficult to test modules in isolation. Furthermore, the complexity of the entire chip produces a very challenging verification task. A very high performance simulation environment is necessary to thoroughly test such a design.

[*]The author has moved to the Robotics Institute at Carnegie Mellon University, 5000 Forbes AVE, Pittsburgh, PA 15213.

Motivated by these concerns, we chose to develop an RTL model of Piranha in C++, rather than in Verilog. We believed the simulation speeds obtained in a C++ environment would be much greater than those obtained in a Verilog environment, and would thereby make our verification effort feasible. Importantly, this choice was made despite the fact that Piranha is to be fabricated in an ASIC design flow based on synthesizable Verilog.

To reach the ASIC design flow, we originally intended to hand translate the C++ code to Verilog. However, as the individual modules inside Piranha grew rather large (between 5,000 and 10,000 lines of C++ code), the challenge of maintaining two separate code bases correspondingly increased. We decided to investigate tools for C++ to Verilog translation. In evaluating the possible tools, we were concerned with the quality of the translated Verilog and with the level of C++ coding restrictions imposed by the translation tool.

After investigation and some in-house testing, we found the C-Level System Compiler [3], a C++ to Verilog translation tool, to perform well on the above criteria, and so we incorporated the tool into our design flow. The tool enables us to use a single C++ source code base to drive a conventional ASIC design flow.

In the rest of this paper, we further discuss our C++-based methodology. In the following Section, we provide details on our C++ RTL, the C++ simulation infrastructure, and the C-Level C++ to Verilog translation process. In Section 3, we discuss the performance of our C++ simulator, along with results from our Verilog translation and synthesis runs. Finally in Section 4, we conclude with a discussion of our experiences.

2. C++-driven Design Methodology

Our design methodology is based on a cycle-accurate RTL model of the Piranha chip[1] written in a "stylized"

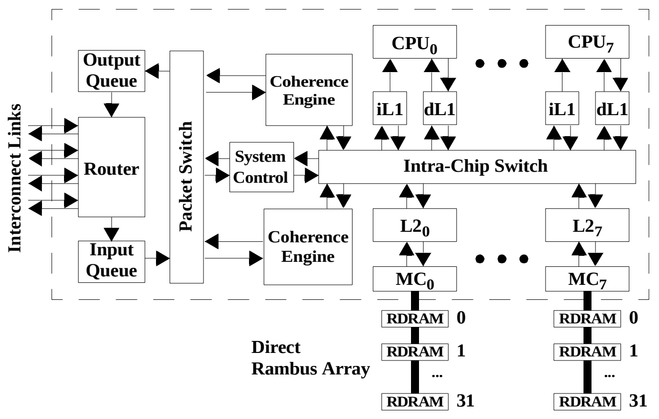

Figure 1. Module diagram of the Piranha chip multiprocessor.The dashed line represent the chip boundary.

version of C++. Figure 1 shows the Piranha chip broken down into its main modules, which include the first level data and instruction caches (dL1 and iL1), second level cache (L2), coherence engines, system controller, packet switch, input and output queues, and network router. These modules have been implemented in our stylized C++, and are targeted for machine translation to Verilog. Of the remaining modules, the Alpha processor (CPU) and Memory Controller (MC) are legacy modules that were originally implemented in Verilog. The Intra-Chip Switch is a physical design challenge and so is generated by a datapath compiler directly to Verilog. To maintain the efficiency of our C++-based simulation, we have simplified C++ representations of these three modules. As described in Section 2.2, we do not require the full functionality of the Alpha Core to test our cache coherence mechanism, so a simplified C++ version of the Alpha Core is acceptable.

In the remainder of this Section, we will discuss the stylized C++ code and special considerations for C++-based hardware design, the Piranha C++ simulator, and the automated C++-Verilog translation process.

2.1 Piranha Stylized C++

Our use of stylized C++ code, rather than standard C++, as our RTL code serves three purposes:

- to hide complex C++ syntax from hardware designers,
- to incorporate source code constructs useful for hardware designs (*e.g.* bit field accesses), and
- to provide a rich source format for preprocessors that enable the design flow

The first two points help maintain the work efficiency of designers trained in more tradition hardware description languages, while the third point enables much of the C++ Piranha Simulator (PS1) infrastructure to be built automatically.

The stylized C++ code deviates from standard C++ code in that each code module (corresponds to a hardware module) is divided into sections marked by reserved keywords. The code sections act as annotations that identify information necessary to build the full simulator, namely the module's input/output port list, the module's internal registers, the logic to execute on a rising clock edge (moore logic), and the logic to execute on a falling clock edge (mealy logic).

The PS1 preprocessor uses this information to automatically construct the source for the simulator executable. Each hardware module is modeled as a C++ class, whose primary member functions implement the moore and mealy logic for the module. The preprocessors use the input/output port list to construct prototypes for a module's moore and mealy functions. The module's internal registers, which need to hold state across moore and mealy function invocations, are modeled as class member variables, and then the moore and mealy function logic is taken directly from the corresponding sections of the stylized C++ code. If desired, the preprocessors can also automatically insert code for test coverage analysis in the modules.

After constructing the C++ source for classes to model each module, the PS1 preprocessor also uses the source annotations to automatically construct the necessary simulation infrastructure, namely the main clock loop which calls the moore and mealy functions for all modules at the appropriate times during a clock cycle. Again, the preprocessor uses the input/output port list in order to determine the inter-module signals and to instantiate the necessary code to model these connections. By automatically creating the clock loop and the "wiring" between modules, the PS1 preprocessor reduces burden on the toolsmith responsible for simulation infrastructure.

The PS1 preprocessor is also designed to support extended source code constructs for the hardware design domain. Currently, the preprocessor recognizes a `variable[END:START]` construct as a bit field operation where `END` and `START` mark range of bits in the named variable.

2.1.1 A Difficulty in C++-based Hardware Modeling

C++ is an inherently sequential language that executes statements as they occur in a sequential block. As such, the

1. The Piranha project consists of two chips: the chip multiprocessor (CMP) and the IO processor (IOP). The IOP is a scaled down version of the CMP that also includes a PCI subsystem. Both chips are being developed under the same methodology. In this paper, we will use the CMP as our illustrative example. Details on the IOP architecture can be found in an earlier paper [2].

language does not model concurrent activities, and this missing feature complicates the modeling of hardware.

This shortcoming specifically impacts a C++-based hardware model in the area of register manipulation. If a register is modeled as a conventional variable in C++, an update to that variable will take effect immediately. The programmer must take care to ensure that subsequent statements executing during the same simulated clock cycle do not access the register, otherwise the statements will see a register value that was intended to be seen in the next clock cycle.

In PS1, we use two techniques to avoid this problem, depending on the scope of the accesses to the register. If the register is private to a module, a programmer can conceivably structure code to ensure the register variable is updated only after all read accesses are performed. This approach is fragile and error prone, so instead, we follow a convention that register updates are performed to a separate variable named for the register with “_ns” suffix (ns is short for “next state”). For example, `count_ns` holds the next state value of the `count` register. After all code for the clock cycle is finished, the programmer must include code to copy the values of all _ns variables into the corresponding register variables, thereby updating their value for the next clock cycle. With this convention, it is easy for the designer to ensure that all register updates are performed after all read accesses.

This approach is not feasible for register values that span modules. For these modules, we require the variables to be modeled with a custom-crafted C++ “Signal” class. This class intercepts all read and write accesses to the variable by overriding the cast and assignment operators, respectively. When an assignment occurs, the Signal class implementation stores the new data value in a pending structure that also includes the clock value at which it should take effect. All cast operations compare the current clock to the clock of a pending data value, update the current value if the appropriate time has passed, and then pass back the current value of the variable. The class also reports an error if the register is updated more than once a clock cycle. The Signal class induces more overhead than plain variable operation, however it ensures correct register semantics.

To ensure that register semantics are maintained across modules, we require all moore cycle functions to only have register values (implemented by the Signal class) as inputs. Mealy cycle functions may have register inputs or combinational logic inputs produced from the Moore cycle in the corresponding module. (PS1 currently does not ensure a Mealy function combinational input is sourced from a Moore function. This requirement is left to the designer to enforce.) With these requirements, we ensure that register values are maintained correctly for the entire simulated clock cycle.

This set of coding conventions facilitates C++ hardware design by providing a structured method to manage the tricky issue of register manipulation. Also, the set of annotations in our stylized C++ allows much of the PS1 simulator infrastructure to be automatically constructed. PS1 is described in more detail in the next Section.

2.2 PS1 Simulator

The PS1 simulator has been built to test the Piranha coherence mechanism as a whole. This task only requires memory operations, and so we have replaced each full-blown Alpha processor[1] with a very efficient BFM that launches simulated memory requests into the system. The BFM is driven by an interpreted, coroutine-based test language. Coroutines are used to model Load and Store memory operations. One of these coroutines begins by issuing the operation to the L1, and then it tracks the progress of the operation by inspecting internal chip state. The coroutines maintain a separate image of memory, and use this image to dynamically check system operation. The interpreted test language and the Load and Store coroutines make it very quick and easy to construct new tests.

As mentioned above, the primary implementations of the Intra-Chip Switch and the Memory Controller have been done in Verilog, however we also have written simplified C++ models of these modules. The remaining modules under test are the actual implementation models, all written in C++.

When the goal is to stress the coherence implementation, PS1 is built with all C++ models. In addition to this functionality, PS1 also has the option to insert Verilog models in place or along side of specified modules. The former allows a Verilog module to be tested in the simulation environment, while the latter allows C++ and Verilog models of the same module to be co-simulated, with PS1 reporting any miscomparisons of the outputs of the modules at each cycle. This co-simulation technique is used to verify the accuracy of our translated Verilog models.

2.3 C-Level C++ to Verilog Translation

The C-Level System Compiler translates C++ code to Verilog. Fortunately, as we had written a substantial amount of C++ code before deciding on using machine translation, the System Compiler places relatively few

1. The actual Alpha processor implementation can be validated in isolation using tools from Compaq’s Modeling, Tools, and Verification group, which is responsible for validating production Alpha processors. For this verification task, the level of performance from Verilog is acceptable.

constraints on the C++ code structure. We already used preprocessors to transform our stylized C++ to standard C++ for simulation. The preprocessors only needed slight modification to allow them to also translate to C-Level style C++. Once this was done, the C-Level C++ could be read directly into System Compiler tool.

However, beyond the preprocessors, some modifications are still required to our original stylized C++ code. The main constraint is that a module's code must be split into distinct sections based on whether the code is implementing combinational logic or updating register variables. Fortunately, our _ns variable usage convention described in the previous Section had largely kept combinational and register update logic separate, therefore for all modules except for one, it was easy to transform our code into the distinct, required sections. The sole exception was the code for the coherence engines, where the register update and combinational logic was carefully intertwined. Significant effort was required to transform this module into a suitable C-Level format.

There is also a small set of programming idioms for which System Compiler automatically creates state machines. A designer must be careful to avoid these idioms to avoid unwanted state machines. These idioms are clearly defined in the System Compiler documentation [3]. In a few cases, our original C++ code contained these idioms and required simple, localized modifications.

In all but the case of the coherence engines, our modules could be changed to a C-Level format with a minimum of changes and without sacrificing code clarity or simulation performance.

As described in this Section, we use a stylized version of C++ to build an RTL model of our Piranha chip. Through automated processes, this C++ representation is used to build a full system simulation environment and also is converted into equivalent Verilog representations. In the next Section, we discuss the practical results of our simulation environment and the quality of our translated Verilog code, along with details on our experiences in building and using this infrastructure.

3. Results and Experiences

In this Section, we will assess the impact of our methodology choice by discussing the efficiency of our C++ simulation, the results from our initial synthesis runs, and also several unexpected issues that arose from our choice.

During this Section, we will frequently refer to the tools listed in Table 1. These are the mix of custom and commercial tools used in our methodology.

Tool	Purpose
Piranha Preprocessors	Perl and LEX/YACC tools used to translate stylized C++ to PS1 or CLevel format and to automatically construct PS1 infrastructure.
PS1	Primary simulator for C++ environment.
cxx (Compaq)	DEC C++ compiler (for building PS1)
System Compiler (C-Level)	C++-Verilog Translator
VCS (Synopsis)	Verilog simulator
DC-Shell (Synopsys)	Synthesis tool
DCPI (Compaq)	Profiling tool used to help optimize PS1 performance

Table 1: The mix of custom and commercial tools used for Piranha. Commercial tools are indicated by including their developer's names in parentheses.

3.1 PS1 Simulator Results

To quantify the execution speed of the PS1 simulator we have performed a number of experiments on our local cluster. The cluster consists primarily of AlphaServer ES40 machines. These machines have four 500MHz Alpha 21264 processors, each with their own on-chip 64K, 2-way associative L1 cache and a board-level 4M L2 cache. The processors share 8GB of RAM. We performed our experiments on one of these machines running Tru64 Unix 4.0F in multiuser mode. During our tests, there was no activity on the system except for the normal system background daemons.

The best comparison would have been to have the entire design in C++ and compare the speed to the entire design in Verilog. But unfortunately the Verilog representations of the full chip were not available at the time we did this evaluation. We created two configurations of PS1 using the Verilog available. The first configuration, which we will call PS1_C++, consisted only of C++ coded modules. These were 46 instantiations of 10 different modules, including 8 dL1's, 8 iL1's, 8 L2's, 2 Coherence Engines, 1 Router, 1 Input Queue, 1 Output Queue, 8 Memory Controller models, 8 DRAM models, and 1 Intra-Chip Switch. The second configuration, which we will call PS1_Mixed consists of 8 dL1's in Verilog, and the rest in C++. Obviously this gives a very low estimate for the

Module	L1	L2	RT	OQ	CE
Approximate Size	5,000 C++ lines 2500 registers 5 RAMs	10,000 C++ lines 8000 registers 8 RAMs	1,000 C++ lines 50 registers 1 RAM	1,000 C++ lines 50 registers 2 RAMs	10,000 C++ lines 3500 registers 6 RAMs
C++ Modification	Complete	Complete	Complete	Complete	Progressing
Verilog Translation (Translation Time)	Complete (10 minutes)	Complete (45 minutes)	Complete (1 minute)	Complete (1 minute)	
Verilog Structural Simulation	Passing tests: 70% coverage	Passing tests: 70% coverage	Passing tests: 80% coverage	Passing tests: 80% coverage	
Synthesis to Gates (Synthesis Time)	Yes (6 hours)	Yes (4 days)			
Verilog Gate-Level Simulation	Passing tests: 70% coverage				

Table 2: Status of Verilog translation, simulation, and synthesis for the Level 1 (L1) and Level 2 (L2) caches, the Router (RT), the Output Queue (OQ) and the Coherence Engines (CE) The other chip modules have not begun the Verilog translation process yet.

simulation speedup. With only one of 10 types of modules in Verilog and the other 9 in C++ the memory image is much smaller than would be the case for all the modules in Verilog. Also, with 8 instantiations in Verilog and the other 38 in C++ there are obviously many events not being simulated in Verilog.

We executed three of tests that exercise all the L1's, L2's, MC's, and Intra-chip Switch extensively (code coverage of approximately 70%). Each of the three tests were executed three times, and then the results of the nine tests were used to calculate an average number of simulated cycles per seconds. (The average values of the three PS1_C++ tests differed by less than 5%, while the average values of the three PS1_Mixed tests differed by less than 1%.)

We found the PS1_C++ full C++ simulation averaged approximately 1050 simulated clocks per second, while the PS1_Mixed mixed Verilog and C++ simulation averaged only 20 simulated clocks per second. The mixed simulation incurs overhead in transferring signals between the Verilog and C++ domain (using a PLI interface), and so we further investigated the results to ensure that this transfer overhead did account for the relatively slow Verilog simulation performance.

We used the DCPI [1] continuous profiling tool to gather information on time breakdown by function for executions of both PS1_C++ and PS1_Mixed. DCPI is highly optimized, and typically adds only 1-3% overhead for most workloads.

In the case of PS1_Mixed, we were only able to identify our C++ functions that simulate the Intra-chip Switch, L2, and MC, and also our routines to transfer signals between domains. The remaining functions were from the Verilog simulation. We found that our C++ routines in PS1_Mixed accounted for slightly less than 5% of the execution time. The remaining 95% of the time was spent in the Verilog simulation, so we can reasonably conclude that the above simulation speeds are representative of a Verilog design.

Without more intimate knowledge of the Verilog simulation internals, we can not point to the exact reasons for the relatively low Verilog simulation speed. Ideally, we would be able to separate out the infrastructure and simulation overhead from the actual time spent simulating the design.

Fortunately, we can extract this separation from the PS1_C++ execution time using DCPI profile information and our knowledge of the source code. We found that only 11% is spent in the simulation infrastructure and overhead (*e.g.* main clock loop, Alpha BFM, and test language interpreter). The remaining 89% of the execution time is spent simulating the entire Piranha. The efficiency of the simulation execution -- almost nine tenths of the time is spent directly simulating the design -- is the key to the C++ environment's superior performance.

3.2 Verilog Translation and Synthesis Results

As mentioned, our physical design tools and the back-end of the ASIC design flow work off of Verilog and not

C++. Our initial plan was to manually translate the C++ modules to Verilog, but toward the end of our C++ development, we found that the CLevel System Compiler could potentially perform a machine C++-to-Verilog translation. The main advantage of this approach is that, if successful, we would only have to manage one code base. The criteria for success is a faithful translation to Verilog, synthesizable Verilog code, and a straightforward path for timing re-designs.

To use the System Compiler, a one-time modification of the C++ code is necessary to change the code to a suitable organization for C-Level. After this one-time modification, the System Compiler can be used to translate the C++ to Verilog, which we then simulate at the structural level and synthesize to a gate-level representation. The gate-level representation is simulated for correctness and checked for long timing paths. We have started this process for five of the modules, and the status of each module is shown in Table 2. Although we are not complete with the entire chip, we feel that we have progressed far enough to draw conclusions on the process. In the following, we discuss each of the listed modules in more detail and also include bits of own experience with this process.

L1: The L1 was the first major module to pass through the Verilog translation process. The original C++ code was written in a very structural manner and followed the _ns convention, described in Section 2.1.1, for manipulating internal registers, so the only major change to support C-Level was to introduce explicit RAM and LPRA models. Originally, these memories were simply modelled by plain arrays in C++. C-Level translates plain arrays into discrete registers, and so to avoid this, we had to model our memories in C++ as distinct classes (or, as an alternative, functions). The System Compiler recognizes these classes as separate modules and does not instantiate discrete registers in their place. This change, along with a code reorganization to conform to C-Level expectations, required approximately a day or two. The L1 C++ code was then ready for translation to Verilog.

For several weeks however, the translation attempts were fruitless, as the complexity and size of the L1 stressed the System Compiler implementation. The compiler successfully produced translations of the L1, however, the translations contained very large numbers of temporary registers, which bogged down the synthesis tool. After a few improvements to the System Compiler by the C-Level team, the L1 now translates to Verilog in approximately 10 minutes on a 448MHz UltraSPARC II machine with 4G of RAM, and then the corresponding synthesis run completes in under 6 hours on a similar machine.

After obtaining a Verilog translation that passed reasonably through the synthesis step, we then worked on simulating the structural and the gate-level version of the L1. This process verified the faithfulness of the Verilog translation. The only problems encountered during the simulation were two variables that had incorrect bit widths in C++ and the proper use of our vendor's RAM macros. Parts of our C++ code use full integers to model all sizes of variables, as this method is faster than using C++ bit fields. With regard to the Vendor RAM macros, it required several iterations to work out the desired RAM timing and to ensure that PS1 transferred signals at the appropriate time during the clock cycle. After working through these two problems, both the structural and gate-level Verilog are passing random stress tests, as validated by our PS1 co-simulation process.

The L1 synthesis runs have also been successful in terms of timing and size. The long timing paths are isolated to areas of aggressive design, and the physical size falls within our conservative estimates.

L2: The L2 began the translation process shortly after the L1. Like the L1, the modifications the C-Level format were largely constrained to explicit RAM and LPRA instantiations and a code reorganization. This step took longer than the L1 however, as the responsible designer was not as familiar with C++ and with the C-Level requirements.

The L2 quickly encountered significant problems with the synthesis step, however. The larger size of the L2 (see Table 2) is proving difficult for our synthesis tool. As can be seen from Table 2, the L2 synthesis step is taking 4 days, which is much too long for iterative design. The synthesis time can be reduced by instantiating the L2 as a hierarchical design. Hierarchy allows synthesis tools to work on smaller, more manageable chunks.

Our synthesis problems with the L2 were exacerbated by the fact that our preprocessors did not support hierarchy within a module, and this led designers to implement flat designs for each module. We are currently working to instantiate hierarchy inside the L2 design. The hierarchy will in essence provide smaller chunks of work for the synthesis tool, thereby reducing synthesis time and improving the quality of the results.

Router, Input Queue, Output Queue: The Router Input Queue and Output Queue are relatively small modules, and were trivial to port to a C-Level format. The Verilog translation and synthesis steps have been very successful. The next step for both of these modules is to begin timing re-designs.

Coherence Engine: The Coherence Engines have proven to be particularly difficult to port to a C-Level format. The original C++ designer did not use the _ns convention, but instead carefully reasoned about the register semantics in the code. The designer was very successful, as the coherence engines have shown by far the fewest of bugs in our simulations. Currently, a different designer, the designer who was to be responsible for the manual Verilog translation, has been restructuring the C++ code to first use explicit memories and second to separate the code into combinational and synchronous logic, as System Compiler requires. The modifications to the code have been significant, and although now complete, are not simulating correctly. We are adding a temporary capability to PS1 to allow side-by-side simulation of two C++ versions of a module to help debug the System Compiler transformations. Also, judging by the size of the module and our experience with the L2, we most likely need to instantiate hierarchy within the module.

We believe our work this point has clearly proven the feasibility of using a C++-to-Verilog translator, in this case C-Level's System Compiler, to drive the physical design tools in an ASIC flow. In our case, our fortuitous decision to use the _ns register manipulation convention simplified the preparation of our code for the System Compiler, however unfortunately, our initial decision to forgo hierarchy within our modules has hampered our physical design process. We have found that the System Compiler's translations are very faithful and that timing re-designs can be made easily as our C++ code is fairly structural. The limiting factor on the time to do re-designs is the synthesis step itself, and this step needs to be improved for one, and probably two, of our modules.

3.3 Practical Issues

An important part of evaluating this methodology obviously lies outside of the above "performance" results. Our experiences have led to a few interesting insights, especially with regard to Verilog-trained designers in this environment and to our simulation infrastructure.

3.3.1 Verilog Designers in New Environment

The Verilog designers in our group have adapted very well to our C++ environment. The designers have been able to make isolated changes, for example for timing re-designs or code reorganization for C-Level, as it has been easy for them to pick up the basic C++ syntax. At times however, their required changes have quickly gone beyond their C++ knowledge. The most notable example is in the implementation of explicit RAM and LPRA models. These models are best implemented as either full C++ classes or templates, and in some cases, we used C++ static storage classes to maintain simulator efficiency. These cases all required considerable assistance from our group's experienced C++ programmers.

The switch to a new coding language obviously has the primary effect on the efficiency of our Verilog-trained designers. Aside from this effect, some of the Verilog designers were initially uncomfortable with aspects of the environment, such as the lack of integrated text editing, simulation, and waveform display tool and also the fact that our C++ environment requires signals to be explicitly registered before they are included in waveform traces. The former issue will most likely be addressed if this type of C++-based methodology matures commercially.

3.3.2 Proprietary Simulation Environment

The primary disadvantage to our methodology is the dependence on and the required development overhead of the custom tools, most notably the PS1 simulator. The simulation infrastructure has required a significant amount of effort to build. In some cases, the custom work has been very beneficial as the functionality has been developed exactly for our needs. A specific example is the coroutine-based test language. The coroutine functionality has greatly simplified our test development and aided in debugging. In other cases however, such as the PS1 code coverage functionality, it would have been more efficient to use commercial solutions.

Although PS1 is relatively mature and stable, its required maintenance overhead is still significant due to the project's changing environment. As a simple example, PS1 uses a commercial waveform dump format and the corresponding viewing tool that is no longer supported. Due to resource constraints, we have not been able to upgrade to a more current waveform utility.

This example of a utility that becomes unsupported is relatively minor in that it does produce a first-order impact on the project. A more serious example is found in our PS1 preprocessors. Due to the long synthesis times, the L2 cache had to be split into a hierarchy of smaller submodules. Our PS1 preprocessors were only set up to handle one level of hierarchy. Instead of incurring the overhead of fixing the preprocessors, we chose the more expedient route of instantiating the hierarchy directly in the L2 C++ module. This choice exposed some of the complex C++ syntax to the hardware developer, an issue that the use of a preprocessor intended to avoid.

3.3.3 Miscellaneous Issues

Being a cycle simulator, the C++ simulation environment does not simulate asynchronous clocks accurately. If there are multiple clocks in the design, they must be made a multiple of the fastest clock. In our design, asynchronous clock crossings are verified by code inspection and review.

Vendor macros are modelled in the C++ by handcoded functions that simulate the macro. Of course in the Verilog simulation, the actual macro is used. This process can also be used for timing critical areas of the design where gates, either built manually or with a gate compiler, are needed.

Overall, our experiences indicate that the key factor in successfully deploying this methodology is to have enough talented and knowledgeable C++ programmers to support the project. These programmers are needed not only to maintain and improve the environment, but also to assist the Verilog designers in implementing their designs.

4. Conclusions

Driven by our project requirements of a high performance verification environment and a standard ASIC design flow, we have developed a novel C++-based ASIC design methodology and the requisite tools. In this methodology, we leverage a C++ RTL representation to enable a high performance simulator that is a factor of at least 50 greater than a comparable Verilog-based simulator, and a C++-to-Verilog translation tool to allow a single C++ source code base to drive a conventional Verilog-based ASIC design flow.

In the process of using this methodology, we have found that Verilog-trained designers can move relatively easily into a C++-based design environment, given a set of coding conventions to help manage the details of modelling concurrency on top of an inherently sequential language and also given reasonable access to team members with C++ programming experience. With regard to the feasibility of a single C++ source code base, we have also found that writing C++ in a cycle accurate manner produces code that is amenable to high quality machine translation to Verilog. If this path is taken however, the C++ RTL should be written with back-end consumers of the Verilog in mind. In particular, the C++ designer should add in sufficient hierarchy so as to simplify the task for synthesis tools.

On the negative side, we have found that the necessary reliance on custom tools can be very burdensome, and absolutely mandates a sufficient resource of C++-capable designers to maintain and improve the environment. This disadvantage may lessen as commercial C++-based hardware design environments mature. Currently missing from these environments are capabilities for automatically building the simulation infrastructure (similar to our preprocessing tools), high level languages for test stimulus construction (similar to our interpreted test language), and of course a high performance simulation environment capable of handling large designs very efficiently.

5. Acknowledgments

The Piranha design methodology is the product of the entire the Piranha team. Two of the authors, Joyce and Stets, are members of the current team, as are Luiz Barroso, Mark Dorland, Kourosh Gharachorloo, Ray Harlan, Tom Heynemann, Sachin Idgunji, Scott Johnson, James Keithan, David Lowell, Uma Maheswari, Joel McCormack, Harland Maxwell, Harold Miller, Mosur Ravishankar, Scott Smith, Jeff Sprouse, Hemendra Talesara, Alex Wakefield, Robert Willey, and Yuan Yu. Former team members, Robert McNamara and Ben Verghese, contributed significantly to the PS1 simulator and the Piranha preprocessing tools.

6. Bibliography

[1] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, D. Sites, M. Vandevoorde, C. A. Waldspurger, and W.E. Weihl. “Continuous Profiling: Where Have All The Cycles Gone?”. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles,* pages 1-14, Saint-Malo, France, October, 1997.

[2] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. “Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing”. In *Proceedingds of the Twenty-Seventh Annual Internation Symposium on Computer Architecture,* pages 282-293, Vancouver, Canada, June, 2000.

[3] CLevel Design, http://www.cleveldesign.com/home.html, July 31, 2000.