UNIVERSITY OF CALIFORNIA
RIVERSIDE

Emulation of SystemC Applications for Portable FPGA Binaries

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Scott Spencer Sirowy

June 2010

Dissertation Committee:

Dr. Frank Vahid, Chairperson
Dr. Tony Givargis
Dr. Sheldon X.-D. Tan

The Dissertation of Scott Spencer Sirowy is approved:

_____

_____

_____

Committee Chairperson

University of California, Riverside

ABSTRACT OF THE DISSERTATION

Emulation of SystemC Applications for Portable FPGA Binaries

by

Scott Spencer Sirowy

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2010
Dr. Frank Vahid, Chairperson

As FPGAs become more common in mainstream general-purpose computing platforms, capturing and distributing high-performance implementations of applications on FPGAs will become increasingly important. Even in the presence of C-based synthesis tools for FPGAs, designers continue to implement applications as circuits, due in large part to allow for capture of clever spatial, circuit-level implementation features leading to superior performance and efficiency. We demonstrate the feasibility of a spatial form of FPGA application capture that offers portability advantages for FPGA applications unseen with current FPGA binary formats. We demonstrate the portability of such a distribution by developing a fast on-chip emulation framework that performs transparent optimizations, allowing spatially-captured FPGA applications to *immediately* run on FPGA platforms without costly and hard-to-use synthesis/mapping tool flows, and sometimes faster than PC-based execution. We develop several dynamic and transparent optimization techniques, including *just-in-time compilation*, *bytecode acceleration*, and *just-in-time synthesis* that take advantage of a platform's available resources, resulting in

orders of magnitude performance improvement over normal emulation techniques and PC-based execution.

Table of Contents

List of Figures

Chapter 1

Introduction

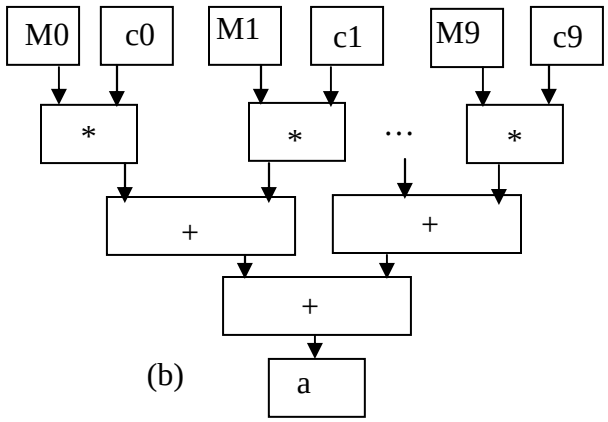
FPGAs (field-programmable gate arrays) support a new form of software whose order of magnitude speedups can enable a class of new high-performance embedded applications not otherwise feasible. However, unlike microprocessor software, two problems severely limit FPGA adoption, and thus prevent the appearance of a range of useful embedded applications. The first problem is that of a highly specialized design process for FPGAs that differs greatly from microprocessor software design, a problem that has been intensively studied by researchers and for which commercial solutions are beginning to appear. The second problem is that FPGA binaries are presently intricately coupled with specific FPGA devices, and cannot be ported across devices or migrated to newer device versions the way that microprocessor binaries can.

FPGAs implement circuits, characterized through their spatial connectivity: component A is connected to B, which is connected to C, etc. Each component computes

Figure 1: FPGAs enable parallel computation. (a) A multiply-accumulate computation, requiring perhaps 30-100 clock cycles on a microprocessor (b) but just 1 or 2 clock cycles on an FPGA.

```
for i in 0 to 9 loop
  a = a + c[i]*M[i]
end loop;
```

(a)



(b)

in *parallel* with all the other components, of which there may be thousands. In contrast, microprocessors implement sequential programs, characterized by their *serial* ordering of computations as a sequence of instructions. It is the parallelism, from the task level down to the bit level, that contributes to FPGAs executing certain computations orders of magnitude faster than microprocessors.

Figure 1 illustrates a computation involving 10 multiplications and additions, which might require 30 to 100 clock cycles to execute on a microprocessor, but could execute in just 1 or 2 clock cycles on an FPGA if enough resources existed. Many embedded system applications are especially amenable to computation speedup from FPGAs. For example, an image processing application may search a camera-provided image for specific objects, such as a tank, a person, or even a specific person's face. Algorithms for such applications may identically search local image portions and then hierarchically compose results – those local search tasks are typically highly-parallelizable, and hence image processing algorithms may execute hundreds of times

faster on an FPGA than on a microprocessor. Likewise, highly parallelizable subtasks exist in other common embedded applications that involve processing streams of video, audio, speech, and other data. Sample domains include television set-top boxes, security cameras, medical imaging and diagnosis equipment, contraband detection systems at locations like airports or borders, fingerprint recognition, speech understanding, and a wide range of military applications. Extensive previous work has shown the speedup advantages of FPGA, typically 10x to 100x, for a wide range of embedded applications [REFS]. Such order-of-magnitude speedups may not just be a change in speed, but rather a "change in kind," as von Neumann originally described the impact of computers over existing desktop calculators, enabling applications not before possible (i.e., the applications enabled by a computer's speed far exceed categorization as that of merely a fast desktop calculator).

One recognized problem preventing FPGA adoption is the different, and more complex, design flow for FPGAs compared to that for microprocessors. The typical design flow requires FPGA users to describe the desired circuit in a hardware description language (HDL), such as VHDL or Verilog, and to use special FPGA-vendor-provided compilers (known as synthesis tools) that convert HDL descriptions to device-specific FPGA binaries. In contrast, microprocessor users utilize "standard" programming languages like C, C++, or Java, and utilize robust high-quality compilers and integrated development environments (IDEs) typically developed by third-party vendors who often specialize in such tools. The massive microprocessor hardware, applications, and tools industries, whose importance need not be stated, have been strongly catalyzed by the

separation of architecture from function enabled by the concept of microprocessor instruction sets, enabling a *standard microprocessor binary*. A standard microprocessor binary is a binary written using instructions of an instruction set, such as an x86 or ARM processor instruction set, that can execute on a variety of existing and evolving versions of a microprocessor, leading to benefits and innovations in the creation of architectures, tools, and applications.

Today, software for FPGAs does *not* benefit from the standard binary concept. Instead, software is compiled by FPGA-vendor-provided tools (typically for free as a means of selling hardware devices), into a proprietary binary that is intricately bound to a specific device. A vendor may offer dozens or hundreds of different devices – Xilinx for example presently supports approximately 100 devices. A binary created for one device cannot run on any other device. The situation hampers development of architecture, software, and tools, and thus the widespread use of FPGAs for embedded computing platforms.

It's natural to ask why industry has not already developed a standard binary concept, if the concept would be so useful. In fact, we do believe that such a concept would eventually begin to evolve over the coming 10-20 year period, with small bitstream portability techniques accumulating into something akin to a standard binary. The lack of portable binaries is becoming recognized as a problem hampering FPGA adoption. For example, an FPGA panel at Supercomputing 2005 noted: "Most applications outlive the hardware. If one is going to invest in an [FPGA] accelerator, what are the options when the accelerator is obsolete? It's a very real issue" [110].

Extensive discussions regularly appear in the newsgroup comp.arch.fpga, and designers have organized to try to make FPGA internal architectures more open (e.g., [109]).

A standard binary concept for FPGAs will certainly incur performance and size overhead compared to the current desktop FPGA CAD approach. Yet, a standard binary concept for FPGAs may catalyze the FPGA hardware, applications, and tools industries, similar to how it catalyzes the microprocessor domain, thus compensating for the incurred overhead. Furthermore, a standard binary for FPGAs that seamlessly integrates with that for microprocessors, may catalyze incorporation of FPGAs into the massive established microprocessor industry, whose hardware and software revenues and number of application developers tower over those for FPGAs by two orders of magnitude. The net result would be the widespread use of FPGAs, especially in embedded systems, whose applications are particularly amenable to FPGA speedup, and hence the appearance of high-performance embedded applications that would otherwise not be developed due to the difficulty of utilizing FPGAs.

We envision opportunities for a portable FPGA distribution format that rides on the success of the "write once, run everywhere" programming paradigm of interpreted languages like Java and C#, wherein a designer captures a design in a high level language, and *any* computing platform that supports a virtual machine for that language can execute that application. At the expense of initial performance, virtual machine technology (like Java's JVM) enables great portability, and is promising as the foundation for a portable FPGA binary. We introduce tools and techniques for an

emulation framework that allows for portable FPGA binary execution which we call *SystemC-on-a-Chip*.

This dissertation can logically be broken into three distinct sections. The first section only includes Chapter 2, and investigates spatial programming and the proper programming constructs and requirements to facilitate a standard FPGA distribution format, and the reasons for choosing SystemC as a possible distribution language. The second section comprises Chapters 3, 4, 5, and 6, and describes tools, frameworks, and experiments to enable the emulation of applications developed in SystemC. Finally, the last section, comprising Chapters 7 and 8, investigate additional uses of the SystemC-on-a-Chip framework.

In Chapter 2, we present an investigation into the proper constructs and languages required to facilitate a portable distribution format for FPGA-based applications. We present a study entitled "C is for Circuits" that closely studied 70 custom-created, clever circuits and attempted to capture those circuits in a sequential language such that a standard C-to-gates synthesis tool could recreate the original custom circuit. Our study complements the question asked by many researchers on whether sequential code (like C) can be analyzed and translated into a high performance circuit. Our study showed that while many custom-created circuits could actually be captured using a sequential language, others could not readily be translated and relied on explicit parallel concepts. Of those that could be translated to a sequential language, many required a radical algorithm change to facilitate synthesis. We thus determined that a portable distribution format would require both *temporal* and *spatial* constructs. We then

investigate the requirements of a language suitable for spatial capture of FPGA applications. We investigate the feasibility of using popular parallel programming frameworks like POSIX, MPI, and RTOS's, but conclude that the SystemC language best captures the temporal and spatial concepts required of a standard FPGA distribution format.

In Chapter 3, we introduce *SystemC-on-a-Chip*, a framework that allows a designer to capture applications in SystemC and have them *immediately* run on any platform that supports the SystemC emulation engine. We introduce the newly developed SystemC bytecode (analogous to Java bytecode), a lean intermediate representation of the SystemC application that preserves both the temporal and spatial features of the application. The SystemC bytecode facilitates a portable representation of the SystemC application that can run any platform assuming SystemC bytecode support. The SystemC bytecode is supported by the *SystemC Emulation Engine.* The SystemC emulation engine can run on any development platform that supports a basic interface of a number of different peripherals, memories, and internal components. The SystemC emulation engine's core is a *SystemC emulation kernel*. The SystemC emulation kernel consists of a lean event-driven kernel, a virtual machine to execute the SystemC bytecode instructions, and hooks and access to the development platform's peripheral set. We demonstrate the usefulness of the SystemC-on-a-Chip framework by developing several complete SystemC-on-a-Chip platforms, highlighting that writing SystemC applications follows the same "Write once, run anywhere" programming paradigm made popular by interpreted languages like Java and C#.

In Chapter 4, we show that for the common case where the SystemC-on-a-Chip is running on an FPGA, we can achieve substantial speedup over a baseline emulation engine by intelligently taking advantage of available FPGA resources. We introduce *SystemC bytecode accelerators*, special coprocessors that can execute the SystemC bytecode natively. SystemC bytecode accelerators are implemented using available FPGA resources, and can be numerous, allowing a SystemC application to effectively to run in parallel (compared to being a parallel simulation). The SystemC bytecode accelerators can improve SystemC emulation execution by approximately 2X. We further demonstrate that the SystemC emulation engine can make intelligent choices about how best to effectively utilize the SystemC bytecode accelerators. We define the *Online Emulation Acceleration* problem and demonstrate that we can achieve 20x improvement over the baseline SystemC emulation engine. With extra available FPGA resources, we also show that we can create custom interconnects among the SystemC bytecode accelerators. Such custom interconnects can effectively *bypass* the SystemC emulation kernel, and result in additional performance improvement.

Unfortunately not all platforms benefit from the resources required to instantiate multiple SystemC bytecode accelerators. In Chapter 5, we address this issue a software-based improvements that *just-in-time* compile the SystemC bytecode to the native processor upon which the SystemC emulation engine is running. Using minimal resources, we modify the SystemC-on-a-Chip framework to be *JIT Aware*, allowing the just-in-time compiled code to execute from resident small, fast memories. Our *JIT Aware* framework includes a JIT Aware Memory, and custom logic for maintaining

emulation signal and event queues. Such modifications result in speedups of approximately 10X compared to a baseline emulation engine, and at near comparable speeds to the same application developed for the native platform.

In Chapter 6, we demonstrate just-in-time synthesis of SystemC applications running on the SystemC-on-a-Chip framework. Just-in-time synthesis is a transparent process(to the SystemC application designer and to the SystemC emulation engine) that synthesizes, place and routes, and maps the original SystemC application to a native implementation that fully takes advantage of the platform's available resources. Just-in-time synthesis of SystemC application results in orders of magnitude speedup of SystemC applications compared to executing natively on the SystemC emulation engine, and several times faster than simulating the SystemC application on a desktop PC.

In Chapter 7, we describe the utility of using the SystemC-on-a-Chip framework for digital physiological model development. We demonstrate *time-controllable* debug features, allowing a physiological model designer to debug using the concept of *time.* This is contrast to traditional debug approaches that require debugging at the instruction level. While instruction level debugging makes sense for traditional sequential programs, *time-level* debugging provides powerful mechanisms to the digital physiological model designer not possible with more traditional approaches.

In Chapter 8, we demonstrate tools and materials useful for teaching a course on spatial programming with SystemC. We develop a freely available Windows-based framework to compile, connect, and download SystemC descriptions to popular teaching

development platforms. We also present possible course materials, including web materials, and course lessons.

We demonstrate the feasibility of using SystemC as a portable distribution language for FPGA applications. We demonstrate the portability of such a portable distribution by developing a fast SystemC emulation framework that transparently optimizes the SystemC application, allowing SystemC applications to *immediately* run without costly and hard-to-use synthesis/mapping tool flows.

Chapter 2

Spatial Algorithms

2.1 Overview

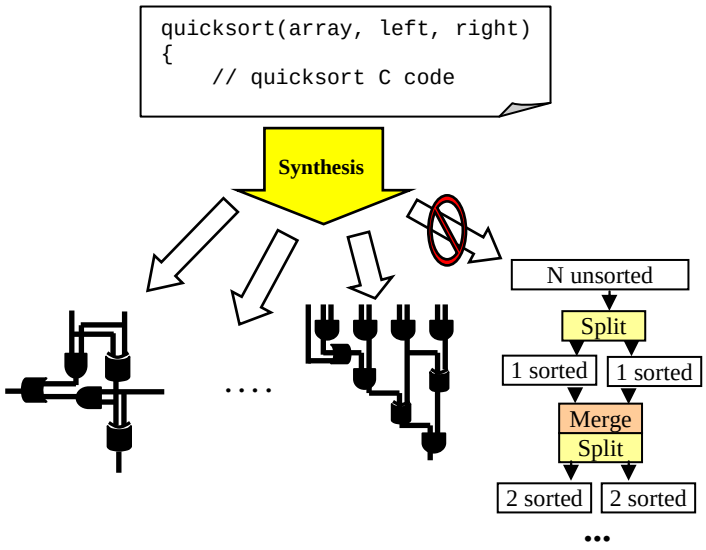
As FPGAs become more common in mainstream general-purpose computing platforms, distributing high-performance implementations of applications on FPGAs will become increasingly important. We present a study entitled *C is for Circuits* that shows that while many manually created circuits can be captured in a sequential language like C for portability purposes, often those implementations would still benefit from explicit parallel concepts. We then investigate the requirements for a language for spatial capture of FPGA applications, and conclude that SystemC satisfies such requirements.

2.2 C is for Circuits

2.2.1 Overview

It is now well-established that many sequential algorithms captured in a language like C can be synthesized to exceptionally fast circuits on field-programmable gates arrays. Numerous FPGA synthesis tools exist [39][49][57][104], with commercial offerings

Figure 2: Although temporally-oriented algorithms in C can be synthesized to a variety of circuits trading off size and performance, many clever circuits representing spatially-oriented algorithms are not reasonably derivable from temporally-oriented algorithms.



beginning to appear [24][25][76], and commercial computing platforms increasingly supporting FPGAs [77][119]. Capturing algorithms in C code (or a similar sequential language, which for simplicity we'll refer to as C code henceforth) provides tremendous portability advantages, as code can be compiled to a microprocessor, or synthesized entirely or partially to FPGAs available on a computing platform. Yet, designers still often conceptualize and capture applications as circuit designs, rather than as C code. While this situation is partly explained by the relatively nascent state of FPGA compilation tools, a significant contributor is also the radically different computation model provided by C than by circuits. The sequential instruction model of C is oriented to time-ordered execution of instructions, while circuits are oriented to spatial connectivity of concurrently-executing components.

In contrast to the advent of compilers causing assembly coding to be almost entirely replaced by C coding, where both coding styles were temporally oriented, the

sharp distinction between temporal and spatial models likely means that spatial models will persist in some form despite continued maturation of C-based FPGA synthesis. Spatial models, such as circuits, possess tremendous degrees of design freedom. Much human ingenuity often underlies the design of both custom circuits and what are known as “hardware algorithms,” which often look radically different from sequential code algorithms designed to solve the same problem. (Because “hardware algorithms” is a misnomer in the era of FPGAs, which implement circuits as software, we use the term “circuit-based algorithms”). Figure 2 shows that while a standard synthesis tool might be able to generate a number of different circuits based on the temporally-oriented Quicksort algorithm, no amount of transformations would be likely to discover a systolic array circuit implementation for fast sorting.

Although circuits represent an important application capture method, capturing applications as circuits suffers from limited portability. A circuit, captured at the netlist level or even at the register-transfer level, cannot readily be adapted to FPGAs differing in their capacities or hard core resources, nor be compiled to execute on a microprocessor. Improved portability could increase the present usefulness of an application across platforms, while also increasing its longevity. In contrast to a circuit, an algorithm captured in C code has much portability. C code can be synthesized to FPGAs of differing capacities and hard core resources, through transformations like loop unrolling and through scheduling, allocation, binding, and technology mapping. C code can even be partitioned among a microprocessor and FPGA, or run on a microprocessor (or several microprocessors) without any FPGAs at all.

We therefore asked the following question:

To what extent can human-designed circuit implementations of an application be captured in a form of C code that can be expected to be synthesized back to the same human-designed circuit?

Note that this question has a subtle but critical difference from most past research that instead seeks to convert an *existing* sequential algorithm to a circuit [39][43][49][64][104][126][129] – research that clearly has much utility. To the best of our knowledge, the above question has not been directly addressed by the codesign or synthesis communities.

Several previous works are related to the question. Stitt [130] provides guidelines for C coders to yield improved circuits. Haubelt [63] formally analyzes a high-level description's flexibility, meaning the extent to which the description can be synthesized to a wide variety of circuits.

Other works are also related. Work on reverse engineering of circuits [40][59] has focused on obtaining low-level behavioral models, like Boolean equations or finite-state machines, for retargeting to different silicon technologies. Those works are not intended for targeting microprocessors. Early hardware/software partitioning work moved non-critical circuit functionality from circuits to microprocessor code [58]. SystemC [46], involving libraries and macros added to C++, allows for temporal and spatial concepts to be captured in a single C++ description.

Of course, circuit designers who use synthesis tools regularly use knowledge of synthesis techniques when writing behavioral (e.g., register-transfer-level) descriptions,

such as writing a for loop that can easily be unrolled. Likewise, parallel architecture programmers write simpler code (e.g., loops) they know compilers will transform to parallel code. The question we seek to answer takes circuit techniques to a higher level, and differs from parallel programming techniques in the finer granularity of parallelism offered by FPGAs compared to more standard parallel architectures.

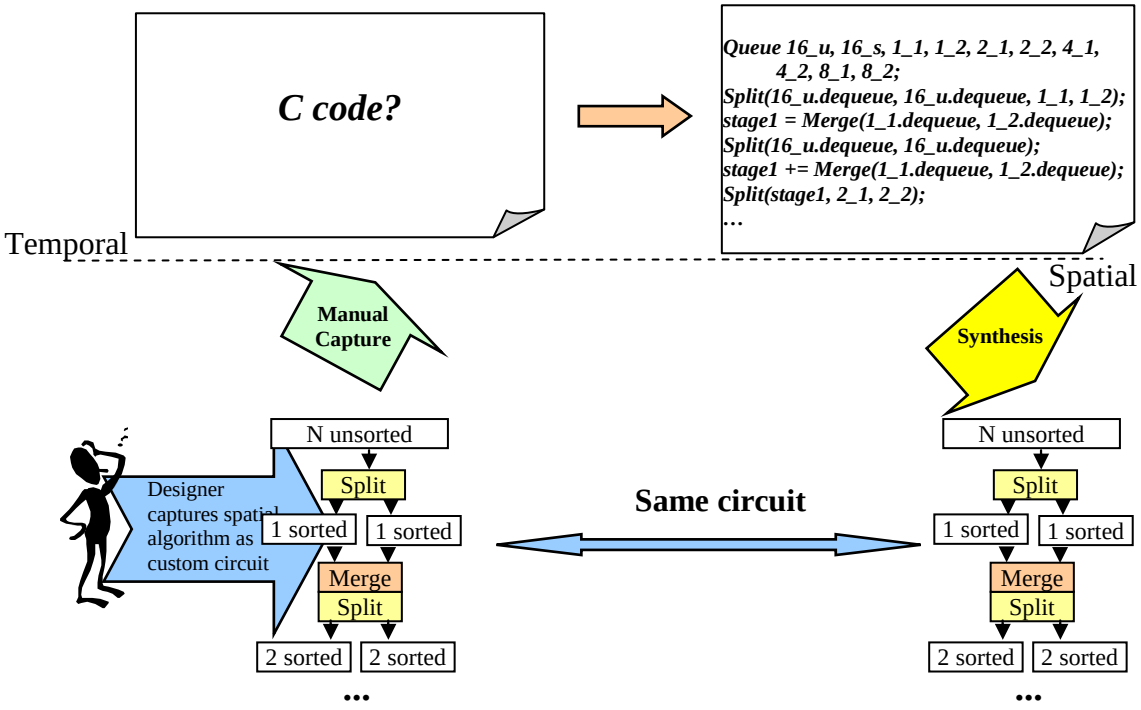
None of the above works explicitly addresses whether existing circuits can be captured in a temporal language. Answering this question is relevant to the FPGA and codesign communities, to determine the extent to which C code can be used to distribute circuit-based algorithms to different compute platforms – algorithms that today are commonly captured and distributed as circuit or register-transfer-level designs.

2.2.2 A Motivating Example – Sorting

There are numerous factors a designer must consider when implementing a sorting algorithm, including data set size, data ordering, and now more recently, the platform on which the algorithm will run.

A software designer targeting a microprocessor platform might use a classic temporal sorting algorithm, such as Quicksort[69], which recursively divides the data into sets greater than and less than a selected pivot. In contrast, a designer targeting an FPGA might approach the problem differently, instead relying on spatial constructs to capture the notion of sorting. The designer might use a systolic Mergesort [154] or Bitonic sort [17], representing highly-parallel, pipelined sorting methods, which cannot reasonably be expected to be derived from a Quicksort algorithm by any FPGA compiler (Figure 2).

Figure 3: C is for circuits: Some circuits might still be captured in a form of C code that is synthesizable back to the original circuit; such C code would provide tremendous portability advantages over other circuit representations



Those methods are radically different than the temporal Quicksort algorithm, even though they accomplish the same task.

Unfortunately, a systolic Mergesort circuit representation is typically not portable, often distributed as a bitstream or at best, some form of netlist. The lack of portability forces distributors to design not only different circuits for different data set sizes, but also for different FPGA sizes and families, which could easily number in the hundreds. Figure 3 illustrates the portability benefits of capturing circuits as C code, showing that if we can capture the systolic Mergesort circuit in *some* form of C code that could be synthesized to the original circuit, we would have a more robust distribution format, capable of being run on a wide range of platforms.

2.2.3 Study Methodology

To investigate whether circuits designed for FPGAs might be captured and synthesized from C code, we examined all papers from six years of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2001-2006), a forum for presentation of clever human-designed circuits for FPGAs (among other topics). We found 70 papers that focused on description of new circuit-based algorithms or clever circuit implementations of standard algorithms for some application. After estimating that each example would require 2-3 days of investigation, we decided to investigate in-depth half of those circuits. We pseudo-randomly chose the subset of 35 circuits to investigate by sorting the 70 circuit papers according to their appearance in the proceedings, starting from oldest to newest. We chose every other paper for investigation – we explain this to make clear that the circuits were not handpicked based on their suitability for C code representation.

We then strove to find C code descriptions for the circuits that would compile back to the same circuit. The goal of the study was to find *any* C description that would compile to the human-designed circuit. Specifically, the claim is not that all functionally equivalent C algorithms would compile to that circuit. Only one is needed, and that one would be used to distribute the circuit-based algorithm. Furthermore, the goal is not to automate the derivation of the C code from the circuit, but merely to determine if a competent designer could capture his/her circuit in C code if necessary.

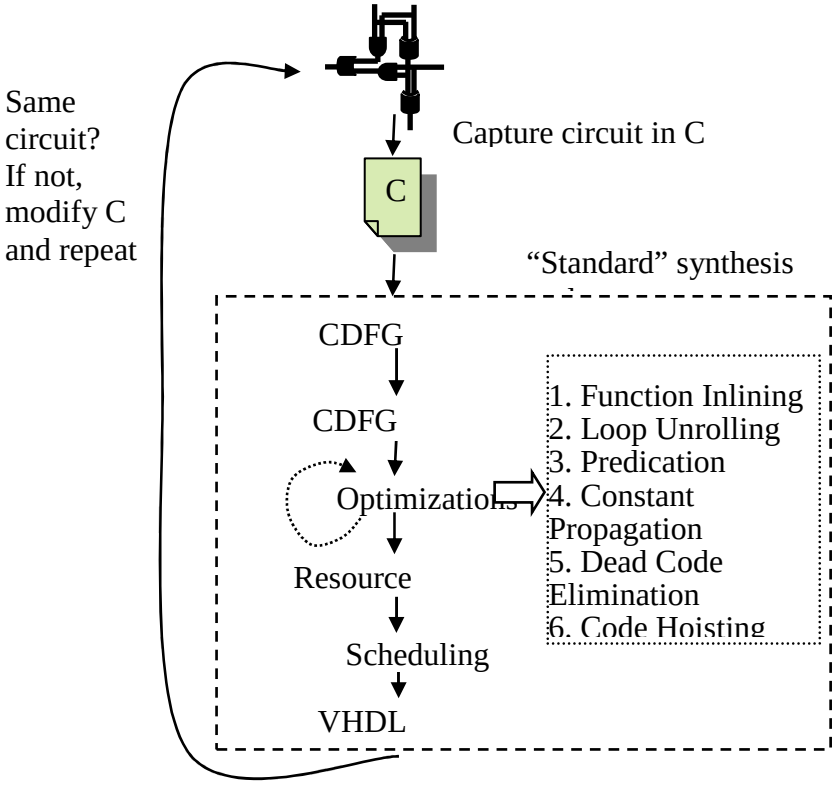
If we were able to capture the circuit in C code that would synthesize back to the same circuit, we classified the circuit as "*re-derivable from C*".

Note that if we failed to classify the circuit as *re-derivable from C*, another C algorithm for the application likely exists that would synthesize to some other circuit with the same functionality, just not the same circuit as the human-designed one. That other circuit would likely have slower performance.

We further sub-categorized the circuits that we found to be *re-derivable from C* as either synthesizable from "*temporally-oriented C"* or "*spatially-oriented C"*. We define "*temporally-oriented C"* as the obvious algorithm that most simply captured the desired behavior of the application (e.g., what we feel is the most "natural" algorithm). If we failed to find such a C algorithm, we next tried to capture the circuit's unique spatial features, through careful use of subroutines and loops, such that a reasonable FPGA synthesis tool should yield the original circuit again. While noting whether circuits were captured in *temporally-oriented* and *spatially-oriented C* was not the main point of the study, the distinction does provide some notion of the effort required by designers to capture their circuit in C code, with spatially-oriented C being harder to write. Furthermore, the distinction also shows the extent of the cleverness of the human-designed circuit, with those derivable from the spatially-oriented C rather than temporally-oriented C likely exhibiting more complex or novel circuit design features.

Because FPGA synthesis tools are still maturing and presently differ widely, we did not simply run the C algorithm through one particular tool. Instead, we defined the transformations and optimizations that could be expected in a mature "standard" synthesis tool. The reader may thus determine for him/herself whether the transformations are "standard" enough to be applied by synthesis tools. To perform synthesis, we followed the methodology shown in Figure 4. If we were able to capture the circuit in C, we converted that C code into a control/data flow graph. We optimized the graph by performing the following optimizations in the order shown: function inlining, loop unrolling, predication, constant propagation, dead code elimination, and

Figure 4: Study methodology. We modeled each circuit in C (when possible). We then performed the following transformations and optimizations in the order shown, representing a "standard" synthesis tool, and observed whether the original circuit was recovered.



code hoisting – straightforward optimizations that could be reasonably implemented in any compilation tool. We performed definition-use analysis to verify that regions of a circuit could be pipelined straightforwardly. We performed resource allocation by allocating a resource for every operation in the dataflow graph. We could have used a more conservative resource allocation, but most of the circuits we investigated were pipelined, and therefore would not allow sharing of resources. We scheduled the graph using resource-constrained list scheduling, inserting registers between each stage of the dataflow graph. Again, we could have used a more conservative pipelining approach to save area, but we were interested in maximizing clock frequency. Next, we converted the scheduled graph into a structural VHDL representation that we then synthesized using Xilinx ISE.

Designers typically define a custom memory interface to best serve the custom circuit, yet our defined standard synthesis tool does not involve synthesis of custom memory interfaces. Since this work concentrates on capturing the compute aspect of custom circuits in C, and not the memory interface, we assume that the synthesis tool is provided with information for each circuit from which the tool can synthesize a custom interface similar to that in the custom circuit. Future work will involve developing mechanisms for providing custom interface information as well as synthesis transformations to generate custom interfaces.

Most of the custom circuits used a standard memory interface consisting of one dual-ported memory, which allows one port for reading and one for writing. This kind of memory interface allows for block transfers and single transfers, similar to many DMAs.

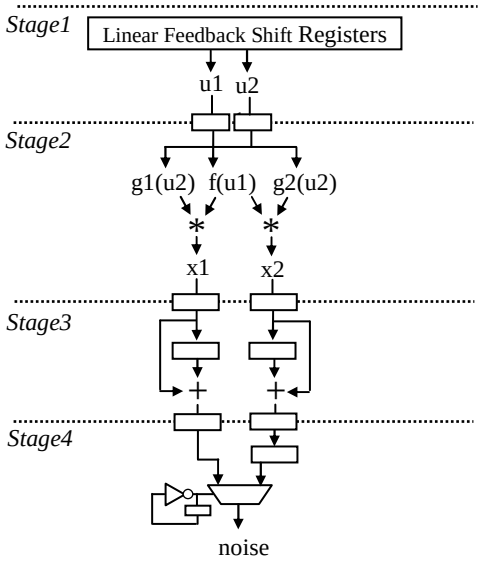
Some circuits implemented streaming data from off-chip memories, while others did not use external memory.

For each example, we targeted the specific FPGA used for each of the custom circuits in their original papers. Although we could have compared both the original circuit and synthesized circuit on newer FPGA fabrics, we felt such comparison might be unfair if the custom circuits were designed based on the characteristics of the original FPGA fabric.

2.2.4 Example – Gaussian Noise Generator

Figure 5 shows the custom circuit in [88] for a Gaussian noise generator. The circuit consists of four pipeline stages. The first stage utilizes linear feedback shift registers (LFSRs) to generate a 32-bit and 18-bit random number, corresponding to *u1* and *u2*. Stage 2 uses the random numbers from the previous step as input to the illustrated functions, which consist of square root, sine, cosine, and log functions. Stage 3 adds every two consecutive results from stage 2. The circuit implements this functionality by delaying one input for a cycle using a register and then adding the output of the register with the output from the previous stage. This buffering results in a delay to the pipeline, potentially causing an output to be generated every 2 cycles. Stage 4 multiplexes the results from stage 3 to the output of the noise generator. By adding a register to the right input of the multiplexor, the circuit generates an output every cycle, instead of two outputs every two cycles.

Figure 5: Circuit for a Gaussian noise generator.



We first tried to determine if the circuit was *re-derivable from temporally-oriented C*. The natural temporal C uses a loop that executes the behavior of stages 1 and 2 twice to generate two samples for the accumulate step in stage 3. FPGA synthesis tools would replicate the circuit used in each iteration of the loop, increasing the area of the circuit without improving performance. We next tried to determine if the circuit was *re-derivable from spatially-oriented C*. Figure 6 shows a portion of the C code to model the Gaussian noise generator circuit in Figure 5. The C code utilizes a single function to describe each pipeline stage of the custom circuit. The output is stored into the array *noise[]*. To handle outputting to an array, we modified the code for stage 4 to store the two noise samples to two memory locations, as opposed to multiplexing the output to a single location. As we will show, this code is synthesized to the same stage 4 circuit shown in Figure 5. For simplicity, the C code uses floating point arithmetic as opposed to the fixed-point arithmetic in the custom circuit. The fixed-point code is similar, with the main difference being that the code uses logical *and* operations to remove unused bits of

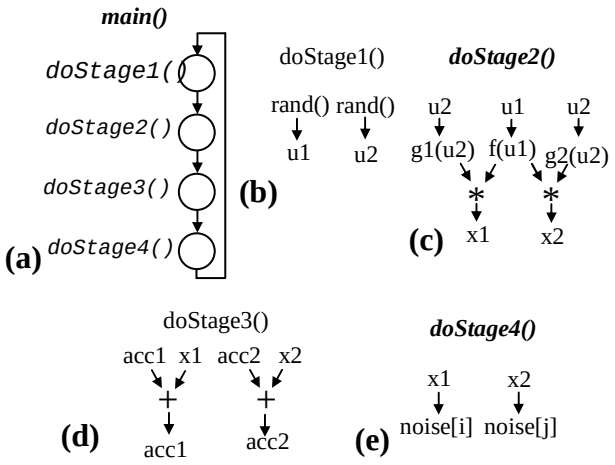
Figure 6: Spatial C code for Gaussian noise generator.

```
inline float rand0_1() {
  return rand()/((float) RAND_MAX+1);
}

inline Stage1 doStage1() {
  Stage1 result;
  result.u1 = rand0_1();
  result.u2 = rand0_1();
  return result;
}

inline Stage2 doStage2( float u1, float u2 ) {

  Stage2 result;
  float f_u1, g1_u2, g2_u2;

  f_u1  = sqrt( -log( u1 ) );
  g1_u2 = sin( 2*M_PI*u2 );
  g2_u2 = cos( 2*M_PI*u2 );
  result.x1 = f_u1*g1_u2;
  result.x2 = f_u1*g2_u2;
  return result;
}

inline Stage3 doStage3( float x1, float x2 ) {

  static float acc1=0.0, acc2=0.0;
  Stage3 result;

  result.x1 = acc1 + x1;
  result.x2 = acc2 + x2;
  acc1 = x1;
  acc2 = x2;
  return result;
}

inline void doStage4( int i, int j,
                float x1, float x2 ) {

   noise[i] = stage3.x1;
   noise[j] = stage3.x2;
}

int main() {

  Stage1 stage1; Stage2 stage2; Stage3 stage3;
  unsigned int i=0;

  while (1) {
      stage1 = doStage1();
      stage2 = doStage2( stage1.u1, stage1.u2 );
      stage3 = doStage3( stage2.x1, stage2.x2 );
      doStage4( i, i+1%NUM_SAMPLES,
                stage3.x1, stage3.x2 );
      i = (i+2)%NUM_SAMPLES;
  }

  return 0;
}
```

the random numbers, essentially specifying the width of each number to be 32 bits for *u1* and 18 bits for *u2*.

The control and data flow graphs generated during synthesis for each function of the C code are shown in Figure 7. Figure 7(a) shows the control flow graph for *main()*,

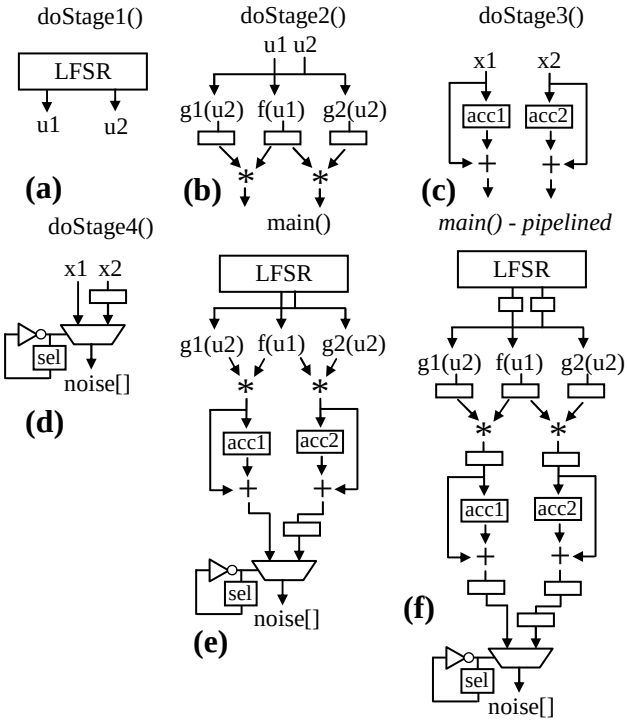
Figure 7: Control/data flow graph for C-level Gaussian noise generator functions (a) main, (b) doStage1, (c) doStage2, (d) doStage3, and (e) doStage4.



where each function call has a corresponding node in the graph. For simplicity, we have omitted the control flow node for the code used to update the variable *i*. Figure 7(b) shows the data flow graph for function *doStage1()*. We omit the control flow graph for this function, and all other functions, because the corresponding graphs consist of only a single node. The data flow graph for stage 1 assigns random numbers to the two outputs of the function. Although not shown, the data flow graph also contains operations to constrain the random numbers to values between 0 and 1. Figure 7(c) and Figure 7(d) show the data flow graphs for the *doStage2()* and *doStage3()* functions. The data flow graph for *doStage4()*, shown in Figure 7(e), produces two outputs instead of the single output from Figure 5.

Figure 8 shows the circuits for each data flow graph for each C function after synthesis performs scheduling, resource allocation, and binding. For stage 1, shown in Figure 8(a), synthesis maps the random number generators to LFSRs. Figure 8(b) shows the circuit for stage 2, for which synthesis utilizes approximation techniques to map the

Figure 8: Datapaths after scheduling, resource allocation, and binding for (a) doStage1, (b) doStage2, (c) doStage3, (d) doStage3, (e) main before pipelining, and (f) main after pipelining. Note the similarity with Figure 5.



functions in stage 2 onto the same resources used to approximate these functions in the custom design. Unlike in the custom circuit, scheduling during synthesis is likely to insert registers between the approximation circuits and the multipliers in order to reduce the critical path length. For stage 3, shown in Figure 8(c), synthesis maps *acc1* and *acc2* onto registers because the outputs from this stage are used again as inputs. Stage 4, shown in Figure 8(d), multiplexes the two outputs from the data flow graph for this stage. Synthesis adds the multiplexor because the outputs from the data flow graph are written to memory, which in this case is a shared resource with only a single port. To allow both inputs to be written to memory, synthesis delays input *x2* one cycle using a register while the circuit stores *x1*.

To optimize the circuit, synthesis can inline all of the functions for each stage into the main function and then perform code hoisting to move the code for each stage into a single control flow node, which is possible since there exists no control in each function. The resulting data flow graph for this single control node is shown in Figure 8(e). During scheduling, synthesis will insert a register at each level of the data flow graph, as shown in Figure 8(f). Note the similarity of the circuit in Figure 8(f) with the custom circuit shown in Figure 5. The only difference in the synthesized circuit is the addition of registers before the multipliers – an addition that may actually improve performance compared to the custom circuit.

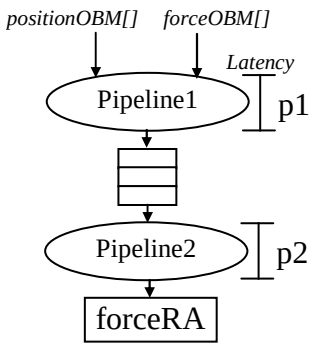
The throughput of the synthesized circuit is identical to the custom circuit, with each circuit producing a noise sample each cycle. The latency of each pipeline is different, but this latency only determines when the initial output from the circuit is valid. We point out that under certain situations, the two circuits are likely to differ in other ways. For example, if the target architecture utilizes a dual-ported memory or a memory with sufficient bandwidth to simultaneously store two results, then stage 4 of the synthesized circuit will not contain the multiplexor or buffer register. This architectural difference does not affect throughput, but does affect timing, resulting in two noise samples every two cycles. To our knowledge, synthesis cannot guarantee the same timing as the custom circuit due to the lack of timing information in the C code. However, the timing difference after synthesis does not appear to be critical.

Thus, we classify this circuit as re-derivable from (spatially-oriented) C.

Figure 9: Molecular dynamics accelerator. (a) Code for calculating nonbonded forces. (b) Custom circuit utilizing a divided pipeline to reduce latency penalty. (c) The synthesized pipeline differs from the custom circuit by utilizing a single pipeline. The synthesized circuit must stall due to a single memory, reducing throughput.

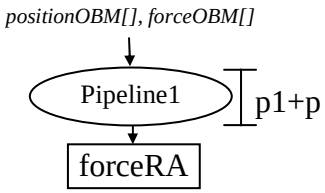
```
foreach atom i do
  ri = positionOBM[i]
  fi = forceOBM[i]
  n = 0
  foreach neighbor j of i do
    if |ri – rj| < rc then
      rj = positionOBM[j]
      fij = calcNBF( ri, rj )
      fi = fi + fij
      fj = forceOBM[j]
      forceRAM[n] = fj – fij
      n = n+1
    end
  end
  forceOBM[I] = fi

  foreach fj in forceRAM do
    forceOBM[j] = fj
  end
end
```

(a)



Max throughput:
1 output per cycle
Latency penalty: p2

(b)



Max throughput:
1 output every 2 cycles

Latency penalty: p1+p2

(c)

2.2.5 Example – Molecular Dynamics Simulator

Scrofano [118] creates a custom FPGA accelerator for molecular dynamics simulations. The authors identify the nonbonded-forces calculations as the most time consuming region of the simulation and provide a custom circuit for those calculations.

Figure 9(a) shows the pseudocode implemented by the custom circuit. For each atom, the inner loop calculates the forces from each neighbor of the atom. The code stores the forces in the array *forceRAM*, which the following loop stores into the *forceOBM* array.

Figure 9(b) shows a high-level view of the custom circuit for the inner loop. Scrofano utilizes two separate on-board memories (OBM) to store the *positionOBM* array and the *forceOBM* array. Utilizing two memories allows the circuit to simultaneously stream position and force data without stalling, therefore achieving a maximum throughput of one force calculation per cycle. Scrofano implements the *forceRAM* array in on-chip memory to minimize the amount of read/write mode switches that would be

required if the forces were stored back immediately to the *forceOBM* array. To optimize the datapath, the authors divided the pipeline into two pipelines separated by a FIFO. Dividing the pipeline reduced the latency penalty that was incurred every time the inner loop executed. The first pipeline generates output faster than the second pipeline and therefore only the latency of the second pipeline has a significant effect on performance.

If we used C code based on the pseudocode in Figure 9(a) to try and model the custom circuit, the inner loop becomes a fully pipelined circuit that streams in the force and position data. Synthesis maps the *forceRAM* array onto block RAMs, which is possible due to the small size of the array, resulting in a single pipeline that performs the same operations as the two pipelines in the custom circuit. To our knowledge, there is presently no common synthesis technique that automatically divides a pipeline as is done in the custom circuit. Such a technique may be possible, requiring analysis to best determine the placement and size of the buffer. By using a single pipeline, the synthesized circuit incurs a larger latency penalty each time the inner loop executes, as shown in Figure 9(c). The designer might instead direct the FPGA synthesis tool by altering the C code in Figure 9(a) to model the buffer that separates the two pipelines. This might be accomplished by inserting a function call to enqueue the intermediate result of the first pipeline and dequeuing a result to the input of the second pipeline. Of course, this relies on a model of a buffer the FPGA compiler can recognize. By modeling the spatial constructs of the circuit, an FPGA tool would be able to effectively recover the original circuit.

Another important difference when using the temporally-oriented code in Figure 9(a) is that the synthesized circuit uses a single memory for input. When synthesizing code to a specific architecture, the synthesis tool must use the appropriate memory architecture, which we assume to be a single off-chip memory. Therefore, the synthesized circuit must read the position and force arrays from a single memory, which does not provide sufficient bandwidth to execute the pipeline without stalls. Therefore, the synthesized circuit has a lower throughput, outputting a force calculation every two cycles. If enough on-chip RAM existed to store both arrays, or the synthesis tool could stream data into two on-chip memories fast enough, then the synthesized circuit could perform similarly to the designer-specified circuit.

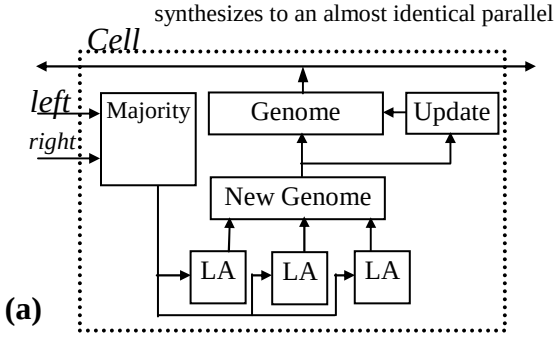
Thus, we classify the molecular dynamics circuit as *re-derivable from (spatially-oriented) C.*

2.2.6 Example - Cellular Learning Automata-Based Evolutionary Computing

In [62], Hariri et al. proposed a custom architecture for cellular learning automata based evolutionary computing (CLA-EC). This architecture consists of a ring of cells, each of which stores a genome. The architecture for each cell is shown in

Figure 10(a). Each cell consists of multiple learning automata (LA) that determine a new genome. The update circuit replaces the existing genome with the new genome if the fitness value of the new genome is better. The majority function uses the genome of

Figure 10: The proposed custom CLA-EC circuit consisting of a ring of (a) custom CLA-EC cells and (b) C pseudocode that synthesizes to an almost identical parallel circuit (code for cell internals is omitted).




```
Cell cells[MAX_CELLS];
int main() {
  for (i=0; i < MAX_CELLS; i++)
     generateNewGenome(i);
  updateGenomes();
  return 0;
}
void updateGenomes(){
  for(i=0; i<MAX_CELLS; i++)
    cells[i].genome = cells[i].newGenome;
}
```

(b)

the left and right neighbor cells to generate reinforcement signals that guide the learning automata.

An abbreviated version of the C code we used to model the CLA-EC is shown in Figure 10(b). This code iterates over some maximum possible number of cells, which is based on the input size. For each cell, *generateNewGenome()* implements the behavior of the majority function, learning automata, and the update function. The *generateNewGenome()* function updates the new genome if the new genome is better, otherwise the function sets new genome equal to the old genome. Because *generateNewGenome()* only modifies a single cell, the loop containing the *generateNewGenome()* function has no loop-carried dependencies, allowing synthesis to parallelize the function calls by performing function inlining, loop unrolling, predication, and code hoisting.

After the *generateNewGenome()* loop completes, *updateGenomes()* updates the genome for each cell with the new genome determined by the *generateNewGenome()* function calls. By modifying the genome of each cell, the *updateGenomes()* function

creates a dependency with the *generateNewGenome()* function, which uses the genome as input. To handle this dependency, synthesis stores the genome in a register. The resulting circuit is almost identical to the custom circuit. The only difference is the addition of a multiplexor before the new genome register that either selects the output of the learning automata or the output of the genome register.

The simplicity of the C code in Figure 10(b) suggests that this implementation may also be the most natural way of writing the application in C. We classify the cellular automata circuit as readily *re-derivable from (temporally-oriented) C.*

2.2.7 More Experiments

We described several examples from the FCCM literature and our attempts to capture those designs in some form of standard C code. We now briefly highlight several other randomly selected examples before summarizing results for the entire examined set.

Tripp [138] designed a circuit to implement a large metropolitan traffic simulation (*Road Traffic*). Each cell computed car velocities and positions based on a specific rule set imposed by the designers which reflected real world traffic conditions. When we focused on the computational aspect of each cell in the network, we found the traffic design to be readily *derivable from (temporally-oriented) C.*

Bogdonav [19] designed a systolic array structure to solve matrix calculations using Gaussian elimination (*Elimination*). The authors in fact modified a temporally-oriented algorithm to achieve their circuit design. We also found the circuit to be *re-derivable from C* code. We decided to model the Gaussian elimination calculation with

Figure 11: 82% of the studied circuits published in FCCM were re-derivable from C, meaning they could be captured in some form of C such that a synthesis tool could be expected to synthesize the same or similar custom design.

Year of Publication	Design	Re-derivable from C?	Method/Reason
2001	3D Vec. Normalization	Yes	Spatial, if online algorithms can be specified
2001	Efficient CAM	No	Uses dynamic FPGA routing
2001	Automated Sensor	Yes	Temporal, floating point -> fixed point
2001	Regular Expression	Yes	Spatial, creative connections of one-bit flip flops
2002	Hyperspectral Image	Yes	Spatial, data reordering
2002	Machine Vision	Yes	Spatial, custom pipelining
2002	RC4	Yes	Temporal, straightforward implementation
2002	Set Covering	Yes	Spatial, data structures for easy hw implementation
2002	Template Matching	Yes	Spatial, heavy modifications to original algorithm
2002	Triangle Mesh	Yes	Spatial, custom encoding scheme
2003	Congruential Sieves	Yes	Temporal, straightforward translation
2003	Content Scanning	Yes	Temporal
2003	F.P and Square	Yes	Spatial
2003	Gaussian Noise	Yes	Spatial, requires the use of spatial C constructs
2003	TRNG	No	Requires sampling a high frequency clock for noise
2004	3D FDTD Method	Yes	Spatial
2004	Deep Packet Filter	No	Requires knowledge of underlying FPGA
2004	Online Floating Point	No	Online algorithm, variable length buffers
2004	Molecular Dynamics	Yes	Spatial
2004	Pattern Matching	Yes	Spatial
2004	Seismic Migration	Yes	Spatial
2004	Software Deceleration	No	Use a uP for its cache
2004	V.M Window	No	Specific timing schemes implemented
2005	Data Mining	Yes	Spatial
2005	Cell Automata	Yes	Temporal
2005	Particle Graphics	Yes	Spatial
2005	Radiosity	Yes	Temporal
2005	Transient Waves	Yes	Spatial
2005	Road Traffic	Yes	Temporal
2006	All Pairs Shortest Path	Yes	Spatial
2006	Apriori Data Mining	Yes	Spatial
2006	Molecular Dynamics	Yes	Spatial, define separate memories, custom pipeline
2006	Gaussian Elimination	Yes	Spatial
2006	Radiation Dose	Yes	Temporal
2006	Random Variates	Yes	Spatial
Totals:		**82% of the circuits were re-derivable from C**	

spatially-oriented C code to ensure synthesis transformations would recover the original systolic array structure.

Krueger [86] designed a floating point unit to add two streaming numbers. The design incorporated variable delays, which we were not able to capture in either temporal or spatial C. We classified their design as *not re-derivable from C*. We again point out that there do exist C algorithms for this application that would synthesize to *some* circuit – just not to the particular published circuit.

Figure 11 summarizes all the designs studied. As described earlier, we identified 70 custom circuit designs published in the last six years of the IEEE Symposium on Field-Programmable Custom Computing Machines, of which we chose every other circuit to study in depth, totaling 35 custom circuit designs. Of the 35 designs, 29 of the designs, or 82%, were found to be *re-derivable from C*. Of the 29 circuits *re-derivable from C*, 9 of those, or 25% of all 35 circuits, were captured in *temporally-oriented C*. Again, this means these designs could have been written in natural high level code, and we could have reasonably expected a synthesis tool to recover the circuit, without much human effort at the circuit level. We note that a benefit of being able to capture the circuit as *temporally-oriented C* is that if the platform on which the circuit runs happens to be a microprocessor, the code may be able to run at or near its best performance, because the algorithm may be the same algorithm one would have written if initially targeting a microprocessor.

20 of the circuits, or 57%, were re-derivable from C were captured in *spatially-oriented C* code. There were several common reasons why a design had to be described in *spatially-oriented C* as opposed to the more natural *temporally-oriented* algorithm. Custom circuit designs often employed a combination of spatial techniques, including intricate pipelining, custom buffering, advanced memory hierarchies, and systolic array connectivity, none of which could reasonably be re-derived from the standard synthesis techniques.

For 17% of the circuits, we were unable to capture the circuit in any form of C code that would be synthesized back to that circuit. James-Roxby et. al [80] proposed

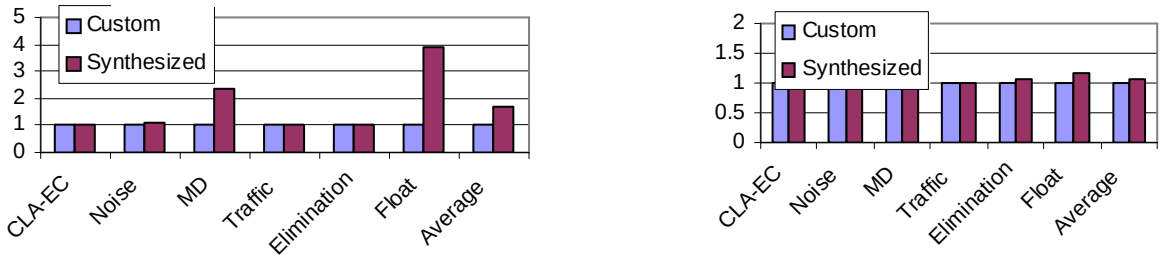
logic-centric systems in which they added microprocessors to the design to make effective use of the cache hierarchy, a technique not reasonably describable using standard C constructs. Several circuits [86][150] utilized low level cores that made re-deriving from C difficult. Others [145] implemented circuits that relied on precise timing, which is also difficult to capture in C. One circuit [81] took advantage of the dynamic reconfigurability of the FPGA to implement dynamic routing, a technique clearly not supported by standard C constructs.

In summary, 82% of the circuit designs published in a forum for circuit-based algorithms could be captured in some form of standard C such that a synthesis tool supporting a basic set of transformations could recover the circuit from that C code.

Figure 12(a) compares the performance of the custom-designed circuits and the circuits synthesized from the C code for several of the examined circuits. All performances are normalized to the performance of the custom-designed circuits. For each example shown, the performance of the synthesized circuit was either identical to the custom circuit or slightly slower than the custom circuit. Had we modeled the molecular dynamics circuit with the original temporal pseudocode shown in Figure 9(a), the synthesized circuit would have been 2.3x slower. This performance decrease would have been caused by the inability of synthesis to split a pipeline into smaller pipelines that communicate using FIFOs. By modeling the molecular dynamics circuit with custom *spatially-oriented C* code, synthesis is able to generate a nearly identical circuit.

Figure 12(b) compares the area, in slices, of the synthesized circuits and the custom circuits. On average, the synthesized circuits required only *6%* more slices. This

Figure 12: Comparison of original custom circuits versus circuits synthesized from derived sequential code representations: (a) Normalized xecution time and (b) Normalized area (slices) Both metrics are normalized to values for stom circuit.



extra area was used by multiplexors and other glue logic that synthesis was unable to remove, and by additional pipeline registers.

advantage of describing a circuit in C is that the C can be distributed to different platforms having different amounts of FPGAs, and an FPGA synthesis tool could thus allocate more or less resources for the application without requiring a designer to distribute a new circuit. In this section, we estimate the changes in performance for each application when being implemented on both a smaller and larger FPGA than the ones used in the previous section.

A larger FPGA for the Gaussian noise generator would not improve the performance of calculating a single noise sample, but would allow for more samples to be generated per cycle by replicating the circuit several times. While the ability to replicate a circuit is not unique to writing the circuit in C, it certainly makes the task easier. Alternatively, a larger FPGA could be used to improve the accuracy of the approximation circuits.

For the molecular dynamics simulator, a larger FPGA could potentially eliminate the memory bottlenecks of the synthesized design. If a large portion of the input could be

stored in on-chip memory, then synthesis could create the same, or even an improved memory architecture compared to the custom circuit. Increased on-chip memory could provide sufficient bandwidth to read multiple positions and forces, improving the throughput of the pipeline to several force calculations per cycle.

For a larger FPGA, *CLA-EC* potentially would achieve significant performance improvements compared to software, due to the ability to implement more cells on the same device. In [86], the authors show an approximately linear speedup compared to software when increasing the number of cells. Based on their results, an FPGA with twice the capacity would result in an approximate 2x speedup. Alternatively, a larger FPGA for *CLA-EC* would allow the circuit to determine an improved result for a given run time.

For the Gaussian Elimination circuit, a larger FPGA would not improve the performance of the custom circuit for existing matrix sizes. However, a larger FPGA would enable circuits for larger matrices, improving performance by at least 2x for a matrix that would not fit in a smaller FPGA.

Similarly, a larger FPGA size for the metropolitan traffic simulation *would* enable simulations of larger road networks.

For the online floating point unit, additional resources would not improve performance because the parallelism of the hardware is limited by non-constant bounded loops that cannot be unrolled.

For smaller FPGAs, the C code for each application could be synthesized by the FPGA to use fewer resources. In fact, every example except the Gaussian noise generator could be implemented with a datapath consisting of only a multiplier, an adder, a register

file, and a corresponding amount of steering logic. The performance of these smaller circuits would be slower than the pipelined implementations of the original circuits, but the C representation would still provide a correct implementation. For the Gaussian noise generator, the C representation would synthesize to a circuit as long as the FPGA had enough resources to implement the sine, cosine, square root, and log functions.

Furthermore, every example could be implemented entirely on a microprocessor, at the obvious cost of slowdown. We leave examining the extent of that slowdown, and partitioning among microprocessor and FPGA, for future work. However, because 25% of the examined circuits could be captured in *temporally-oriented C* code, the microprocessor performance of these captured circuits is likely comparable to corresponding software-oriented implementations, since these implementations are likely to be similar.

2.3 Other Related Work

2.3.1 C-based Synthesis Tools

There is a growing community that seeks to convert existing sequential algorithms into structures suitable for FPGA implementation. Numerous FPGA synthesis tools exist, with several commercial offerings beginning to appear. Most offerings extend the C language with parallel constructs or compiler-specific pragmas that aid in exposing parallelism and pipelining opportunities. Other efforts [106][143] automatically attempt to extract as much parallelism and pipelining opportunities, while still allowing the original C code to compile for a traditional CPU.

2.3.2 Parallel Languages

There are a number of models of computation and circuit capture methods. Brown [21] shows that a parallel model of computation requires machine primitive units, control constructs, communication mechanisms, and synchronization mechanisms. Circuits are usually captured in a hardware description language (HDL) like Verilog [141], or VHDL [142], although circuits can also be captured using schematics.

2.3.3 Portability

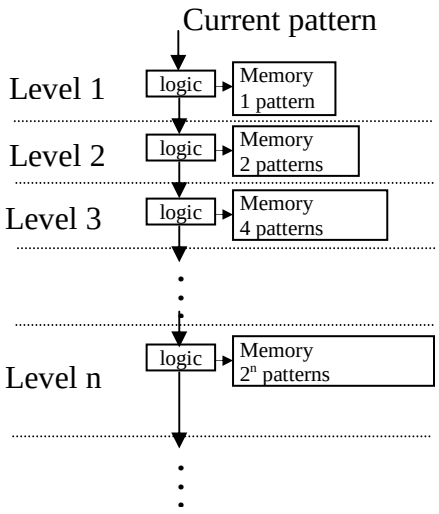
There has been previous work in capturing applications and circuits to increase portability. Andrews and Anderson [3][4] focus on creating operating system and middleware abstractions that extend across the hardware/software boundary, enabling a designer to create applications for hybrid platforms with one executable. Levine [91] describes hybrid architectures with a single, transformable executable. They argue that an executable described for a queue machine (converse of a stack machine) makes runtime optimizations to a specialized FPGA fabric feasible. Moore [100] describes writing applications that dynamically bind at runtime to reconfigurable hardware for the purposes of portability. Similar to Andrews and Andersons, the authors develop hardware/software abstractions by writing middleware layers that allow application software to utilize reconfigurable DSP cores. Vuletic [146] proposes a system-level virtualization layer and a hardware-agnostic programming paradigm to hide platform details from the application

designer and lead to more portable circuit applications. Lysecky and Stitt [93][131] showed that a temporally-based binary could potentially be used as part of a standard FPGA binary approach. They introduce *Warp Processors*. Warp processors can dynamically profile, extract, and synthesize computationally expensive temporal kernels into fast FPGA circuits. Their approach makes FPGA tool flows completely transparent, and result in application speedups up to 10X, and energy savings of up 80%.

2.4 Requirements of a Language for Spatial Capture

C is for Circuits demonstrated that sequential languages possess many constructs that would form part of a viable distribution format for FPGA applications. In some cases, the sequential programming constructs (sequential instructions, function calls, etc) were sufficient to capture an FPGA application. In many other cases, the sequential programming model was limited, forcing awkward implementations, or at worst not being able to capture the same behavior. For the FPGA applications that were difficult or impossible to capture using only a sequential programming model, we identified several programming constructs that would have made such implementations feasible, or simpler to capture. One requirement is the ability to spatially connect two components together through the use of a specified interface. Another requirement is the ability to control precise timing synchronization between spatially connected components. The third requirement is that the language should be able to be executed on both a microprocessor and an FPGA. Such a language requirement will include sequential constructs found in a

Figure 13: Pipelined Binary Tree [94]. Each level operates concurrently, taking the pattern and address information from the previous level, and passing information to the next level. Such a design cannot readily be captured in a sequential language, and requires explicit parallel constructs to capture for portable distribution



language like C, with the addition of spatial and timing constructs found in explicitly parallel languages like VHDL and Verilog.

For illustrative purposes, we use the pipelined binary tree, developed by Lysecky [94] to guide decisions on which parallel programming model best suits the constructs required for such a portable distribution format. Figure 13 shows an n-level pipelined binary tree, a high throughput circuit for the pattern counting problem. Target patterns are stored in the tree in breadth-first order. The first level (root) contains only one pattern, the second level contains two patterns, the third four patterns, the fourth eight patterns, and so on. Each level consists of control logic and a memory to store the patterns, and another memory of the same size (not shown in the figure) to maintain pattern counts. Each level operates concurrently, taking information from the previous level, and sending information to the next level.

Level 1 receives the current pattern and compares with the target pattern. If equal, level 1's logic increments the count associated with that target pattern. If less, the logic passes the pattern to level 2, informing level 2 to look in its left node (because in a binary tree, if the pattern is less than the root, then search proceeds down the left sub-tree) – in particular, by telling level 2 to look at address 0. If greater, level 1 tells level 2 to look in address 1. Level 2 then compares the pattern with the target pattern located in the address it received from level 1 (while level 1 meanwhile processes the next incoming pattern). If equal, level 2's logic increments the count associated with that target pattern. If less, level 2 appends a 0 to the address, so if the address was 0, the new address is 00; if it was 1, the new address is 10. If greater, level 2 appends a 1 to the address, yielding either 01 or 11. Subsequent levels operate similarly, either incrementing their count, or appending 0 or 1 to the address as they pass the address to the next level. The pipelined binary tree is unique in the sense that it's an explicitly *parallel* algorithm which dedicated interconnections, precise timing, and that which cannot readily be captured in a sequential language like C for distribution purposes.

2.4.1 POSIX

One popular method for implementing parallel based applications is to use the POSIX-based approach. POSIX is a thread-based library targeting C-based languages that allows a designer to capture parallel programs with a predefined set of library function calls to create, spawn, and join parallel computations (processes) together. POSIX-based programming represents a possible method for capturing FPGA-based applications

Figure 14: Snippet of POSIX-based implementation of one level of the pipelined binary tree and how levels are connected and how they communicate.

```
unsigned char level1_pattern;
unsigned char level1_address;
unsigned char level1_enable;

unsigned char level2_pattern;
unsigned char level2_address;
unsigned char level2_enable;

sem_t  timestep_done,computeLevel1Done;
sem_t level1_pattern, level1_address, level1_enable;
sem_t level2_pattern, level2_address, level2_enable;


void * ClockTick( void * arg ) {
  while(1){
    sem_wait(&computeLevel0Done);
    sem_wait(&computeLevel1Done);
    sem_wait(&computeLevel2Done);
    sem_post(&timestep_done);
  }
}

int main(){
  pthread_t timeStepFunction;
  pthread_t computelevel0;
  pthread_t computelevel1;
  pthread_t computelevel2;
  …
  pthread_create(&computelevel0);
  pthread_create(&computelevel1);
  pthread_create(&computelevel2);
  pthread_create(&timeStepFunction);

  pthread_join(computelevel0, NULL);
  pthread_join(computelevel1, NULL);
  pthread_join(computelevel2, NULL);
  pthread_join(timeStepFunction, NULL);

  return 0;
}
```

```
void * computeLevel1( void * arg ) {
  static TPM[2];
  TPM[0] = 10;
  TPM[1] = 20;

  while (1) {
    sem_wait(&timestep_done);
    sem_wait(&level1_pattern);
    sem_wait(&level1_address);
    sem_wait(&level1_enable);

    //actual behavior of level1
    level2_pattern = level1_pattern;
    if(level1_pattern == TPM[level1_address]){
      level2_address = (level1_address << 1) | 1;
    }
    else{
      level2_address = (level1_address << 1) | 0;
    }

    if(level1_pattern == TPM[level1_address]){
      level2_enable = 0;
    }
    else{
      level2_enable = 1;
    }

    sem_post(&level2_pattern);
    sem_post(&level2_address);
    sem_post(&level2_enable);
    sem_post(&computeLevel1Done);
  }
}
```

because the combination of parallel constructs (from POSIX) and the sequential constructs (from the C-based language) seem to match the requirements needed of many FPGA based applications.

Figure 14 shows the earlier described pipelined binary tree implemented using a POSIX-based approach. The implementation works, but suffers from several disadvantages. Because the pipelined binary tree benefits from precisely timed communication in which every level is speaking with the next level every cycle,

modeling such behavior using a thread-based approach is difficult, hard to read, and difficult to extend. As shown in the figure, the designer must explicitly create a new thread that manages global time. Also, the POSIX design relies on using a complicated set of locks and semaphores to guard the global memory space from being incorrectly written to and/or read from. Whereas one hallmark trait of FPGA-based circuits is the precisely-timed communication between concurrently executing components, POSIX-based approaches typically benefit most coarse communication mechanisms, and begin to suffer both in performance and robustness as the implementation tries to capture finer grained communication granularity.

2.4.2 Other Thread-Based Approaches

There are other thread-based approaches we considered as a possible portable distribution format for FPGA-based applications. The Message Passing Interface (MPI) [97] represents one such approach. In contrast to a POSIX-based approach which uses shared memory to communicate among concurrently executing components, concurrently executing in MPI-based applications pass explicit messages to each other, both synchronously and asynchronously. MPI-based approaches work well for large distributed systems, but still don't match the precisely timed communication model often seen of FPGA applications.

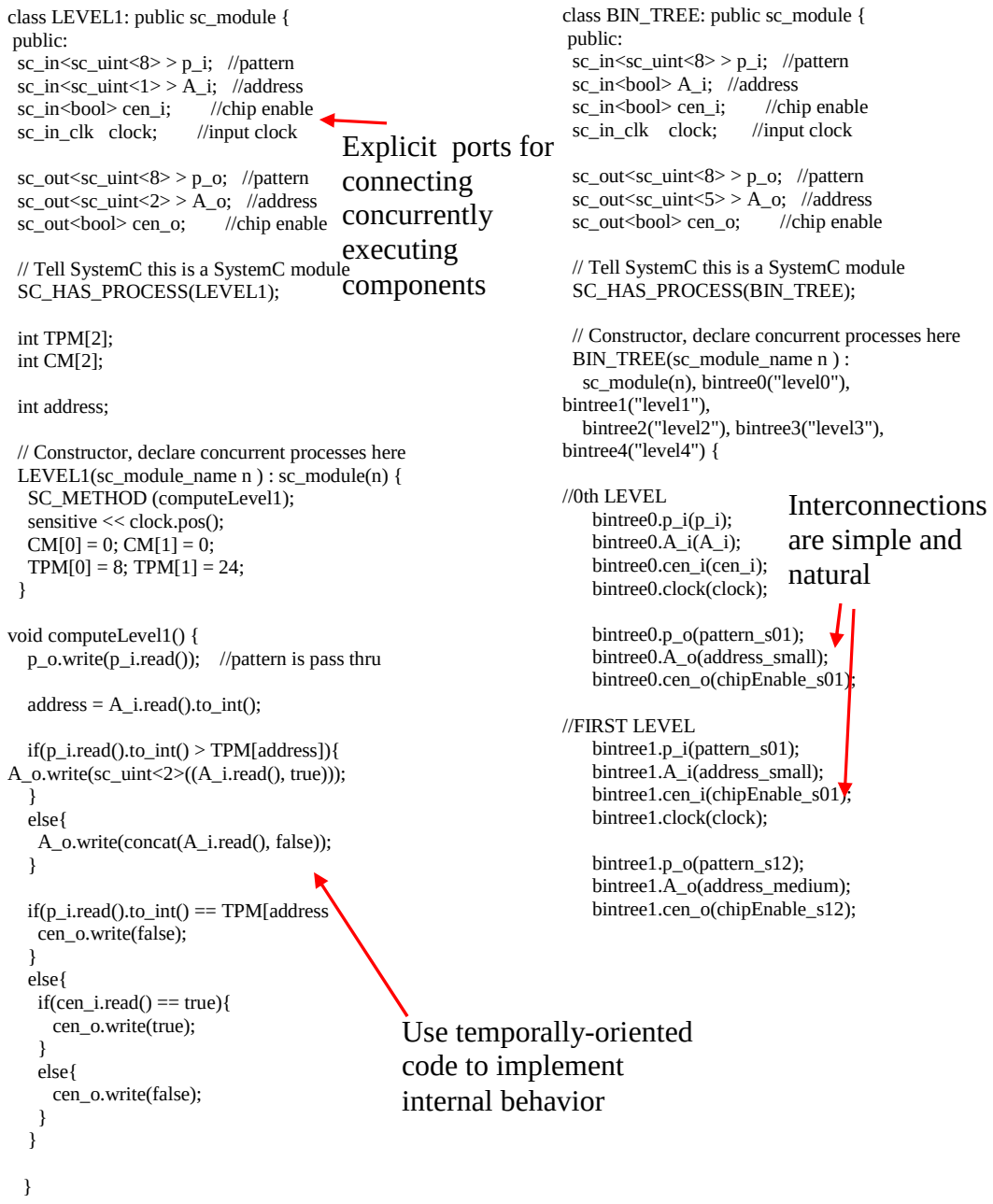
Real-time operating system (RTOS's) represent a finer grained approach, allowing the user to time at some granularity the period at which parallel processes should execute, but still fall shy of the precisely-timed communication required of many

FPGA applications. For instance, while an RTOS might allow the designer to specify that a set of processes execute every millisecond, such granularity is often insufficient, and there is usually no guarantee as to the ordering of the execution of the processes, which could lead to incorrect behavior.

2.4.3 SystemC

Another method for capturing FPGA applications is to use SystemC. SystemC is a set of libraries that seeks to bridge the gap between HDLs and the standard programming language C++, by achieving HDL functionality using C++ objects, thus enabling a designer to describe a complete system, including both sequential program behavior and circuit behavior, in a single language environment. Figure 15 shows the same pipelined binary tree captured using SystemC. The SystemC method is attractive, allowing a designer to capture concurrently executing components using well known C++ practices (class creation, templates, etc) while still allowing for precisely timed communication because each component be “clocked” by a global clock that manages simulated time, and need not be explicitly introduced into the design.

Figure 15: Snippet of SystemC implementation of a level of the pipelined binary tree and how multiple levels are connected.



We have chosen to use SystemC as distribution format for several reasons. SystemC allows for the spatial connection of concurrently executing components, the ability to precisely time the communication between multiple components, and the ability

describe the behavior of components using temporally-oriented constructs. Additionally, the SystemC libraries are freely available and becoming more widely adopted.

Chapter 3

SystemC-on-a-Chip Framework

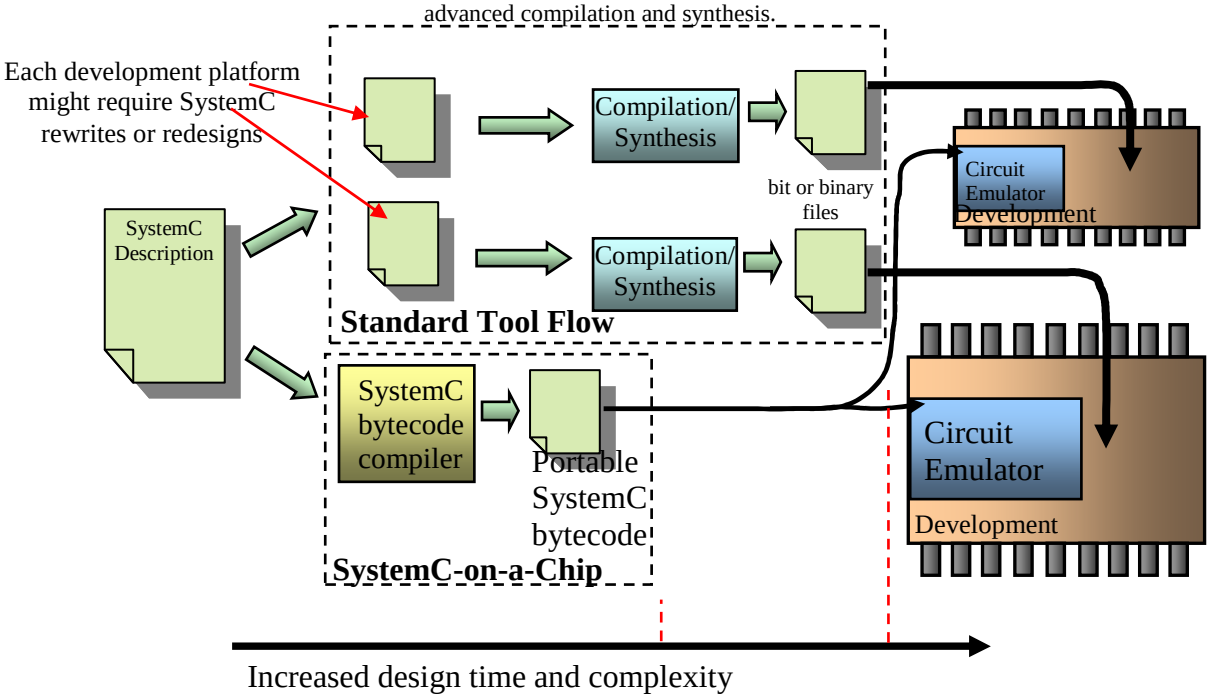
3.1 Overview

SystemC [133] represents a digital system description approach based on C++. SystemC uses object-oriented features of C++ to enable descriptions that include features common in previous hardware description languages (HDLs), such as creation of components, instantiation and connection of components to form a circuit, and precisely-timed communication and execution among concurrently-executing components, all using existing C++ syntax. Regular C++ code can be included in descriptions, and SystemC also provides a thread library, thus supporting description of both the "software" (sequential instructions coupled with parallel threads) and "hardware" (circuit) parts of an entire system in a single description language.

While a SystemC description can be executed on a PC for simulation purposes before eventually synthesizing the description to an ASIC, FPGA, or board-level customized implementation, in-system SystemC emulation, wherein the executing description would interact with physical inputs and outputs (I/O), would also be useful.

In-system emulation is common for embedded processors. Though slower than a custom implementation, emulation enables early prototyping, and benefits from real I/O rather than fabricated I/O in simulation, whose creation can be difficult and time-consuming while still not matching the complexity and nuances of real I/O. Emulation can be especially useful for SystemC, as illustrated in Figure 16, due to the fact that synthesis tools can be expensive (compared to compilers), may only run on limited PC platforms and be challenging to install (especially on lower-end PCs), may be unpredictable with respect to circuit size/speed or tool runtime, often require long runtimes (such as hours or days), may not support particular target devices or platforms, and can only synthesize the parts of the code written for synthesis. The main tradeoff is

Figure 16: SystemC-on-a-Chip allows a designer to emulate SystemC descriptions on various supported development platforms. Emulation enables early prototyping and interaction with real peripherals and I/O, while reducing the need for advanced compilation and synthesis.



that emulation is typically much slower than native platform execution. Another tradeoff is that the emulation engine must be present on a target platform, but this is a one-time task, which may be done by the platform's developers or by platform users (such as teaching assistants in an educational setting).

For education, where system execution speed may not be a top priority, emulation may be entirely sufficient, such as when describing a microprocessor system as is commonly done in computer architecture courses, where such descriptions may never be intended for synthesis, but execution on a physical platform is desired. In fact, for some systems (in education settings or otherwise), emulation may be fast enough to serve as a final implementation, obviating the need for synthesis, akin to virtual machines sometimes being sufficient for executing processor bytecode such as Java bytecode. For example, a "human reaction timer" system may involve several interacting components interfacing with buttons, LEDs, and LCDs, with emulation speed being fast enough to interact with all these items. In such cases, SystemC ultimately represents a parallel programming approach such as an approach using POSIX threads, with the added benefit of supporting circuit-style spatial connectivity, but the drawback of not (presently) supporting real-time scheduling as in a real-time operating system approach.

We introduce an approach to SystemC emulation, involving several parts. We created a compiler to convert SystemC to a new bytecode format that we developed, which possesses MIPS-like instructions supplemented with *new* SystemC-specific instructions that convey spatial and timing information. We developed an emulation engine that can run on a microprocessor on a development platform and that executes the

SystemC bytecode while interacting with I/O and (optional) peripherals (frame buffers, UART, etc.). Because portability is important in the approach, we introduce a USB flash-drive method for programming, wherein the compiler-generated textual bytecode file is copied to a USB flash-drive, which is then read by the development platform and just-in-time translated to the machine-level bytecode used by the emulation engine. For the common situation where the emulation engine is implemented on (or with access to) an FPGA, we developed FPGA-based custom emulation accelerators that substantially increase the emulation speed, enabling SystemC execution speeds comparable to middle-to-high-end PCs.

3.2 Related Work

There has been research in the field of hardware emulation for verification and testing, including the BEE reconfigurable platform [27], and network-on-chip emulation platforms [52]. Nakamura [105] describes a hardware/software verification platform that uses shared register communication between a processor simulator and FPGA emulator. Benini [15] describes virtual in-circuit emulation of SystemC circuits for co-verification and timing accurate prototyping. Rissa [116] evaluates the emulation speeds of several SystemC models compared to standard HDL models.

Much research has involved virtualization [92][124], with several commercial products developed in response to the need for portable virtual machines. VMware [147] and the open source product Xen [153] concentrate on developing virtual machines that allow the end-user to run multiple operating systems concurrently. The Java Virtual

Machine [127] allows the programmer to write operating system independent code, and tools like DOS Box and console emulators allow the user to run legacy applications in modern operating systems. Fornaciari [47] extends virtualization to FPGA platforms, giving the application designer a virtual view of an FPGA that is then physically mapped via operating system functionality. Virtualization has also been used to abstract complex microcontroller details from the beginning embedded systems student [123].

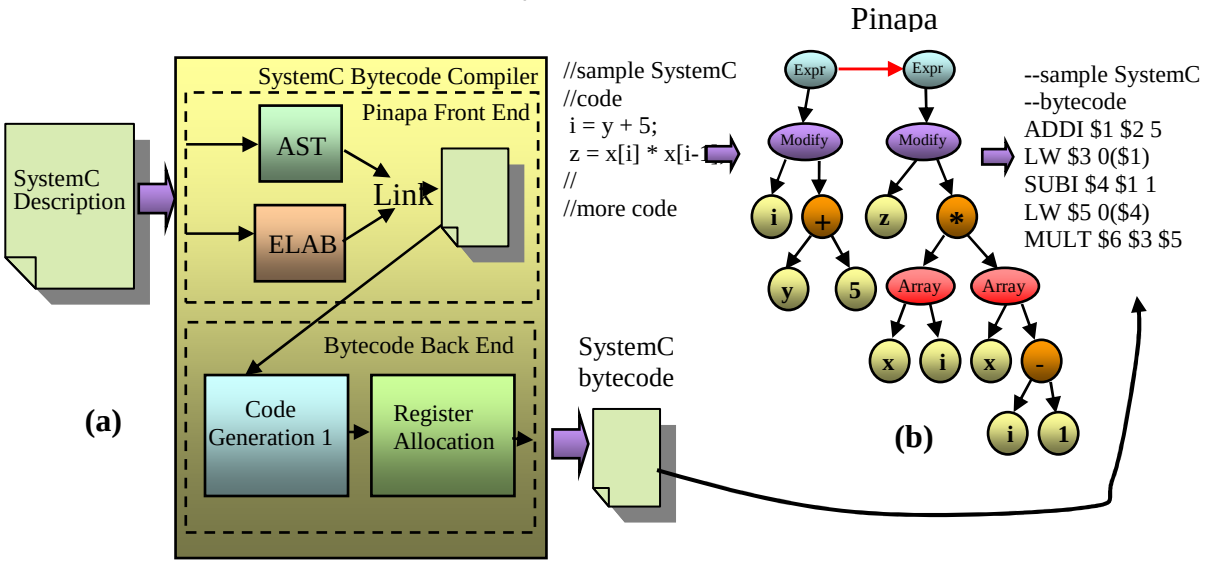
3.3 SystemC-on-a-Chip Components

The SystemC-on-a-Chip platform consists of four main parts, including a SystemC bytecode compiler, a new intermediate SystemC bytecode format, a portable USB flash drive download interface, and an emulation engine.

3.3.1 SystemC Bytecode Compiler

We considered several options to achieve in-system emulation of SystemC descriptions. One approach was to port the publicly available SystemC libraries to each development platform, and add support for I/O and peripheral interaction. Such an approach would allow the same SystemC binary to run on any supported development platform, including standard PCs. Also, the SystemC circuit would run natively on the development platform's microprocessor. However, the SystemC libraries are large and require OS support, thus limiting the number of platforms that could support the SystemC-on-a-Chip framework. Furthermore, the SystemC libraries build a simulation kernel into the circuit

Figure 17: SystemC bytecode compiler: (a) The SystemC bytecode compiler builds on PINAPA, a SystemC front-end tool, and uses a custom SystemC bytecode backend; (b) Sample code generation during the first phase of the SystemC bytecode back end.



executable, increasing the size of the executable and making testing multiple SystemC descriptions quickly more difficult.

Another option was to decompile the SystemC executable, extract the circuit, and retarget that circuit for a custom emulation framework. The decompilation approach separates the circuit from the simulation kernel, allows testing multiple circuits quickly, and potentially a smaller circuit executable. A custom emulation framework also allows smaller development platforms to take advantage of in-system SystemC emulation. However, decompilation is difficult, and solutions that operate at the source SystemC level seemed more feasible.

The option that we chose was to directly operate from SystemC source code to produce bytecode, as shown in Figure 17. Our SystemC compiler builds upon the *PINAPA* tool [102]. Originally intended as a front-end for circuit verification tools,

PINAPA provides a *gcc* compiler front-end to SystemC circuits that extracts a circuit's spatial and architectural features from the SystemC description.

The PINAPA front-end performs two operations on the SystemC program. PINAPA uses a modified version of the *gcc* compiler to extract behavioral information about each process and component in the circuit to generate the corresponding abstract syntax trees (AST), and uses a modified SystemC kernel to extract the circuit's architectural features, like ports, signals, and spatial connectivity. Finally, PINAPA links the architectural description (ELAB) to each component's AST to form the intermediate output.

We created a custom two-pass back-end to the PINAPA compiler that accepts PINAPA's AST+ELAB output and generates SystemC bytecode. The first pass traverses each ELAB component's AST. The first pass inlines auxiliary functions, flattens hierarchical descriptions, and generates initial SystemC bytecode assuming an infinite amount of available registers, shown in Figure 17(b). The second pass performs a linear scan register allocation [114] on the first pass output to constrain the intermediate code to a fixed number of registers. The output of the register allocation pass is a readable text file of the SystemC description in SystemC bytecode.

3.3.2 SystemC Bytecode Format

The SystemC-on-a-Chip platform accepts a bytecode version of the SystemC description, and not a traditional SystemC binary, nor the SystemC source code. A traditional SystemC binary includes much more information than is actually required to emulate the

application, including constructs to support object-oriented C++ programming, and the simulation kernel. SystemC source code separates the circuit from the simulation kernel, but requires compiler support on each development platform. Similar to Java bytecode and a Java Virtual Machine, an intermediate SystemC bytecode format separates the SystemC description behavior from the simulation kernel, doesn't require a platform compiler, and can run on any development platform that supports the SystemC bytecode format.

The format of the SystemC bytecode is shown in Figure 18. The SystemC bytecode is a flattened version of the original SystemC description. The SystemC bytecode compiler flattens the SystemC description to more efficiently emulate the SystemC bytecode. A SystemC *circuit* is composed of a list of signals and a list of processes. A *signal* is a wire or set of wires that connects independently running processes, and is defined by a signal name and bit width. A *process* is a behavioral description of a circuit entity. A process is defined by a *sensitivity list,* a list of signals the process is sensitive to, and a list of sequential instructions which define the process's behavior.

Figure 18: SystemC bytecode format. Each process is described by a number of MIPS-like instructions, with additional instructions added for SystemC specifics, like reading signals, extracting bit ranges, etc.

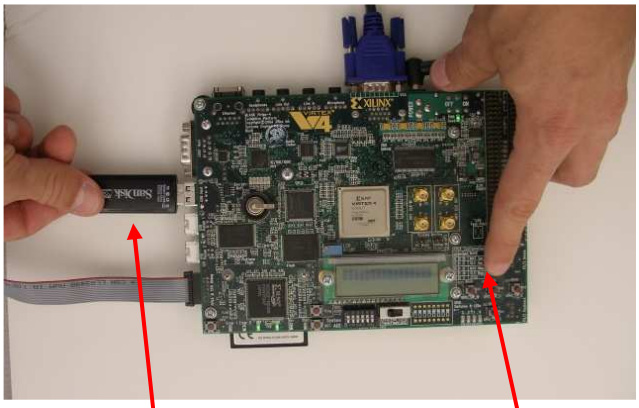
circuit: signals processes
signals: signal or
signals signal
processes : process or
processes process
signal : SIGNAL NAME COLON NUMBER
process : PROCESS sensitivity_list code
sensitivity_list: NAME or
sensitivity_list NAME
COMMA
code: instruction or
code instruction
instruction:
SRL *or* SLL *or* SLLV *or* SRLV
or MULT *or* MFLO *or* ADD
or SUB *or* AND *or* OR *or* ADDI
or ANDI *or* ORI *or* XORI
or SUBI or LW *or* SW
} Computation instructions

or J *or* JR *or* BEQ *or* BNE
or BLE *or* BGT *or* BLT *or* BGE
} Control instructions

or BIT *or* RANGE *or* READ
or WRITE *or* CONCAT *or* WAIT
or END
} SystemC-specific instructions

A process is captured as a sequence of sequential instructions. The SystemC bytecode instructions are a derivative subset of the MIPS RISC register machine instruction set [67], shown in the bottom half of Figure 18. We also considered targeting virtual stack or queue machines. The Java Virtual Machine [127] executes bytecode instructions intended for a stack machine, and [91] executes bytecode instructions for a queue machine. Proponents of stack and queue based bytecode formats argue that the stack/queue bytecode can more efficiently run on a virtual machine because the operands are implied. Other studies [37] have shown that the advantages of stack machines aren't

Figure 19: USB interface. The user copies SystemC bytecode to a USB flash drive, plugs the drive into a platform and pushes a button—the platform then begins emulating the SystemC description.



as clear. The authors show the bytecode targeted towards a register machine can be competitive with stack machine code, and usually results in more compact code. An additional advantage is that register bytecode is more readable, potentially allowing a student to write bytecode in the absence of a SystemC bytecode compiler.

SystemC bytecode format supports three different types of instructions: computation/memory instructions, control instructions, and SystemC-specific instructions. The computation and control instructions are derived from the MIPS instruction set [67]. We chose the RISC MIPS instruction set because the SystemC bytecode is easy to generate, because a RISC-based emulator can be efficient [37], and because the code is understandable to the beginning student. We also chose a representative subset of the MIPS instructions that would allow specifying all circuits described in the synthesizable subset of SystemC[134].

We added a number of SystemC-specific instructions to the base MIPS instruction set, including the *BIT, RANGE, READ, WRITE, and WAIT* instructions. The *BIT* and *RANGE* instructions extract either one bit or a range of bits from a given register. The *READ* and *WRITE* instructions allow a process to read and write signals, much as the process can load or store values to memory. We added the SystemC-specific instructions to more efficiently execute frequently occurring SystemC primitives and function calls. Most of the SystemC-specific instructions could have been implemented as a sequence of the basic computation instructions except for the WAIT instruction. The *WAIT* instruction allows a SystemC description to wait a fixed number of simulated time steps. The WAIT statement is the only supported feature that does not follow the synthesizable SystemC guidelines, but allows designers to test their SystemC applications with custom bytecode test benches. The *END* instruction instructs the emulation engine that a process is done executing.

3.3.3 USB Download Interface

Our SystemC-on-a-Chip platform supports USB programming via a USB flash drive, rather than a traditional hardware programmer or USB cable. A traditional hardware programmer requires non-volatile memory and a removable chip, greatly limiting the number of supportable development platforms. An alternative programming approach is to program a device in-system using a USB cable. While eliminating the need for a programming device, such an approach still requires a PC every time a designer wishes to load a new SystemC description.

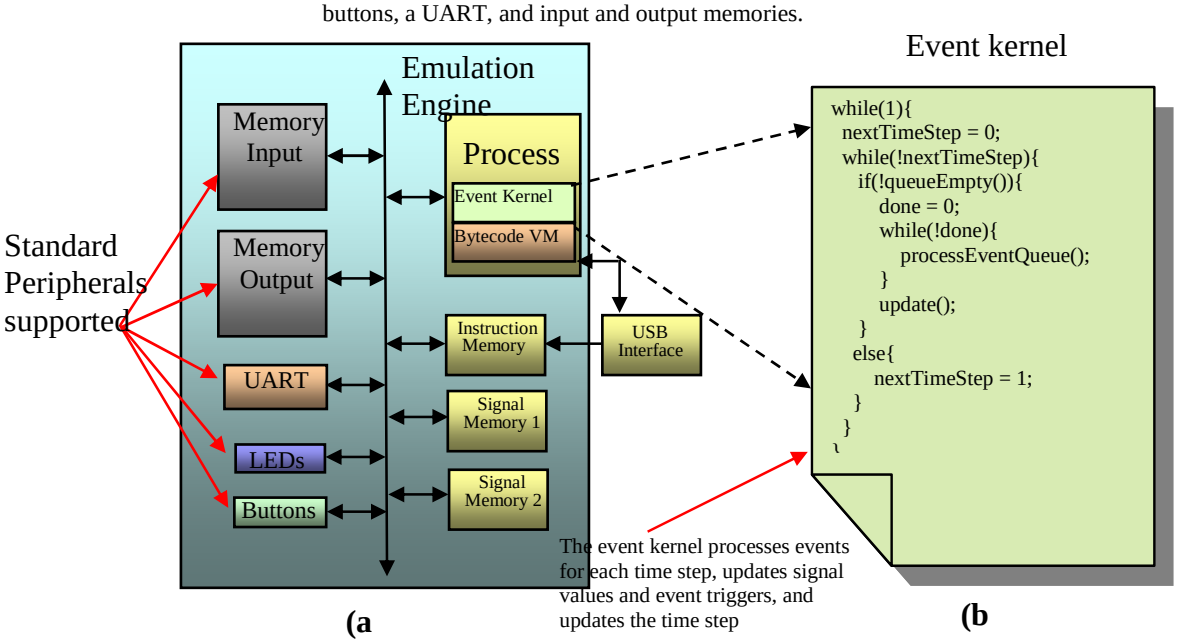
Instead, we chose a USB flash drive programming approach, illustrated in Figure 19. A user (such as a student) copies the desired SystemC description (in bytecode format) onto a USB drive as a file, plugs the drive into the SystemC-on-a-Chip platform, and presses a button on the platform that downloads the program from the flash drive to the internal emulation engine. The approach eliminates the need for non-volatile memory in the development platform. The approach enables loading and changing circuits by inserting and swapping flash drives, enabling more mobility and portability. The approach also matches current usage schemes for popular electronic devices, allowing a beginning student to start programming with minimal effort, and using a familiar paradigm. The cost is that the SystemC-on-a-Chip platform must contain an internal USB flash drive reader.

3.3.4 SystemC Emulation Engine

The basic emulation engine supports SystemC bytecode written or generated for the synthesizable subset of SystemC. We currently do not support higher level features of SystemC like transaction level and system level modeling because we are presently targeting SystemC descriptions that could eventually run natively on an FPGA. Figure 20(a) shows the architecture of the basic emulation engine.

The basic emulation is driven by a processing core that runs a lean, event-driven simulation kernel [48]. Figure 20(b) shows the pseudocode for the event-driven kernel. For each time step, the event-driven kernel processes a queue of ready-to-run *events*. An *event* is placed on the queue when a signal value is updated and that signal is on the

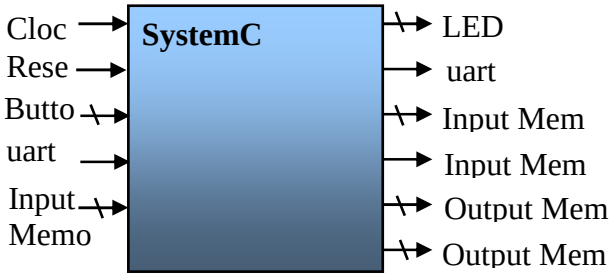
Figure 20: Basic emulation engine. The emulation engine consists of a hybrid event-driven kernel to allow a variety of different circuit implementations. Circuits can also take advantage of a range of standard peripherals, including lights, buttons, a UART, and input and output memories.



sensitivity list of a process. Each time step might consist of multiple *delta* time steps, in which a process may execute multiple times during a time step. After each delta step, the event kernel updates the signal values, and places any new sensitive processes onto the event queue.

The signal's values are located on the system bus in *Signal Memory 1* and *Signal Memory 2*. Processes and peripherals write to *Signal Memory 1*, and read from *Signal Memory 2*. After each delta step, the event kernel copies the contents of *Signal Memory 1* to *Signal Memory 2*. The advantage of putting the signal memories on the bus is that peripherals have easy access to the signal values, and gives access to emulation accelerators. The disadvantage is that multiple peripherals might try to access the signal memories at the same time as the event kernel, blocking the bus, and degrading emulation efficiency.

Figure 21: SystemC-on-a-Chip circuit interface. The emulation engine supports access to multiple peripherals, including buttons, LEDs, and memory.



The event-driven kernel calls a bytecode virtual machine to execute each event in the event queue. The *bytecode virtual machine* supports the SystemC bytecode instruction set described in the previous sections. Each process is allocated an instruction memory, register file, and local data memory. The virtual machine also contains proper hooks to communicate with the standard peripheral and I/O set. We designed the bytecode virtual machine using standard techniques from [124] to increase the efficiency of execution.

The emulation engine supports platform I/O and peripheral access. The set of peripherals includes buttons, LEDs, UART, and input and output memories. We chose the peripherals to be a representative subset of peripherals that most development platforms could support. For development platforms with a larger set of peripherals, emulation designers could easily add extra support. The basic emulation engine supports SystemC descriptions that implement the interface shown in Figure 21. The description writer does not have to follow the standard interface, but the standard interface provides a convenient mapping between description's signals and the available peripherals.

3.4 Experiments

We built two complete SystemC-on-a-Chip platforms, and implemented dozens of SystemC descriptions to demonstrate the applicability of in-system SystemC emulation. The systems we built are summarized in Figure 22. One platform used the Virtex4 Ml403 FPGA development board, and the other used a Spartan 3E FPGA development board. On the Virtex4 ML403 FPGA, we built the emulation engine on a PowerPC processor and used the PLB bus framework to access I/O and peripherals. On the Spartan 3E FPGA, we built the emulation engine on a Microblaze soft-core processor, using the OPB bus framework to access peripherals and I/O. The instruction memory, stack, and heap for the PowerPC based basic emulation engine were all stored in SRAM. In contrast, the instruction memory, stack, and heap for the Microblaze-based system were all located in on-chip BRAM. Due to limited BRAM resources, some SystemC descriptions would not run on the Microblaze-based platform. No SRAM existed on the Spartan 3E platform.

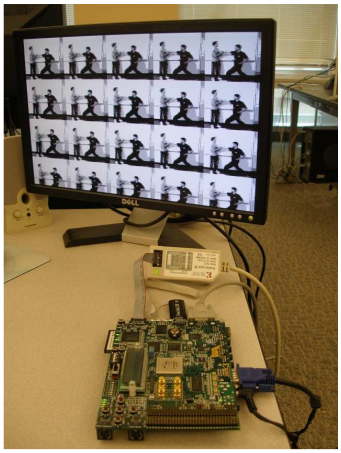
Figure 22: SystemC-on-a-Chip prototypes. Each system differed in size, processor, memory, and number of emulation accelerators, but each could run the same SystemC bytecode for a given SystemC description.

Development Platform	Main Processor	Bus Platform	Memory Location	# of emulation accelerators
Xilinx Virtex4 Ml403 FPGA	PowerPC	PLB	SRAM	2
Xilinx Spartan3E FPGA	Microblaze	OPB	BRAM	1

Figure 23: SystemC experiments. (a) SystemC code for Image Edge Detection. The code took only minutes to create and compile before being put on a Virtex4. (b) Edge Detection running on a Virtex4. We connected the memory output to a frame buffer to see the results on a VGA screen.

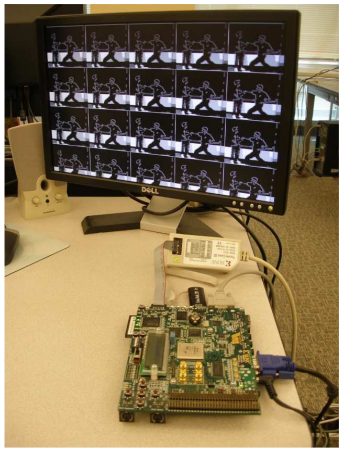
```
class EDGE_DETECTOR : public sc_module {
//signal declarations
…
EDGE_DETECTOR() {
   SC_method(mainComp);
   sensitive << dataReady;

   SC_method(getPixel);
   sensitive << clock.pos();

void getPixel(){
   …
   dataReady.write(1);
}

void mainComp(){
    int i, j;
    for(i = 0; i < 3; i++){
        for(j = 0; j < 3; j++){
   sumX = sumX +   mem.read()*GX[i][j]
        }
    }
```

(a) SystemC Snippet



Before

(b)



During

We implemented a number of different circuits in SystemC, including an edge detector, encryption/decryption applications, various state machines, and several smaller combinational logic components to exercise the entire SystemC bytecode set. We implemented the edge detector with two communicating processes in about 200 lines of SystemC. The encryption/decryption units required about 300 lines of SystemC, and consisted of five processes. One of the combinational components, a structural implementation of a 32-bit adder, required 500 lines of SystemC and consisted of 66 processes. The SystemC bytecode compiler compiled each example in seconds, and generated between 50-2000 bytecode instructions. Figure 23(a) shows a snippet of the SystemC source code for the edge detection circuit. The edge detection circuit was written with two processes, one process to gather pixel data from the input memory, and one process to perform the edge detection and output to the output memory. We configured each platform to use the output memory as a frame buffer, allowing visual

inspection of the output on a VGA screen (Figure 23(b)). The edge detection circuit could process a 128x128 image in approximately 30 seconds on the base emulation engine. While slow, in an early prototyping scenario, or in a classroom setting, such times might be acceptable. We also compared the edge detection circuit running on the SystemC-on-a-Chip platforms to the same SystemC circuit description running on an Intel-based PC running at 2 GHz. The SystemC edge detection circuit took 0.5 seconds to complete the same 128x128 image.

We compared a variety of SystemC descriptions on a base SystemC-on-a-Chip platform on both the Virtex4 Ml403 platform and on the Spartan 3E platform to running a native application on the underlying platform and to PC simulation. On the Spartan 3E development platform, the Microblaze system clock was half the speed of the PowerPC on the Virtex4, but fetched memory more efficiently since the Microblaze had a dedicated bus to the BRAM instruction memory. In all cases, the basic emulation engine executed the SystemC descriptions ~100X slower than the executing an implementation of the application on the native platform and up to 1000X slower than PC simulation. If we normalize for clock speed since the PC is running several orders of magnitude faster than the Xilinx platforms, the performance is comparable. In all cases, the SystemC bytecode was portable, allowing us to write the SystemC application once, and run on any of the supported platforms. The basic emulation engine has the advantage that many smaller development platforms could still support its software (like the Spartan 3E implementation), enabling in-system SystemC emulation for less capable systems, or for

systems without FPGA resources. In future chapters, we seek techniques and architectural enhancements to improve the performance of base SystemC emulation.

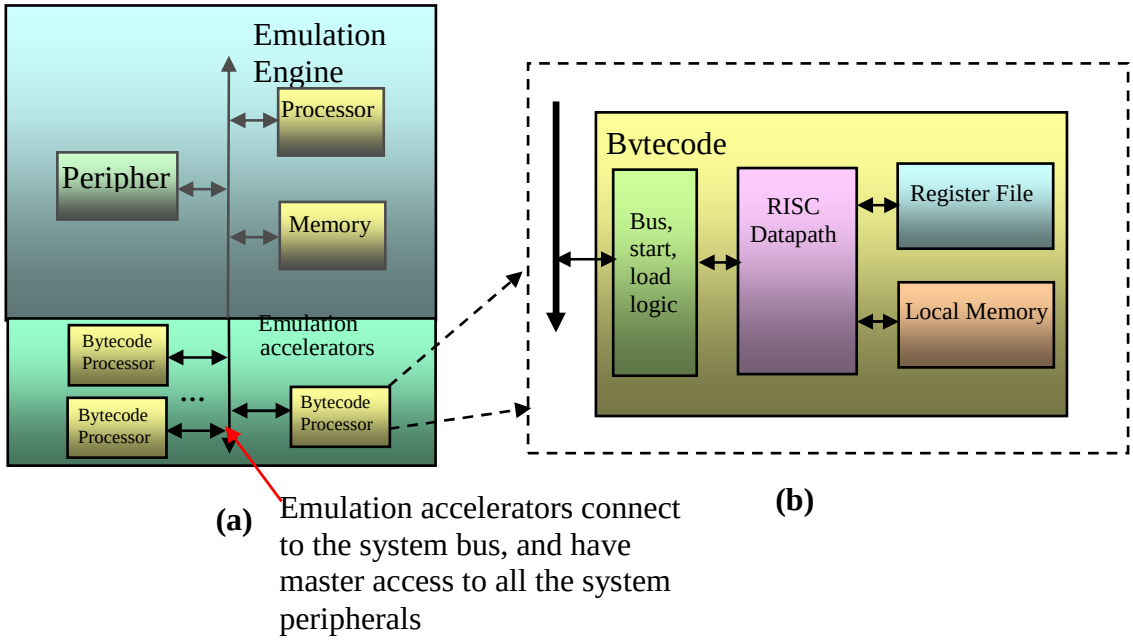
Chapter 4

SystemC Bytecode Accelerators

4.1 Overview

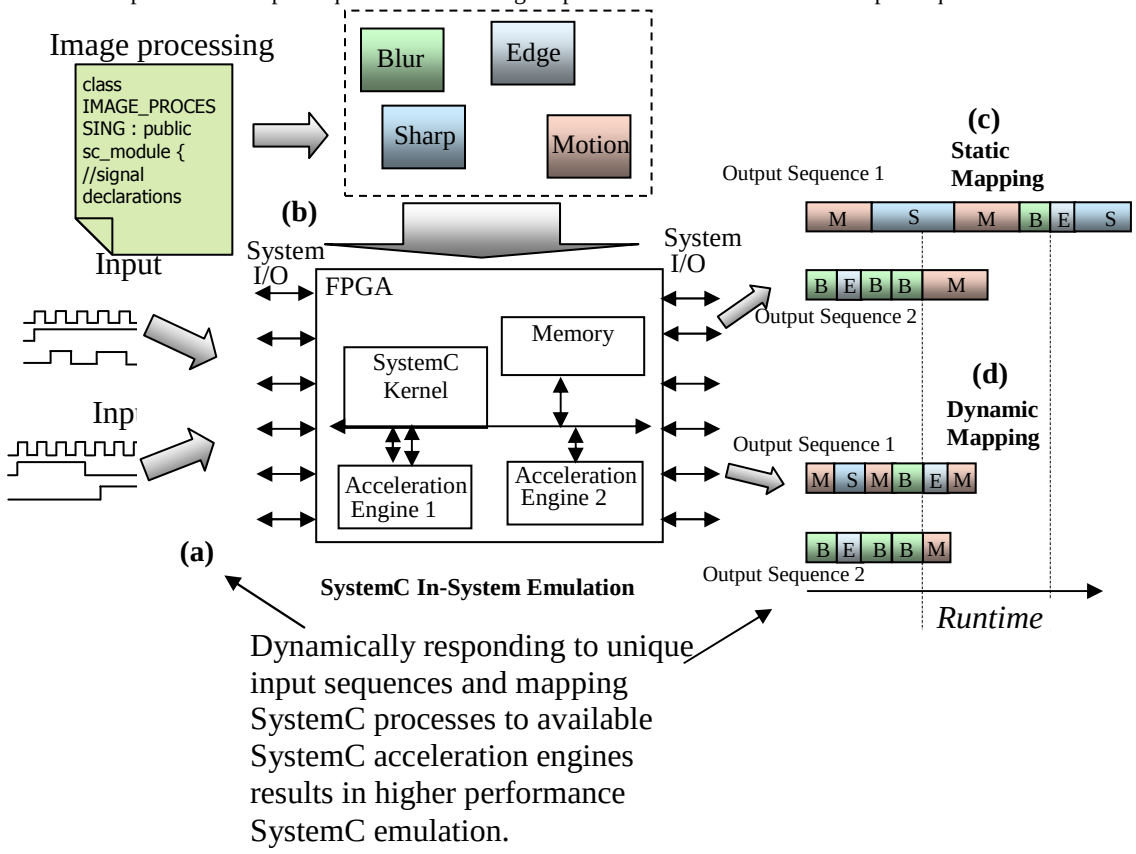
For the common situation where the SystemC-on-a-Chip platform is implemented on an FPGA, we've developed emulation accelerators that substantially increase the SystemC emulation speed. Figure 24(a) shows multiple emulation accelerators connected to the basic emulation engine. Each emulation accelerator runs in parallel to the other emulation accelerators and the main emulation processor. Figure 24(b) shows the internals of one of the emulation accelerators. The emulation accelerator consists of a small SystemC bytecode processor and bus steering logic. The bytecode processor is a modified multi-cycle MIPS datapath, with connections to a register file and local data memory. The emulation accelerator can complete most instructions in 3-4 cycles, with the exception of the READ instruction, which has a nondeterministic execution time since the accelerator must read data from the system bus. The emulation accelerator is configured as a master on the system bus to allow the accelerator to read and write the emulation engine's signal memories independent from the emulation processor, and as a slave to allow the emulation processor to command the start of its execution.

Figure 24: Emulation accelerators. The emulation accelerator consists of a multicycle MIPs-like datapath than can execute one instruction in about 3-4 cycles, almost 100X faster than executing the same instructions in the base emulator.



The number of emulation accelerators can substantially increase the performance of the SystemC emulation since each emulation accelerator runs in parallel. The emulation accelerators do contend for the signal memories, but typical SystemC behavioral descriptions only read/write signals at the start and end of their descriptions. The advantages of emulation accelerators increase as the size of the SystemC processes increase since the emulation accelerator can execute bytecode instructions orders of magnitude faster than the basic emulation engine can. There are tradeoffs though. Assuming circuit emulation doesn't require fast execution, the FPGA area required to implement emulation accelerators could be allocated for other circuitry, including more advanced peripherals or I/O. Also, smaller process descriptions may not benefit much from emulation acceleration, or other SystemC execution times might be perfectly acceptable in without acceleration.

Figure 25: SystemC in-system emulation: **(a)** In-system emulation of a description allows testing with real I/O, thus creating dynamic test bench input vectors that cannot be analyzed statically. **(b)** Sample image processing system that invokes several different filters depending on the input. **(c)** Statically mapping each process to either software or an acceleration engine results in widely varied runtimes for different input sequences. **(d)** Dynamically mapping SystemC processes in response to the input sequence results in higher performance emulation for all input sequences.



Because the SystemC emulation engine benefits from connecting to real I/O compared to modeled I/O, shown in Figure 25(a and b), another potential drawback of SystemC in-system emulation is that the ordering of events on the event queue is not known before runtime, making some existing static acceleration techniques like queue reordering [82] and process splitting [103] less effective. Figure 25(c and d) shows how two different input sequences into a SystemC emulation image processing system can generate two different output sequences, of which only an adaptive mapping of processes to acceleration engines can guarantee higher emulation performance. The SystemC

emulation framework allows for dynamic decisions of whether to execute a process' bytecode on the microprocessor SystemC kernel, or to load and execute that bytecode on an acceleration engine. However, acceleration engines are limited, and loading acceleration engines involves time overhead, so load decisions should be made so as to minimize total execution time.

Thus, a problem exists as to how to efficiently utilize the finite number of SystemC acceleration engines to execute a dynamically changing event-driven SystemC emulation event queue such that the total emulation time is minimized. We define the *online SystemC emulation acceleration* problem, and apply online heuristics to dynamically improve the performance of SystemC emulation.

4.2 Related Work

Improving the performance of event-driven simulations has been extensively researched. Much research has concentrated on developing parallel frameworks for general event-driven simulation. Fujimoto [51] presents a comprehensive survey of several parallel simulation techniques. Jefferson [82] analyzes the critical paths of event-driven simulations, and discusses techniques to achieve supercritical speedups in simulation. Das [36] discusses adaptive protocols for parallel simulations.

Other work has focused on specifically improving SystemC simulations. Naguib [103] automatically splits SystemC processes to prevent unnecessary wake up calls to the SystemC event kernel. Perez [111] creates an optimized implementation of the SystemC kernel that utilizes acyclic scheduling. Wang [149] uses compiled simulation to eliminate

unnecessary evaluations, and to improve simulation time. Our work focuses on dynamic SystemC emulation (rather than static SystemC simulation) whose behavior requires dynamic scheduling techniques to improve performance.

Another area of research combines both of the above approaches to parallelize the SystemC *simulation* kernel. Chopard [30] and Combes [32] show how relaxing a number of constraints on the event queue makes feasible a parallel SystemC event-driven kernel. Chandran [26] identifies methods to execute the SystemC kernel on simultaneous multiprocessor machines for faster performance. Our work utilizes FPGA resources to accelerate the execution of SystemC processes for higher performance emulation.

Dynamic load balancing has been studied extensively in previous works [61][78][98]. The idea of dynamic load balancing is that migrating processes across a network from high load hosts to lower load hosts can minimize application execution time despite overhead in migrating processes between processors. Our online SystemC emulation acceleration problem can be considered a special case of dynamic load balancing with heterogeneous processing units and high migration overheads.

Dynamic system optimizations have also been the focus of much research. Balarin [6] presents a survey of real-time embedded system scheduling, which classifies the problem into static scheduling and dynamic scheduling. Danne [35] introduced real-time scheduling algorithms for periodic applications in an FPGA. Ghiasi [53] uses the task graph model to reorder task execution offline to minimize reconfiguration overhead. Huang and Vahid [72][73] develop new online heuristics for managing FPGA coprocessors in a dynamic environment. Noguera [108] proposed dynamic run-time

hardware/software scheduling techniques for FPGAs emphasizing dynamic concurrent task scheduling. Steiger [128] proposed the use of a reconfigurable operating system to manage dynamically incoming tasks and online scheduling problem. Our work applies these dynamic techniques to improve the performance of SystemC emulation.

4.3 Online SystemC Emulation Architecture

4.3.1 Base Architecture with Acceleration Engines

A SystemC emulation architecture enables the execution of SystemC descriptions on real platforms without the need to synthesize/map for the particular platform, by executing an intermediate form of SystemC called *SystemC bytecode*. Figure 26 shows a basic SystemC emulation platform. The platform consists of a main processor that executes the

Figure 26: SystemC emulation platform. A limitation of the SystemC emulation platform is that the acceleration engines and the SystemC kernel within the emulation platform are connected via a single bus structure, thereby creating a bottleneck for shared memory usage when multiple processes (*p1*, *p2*, *p3*) are scheduled in parallel, hindering performance.

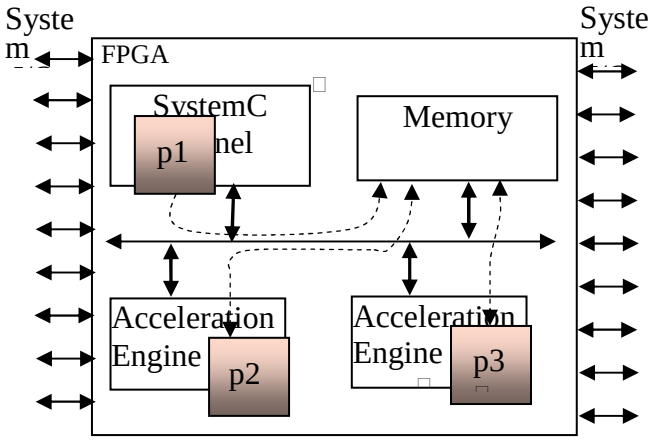
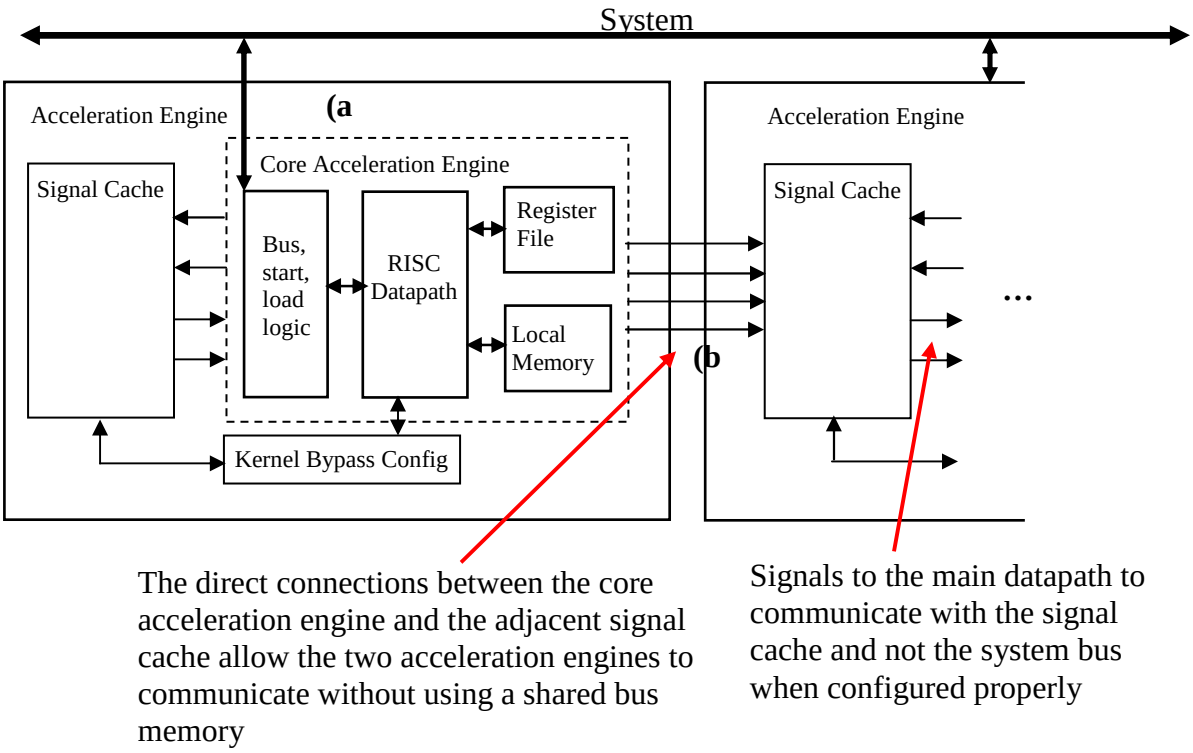


Figure 27: SystemC acceleration engines: **(a)** Internal structure. **(b)** Direct connection of two SystemC acceleration engines using a kernel bypass connection. In some situations, bypassing the bus and SystemC kernel can lead to significant performance benefits for a given SystemC description.



SystemC kernel, which is a combination of a virtual machine and event-driven kernel. The SystemC kernel connects to the platform's peripherals (memories, lights, buttons, timers, general I/O) through a shared bus, allowing a SystemC description full access to a variety of peripherals.

For the common situation where the emulation engine is implemented on (or with access to) an FPGA, the SystemC kernel can offload process emulation to a SystemC *acceleration engine*. An acceleration engine, shown in Figure 27(a), consists of a MIPS-like datapath, communicates with the SystemC kernel via memory-mapped registers, and executes SystemC bytecode orders of magnitude faster than the SystemC kernel.

4.3.2 Kernel Bypass

We observed that the SystemC emulation platform possesses a memory bottleneck when both the main emulation kernel and the SystemC acceleration engines attempt to read and write the shared signal memories, as highlighted in Figure 26. To mitigate the memory bottleneck, we introduce *kernel bypass connections,* which are direct one-way connections between neighboring accelerators that allow the SystemC accelerators to communicate without having to read and write their values to shared memories on the system bus. Figure 27(b) shows the kernel bypass architecture for two SystemC accelerators. An additional advantage of kernel bypass connections is that the emulation kernel also reduces some overhead of maintaining the event queue since the writing accelerator can directly flag the reading accelerator to start execution once the writing accelerator is done.

To facilitate direct communication between two neighboring accelerators, we add a SystemC kernel-controlled configuration register and small signal cache. A *signal cache* is a small memory data structure that holds a signal identifier, the signal's value, and a valid bit. If an accelerator is configured to be a kernel bypass reader, the acceleration engine will instead first look for a signal value in signal cache prior to fetching the value from the signal memory on the bus. Similarly, if a SystemC accelerator is configured as a kernel bypass writer, the SystemC accelerator will write to the connected accelerator's signal cache by sending the signal's ID and its current value. In contrast to the system bus which can take tens of cycles, the signal cache allows one-cycle signal writing and retrieval. For each simulated time step, a utilized kernel bypass

connection can save between tens and hundreds of cycles, depending on the number of signals written to and read from.

The signal cache size is currently limited to ten signals. If two processes communicate with more than ten signals, the two processes must communicate through the bus-connected signal memories. Processes that communicate with more than ten signals can still see some speedup because ten read and writes to the system bus are eliminated every simulated time step.

4.4 Online Acceleration Assignment

4.4.1 Problem Definition

We define the *Online SystemC emulation acceleration* problem as follows. Given are:

- A process set P = {p1, p2, p3, ..pn} containing the n processes that comprise a given SystemC description.

- A set of execution times Tp = {tp1, tp2, tp3,… tpn} containing the execution time of each process i running on the SystemC kernel without communication overhead.

- A set of execution times Tc = {tc1, tc2, tc3...,tcn} for each process i when running on a SystemC acceleration engine; the times do not include communication overhead.

- A set of sizes S = {s1 ,s2, s3,…, sn} giving the size of each process i in terms of number of bytecode instructions..

- The total number of acceleration engines AE in the SystemC emulation framework.

- The time to load one instruction into a SystemC acceleration engine TR.. The total time to load an acceleration engine with process i can be thus be written as: loading time(i) = TR*si

The online SystemC emulation acceleration problem must satisfy the following constraints:

- Processes running on the SystemC kernel and on the acceleration engines may run in parallel, unless that process is the same process i. For instance, in the queue <p2, p1, p1, p1, p3>, the three instances of p1 must execute sequentially, but p2 and the first instance of p1 can run in parallel.

- The SystemC kernel cannot be interrupted to run a process when the SystemC kernel is loading a process onto an acceleration engine or when the SystemC kernel is itself running a process.

We define several additional constraints to the online SystemC emulation acceleration problem that take advantage of the number of kernel bypass connections within the SystemC emulation framework:

- A set O of process pairs (Oi, Oj) that satisfy the condition that all of the inputs into Oj are outputs from Oi. These process pairs can be determined statically and sent to the SystemC kernel at download time

- Number of kernel bypass connections: The number of kernel bypass connections in the SystemC emulation platform

- Kernel bypass connection pairs: For each Kernel bypass connection, there exists two acceleration engines AEi and AEj that the connection is made up of

- Number of signal connections between each process pair (Oi, Oj)

The dynamic input to the problem is an event queue *Q*, such as *<p2, p1, p4, p2, p1, p1….>*, that lists and orders the process instances that run on the platform for a given time step.

The Online SystemC Emulation Acceleration problem is defined as an online problem: For each process in the event queue, using only knowledge of *prior* and *current* processes in the queue, determine whether to load that process into a SystemC acceleration engine, such that the time for the *entire* event queue (including future instances of the process in the queue) is minimized. When a process is already loaded into a SystemC acceleration engine, we refer to the process as being *acceleration engine resident*. The *current process* is the process that at a given time is to be executed next and for which the acceleration engine load determination must be made. Thus, the solution to the online SystemC emulation acceleration problem consists of an acceleration engine management decision for each process instance in the event queue. Each decision is either: load, don't load, or already loaded. For a decision to load, the decision also lists a process that must be unloaded to make room for the new process being loaded.

4.4.2 Communication Overhead

The SystemC accelerators communicate with the SystemC kernel through memory mapped registers and *signal memories,* which store the current and next values of each signal in the SystemC description. We use queuing theory [55] to estimate average memory access delay, and model memory contention by the M/M/1 queue. The processes in the SystemC kernel and in the SystemC acceleration engines generate memory access requests through *READ* and *WRITE* bytecode instructions. We define the following:

- Random memory access rate: The random memory access rate is the number of times a process i reads from memory, where λi is the memory access rate of running process i.

- Bus service rate: μ. The bus service rate is the number of requests the system bus can process in a second. E.g. Assuming a 100Mhz memory bus, one access takes 20 cycles, so μ=5M/s.

- Average delay: The average delay is the number of cycles for one memory access. According to queuing theory, average delay for one access is $D=\lambda/(\mu(\mu-\lambda))$.

- System delay: delay = $D\lambda$.

4.5 Online Heuristics

4.5.1 Upper and Lower Bounds

An upper bound on total execution time can be determined by running every process on the SystemC kernel. A lower bound can be determined by assuming every process is

preloaded onto an infinite set of existing SystemC acceleration engines, and considering communication overhead, referred to as the *Infinite Accelerators.*

4.5.2 Accelerator Static Assignment

To see the advantage of dynamically loading bytecode to the SystemC acceleration engines for higher performance emulation, we compare to a *statically preloaded* approach, which assumes the SystemC acceleration engines are initially loaded with one process's bytecode each, and are not reloaded during runtime. At the beginning of SystemC emulation, the SystemC kernel assigns each acceleration engine a process to always execute when an instance arrives on the event queue. The acceleration engines are loaded with the processes that have the largest speedup potential (*tpi-tci)*. Compared to dynamic techniques, the benefits of static accelerator assignment are one-time acceleration engine loading, and a simpler emulation event kernel. The drawbacks are that there might only be a few acceleration engines, and running the rest of the SystemC processes on the software SystemC kernel could be computationally expensive. An alternative method for static assignment would have been to utilize profile information to predict which processes execute most frequently. However, due to simulation complexity, profiling information was not available.

4.5.3 Greedy Heuristic

A greedy heuristic can be defined that always loads the current process into a SystemC acceleration engine before executing. If the process is *acceleration engine resident*, the

SystemC kernel just instructs the SystemC acceleration engine to begin executing. Otherwise, the SystemC kernel randomly chooses an idle SystemC acceleration engine to load the process' bytecode instructions. In the case that all the SystemC acceleration engines are busy running, the emulation kernel will wait until the one of the acceleration engines becomes idle. The time complexity of the greedy heuristic is *O(1)*. However, the greedy heuristic may incur lots of loading overhead since it loads a SystemC acceleration engine with bytecode on every execution. Further, the greedy heuristic attempts to use all the available acceleration engines, which increases the amount of communication overhead on the system bus.

4.5.4 Aggregate Gain

We use the aggregate gain (AG) heuristic introduced in [72] to address the online SystemC emulation acceleration problem. The AG heuristic uses the history of application executions to attempt to predict future executions and hence to predict when reconfiguration overhead is worthwhile. The AG heuristic considers reconfiguration and communication overhead. The basic idea of AG is that we maintain an aggregate gain table for each process type running in the system. The gain is the time saved by running the process instance with the accelerator. The AG table gets updated when a new process arrives. The AG table shows which processes make most of the gains by running in the SystemC acceleration engine.

Sequences of processes on the event queue often exhibit temporal locality—recently-executed processes are more likely to execute in the near future than are

processes from long ago. A fading factor *f* is introduced to refresh the AG table. *f* is adaptive to the average loading time. The intuition of the loading, replacement and wait decision is to make the total gain of the acceleration engine resident processes high. Thus the load, replace and wait decisions will be made only if the decision would not decrease the total gain resident processes.

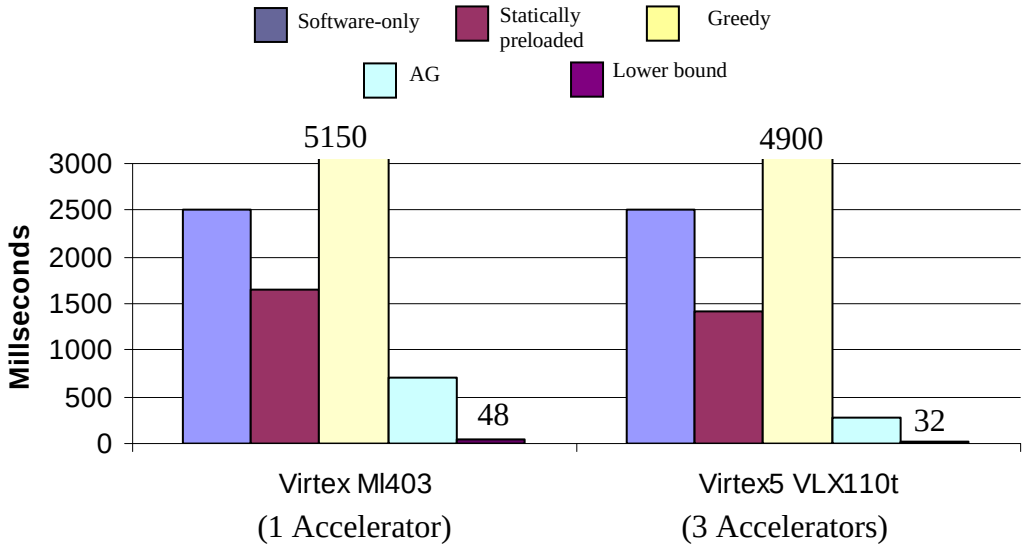
We can alter the AG heuristic to support the additional kernel bypass feature. The modified AG heuristic treats tightly coupled processes as one large process. The large process takes multiple acceleration engines and we assume the acceleration engines of the large process must be loaded together. The load, replacement, and wait policies of the large process are similar to the definitions in original AG heuristic.

4.6 Experiments

4.6.1 Framework

We developed a simulator in C++ to test our heuristics, and applied the simulator to several SystemC descriptions. We also fully implemented two SystemC emulation platforms, one on a Xilinx Virtex4 Ml403 development platform, and one on a Xilinx Virtex5 vlx110t development platform. The SystemC kernels ran on a PowerPC and Microblaze processor respectively, both operating at 100MHz. The SystemC kernels communicate to the acceleration engines and the rest of the peripherals through the PLB bus. The average memory access time is 40 cycles. The SystemC kernel uses a handshaking protocol over the PLB bus to communicate and load instructions into each of the acceleration engines. The total time to load one instruction (*TR)* onto an

Figure 28: Emulation runtime results of image filtering, lung, and radiosity examples emulated on two different emulation platforms. *AG* performs up to 9x faster than software-only emulation, and 5x faster than a *statically preloaded* approach.



acceleration engine is approximately three microseconds. The Virtex4 Ml403 development platform could hold one acceleration engine, and the Virtex5 vlx110t development platform could hold three. For two of the accelerators in the Virtex5 vlx110t, we connected them for kernel-bypassed enabled execution. One accelerator was configured as a reader, and one was configured as a writer. We chose this configuration because many of the image processing SystemC circuits mapped to this architecture well. The kernel bypass circuitry only consumed a few hundred more slices than the core acceleration engine. The SystemC emulation kernel was written in approximately 2500 lines of C code. The online heuristics consisted of only a few hundred lines of code.

We applied our heuristics to an image filtering system (including a blur filter, an emboss filter, a sharpen filter, and several implementations of edge detection), a digital lung model [107], and a reconfigurable radiosity design [7]. We wrote the image filters,

lung model, and reconfigurable radiosity design in SystemC, capturing each design using multiple processes. We modeled several dynamic scenarios in which the image filters, lung model, and radiosity design might be used.

For all experiments, because sequences involve some random ordering, we generated 20 sequences, and report the arithmetic average. The heuristic runtimes were negligible.

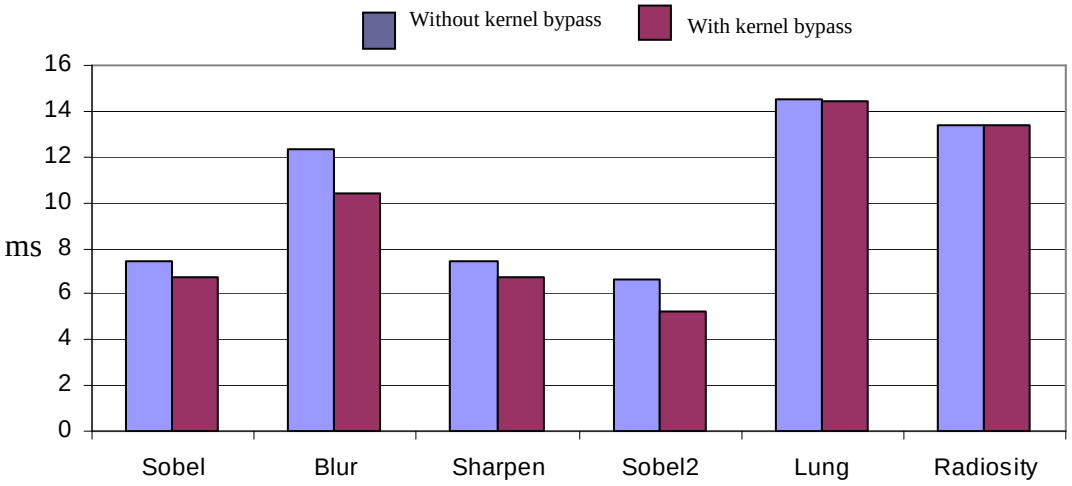
4.6.2 Evaluation

Figure 29 shows total execution times of a suite of SystemC image processing, lung, and radiosity descriptions running on Virtex4 Ml403 and Virtex5 vlx110t implementations of the SystemC emulation framework without the kernel bypass mechanism enabled.

For the Virtex4 Ml403 implementation, the *statically preloaded* accelerator approach yielded ~1.5x speedup compared to software-only emulation (i.e., only running on the SystemC kernel and no acceleration engines). The greedy heuristic results in a slowdown of 50% compared to software-only emulation. This is because the *greedy* attempts to reconfigure the accelerators without consideration of the high reconfiguration cost of downloading new bytecode instructions. The dynamic *AG* approach yields more speedup. The execution time *AG* obtains over software-only emulation and a statically preloaded approach is 3.5x and 2.3x respectively. *AG* performs approximately 7x faster than the *greedy* heuristic.

For the Virtex5 vlx110t implementation, the *statically preloaded* accelerator approach yielded ~1.75x speedup compared to software-only emulation. Compared to the

Figure 29: Emulation runtimes without and with kernel bypass using the AG heuristic on the image processing examples. Kernel-bypass-enabled emulations performed on average 11% better than without kernel bypass, and up to 20% in some examples.



Virtex4 Ml403 implementation which only had one accelerator, the nominal speedup achieved with the Virtex5's three accelerators was unexpected, and could have resulted due to a poor mapping between processes to accelerators. The penalty could also have been due to increased communication costs on the system bus. The *greedy* heuristic was again about 50% slower than software-only emulation because of the high cost to reload the acceleration engines with new bytecode instructions. The *AG* heuristic performed 9x, 5x, and 18x better than *software-only emulation*, *statically preloaded*, and *greedy* solutions respectively. The *AG* heuristic takes the accelerator reloading cost into account and thus decided not to reload the accelerators every time there was a new process on the event queue.

Comparing with the *Infinite Accelerators* lower bound (i.e., all processes are accelerated and without the need to reload the bytecode instructions onto the accelerator) shows that the AG heuristic obtains execution times on average within 15x slower on a

platform with one accelerator because of the high loading time, and 8x slower on a platform with three accelerators of this lower bound. The lower bound solution does not need to contend with the high reconfiguration time the other heuristics do. Future work could look into modifying the architecture for decreased reconfiguration times.

Figure 29 shows the effect of enabling a kernel bypass connection between two accelerators on the Virtex5 vlx110t emulation platform (the Virtex4 Ml403 could only hold one acceleration engine, so kernel bypass was non-applicable). On average, the SystemC examples improved their speedup by 11%. *Blur* and *Sobel2* achieved 20% speedup with kernel bypass because they contained a few processes that had heavy communication. Other examples like the *Lung* and *Radiosity* only improved by a few percent. This was because the inter-communication between processes was light. More kernel bypass connections could increase performance by more significant gains.

Chapter 5

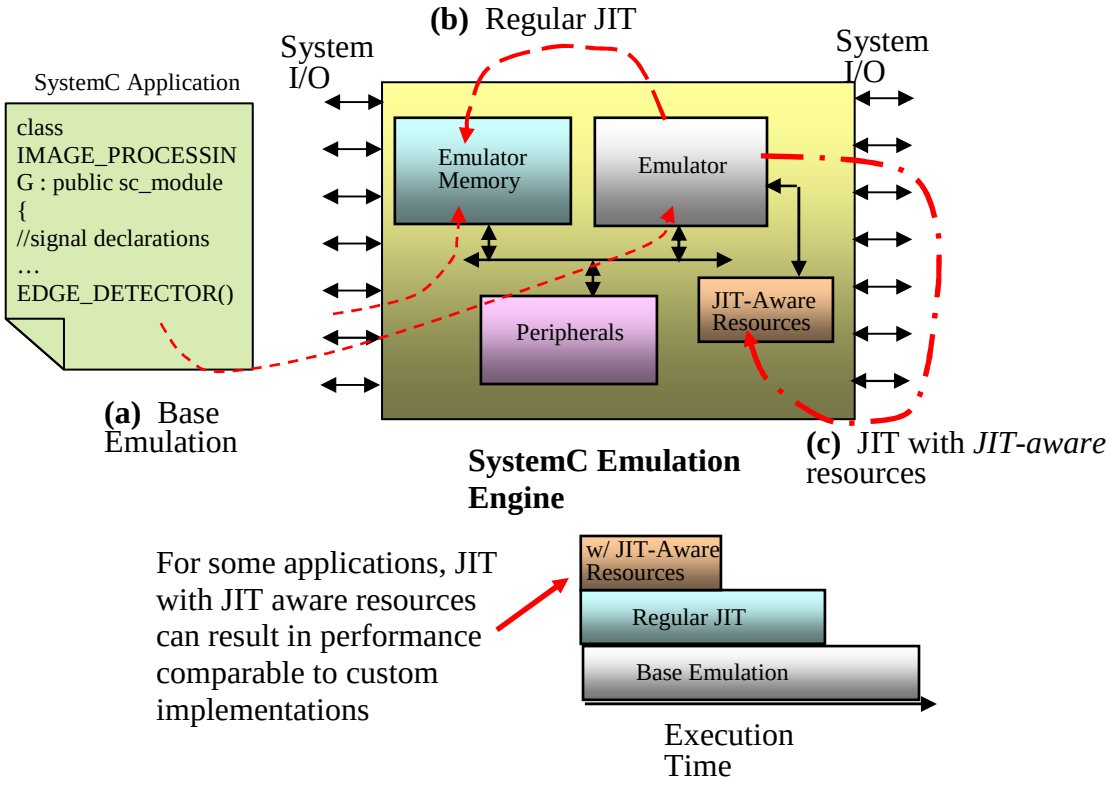
Just-in-Time Compilation of SystemC

5.1 Overview

The previous two chapters detailed the SystemC-on-a-Chip framework, enabling portable execution of SystemC applications on any platform the supports the SystemC emulation engine, and SystemC accelerators and kernel bypass mechanisms that could substantially increase the performance of SystemC emulation with dynamic system optimizations. However, the acceleration engines require FPGA resources. We take a different approach to speedup, wherein we just-in-time compile the SystemC bytecode into native instructions of the soft-core processor, as shown in Figure 30(b). Just-in-time compilation has been used with wide success to speed up emulated commercial applications in the CLR format (from C#) and Java bytecode, for PC-based platforms [83]. Our work is the first JIT approach for an FPGA soft-core processor.

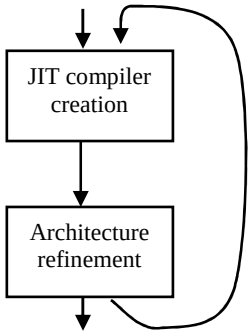
More significantly, however, is that JIT for an FPGA soft-core processor provides even more optimization possibilities than JIT for a traditional processor. The reason is because the soft-core processor architecture can be changed. As such, we could carry out an iterative process, whereby after creating an initial JIT compiler, we could analyze

Figure 30: While the performance of the base SystemC emulation engine is acceptable for some applications, for others it is not (a). Just-in-time compiling the SystemC bytecode to the emulator's memory improves performance (b), but can be made to be competitive with custom implementations if the emulation engine is made *JIT aware* (c).



system performance to detect the new performance bottleneck. We could then change the processor architecture in order to alleviate that bottleneck, modify the JIT compilation accordingly, and repeat until no further improvements were found, as shown in Figure 31. The resulting JIT compilation, with the architecture containing JIT-aware resources as illustrated in Figure 16(c), showed substantial further speedups over the original JIT compilation.

Figure 31: The JIT/architecture codesign process.



5.2 Related Work

There has been much previous work in the field of dynamic binary translation and just-in-time compilation to improve the performance of software interpretation. Of the many techniques to improve execution of Java bytecode, just-in-time compilation often improves execution runtimes to near native speeds [83]. The Transmeta Crusoe processor [38] dynamically translates x86 code into native VLIW instructions for improved performance and reduced power. Other architectures, like accumulation-based computer architectures [84], have also benefited from just-in-time compilation techniques. Gligor [54] used dynamic binary translation to improve the speed and flexibility of MPSoC simulations.

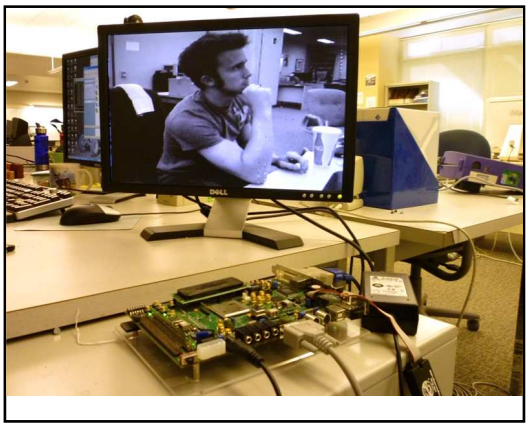
There has been an increased amount of work done to improve virtualized software execution with supporting hardware. Adams [2] presents a survey of techniques for improving x86 virtualization execution, discussing both software and hardware optimizations. Rosenblum [117] discusses the advantage of hardware-level virtual machines, and the need to make them as fast, efficient, and transparent as possible. Enzler

[45] uses reconfigurable arrays to virtualize hardware. Bauer [12] uses reconfigurable arrays to improve the execution time of event-driven simulation.

5.3 Experimental Setup

For the upcoming experiments in this paper, we built three complete SystemC-on-a-Chip platforms, each with differing memory subsystem implementations, and differing performance profiles. We built one system on a Xilinx Spartan 3E FPGA platform that required that the SystemC engine reside in DRAM memory. We built a SystemC engine on an SRAM-based memory structure on the Virtex4 Ml403 development platform. Finally, we implemented the SystemC emulation engine on a larger Virtex5 vlx110t, and shown in Figure 32(a). The Virtex5 implementation also executed from a large SRAM. To highlight the benefits of the new emulation architecture changes, we built two versions of each platform, one with the dedicated just-in-time emulation architecture changes, and one without. Each system is briefly summarized in Figure 32(b). The emulation architectures were described using approximately 10,000 lines of VHDL. We wrote the SystemC emulation using approximately 3,000 lines of C. The emulation architectures were built using Xilinx ISE 11, and the software was compiled using Xilinx EDK 11.

Figure 32: Experimental Prototypes. **(a)** The Virtex5 vlx110t implementation connected to a large screen buffer for testing image processing applications. **(b)** A summary of each experimental system. Each version was built with and without dedicated hardware to improve the impact of just-in-time compilation of the SystemC bytecode.



(a)

Development Platform	Main Processo	Memory	
Xilinx Spartan3E FPGA	Microblaze	Emulation Engine	DRAM
		JIT Memory	BRAM
Xilinx Virtex4 Ml403 FPGA	PowerPC	Emulation Engine	SRAM
		JIT Memory	BRAM
Xilinx Virtex5 vlx110t	Microblaze	Emulation Engine	SRAM
		JIT Memory	BRAM

(b)

We picked a variety of benchmarks to test our SystemC just-in-time compilation approach. The benchmarks range from image processing applications like Sobel edge detection to encryption algorithms like an A5/1 stream cipher. We carefully chose applications with varied amounts of complexity to show where just-in-time compilation for SystemC excels and where it doesn't. To compare the speed of the JIT compiled code with an "upper bound," we rewrote each benchmark directly in C code (not SystemC), performing a manual scheduling of processes so as to eliminate the need for the scheduling done in the SystemC simulation kernel. The C descriptions are less intuitive than the SystemC descriptions, and the parallelism in the application is less exposed, but the C descriptions provide an upper bound as to how fast the SystemC bytecode could possibly execute on a Microblaze—essentially, the C code strips away all SystemC overhead and describes just the application code. We compiled the C descriptions directly to Microblaze machine code using the Xilinx tools and the highest levels of optimization (O3). We refer to this implementation as *native software*.

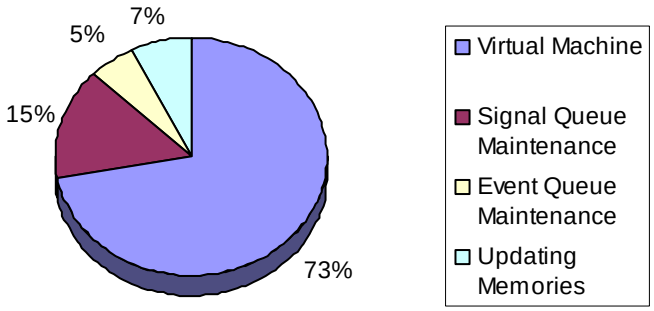
5.4 Just-in-Time Compilation of SystemC

We began by profiling the SystemC emulator's execution for the benchmarks. Figure 33 shows, which clearly show that the virtual machine executing on the Microblaze contributes to most of the execution time, namely 73%; the other contributors relate to architectural features. The virtual machine's dominance is due to each bytecode instruction requiring dozens of Microblaze instructions to execute. Just-in-time compilation from bytecode instructions directly to Microblaze instructions should thus greatly decrease that time, because almost all SystemC bytecode instructions can be translated to just 1 or 2 Microblaze instructions.

5.4.1 Compilation

Just-in-time compilation from the SystemC bytecode to the target platform is

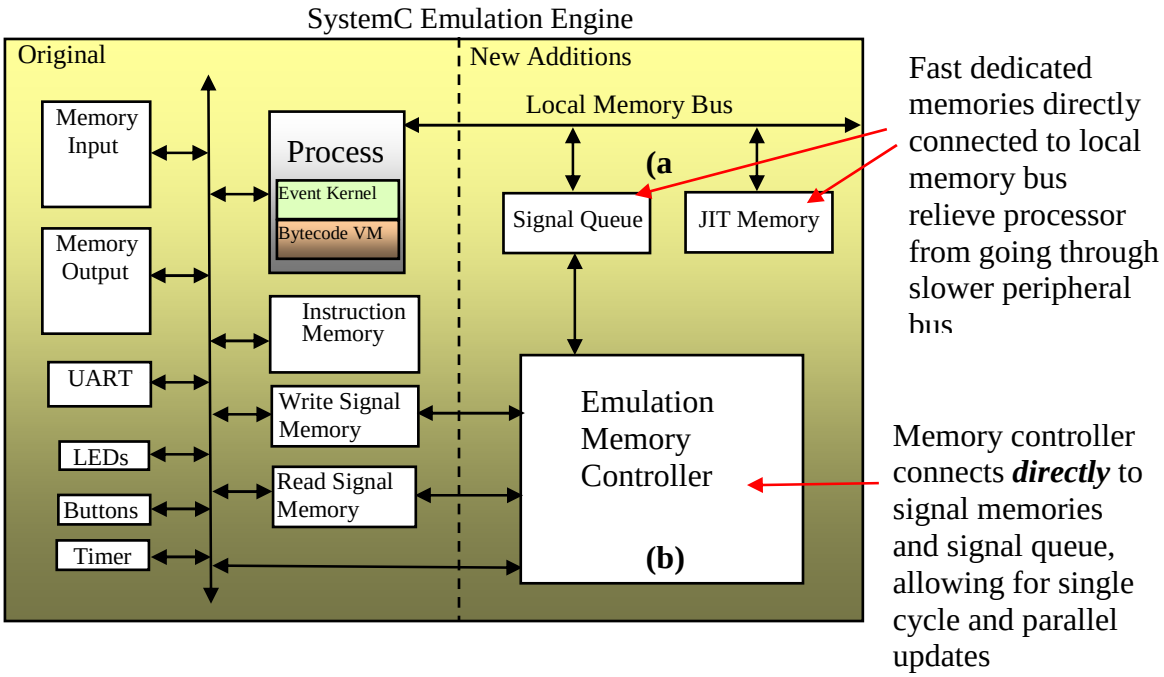
Figure 33: Results of our initial profiling of the SystemC bytecode emulator.



straightforward. The SystemC just-in-time compilation process consists of three analysis phases and three translation passes. The first analysis phase determines how many instructions each source SystemC bytecode instruction will require in the target architecture. The second analysis phase determines which bytecode registers depend upon values from previous executions of the process. The third analysis phase determines which register conventions might be violated by naïvely translated code – for instance, any registers that must be saved across function calls should not be overwritten.

The first translation pass directly copies bytecode instructions to appropriate locations in the JIT memory, which can be calculated from the information gleaned in the first analysis phase. The second pass translates each bytecode instruction, which also requires information from the first analysis phase (to recalculate relative branches). The third pass adds a function prologue and epilogue to ensure compliance with the emulation engine and architecture register conventions, which requires the information from the latter two analysis phases.

Figure 34: Modifications to the SystemC emulation engine that increase the utility of just-in-time compilation. The new SystemC emulation engine supports a local memory bus with a dedicated JIT memory and a static signal queue for fast access to commonly executed software operations **(a).** The new SystemC emulation engine also has a dedicated *emulation memory controller*, which offloads costly memory updates from software, and magnifies the impact of just-in-time compilation



5.4.2 JIT Compilation with Dedicated JIT Memory Resources

Unfortunately, straightforward just-in-time translation often results in unimpressive performance improvements. There are a number of reasons for this, but perhaps the most obvious is the emulation memory architecture. The entire emulation engine requires a large instruction memory, heap, and stack, and does not lend itself easily to small, fast memories (which are often very limited, and sometimes non-existent) Thus, the emulation engine usually resides in a larger, slower memory (e.g., DRAM, or SRAM). Naïvely placing the native code resulting from just-in-time compilation back into this same memory shows performance improvement, but this improvement will be greatly hampered by memory latency.

We observed that since the native code returned by the just-in-time compilation process is much smaller than the emulation engine needed to execute bytecode, the SystemC emulation engine would benefit from using a small fast memory dedicated for storing the just-in-time compiled native code, as shown in Figure 34(a). The dedicated JIT memory directly connects to the base SystemC kernel via a local memory bus, can hold small amounts of natively translated SystemC code, and can execute orders of magnitude faster than the original interpreted SystemC bytecode. The just-in-time compiled code is also several times faster than translated code executed from the original slower memory.

We implemented the just-in-time compilation routines in approximately 1,500 lines of C. For our experiments, we assume the emulator can just-in-time compile the *entire* SystemC application to the dedicated just-in-time memory. Of course, assuming enough just-in-time memory isn't necessarily a constraint as the emulator can fall back on just-in-time compiling the SystemC bytecode to the larger, slower memory resources and still see performance improvement. For each example, the emulator just-in-time compiles the SystemC circuit to the dedicated memory *prior* to emulation execution. The time required to just-in-time compile is a one-time cost, and runs in milliseconds, even for large System C applications. Future work might investigate methods for just-in-time compiling dynamically as the SystemC application is running.

Figure 35: JIT compilation with dedicated JIT resources performed 4X faster than the base SystemC emulation platform, yet still fell short of native software implementations by another 10X.

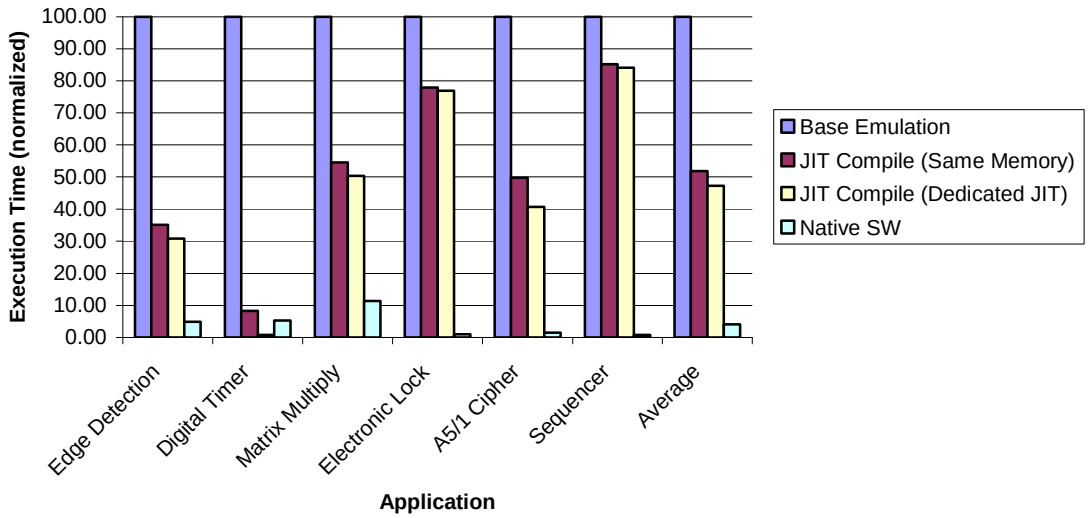


Figure 35 shows the advantages of using just-in-time compilation for SystemC emulation on the Virtex5 development platform for a number of SystemC applications. For each example, we compare just-in-time compilation to an implementation of the SystemC emulation engine implemented entirely in DRAM. We also compare the just-in-time compiled version to an implementation of the application running natively on the development platform. The results for the Spartan 3E and Virtex 4 SystemC emulation implementations were similar to the Virtex5 implementation. Figure 35 shows results running JIT compilation using dedicated JIT memory resources compared a more straightforward approach using the platform's normal resources. On average (geometric mean), JIT compilation with dedicated resources achieves approximately 4X speedup compared to base emulation, and 1.6X speedup compared to just-in-time compiling to the emulator's same memory resources. For computationally intensive SystemC applications, like the digital timer, just-in-time compiling to dedicated JIT memory resources resulted in over 100X speedup. For others, like the electronic lock, the

speedups we less impressive. While still achieving 1.5X speedup, the electronic lock lacked computationally challenging routines, meaning other components of the SystemC emulator became a new bottleneck.

5.4.3 Emulation Memory Controller

Dedicated just-in-time memories improved the performance of SystemC emulation by over 4X on average. However, the improved performance still fell short of the software running natively on the development platform by 10X. While this can partly be explained by the different software implementations required for the native platform (sequential implementation) compared to the original SystemC implementation (structural and spatial implementation), a more concerning factor was the overhead the emulation engine incurred managing queue and memory resources to preserve correctness of the SystemC application, shown in Figure 33.

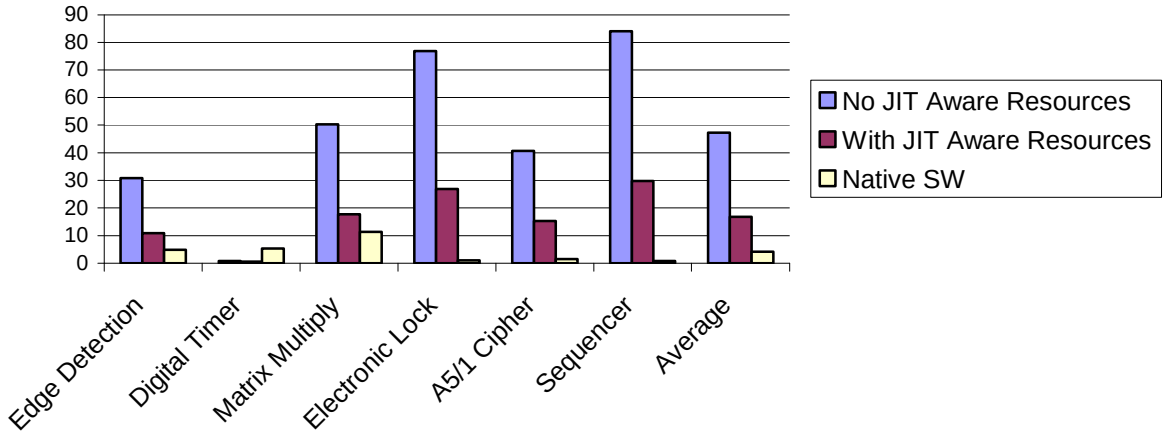
To facilitate the described just-in-time compilation techniques, we introduce several additional architectural changes to the base SystemC emulation engine, shown in Figure 34. The new architectural changes address the remaining 27% of the software time spent concerned with updating the read and write signal memories, and maintaining the signal and event queues, and thus greaten the impact of replacing the just-in-time compiled SystemC bytecode with the interpreted code of the SystemC virtual machine.

The first change to the SystemC emulation engine is the addition of dedicated fast memory connected directly to the processor to act as the new *signal queue.* Original implementations dynamically managed the signal queue, making unnecessary low-level

memory allocation calls. We observed that the signal queue is bound in size by the size of the read and write signal memories, and thus decided that a statically created memory would mitigate the effect of dynamic signal queue management. We also observed that instead of *enqueueing* and *dequeuing* signals to and from the signal queue, the emulation engine only needed to store a signal identifier, reducing the overhead on the bus to which the signal queue is attached to the processor.

We further observed that *signal queue maintenance* (15%) and *updating memories* (7%) were a series of interleaving function calls that worked with highly dependent data (updating the write and read signal memories involved enqueuing the signals that changed values), we could offload the tasks of updating the memories *and* the maintenance of the signal queue to a dedicated *emulation memory controller*. On completion of a delta time step, the emulation engine kernel commands the emulation memory controller to update the signal memories and populate the signal queue. The emulation memory controller iterates over the write signal memory, finds any signals that have been updated, updates the read memory signal value, and adds the updated signal to the signal queue. The actions can be pipelined, meaning that the emulation memory controller can check, update, and enqueue every signal in the system in one pass. For a typical SystemC application with 40-50 signals, the emulation memory controller can finish updating all signals in 40-50 cycles. This is in contrast to a software approach which requires high hundreds-thousands of cycles for the same SystemC application.

Figure 36: JIT Compilation with JIT Aware Resources speeds execution by 10X compared to base emulation, and by 2.5X compared to JIT compilation without the same resources.



Dedicated just-in-time memory resources improve performance, but are limited by software calls to manage queue and memory resources. Fortunately, for FPGA platforms the software can be replaced by a dedicated signal queue and memory controller, which can update the signal queue and the signal memories in tens of cycles (compared with hundreds to thousands). Figure 36 shows the effect of including the dedicated JIT aware resources into the SystemC emulation architecture. The results are again shown using the Virtex5 vlx110t as the example development platform. The results compared just-in-time compilation running in dedicated JIT memory resources with and without the additional JIT Aware resources. On average, just-in-time compilation with JIT Aware Resources improved execution times by 10X compared to the base emulation architecture, and by 2.5X compared to JIT compilation without JIT aware resources. Again, for computationally demanding applications, JIT compilation with JIT Aware resources could actually attain better execution times than the native application. This is

due to the fact the computationally demanding SystemC application now runs in local fast BRAM memories, and the native application still executes in slower memory resources. For other applications, the speedup isn't quite as dramatic, but JIT compilation with JIT Aware resources comes with 4X of native application execution on average.

Chapter 6

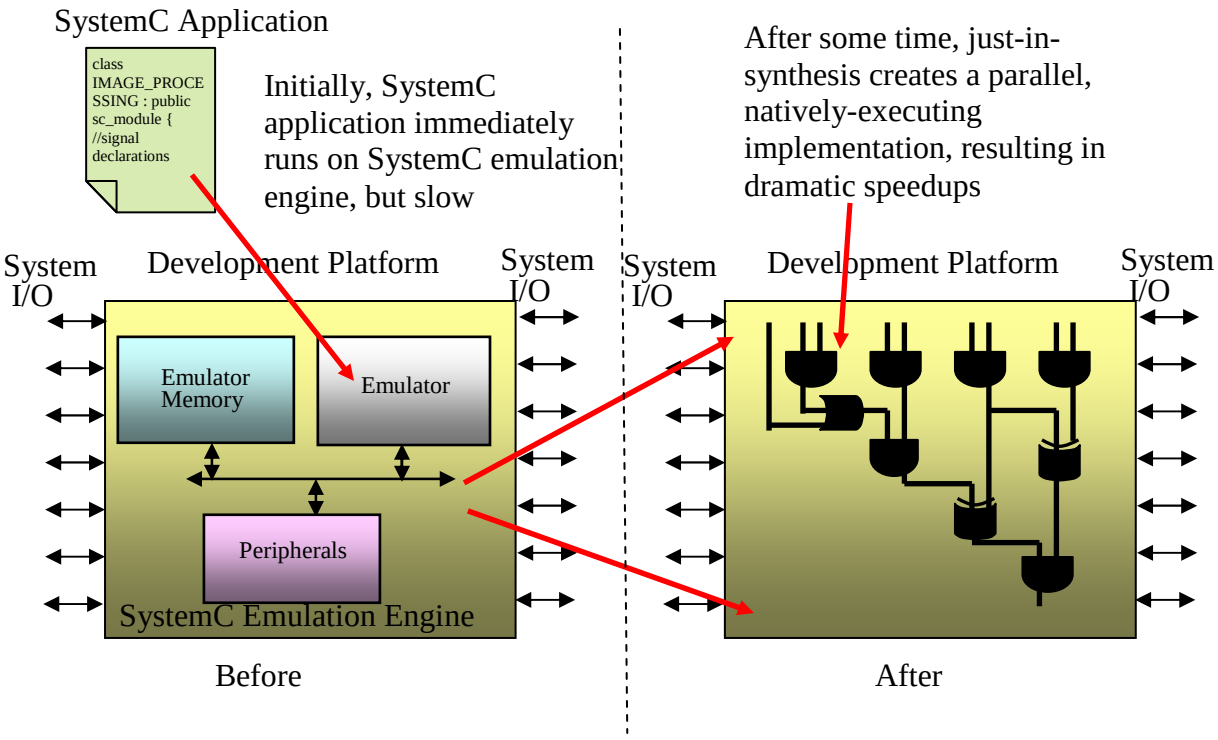
Just-in-Time Synthesis of SystemC

6.1 Overview

The performance of SystemC emulation can be improved greatly using online bytecode acceleration and just-in-time compilation techniques, but still pales in comparison to the potential of native FPGA implementations. While SystemC bytecode accelerators are able to expose some of the parallelism present in the SystemC application, each process is still executed *temporally*, greatly limiting opportunities for high-performance FPGA speedups (and the reason the SystemC application might have been written in the first place).

Analogous to Java-like approaches where just-in-time compilation can improve application execution times by orders of magnitude by a translation of the bytecode to native platform instructions, the availability of FPGA resources on platforms that support the SystemC-on-a-Chip framework lend themselves to being utilized for native execution of the SystemC application.. Figure 37 illustrates just-in-time synthesis of SystemC applications to a native FPGA implementation.

Figure 37: Just-in-Time Synthesis of SystemC applications leads to natively executing applications that can run orders of magnitude faster than baseline simulation and several times faster than PC simulation.



We introduce a transparent, server-side just-in-time synthesis framework to the SystemC-on-a-Chip framework that can override software emulation of SystemC applications and instead execute the application natively, yielding orders of magnitude speedups improvement over software emulation, and faster performance than native PC simulation. We demonstrate the usefulness of the framework by developing a full prototype for the Xilinx Virtex4 Ml403 development board using partial reconfiguration.

6.2 Related Work

There has been a large body of work devoted to decompilation. Many efforts used decompilation to port legacy binaries to updated computer architectures, to convert

binaries between two different languages, and to document and maintain applications written in assembly. Software developers have also used decompilation as a debugging tool for assembly code, by recovering a high-level representation that is easier to check for errors. A more complete treatment of decompilation can be found in [31]. We use decompilation techniques for synthesis first proposed by Stitt [131].

Increasingly powerful FPGA platforms (and more usable tools) have made partial reconfiguration more attractive and the subject of much research. Horta [71] describes the use of dynamic plug-ins for FPGAs with partial run-time reconfiguration support. Forin [132] uses partial reconfiguration to create an extensible MIPS-like processor called eMips. Emmert [44] uses partial reconfiguration for fault tolerance purposes.

Approaches for dynamic software optimization and binary translation have been proposed to maintain binary compatibility, to reduce compilation time, and to perform runtime optimizations. Dynamo [10] is a dynamic optimization approach that profiles an application during execution to determine frequent paths, optimizes the code for those paths, and stores the optimized code in a special fragment cache. When software execution reaches a frequent path, the microprocessor fetches instructions from the fragment cache to execute the optimized code. FX!32 [28] dynamically translates x86 binaries into Alpha binaries by first emulating the application and profiling to determine frequent regions that should be translated to native Alpha instructions. BOA [56] dynamically translates PowerPC instructions into smaller microinstructions that can be more easily pipelined and scheduled in parallel. BOA also detects frequent paths, performs path-specific optimizations, and translates paths from PowerPC code into native

VLIW code. The IA-32 execution layer for the Itanium microprocessor uses software to convert IA-32 instructions into native Itanium instructions [11]. Warp processing [93][94][129][131] has demonstrated the feasibility of performing binary synthesis at runtime, allowing binary synthesis to also take advantage of runtime information to optimize hardware.

6.3 Just-in-Time Synthesis

6.3.1 Server-Side Synthesis Framework

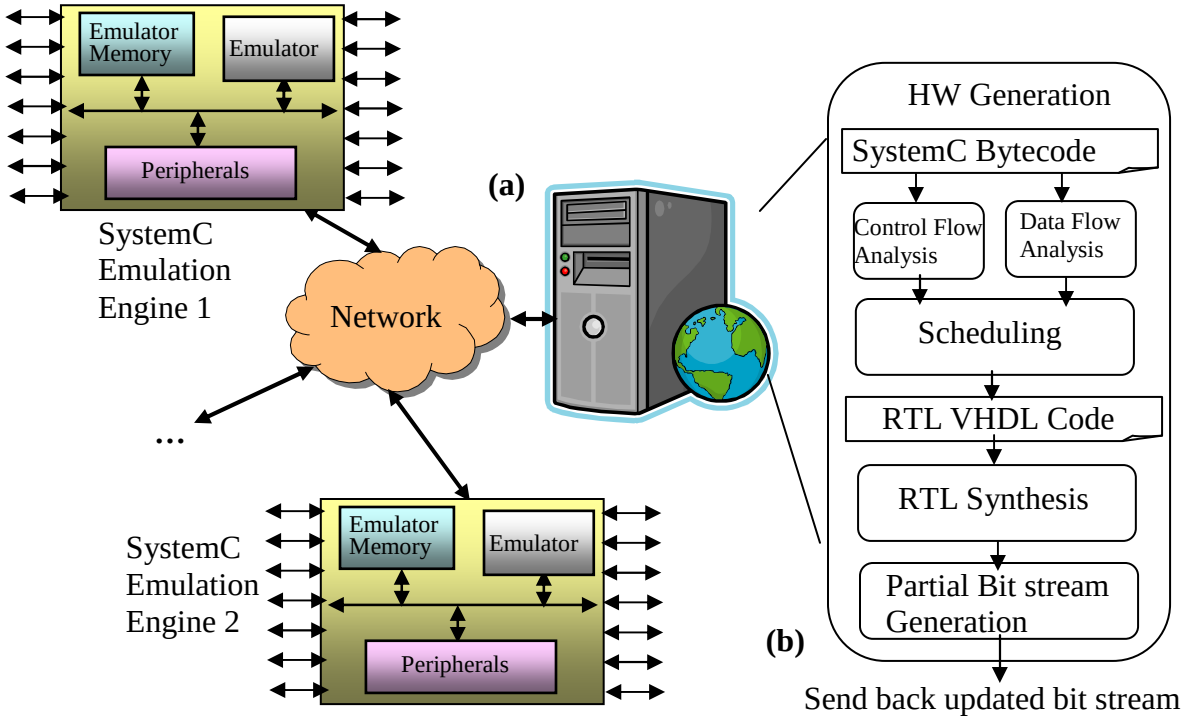
We investigated two options for synthesizing SystemC applications native platform execution. One option is to utilize another local processor within the SystemC-on-a-Chip framework to perform synthesis, place and route, and mapping for the platform. Lysecky and Stitt [93][94][131] showed that the computationally complex algorithms used by synthesis and place and route can be made lean enough to run on a small Arm7 processor. However, they assumed a simple architecture model with a much-reduced complexity FPGA platform. Modern FPGAs are so complex that they render on-chip synthesis with a small embedded processor infeasible, and instead require powerful computing platforms to perform the synthesis process.

Another option is to use an external server to perform the complex synthesis process. The server-side approach requires an internet connection, but this is plausible as most modern FPGAs platforms have existing internet connectivity, or can be programmed to have such behavior. Figure 38(a) shows server-side synthesis for SystemC-on-a-Chip. A SystemC application initially runs on the platform using the

emulation techniques described in previous chapters. If additional performance is required, the emulation engine sends the currently executing SystemC bytecode through the internet connection to a remote server. The remote server converts the SystemC bytecode to a circuit representation, performs synthesis, place and route, and mapping, and finally sends the updated bit stream back to the FPGA platform. The new bit stream overrides execution of the emulation engine, and executes as a native implementation. The server approach is not limited to an external remote server. The approach also works in the case where an FPGA platform is directly connected to a PC platform, like Intel's QuickAssist Technology, in which case the PC can perform synthesis externally.

There are several advantages to performing synthesis for SystemC applications running on SystemC-on-a-Chip. The first is performance. After some initial time spent sending the SystemC bytecode to the server to be converted, synthesized, and sent back, the SystemC application can potentially run orders of magnitude faster than when running the base SystemC emulation engine, and also faster than simulating the same SystemC application on a desktop PC. The second advantage is the synthesis process is completely transparent to the SystemC application designer. The SystemC application designer does not to need to use costly, difficult, and hard-to-use synthesis flows that often differ greatly from traditional compilation flows (of which the SystemC-on-a-Chip flow follows). Instead, the SystemC application *immediately* runs on the emulator using a more traditional compiler, and if needed, will transparently synthesize itself to a circuit that takes advantage of the available FPGA resources.

Figure 38: Just-in-Time Synthesis SystemC-on-a-Chip framework**.** (a) The server responds to requests from SystemC-on-a-Chip platforms that require native execution speeds. (b) The server *decompiles* the SystemC bytecode, recovers the high-level information, and synthesizes a circuit tuned to that platform's available resources.



There are also a several disadvantages to the server-side just-in-time synthesis approach. One disadvantage is that the server-side synthesis approach can be slow. In extreme cases, the server may finish generating a native platform *after* the emulated circuit has finished executing. Synthesis, place and route, and mapping are complicated NP complete problems, and often require extensive resources (and time) to complete. For such situations, the SystemC-on-a-Chip platform would not be able to take advantage of the native implementation. The server-side approach *can* be beneficial for long-running SystemC applications or scenarios where a SystemC application repeatedly executes. While the first execution instance may not take advantage of the newly-created native

bitstream, repeated executions could immediately take advantage, resulting in on average high performance SystemC execution. Such cases lend well to a server-side approach.

6.3.2 Decompilation and Synthesis

The server-side synthesis framework requires methods to recover high-level information from the SystemC bytecode in order to generate a high-performance circuit. With modifications, we use Stitt's binary synthesis/decompilation [131] tools to recover and generate RTL VHDL for the SystemC application. Shown in Figure 38(b), the decompilation tools perform dataflow and control flow analysis, scheduling, and generate RTL VHDL to input into commercial synthesis and place and route tools.

The decompilation tools required some modifications. The major modification required modifying the tools to accept a binary (the SystemC bytecode) that already has explicit parallel constructs, and maintain those constructs through the optimization process. For instance, if a SystemC application was written using two explicit processes with custom communication, the decompilation tools must preserve those connections, while still correctly optimizing the behavior of each process. A more complete description of the optimizations performed by the decompiler can be found in [131].

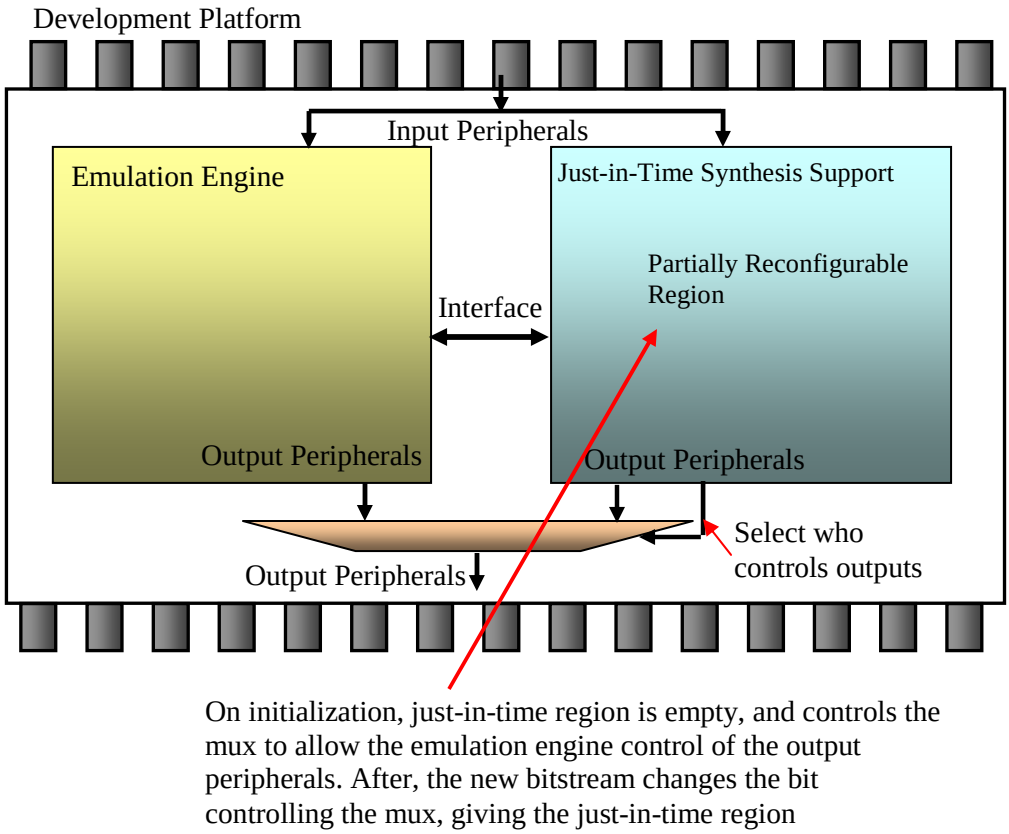
The output of the decompilation framework is RTL VHDL. The VHDL is input into a commercial synthesis tool framework that generates a partial bit stream. The partial bit stream is finally sent back to the SystemC-on-a-Chip framework, the bit stream is downloaded, and begins execution as a custom implementation.

6.3.3 SystemC-on-a-Chip Architectural Support

We considered two options on how best to override the emulation engine once synthesis has completed. The first option is to create a fully custom bit stream that completely erases the emulation engine architecture and takes control of all of the platform resources once downloaded. This approach has the advantage that the SystemC application has access to 100% of the resources on the chip, giving potential for a higher performance implementation. One disadvantage is the platform must store the emulation engine framework in memory for additional uses of the platform, or be forced to download the original bit stream for future uses. Another disadvantage is there is the possibility that the synthesis job cannot create a custom bit stream for the application, possibly due to area constraints, timing constraints, etc. in which case it might be more advantageous to synthesize *part* of the SystemC application to a custom implementation, and leave the rest to run on the emulation engine.

We chose an approach where the SystemC emulation engine remains persistent on the development platform, and is overridden at the right time by a *just-in-time synthesis architecture* supported by partial reconfiguration, and shown in Figure 39. For the common case where the SystemC-on-a-Chip framework is executing on an FPGA platform, a portion of the FPGA platform is now dedicated to be a partially-reconfigurable region called the *just-in-time synthesis* support section. The just-in-time synthesis support section statically interfaces to the emulation engine, and to a static multiplexor that multiplexes the control of the output peripherals. On platform initialization, the just-in-time synthesis support is mostly blank, with the exception of a

Figure 39: Just-in-Time Synthesis Architectural Support. The partially-reconfigurable region multiplexes the use of the input and output, and can override the execution of the emulator once programmed.



few control bits that control the multiplexor to choose the emulation engine as having sole control over the output peripherals, and telling the emulation engine the just-in-time synthesis section is empty.

The SystemC emulation engine sends a request to the server-side synthesis tool to create a custom implementation. The server-side synthesis tool is aware of the partially-reconfigurable configuration, and not only synthesizes a custom implementation of the SystemC application, but also generates small pieces of control logic that interface with the SystemC emulation engine, and which switch control of the output peripherals to the just-in-time synthesis region.

The interface between the SystemC emulation engine and the just-in-time synthesis region serves several purposes. The first is to instruct the emulation engine that the just-in-time synthesis region is ready to execute (thus stopping the emulation engine). The second purpose is to transfer a notion of state between the emulated application, and the newly-created custom implementation ready to run in the just-in-time synthesis region. Without a transfer of state, the newly-created custom implementation must begin running, and lose the potential for starting where the emulated application left off. The just-in-time synthesis region also uses the interface to tell the emulation engine whether it is emulating all or part of the SystemC application. If the just-in-time synthesis region is executing the entire SystemC application, the just-in-time synthesis region will take complete control over the output peripherals, and also minimally communicate with the emulation engine. If it is only executing part of the SystemC application, the just-in-time synthesis region registers the correct SystemC processes into the emulation engine so as to maintain correctness, and to instruct the emulation engine to use the custom implementations of the desired SystemC processes.

Using the partially-reconfigurable approach, the SystemC emulation engine can persist in the background, potentially allowing other SystemC applications to run as a custom implementation uses the partially reconfigurable region. The approach also allows the server-side synthesis tools to selectively choose how best to use the SystemC-on-a-Chip platform, either creating full custom versions of the SystemC applications, or only synthesizing parts, and emulating the rest. The tradeoffs include performance, complexity of the design, and how many applications the SystemC-on-a-Chip framework

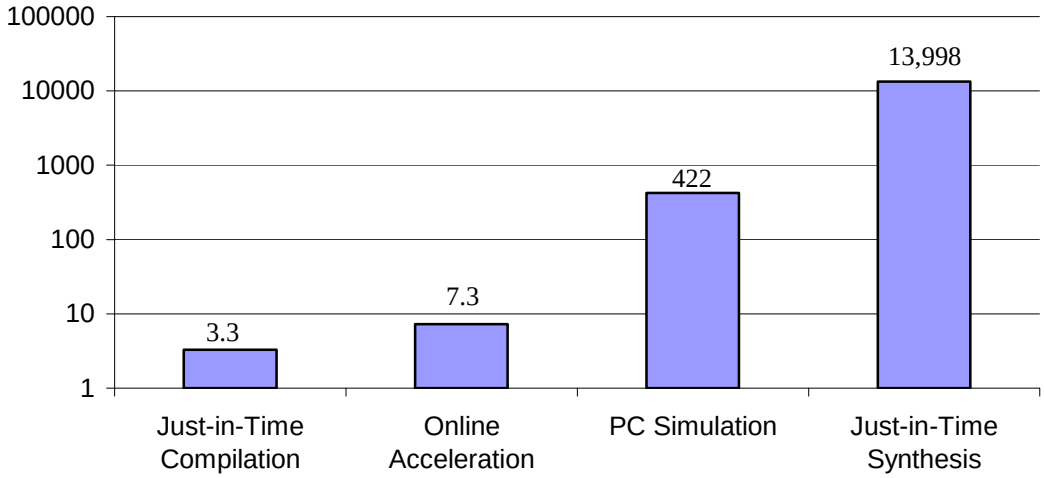
can independently support. One disadvantage of this approach (compared to deleting the emulation engine) is the emulation engine itself is consuming resources that might best be used by a custom implementation of the SystemC application.

6.4 Experiments

We built a full prototyping framework to test and demonstrate the usefulness of the just-in-time server-side synthesis framework for SystemC-on-a-Chip. We built our prototype using the Xilinx Virtex4 Ml403 development board. We used the 9.2 series of Xilinx's ISE, EDK, and PlanAhead tools to implement the partially reconfigurable region for just-in-time synthesis support. We implemented the architectural support for just-in-time synthesis using an additional several hundred lines of VHDL (mostly for bus macro instantiation), and a two hundred lines of C for the emulation engine.

We built the server-side framework by modifying Stitt's binary synthesis framework to support SystemC bytecode. The original binary synthesis tool was written using approximately 30,000 lines of code; the additional SystemC bytecode support required approximately 2,000 extra lines of code. For our experiments, we synthesize the entire SystemC application, and leave deciding how best to synthesize only parts of the SystemC application to future work. The binary synthesis tools generated RTL-level HDL code that served as input into Xilinx's ISE and PlanAhead tools for synthesis, placement, routing, and partial bit stream generation. Currently the server-side framework only supports one synthesis request at a time, but will be augmented to allow

Figure 40: Speedups compared to base SystemC emulation for some common image processing filters. Factoring out the time required to synthesize the SystemC application, just-in-time synthesis is almost 14,000X faster than base emulation, and 30X faster than PC simulation



for multiple requests in the future. The server-side decompilation framework was built on a 2GHz PC using 2GB RAM.

We implemented a suite of image processing applications in SystemC, including an edge detector, an emboss filter, and a sharpening filter. Each image processing algorithm was implemented using various numbers of processes to test the correctness of the decompilation framework, and its ability to generate the high performance circuit implementations given different SystemC implementations. Each SystemC application was written using the synthesizable subset of SystemC, guaranteeing the decompilation framework could create a circuit. Future work might investigate decompiling a less-constrained version of SystemC bytecode

Figure 40 shows a comparison of the speedups achieved by online emulation acceleration, just-in-time compilation, PC simulation, and just-in-time synthesis compared to base SystemC emulation. The data shown is for only one of the image

filters. The data for the other image filters was very similar. As shown in earlier chapters, just-in-time compilation earns modest speedups compared to base SystemC emulation. PC simulation is approximately 400x faster than base emulation, but part of this speedup is due to the disparate clock speeds between the PC (running at 2 GHz) and the base emulation engine (running at 100 MHz). After just-in-time synthesis, the native SystemC application runs approximately 14,000x faster than base emulation and approximately 30x faster than PC simulation. The speedup over the base emulation engine is due to a completely parallel implementation the server created (for each implementation, the server side decompilation framework was able to recover and create the same circuit that we hand-created from the same SystemC application). Just-in-time synthesis was 30x faster than PC simulation because the SystemC application on the PC itself is wrapped within a simulation kernel that causes slowdown. The 14,000x performance improvement does assume synthesis took zero time. In all three cases, the decompilation and synthesis process took approximately 20-30 minutes. In these particular cases, the SystemC application still ran on the emulation engine until the new bit stream was created. Once bit stream generation finished, there was a noticeable quantitative and visual difference in how fast the SystemC-on-a-Chip performed.

Chapter 7

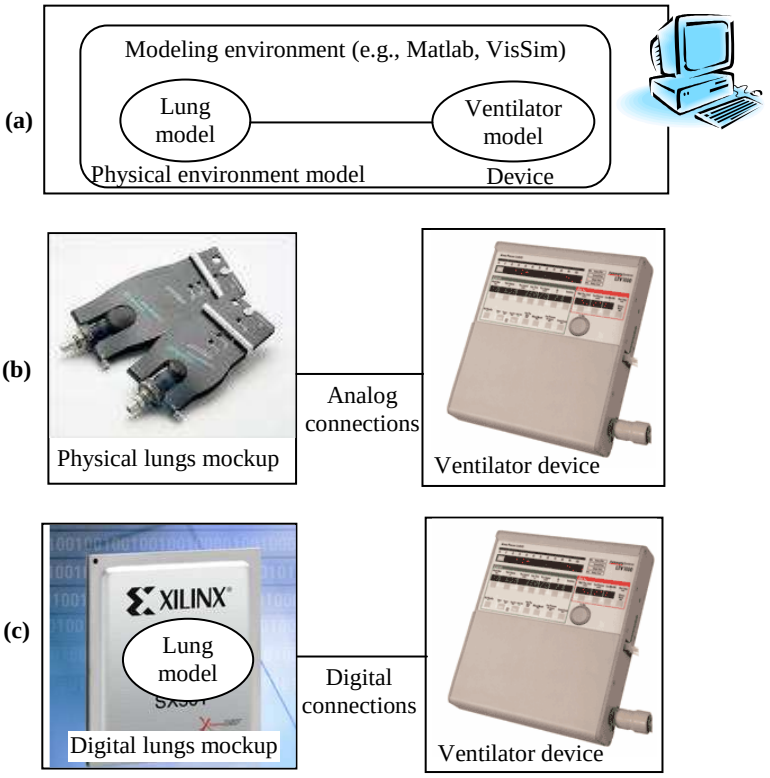
Controlling Time with SystemC Emulation

7.1 Overview

Emulation of SystemC applications allow for portable execution over a variety of devices and platforms, saving time and programming effort, and allowing a designer the opportunity to create a device-independent FPGA application. An additional advantage of emulation is the power to start, stop, and control time. Controlling time might allow a designer to debug a SystemC application in-system, giving access to internal variables, signals, and state of the SystemC application as it running and connected to real peripherals. Such control might be beneficial in a number of domains. We will use the development of physiological models for medical device testing as a case study into the usefulness of time-controllable SystemC emulation.

Medical device software is commonly developed using one of several approaches. One approach involves modeling on a PC, shown in Figure 41(a). A designer develops models for both a medical device, such as a pacemaker or ventilator, and for the physiological system with which the device interacts, such as a heart or lung. Such a modeling approach supports rapid device software changes, supports simulations that

Figure 41: Approaches to integrating an embedded device with the physical environment during design: (a) system model, (b) physical mockup, (c) digital mockup**.**



execute faster (or slower) than real-time, and avoids potential safety issues that could arise when interacting with an actual physical system.

A second approach, used after or instead of the modeling approach, runs the medical device software on the actual medical device, which is connected to a physical mockup on the physiological system. Physical mockups range from simple structures, such as a balloon representing a lung, to computerized mechanical parts that dynamically react [Michigan Instruments], that can be set to mimic a range of conditions, and whose internal sensors can be interfaced to a computer for analysis and debugging.

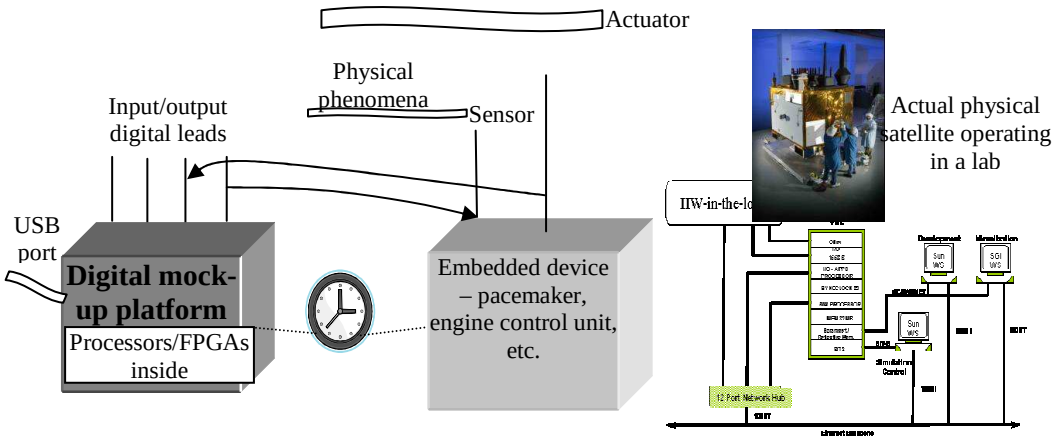
One disadvantage of interfacing to off-the-shelf physical mockups is the inability to adapt to new features, especially features not easily mimicked via mechanical means.

For example, a future ventilator may sense human-generated nitric oxide concentrations (recently discovered to be significant in respiratory issues [120]) and adapt the output gas mix in response. However, no existing computerized mechanical test lung generates nitric oxide, nor is it clear how to create one.

An alternative is to connect the actual medical device to a digital mockup of the physiological system. A *digital mockup* is a behavioral model that emulates the physical system. In such a case, the medical device software executes as if it were interacting with a physiological system, but in fact all interaction is through a digital interface, as in Figure 41(c). We consider a digital mockup platform with a sensor/actuator bypass method of integration [122] as shown in Figure 42(a). Under this scheme, the digital mockup taps directly into the information packets that carry the control and data bits to and from the device's sensors and actuators. The digital mockup includes models of the physiological system, of the physical connections between the device and physiological system, and of the sensors and actuators. A supervisory system coordinates execution of the digital mockup and medical device. The sensor/actuator bypass method is in harmony with methods used in industrial "hardware-in-the-loop" practice today, shown in Figure 42(b). Digital mockups combine the flexibility and faster-than-real-time execution benefits of PC simulation models with the advantages of developing software on an actual medical device. Digital mockups are also potentially less costly than physical mockups, which can cost tens of thousands of dollars.

However, no common methodology exists for creating digital mockups. Towards this end, we sought to develop a general approach for time-controllable digital mockup execution. Digital mockups can be implemented through a variety of methods and on a variety of different platforms, trading off performance, complexity, size, and accuracy. While a medical device software developer may run a digital mockup directly on the physical development platform for increased performance and/or accuracy, another approach is to run the digital mockup on top of *virtualized platform* like an in-circuit emulator. By varying the rate at which the digital mockup generates samples, the digital mockup can still run faster than or in real-time to interface with the medical device software under test. A virtualized environment can also provide built-in and unobtrusive debug capabilities, allowing the designer to stop, start, and step through the digital mockup to examine important system variables. The virtualized environment can exploit

Figure 42: Digital mock-up platform: (a) The bypass method of integration taps directly into the digital information packets that indicate the data/control values to/from the device sensors/actuators, (b) the method matches hardware-in-the-loop approaches used in industrial practice (*figure courtesy of Boeing, 2009*).



the digital mockup's explicit notion of a simulated time step, allowing the designer to monitor the mockup using *time-controllable* debug. For example, a medical device software developer may wish to step through a wheezing lung that just coughed one time step at a time (physiological models are defined to compute the next system values in time based on a *delta time* parameter), observing subtle differences in pressure and volume in the digital lung that might not easily be observed when running in real-time.

We describe a *time-controllable* SystemC-on-a-Chip framework that allows a medical device software developer to interface a medical device to a SystemC-based digital mockup, and start, stop, profile, and advance execution using explicit time-granularized debug commands. This is contrasted to a more traditional debugging approach, where debugging is performed at the *instruction* granularity, and which does not include an explicit notion of time. A time-granularized approach is generally more useful for physiological digital mockups, and provides a more powerful abstraction for developing and testing medical device software.

7.2 SystemC for Synchronized Physiological Models

There are a number of approaches to capturing and implementing physiological systems models. Physiological systems are usually first modeled using systems (hundreds or thousands) of partial and ordinary differential equations. The model can then be captured for PC execution using a particular programming language, typically an expressive mathematical language like MML, Matlab, or VisSim.

Figure 43: Capturing physiological models in SystemC. (a) Portion of a mathematical model of the human lung. (b) Description of the model in SystemC. (c) Description using POSIX threads. The POSIX threads approach requires implementing explicit lock-stepping mechanisms that detract from the model's readability.

(a)

$$C_{br} = Q_{br} / V_{br}$$
$$F_{br} = (P_{air} - P_{br}) / R_{br}$$
$$dQbr/dt = F_{br} * (C_{air +} C_{br}) + F_{alv} * (C_{alv} - C_{br})$$

(b)

```
Class model : public sc_module {
  sc_in_clk clock;

  integrator Q_integ;

  sc_signal<sc_uint<32> > Qbr, Qbr_t;
  sc_signal<sc_uint<32> > Cbr, Fbr;

  SC_CTOR {
    Q_integ.clock(clock);
    Q_integ.func(Qbr_t);
    Q_integ.dt(dt);
    Q_integ.out(Qbr);

    SC_METHOD(cbr_func);
    sensitive << clock;
    SC_METHOD(fbr_func);
    sensitive << clock;
    SC_METHOD(qbr_t_func);
    sensitive << clock;
  }

  void cbr_func( void ) {
     Cbr = Qbr / Vbr;
  }
  void fbr_func( void ) {
     Fbr = (Pair – Pbr) / Rbr;
  }
  void qbr_t_func( void ) {
     Qbr_t = Fbr * (Cair + Cbr) + \
             Falv * (Calv – Cbr);
  }
};
```

SystemC implementation much closer to the

Extraneous code to implement synchronous locksteps detract from actual model

(c)

```
int cbr,fbr,qbr_t;
sem_t  timestep_done,cbr_done;
sem_t fbr_done,qbr_t_done;

void * Cbr( void * arg ) {
   while (1) {
      sem_wait(&timestep_done);
      cbr = Qbr / Vbr;
      sem_post(&cbr_done);
   }
}

void * Fbr( void * arg ) {
   while(1) {
      sem_wait(&timestep_done);
      fbr = (Pair – Pbr) / Rbr;
      sem_post(&fbr_done);
   }
}

void * Qbr_t( void * arg ) {
   while(1) {
      sem_wait(&timestep_done);
      sem_wait(&cbr_done);
      sem_wait(&fbr_done);
      qbr_t = fbr*(Cair + cbr) +
Falv*(Calv – cbr);
      sem_post(&qbr_t_done);
   }
}

void * ClockTick( void * arg ) {
   while(1){
      sem_wait(&qbr_t_done);
      sem_post(&cbr_done);
      sem_post(&fbr_done);
      sem_post(&qbr_t_done);
      sem_post(&timestep_done);
   }
}

int main(){
   pthread_t pCbr;
   pthread_t pFbr;
   pthread_t pQbr_t;
   …
   pthread_create(&pCbr);
   pthread_create(&pFbr);
   pthread_create(&pQbr_t);
   pthread_join(pCbr, NULL);
   pthread_join(pFbr, NULL);
   pthread_join(pQbr_t, NULL);

   return 0;
}
```

Another method to capturing physiological systems models is to use SystemC. SystemC is a set of libraries built on top of the C++ language that provides an event-driven simulation kernel, allowing a designer to simulate a number of concurrently executing processes, and which supports precisely-timed communication based on simulated time. SystemC is a natural fit for capturing physiological systems models for a number of reasons. The equations that represent most physiological systems are naturally expressed as a number of concurrently executing interconnected processes that execute in lockstep. Digital physiological mockups implemented in SystemC have the added advantage that freely available SystemC simulation environments exist that enable comprehensive PC testing. Further, the developer can run SystemC on a real development platform using an in-circuit emulation approach like *SystemC-on-a-Chip* [Sirowy], with the advantage that the SystemC-based digital mockup executes with real peripherals, and with real devices, like medical device platforms.

While solutions can be implemented in other parallel programming paradigms like POSIX threads or Java threads that also operate with precise timing constraints, physiological models are more naturally represented in SystemC, where lock-stepped execution is an intrinsic part of the language. A SystemC description can require less code, is more readable, and is also more extendable. Figure 43(a) shows a portion of a human lung model captured with three interconnected equations. Figure 43(b) shows the SystemC description and Figure 43(c) shows a more traditional POSIX–based parallel programming description of the model. The POSIX threads approach requires describing explicit tightly-coupled, time lock-stepping mechanisms that make the description more

difficult to read, maintain, and extend. Additionally, there is no clear way to step through a POSIX implementation at the simulated time level without further introducing extraneous code into the model. Matlab can also model a number of interconnected equations using a mathematical approach, but like POSIX and Java descriptions, Matlab does not support explicit timing constructs, and debugging is still performed using standard instruction-granularity debug features.

7.3 Related Work

Pimentel and Tirat-Gefen [112] developed real-time digital mockups that interfaced to medical devices by connecting symmetric D/A (digital-to-analog) and A/D (analog-to-digital) cards to each side. Previous work by Sirowy [122] focused on modest modifications to the medical device hardware and software such that a digital mockup could be connected directly. Sirowy's approach still allows the addition of D/A and A/D attachments, but with the added advantage of allowing a designer to completely stay in the digital domain, and to accommodate situations where D/A or A/D conversions are complex (e.g., in the case of gas generation or sensing). Other researchers have developed real-time physiological models [20] with a focus on describing the necessary architectures to achieve real-time.

Several research efforts have emphasized creating and cataloging detailed physiological models [79][107][135]. Those models are targeted for PC-based simulation, yet could be used as a basis for digital mockups. Further, many physiological

models are highly complex, often requiring hours or days to simulate a few seconds [96]. Our initial focus is on real-time digital mockups.

There has been much work in the domain of synchronization mechanisms for distributed systems. Lamport [87] describes methods to order events in a distributed system. Kopetz [85] also specifies clock synchronization methods, but describes techniques used for a more general network topology. In contrast, our system consists of only two directly connected components, and thus is a simpler synchronization problem because uncertainties in a general network need not be considered

There have been some efforts that focus on making time an explicit first class entity when designing and programming systems. Lee [90] calls for the need to bring time to the forefront of programming languages and models, especially with the rise in cyber physical systems research. Lee [89] presents a taxonomy detailing several timing properties that should be explicitly expressed in programming languages for timing oriented behaviors. Benini [14] develops methods for performing time granularity debugging by calculating time through knowledge of the system's clock speed and the number of cycles between breakpoints

7.4 Time-Controllable Digital Mockup Execution

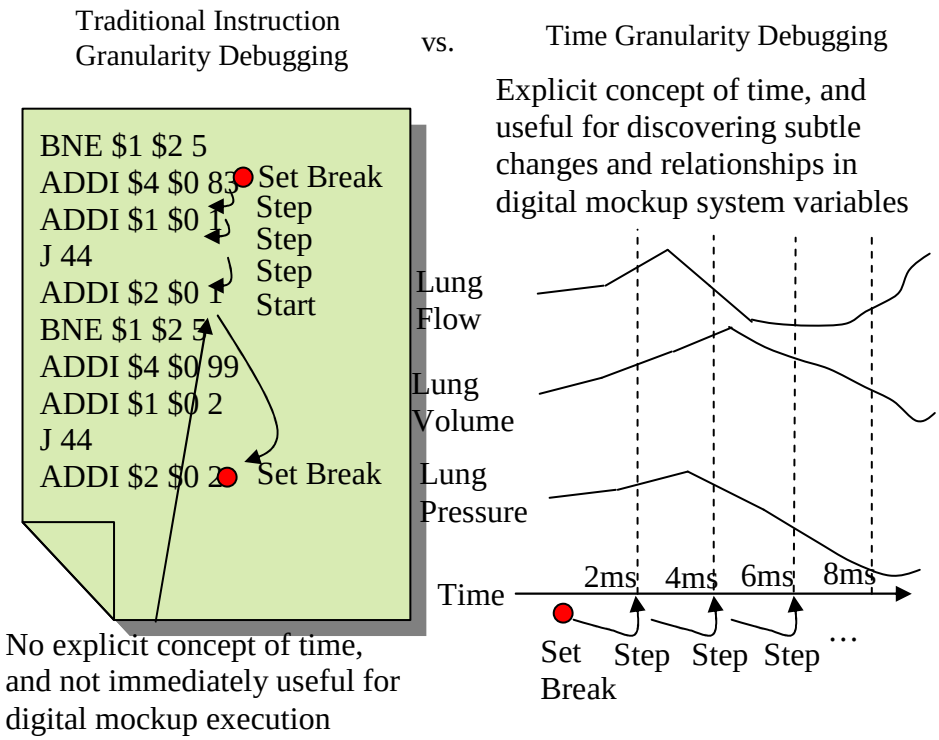
The SystemC-on-a-Chip framework can be augmented to give the developer unobtrusive *time-granularized* debug and test capabilities. In contrast to the standard *instruction granularity* debugging approaches, the SystemC-on-a-Chip framework can start, stop, and step a digital mockup's simulated time, advancing time forward as slow or

fast as the developer requires. Figure 44 highlights the differences between instruction level and time granularity debugging.

Time-controllable SystemC emulation possesses a number of advantages for digital mockup execution. First, the medical device software developer can control time by running simulations between the digital mockup and medical device faster than real-time. Running faster than real-time might allow a developer to simulate a night's worth of breathing in just a few hours, or make possible the ability to test several different control algorithms on the medical device in a timely manner. The ability to run faster than real-time is of course determined by the delta time step at which the digital mockup is executing and how powerful the underlying platform is, but for many examples, running faster than real-time is feasible.

Another advantage is the debugger can step through the execution of the digital mockup at the level of time granularity the digital mockup computes. Stepping using an explicit notion of time might allow a medical device software developer to step through a simulated cough of a digital lung mockup, a heart murmur in a digital heart mockup, or other anomalies and subtleties that might not otherwise be seen, or easily observed, executing at faster speeds

Figure 44: Time-Controllable Debugging. In contrast to traditional instruction granularity debugging, time granularity debugging allows a developer to monitor system variables by explicitly controlling simulated time.



7.5 Experiments

We conducted several experiments to test the feasibility of capturing digital mockups using SystemC, interfacing those models using the SystemC-on-a-Chip framework to a medical device, and testing the ability to control time by configuring faster than real-time execution and incrementally stepping through time. We built a SystemC-on-a-Chip framework to run on a Xilinx Virtex5 FPGA platform. We wrote the SystemC-on-Chip framework in approximately 20,000 lines of C, C++, and VHDL. The main emulation kernel was built on top of a Xilinx Microblaze processor, with custom bytecode accelerators [Sirowy] built on the native FPGA fabric for increased performance. We also

built SystemC-on-a-Chip frameworks for a Xilinx Virtex4 Ml403 platform, and a Xilinx Spartan 3E platform. The Virtex4 implementation was built on top of a PowerPC-based system. All of the SystemC-on-a-Chip implementations could execute the same SystemC bytecode without recompiling for any particular platform.

We described a number of physiological models in SystemC that we obtained from the NSR Physiome Project. Figure 45 shows a portion of the SystemC code used to capture a two-compartmental respiratory system, one bronchial compartment and one alveolar compartment. The respiratory system model computes airway pressure, lung pressure, flow, and volume values for a healthy human lung at a simulated time step of approximately 4 milliseconds. The respiratory system was modeled using a series of four ordinary differential equations, and nine linear equations. We modeled the respiratory system using approximately 400 lines of behavioral SystemC. The SystemC description compiled to approximately 500 lines of SystemC bytecode, and compiled through the SystemC bytecode compiler in less than a second.

Figure 45: SystemC Implementation of a two-compartment respiratory system digital mockup.

```
#include “systemc.h”

template<int bit = 32>
class integrator : public sc_module {
  sc_in_clk clock;
  sc_in<sc_uint<32> > dt;
  sc_in<sc_uint<32> > funct;
  sc_out<sc_uint<32> > out;

   sc_signal<sc_uint<32> > reg;

  integrator( sc_module_name n ) sc_module (n)
{
    sc_method(process);
    sensitive << clock;
  }
  void process(void) {
    reg = funct.read() * dt.read() + reg;
    out.write(reg);
  }
};

class model : public sc_module {
  sc_in_clk clock;
  sc_in<sc_uint<32> > qalv, valv, qbr, vbr;
  sc_out<sc_uint<32> > qalv_t, valv_t;
  sc_out<sc_uint<32> > qbr_t,    vbr_t;

  sc_signal<sc_uint<32> > pbr, palv, fbr;
 sc_signal<sc_uint<32> > falv, cbr, calv;

  model( sc_module_name n ) : sc_module(n)   {
    SC_METHOD(pbr_func);
    sensitive << clock;
    //…
    SC_METHOD(qalv_t_func);
    sensitive << clock;
  }

  void pbr_func(void) {
    int COM_BR = 0x100;
    int VBR_0 = 0x9600;
    pbr = vbr.read() - VBR_0 / COM_BR;
  }

  //…

  void qalv_func(void) {
    qalv_t.write(falv * (cbr + calv));
  }
};
```

```
class top : public sc_module {
  sc_in_clk clock;
  sc_in<sc_uint<4> > buttons;
  sc_in<sc_uint<32> > memory_in;
  sc_in<sc_uint<8> > uart_rx;
  sc_out<sc_uint<8> > uart_tx;
  sc_out<sc_uint<32> > fb_h;
  sc_out<sc_uint<32> > fb_v;
  sc_out<sc_uint<32> > fb_data;
  sc_out<sc_uint<4> > leds;

  sc_signal<sc_uint<32> > Qbr_t, Qalv_t;
  sc_signal<sc_uint<32> > Vbr_t, Valv_t;
  sc_signal<sc_uint<32> > Qbr, Qalv;
sc_signal<sc_uint<32> > Vbr, Valv;
  sc_signal<sc_uint<32> > dt;

  model model_1;
  integrator<32> integrator_Qalv;
  integrator<32> integrator_Qbr;
  integrator<32> integrator_Valv;
  integrator<32> integrator_Vbr;

  top( sc_module_name n ) : sc_module(n)
  {
    dt.write(0x1);

    model_1->clock(clock);
    model_1->qalv(Qalv);
    model_1->qbr(Qbr);
    model_1->valv(Valv);
    model_1->vbr(Vbr);
    model_1->qalv_t(Qalv_t);
    model_1->qbr_t(Qbr_t);
    model_1->valv_t(Valv_t);
    model_1->vbr_t(Vbr_t);

    integrator_Qalv->clock(clock);
    integrator_Qalv->dt(dt);
    integrator_Qalv->funct(Qalv_t);
    integrator_Qalv->out(Qalv);

    //…

    integrator_Vbr->clock(clock);
    integrator_Vbr->dt(dt);
    integrator_Vbr->funct(Vbr_t);
    integrator_Vbr->out(Vbr);
  }
};
```

We executed the digital respiratory mockup on the Xilinx Virtex5 implementation of the SystemC-on-a-Chip development platform. At full speed, the SystemC-on-a-Chip platform could execute a full simulated time step in 1.6 milliseconds, or about 3X faster than real-time. We also modeled an alternate implementation of a lung that computes

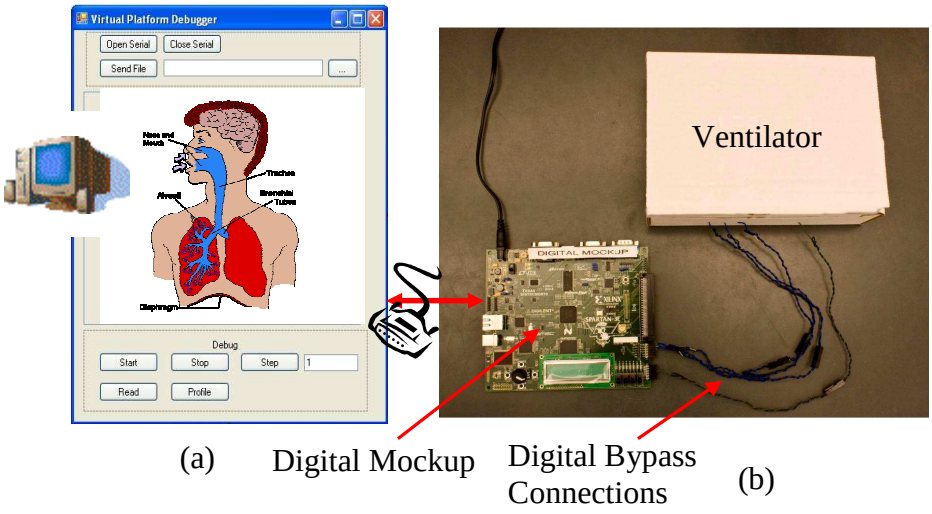
Figure 46: SystemC Digital Mockup Implementation Summary. Both respiration models were obtained from the NSR Physiome Project and manually converted to concurrently executing SystemC implementations.

Digital Mockup	# of Eqns.	# of ODEs	SystemC LOC	Simulate Dt	Simulated Freq
Alveolar Bronchial Lung w/ Gas Exchange	13	4	430 (Behavioral)	2^{-8} s	~800 Hz
First Order Non-Linear Lung	4	1	570 (Structural)	2^{-8} s	~600 Hz

concentration, lung mass, flow, bronchial pressure, and alveolar pressure. The system consisted of four equations, one of which was an ordinary differential equation. We modeled the system using 600 lines of structural SystemC. The SystemC bytecode compiler compiled the model to approximately 300 lines of SystemC bytecode. While the model computed fewer equations than the previous model, the SystemC-on-a-Chip framework took longer to compute one time step because the model was captured structurally with more interconnected processes. Figure 46 summarizes the models.

Figure 47 illustrates one of our prototype setups for a ventilator and the respiratory system digital mockup. The digital mockup communicates to the ventilator through four dedicated serial connections and one synchronization channel. The dedicated serial connections bypass the ventilator's airway pressure, lung pressure, flow, and volume transducers. The synchronization channel is used to ensure that both models are sampling at the same frequency. Since the digital mockup can simulate time 3X faster than real-time when running on the virtualized platform, the medical device and digital mockup use the synchronization channel to agree on a rate at which both devices operate

Figure 47: Medical device(ventilator) and digital mockup(lung) prototype setup. (a)The digital mockup can be time-controlled using a simple PC-based debug interface. (b)The digital mockup and ventilator communicating digitally.



[Sirowy]. The rate at which the devices operate is user-defined by a separate PC-based debug interface, and shown in Figure 47(a).

We tested the usefulness of the time controllability of the test platform by developing a prototype PC-based debugging application. The debugger is able to stop, start, and advance time at the smallest simulated time rate the digital mockup can achieve (approx 4 milliseconds). Figure 47(b) shows that even with a simple debugging interface we can step through several steps of lung breathing, monitor pressures, volumes, and gas concentrations, and also make sure the ventilator software is performing correctly. The time-controllable debug commands given to the digital mockup propagate to the ventilator via the synchronization channel.

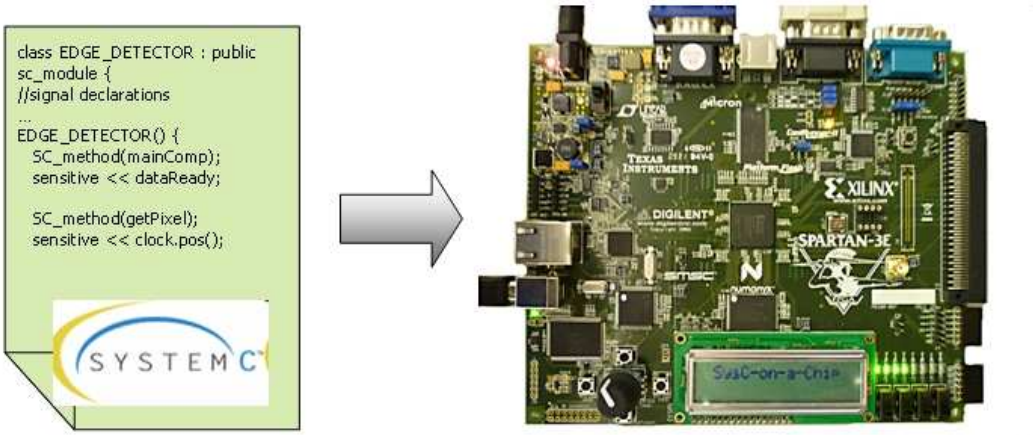
Chapter 8

SystemC Emulation in the Classroom

8.1 Overview

Computing was originally dominated by desktop and hence data-oriented systems. However, embedded and hence time-oriented systems, which must measure input events or generate output events of specified time durations, or must execute at regular time intervals, are increasingly commonplace. Blinking a light on and off for 1 second

Figure 48: SystemC-on-a-Chip in the classroom.



represents a “Hello World” example of a time-oriented system. Time-oriented programming differs significantly from the more common data-oriented programming, and developing correct maintainable time-oriented programs is challenging.

Similarly, many embedded systems possess spatial connectivity, wherein component A is connected to B, component B is connected in component C, etc, and requires a fundamentally different model and structured approaches for teaching correctly.

We can address both the spatial and time-oriented requirements of many embedded systems using SystemC. We present a spatial and time-oriented approach to teaching embedded systems using SystemC. Our approach involves creating an easy-to-use front end for the SystemC-on-a-Chip framework for the popular Xilinx Spartan 3E board (shown in Figure 48), a website with a number of available materials for the instructor wanting to use the SystemC-on-a-Chip in the classroom, including a course worth of lab assignments.

8.2 Related Work

Several research projects attempt to improve engineering education. Hodge [70] introduces the concept of a *Virtual Circuit Laboratory*, a virtual environment for a beginning electrical engineering course that mimics failure modes in order to aid students in developing solid debugging techniques. The environment not only provides a convenient test environment, but also allows an instructor to concentrate more on teaching. Butler [22] developed a web-based microprocessor fundamental course, which

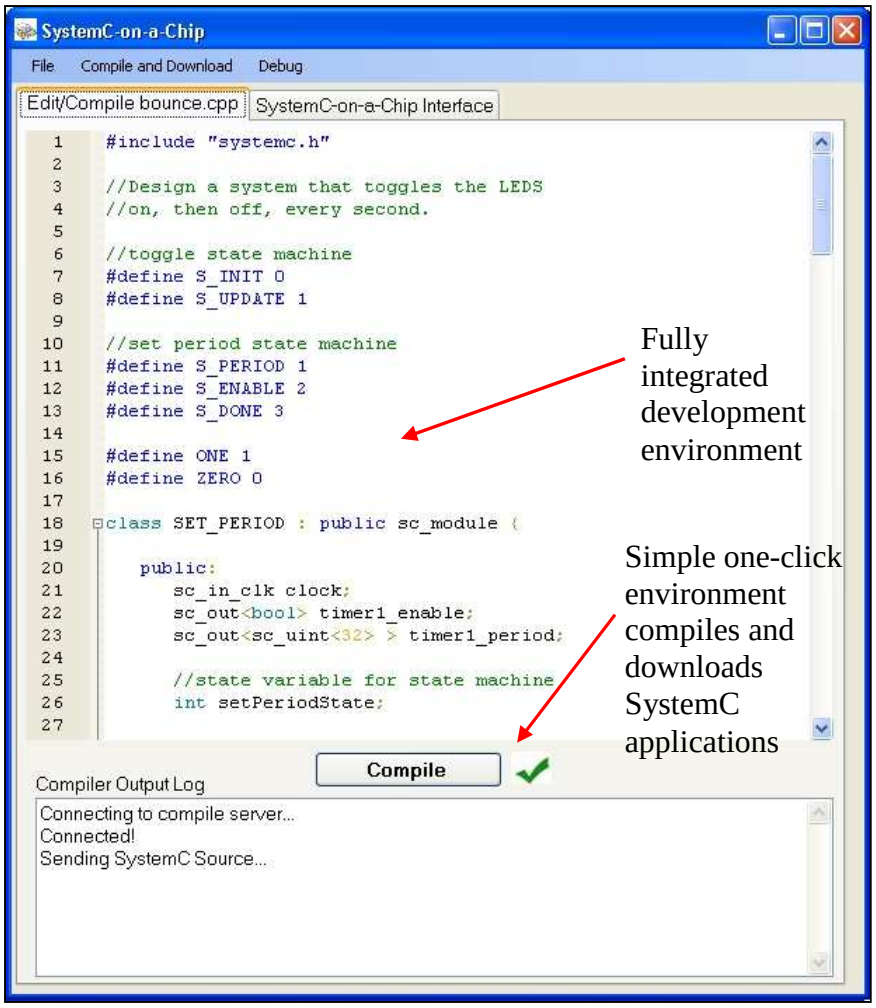
includes a *Fundamental Computer* that provides students in a first year engineering course a less threatening introduction to microprocessors and how to program.

Other researchers have concentrated on developing or evaluating computing architectures for beginning students or non-engineers. Benjamin [16] describes the *BlackFin* architecture, a hybrid microcontroller and digital signal processor. The architecture provides a rich instruction set based on MIPS with variable width data, and parallel processing support. Ricks [115] evaluates the *VME Architecture* in the context of addressing the need for better embedded system education. The Eblocks project [33] concentrated on developing sensor blocks that people without programming or electronics knowledge could connect to build basic customized sensor-based embedded systems.

A number of real time operating systems have been introduced to provide a higher level of abstraction between the application software and embedded hardware, including the open source eCos [42], and VxVorks and RTLinux from WindRiver [152].

There have also been several efforts to create virtual environments of microcontrollers suitable for running from the convenience of a standard desktop computer. The Virdes [144] virtual development system provides a virtual prototyping environment for anyone learning to program using the popular 8051/8052 microcontroller. Virdes ships with several already built layouts to blink LEDs, work with analog to digital converters, and a virtual UART and terminal. Images Scientific Instruments [75] developed a virtual system for prototyping PIC microcontrollers, while

Figure 49: Windows-based interface for programming SystemC-on-a-Chip.



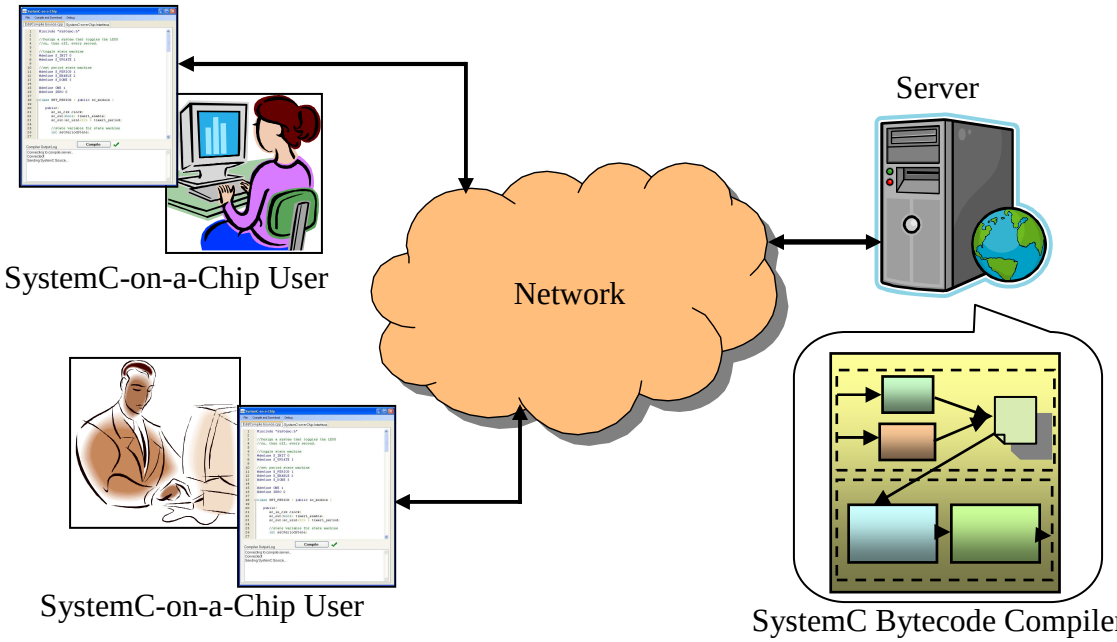
other work has concentrated on developing virtual peripherals [60] for the AVR microcontroller.

8.3 SystemC-on-a-Chip Software

8.3.1 Using the SystemC Bytecode Compiler

We considered a few approaches to distributing the SystemC bytecode compiler to students and teachers. The first approach was to make the SystemC bytecode compiler

Figure 50: Remote Compilation for SystemC-on-a-Chip.



source freely available, allowing students and teachers to install the compiler framework with no restrictions, and the freedom to make changes at their leisure. However, the SystemC bytecode compiler is currently difficult to install, Linux-based, and not desirable to setup. We instead chose to wrap the SystemC bytecode compiler framework in a simple, but full functional Windows interface, shown in Figure 49. The Windows-based environment showcases a full-featured editor, allowing students to begin coding immediately. The Windows-based approach is more familiar to most students, and allows more novice users to quickly begin. The Windows-based environment supports a simple compile interface, wherein a student simple clicks the big “Compile” button in the middle of the screen.

In contrast to most integrated development environments wherein the backend compiler is located on the local machine for which compilation is taking place, we take a

remote compilation approach. Modeled after approaches taken by companies like Tensilica [136], remote compilation for SystemC-on-a-Chip has a number of advantages, including a simpler and smaller Windows-based front-end, and the ability to make transparent updates to the compiler backend. Figure 50 shows the SystemC-on-a-Chip remote compilation framework. We can currently support dozens of concurrent users, allowing each to write and compile SystemC code as if the compiler was on the local machine. Such support enables classrooms of students to work concurrently. Such support is limited is though, and we are currently investigating approaches to reduce latency when multiple users begin overloading the compiler server.

8.3.2 Downloading SystemC to Development Platform

We take two approaches to downloading the SystemC bytecode to the development platform. In a previously explained approach, the user places the SystemC bytecode onto a USB thumb drive and inserts the thumb drive directly into the platform. The approach is simple, intuitive, and allows a student to migrate his code to different platforms for portability purposes. We offer an additional approach using the Windows-based environment. After successfully compiling a SystemC application, the student has the option of downloading the SystemC bytecode by accessing a “Download” menu option, or by pressing the “Download” button on the second tab. Assuming serial connectivity with the development platform, the Windows-based environment maintains the software (the emulation engine) and circuitry of the SystemC-on-a-Chip platform and will download the SystemC bytecode automatically. The approaches are complementary, and give the user additional options for interfacing with their development platform.

8.4 Spatial and Time-Oriented Programming

8.4.1 Course Plan

We previously developed virtual microcontroller [123] technology for the purposes of teaching structured time-oriented programming to beginning students to complement traditional data-oriented programming paradigms without having to focus on the complexities and nuances of real microcontrollers. The SystemC-on-a-Chip teaching framework focuses on more advanced time-oriented programming while also introducing the concept of spatial programming to college students. Additionally, the SystemC-on-a-

Figure 51: Time Oriented and Spatial Programming with SystemC. We have developed a complete set of labs and materials to complement a course in spatial and time-oriented programming.

Example	Title	Purpose
1	Input/Output with LEDs	Beginning example on how to write SystemC to interface with peripherals
2	Seatbelt Warning Light System	Connecting Components. Spatial Programming
3	Toggle Switch	Introduction to Time-Ordered Behavior
4	Data Transmission and Encryption Systems	Introduction to Time-Interval Behavior
5	Working with an LCD	More advanced peripheral interfacing and time-interval programming

Chip framework gives students access to a number of powerful peripherals often seen in commercial systems, including LCDs, UARTs, and a video screens. The SystemC-on-a-Chip teaching approach is complementary to the virtual microcontroller approach, and could fit well as a more advanced course on embedded systems programming.

8.4.2 Sample Labs

Figure 51 shows a listing of several exercises intended we developed to teach college students about time-oriented and spatial programming using SystemC, and within the context of the SystemC-on-a-Chip platform. The listing is part of a complete set of materials available on *http://systemc.cs.ucr.edu* intended to give an instructor ample materials to serve as a basis for time-oriented and spatial programming. The examples follow a progression that teach students the basics of SystemC, spatial programming, time-ordered and time-interval programming, and then more advanced programming concepts. For each example, we introduce a new concept, and how that concept is implemented using SystemC. For the instructor, we provide our own source code solution. The source code solution might be used in the classroom showing the students the particular concept, or may be used to check student solutions in a lab setting. We also provide a series of additional exercises that further aid understanding in the particular concept just learned. The additional exercises can be presented to the students in numerous ways, including homework assignments, extra practice, or as supplemental laboratories.

Chapter 9

Contributions

9.1 Summary

We have demonstrated that SystemC serves as a viable distribution format for portable FPGA binaries. Combined with a fast emulation framework that dynamically and transparently optimizes the SystemC application, such a distribution format can attain high performance and still remain highly portable .

As FPGAs become more common in mainstream general-purpose computing platforms, distributing high-performance implementations of applications on FPGAs will become increasingly important. Even in the presence of C-based synthesis tools for FPGAs, designers continue to implement applications as circuits, due in large part to allow for capture of clever circuit-level implementation features leading to superior performance and efficiency. We demonstrated that while the distribution of sequential code (like C) for FPGA applications worked for 82% of the clever circuits we studied, many circuits required explicit parallel concepts, and of the 82% that we could capture as sequential code, 70% required *spatially-oriented* C code. Clearly the distribution format

of the FPGA application should include parallel programming constructs, along with the already established sequential constructs.

We chose to use SystemC as a possible distribution format for FPGA applications. SystemC allows description of a digital system using traditional programming features as well as spatial connectivity features common in hardware description languages. We described an approach for in-system emulation of SystemC descriptions. The approach centers around a new SystemC bytecode format that executes on an emulation engine running on a microprocessor and/or FPGA on a development board. Emulating SystemC allows a designer to test a circuit design using real peripherals while eliminating the need for eliminating the need for expensive, complicated, and often long-running synthesis tools at the cost of slower execution speed compared to a circuit. We described a full SystemC-on-a-chip framework that includes a SystemC bytecode compiler, the SystemC bytecode format, emulation engine, and emulation accelerators. We demonstrated that a number of examples could be written once in SystemC, and then run unaltered on several prototype platforms from a USB flash drive.

We observed that with the inclusion of SystemC bytecode accelerators that SystemC emulation could further benefit by adapting to a dynamically changing event queue. We defined the *Online SystemC Emulation Acceleration* problem and applied several online heuristics to improve emulation performance by 9x over emulating all of the SystemC on the SystemC emulation kernel, and 5x over statically preloading the

acceleration engines. Online heuristics could further speedup emulation by up to 20% with kernel bypass.

While many SystemC-on-a-Chip implementations benefit from FPGA resources, which directly affect the use of SystemC bytecode accelerators and their dynamic management, others do not, and are penalized with slow performance. We introduce JIT compilation techniques that on average improve the performance of SystemC emulation by 10x compared to basic SystemC emulation on a Microblaze processor. The speedup was obtained via a JIT/architecture codesign process wherein the architecture was refined and JIT compilation modified to yield additional speedups. The net result is that our SystemC emulator with JIT compilation on a Microblaze processor runs nearly as fast as C code written for and compiled directly to the Microblaze processor. Such fast emulation can greatly broaden the usefulness of SystemC emulation.

We demonstrated that the SystemC-on-a-Chip framework works well with developing digital mockups for medical device testing. Developing medical device software by interfacing with a digital mockup enables development without costly or dangerous physical mockups, and enables execution that is faster or slower than real-time. Developing digital mockups in SystemC has the added advantages that the description closely models the high level mathematical and physical model, can be tested extensively with freely available SystemC support libraries, and can interface to real medical device software through the use of the SystemC-on-a-Chip framework. The SystemC-on-a-Chip framework enables *time-controllable* debug features, making possible the ability to step through a digital mockup's execution through simulated time.

We tested the feasibility of such an approach by modifying the existing SystemC-on-a-Chip framework to support time-controllable debug, and also tested multiple respiratory digital mockup examples. We currently are modifying a commercial ventilation system to interact with SystemC-based digital mockups.

We developed and demonstrated a working framework to allow SystemC to be taught and used in the college classroom. Our framework includes a networked compiler, a simple and powerful Windows front end graphical interface, and a series of lessons to guide the beginning student from beginning SystemC constructs to more advanced embedded system design.

9.2 Remaining Challenges

We are currently working to improve the SystemC emulation tools in many respects, including developing new hardware-based emulation architectures, reducing the footprint of the emulation software, and developing frameworks for a number of different platforms. Possibly future improvements to the SystemC-on-a-Chip architecture include migrating the event queue kernel to hardware for improved performance, exacerbating the speedups seen by both JIT compilation and online SystemC acceleration. Another future improvement is profile a number of SystemC applications to identify various topologies of the SystemC bytecode accelerators that would improve SystemC emulation. We currently have only developed one kernel bypass mechanism, but many such bypass mechanisms might exist. Eventually, the entire SystemC-on-a-Chip framework might be an array of connected SystemC accelerators that require no overhead for maintaining event and signal queues. Another future improvement to the emulation framework is to

integrate the JIT compilation framework with the online acceleration management problem, further increasing the performance of the emulation framework.

Further improvements include supporting a larger set of the SystemC language (constructs like memories, queues, fifos, etc), as well as higher level programming paradigms like transaction level modeling (TLM). The SystemC-on-a-Chip framework should eventually be built for PC-based platforms that already support FPGA additions (like Intel Quick assist), increasing the utility of such a framework.

References

[1] Altera Corp. http://www.altera.com, 2005.

[2] Adams, K. and Agesen, O. 2006. A comparison of software and hardware techniques for x86 virtualization. In Proceedings of the 12th international Conference on Architectural Support For Programming Languages and Operating Systems (San Jose, California, USA, October 21 - 25, 2006).

[3] Anderson, E., Agron, J., Peck, W., Stevens, J., Baijot, F., Komp, E., Sass, R., and Andrews, D. 2006. Enabling a Uniform Programming Model Across the Software/Hardware Boundary. In Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (April 24 - 26, 2006). FCCM

[4] Andrews, D., Sass, R., Anderson, E., Agron, J., Peck, W., Stevens, J., Baijot, F., and Komp, E. 2008. Achieving programming model abstractions for reconfigurable computing. IEEE Trans. Very Large Scale Integr. Syst. 16, 1 (Jan. 2008), 34-44.

[5] Azizi, N., Kuon, I., Egier, A., Darabiha, A., and Chow, P. 2004. Reconfigurable Molecular Dynamics Simulator (April 20 - 23, 2004). FCCM

[6] Balarin, F. , Lavagno, L., and Murthy P. Scheduling for Embedded Real-Time Systems. IEEE Design and Test of Computers, 1998

[7] Baker, P., Todman, T., Styles, H., and Luk, W. 2005. Reconfigurable Designs for Radiosity. - Volume 00 (April 18 - 20, 2005). FCCM

[8] Baker, Z. K. and Prasanna, V. K. 2005. Efficient Hardware Data Mining with the Apriori Algorithm on FPGAs. (Fccm'05) - Volume 00 (April 18 - 20, 2005). FCCM

[9] Baker, Z. K. and Prasanna, V. K. 2006. An Architecture for Efficient Hardware Data Mining using Reconfigurable Computing Systems. (Fccm'06) - Volume 00 (April 24 - 26, 2006). FCCM

[10] Bala, V., Duesterwald, E., and Banerjia, S. 2000. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, Vol. 35, No. 5, pp. 1-12

[11] Baraz, L., Devor, T., Etzion, O., Goldenberg, S., Skaletsky, A., Wang, Y., and Zemach, Y. 2003. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. *In Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 191-201.

[12] Bauer, J., Bershteyn, M., Kaplan, I., and Vyedin, P. 1998. A reconfigurable logic machine for fast event-driven simulation. In Proceedings of the 35th Annual Design Automation Conference (San Francisco, California, United States, June 15 - 19, 1998). DAC '98

[13] Beeckler, J. S. and Gross, W. J. 2005. FPGA Particle Graphics Hardware (April 18 - 20, 2005). FCCM

[14] Benini, L., Bertozzi, D., Bruni, D., Drago, N., Fummi, F., and Poncino, M. 2003. SystemC Cosimulation and Emulation of Multiprocessor SoC Designs. /Computer/ 36, 4 (Apr. 2003), 53-59

[15] Benini, L., Bruni, D., Drago, N., Fummi, F., and Poncino, M. "Virtual in-circuit emulation for timing accurate system prototyping," in Proc. IEEE Int. Conf. ASIC/-SoC, 2002

[16] Benjamin, M., Kaeli, D., and Platcow, R. 2006. Experiences with the Blackfin architecture in an embedded systems lab.. WCAE '06

[17] Bitton, D. , Dewitt, D.J, Hsaio, D.K, and j. Menon. 1984. A taxonomy of parallel sorting. ACM Comput. Surv. 16, 3 (Sep. 1984)

[18] Bondhugula, U., Devulapalli, A., Dinan, J., Fernando, J., Wyckoff, P., Stahlberg, E., and Sadayappan, P. 2006. Hardware/Software Integration for FPGA-based All-Pairs Shortest-Paths. (April 24 - 26, 2006). FCCM.

[19] Bogdanov, A. and Mertens, M. C. 2006. A parallel hardware architecture for fast Gaussian Elimination over GF(2). FCCM, pp. 237-248.

[20] Botros, N., Akaaboune, M., Alghazo, J., and Alhreish, M. 2000. Hardware Realization of Biological Mechanisms Using VHDL and FPGAs

[21] Brown, J.C Parallel Architectures for Computer Systems. IEEE Computer vol 37, no. 5. pp83-87 1989.

[22] Butler, J. and Brockman, J. Web-based Learning Tools on Microprocessor Fundamentals for a First-Year Engineering Course. 2003. American Society for Engineering Education

[23] Cadence Design Systems. http://www.cadence.com/us/pages/default.aspx

[24] CatapultC. http://www.mentor.com/products/c-based_design/

[25] Celoxica. http://www.celoxica.com/

[26] Chandran, P., Chandra, J., Simon, B. P., and Ravi, D. 2009. Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines.

[27] Chang, C., Kuusilinna, K., Richards, B., and Brodersen, R. W. 2003. Implementation of BEE: a real-time large-scale hardware emulation engine. In Proceedings of the 2003 ACM/SIGDA Eleventh international Symposium on Field Programmable Gate Arrays (Monterey, California, USA, February 23 - 25, 2003). FPGA '03. ACM, New York, NY, 91-99

[28] Chernoff, A., Herdeg, M., Hookway, R.,Reeve, C., Rubin, N., Tye, T., Bharadwaj Yadavalli, S., and Yates, J. 1998. FX!32 a profile-directed binary translator. *IEEE Micro*, Vol. 18, Issue 2, pp. 56 – 64.

[29] Cho, Y. H. and Mangione-Smith, W. H. 2004. Deep Packet Filter with Dedicated Logic and Read Only Memories. (April 20 - 23, 2004). FCCM

[30] Chopard, B., Combes, P., and Zory, J. A Conservative Approach to SystemC Parallelization. Lecture Notes in Computer Science. Volume 3994. 2006.

[31] Cifuentes, C. 1994. Reverse compilation techniques. *Queensland University of Technology, Department of Computer Science, PhD thesis*

[32] Combes, P., Caron, E., Desprez, F., Chopard, B., and Zory, J. 2008. Relaxing Synchronization in a Parallel SystemC Kernel. In Proceedings of the 2008 IEEE international Symposium on Parallel and Distributed Processing with Applications

[33] Cottrell, S. and F. Vahid. A Logic Enabling Configuration by Non-Experts in Sensor Networks. HFC. 2005.

[34] Coware. http://www.coware.com/

[35] Danne, K., Platzner, M. Periodic Real-Time Scheduling for FPGA Computers. Intelligent Solutions in Embedded Systems, 2005

[36] Das, S. R. 1996. Adaptive protocols for parallel discrete event simulation. In Proceedings of the 28th Conference on Winter Simulation

[37] Davis, B., Beatty, A., Casey, K., Gregg, D., and Waldron, J. 2003. The case for virtual register machines. In Proceedings of the 2003 Workshop on interpreters, Virtual Machines and Emulators (San Diego, California, June 12 - 12, 2003). IVME '03. ACM, New York, NY, 41-49

[38] Dehnert, J. C., Grant, B. K., Banning, J. P., Johnson, R., Kistler, T., Klaiber, A., and Mattson, J. 2003. The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In Proceedings of the international Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization

[39] Diniz, P., Hall, M., Park, J., So, B., and Ziegler, H. 2005. Automatic mapping of C to FPGAs with the DEFACTO compilation and synthesis systems. Journal on Microprocessors and Microsystems, Vol. 29, Issues 2-3, pp. 51-62.

[40] Doom, T.; White, J.; Wojcik, A.; and G. Chisholm. 1998. Identifying high-level components in combinational circuits. Proceedings of the 8th Great Lakes Symposium on VLSI 1998

[41] Durbano, J. P., Ortiz, F. E., Humphrey, J. R., Curt, P. F., and Prather, D. W. 2004. FPGA-based acceleration of the 3D finite-difference time-domain method. FCCM.

[42] eCOS. http://ecos.sourceware.org/

[43] Eles, P., Peng, Z., Kuchchinski, K. and Doboli, A. 1997. System level hardware/software partitioning based on simulated annealing and tabu search. Journal on Design Automation for Embedded Systems, Vol. 2, No. 1, pp. 5-32.

[44] Emmert, J, and Bhatia, D. Partial Reconfiguration of FPGA mapped designs with Applications to Fault Tolerance and Yield Enhancement. Lecture Notes in Computer Science. April 2006

[45] Enzler, R. Plessl, C. and Platzner, M. Virtualizing Hardware with Multi-Context Reconfigurable Arrays. Lecture Notes in Computer Science. Springer Publishing. September 2003.

[46] Fin, A., Fummi, F., and Signoretto, M. 2001. SystemC: a homogenous environment to test embedded systems. CODES, pp 17-22

[47] Fornaciari, W. and Piuri, V. Virtual FPGAs: Some Steps Behind the Physical Barriers. In Parallel and Distributed Processing (IPPS/SPDP'98 Workshop Proceedings), LNCS. 1998

[48] French, R. S., Lam, M. S., Levitt, J. R., and Olukotun, K. 1995. A general method for compiling event-driven simulations. In Proceedings of the 32nd ACM/IEEE Conference on Design Automation (San Francisco, California, United States, June 12 - 16, 1995). DAC '95. ACM, New York, NY, 151-156

[49] Frigo, J., Gokhale, and M., Lavenier, D. 2001. Evaluation of the streams-C C-to-FPGA compiler: an applications perspective. FPGA, pp. 134-140

[50] Fry, T. W. and Hauck, S. 2002. Hyperspectral Image Compression on Reconfigurable Platforms. (September 22 - 24, 2002). FCCM.

[51] Fujimoto, R. M. 1989. Parallel discrete event simulation. In Proceedings of the 21st Conference on Winter Simulation E. A. MacNair, K. J. Musselman, and P. Heidelberger, Eds. WSC '89.

[52] Genko, N., Atienza, D., Micheli, G. D., Mendias, J. M., Hermida, R., and Catthoor, F. 2005. A Complete Network-On-Chip Emulation Framework. In Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1 (March 07 - 11, 2005). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 246-251

[53] Ghiasi, S. and Sarrafzadeh, M. 2003. Optimal reconfiguration sequence management. ASP-DAC '03. ACM, New York, NY, 359-365

[54] Gligor, M., Fournel, N., and Pétrot, F. 2009. Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In Proceedings of the 7th IEEE/ACM international Conference on Hardware/Software Codesign and System Synthesis (Grenoble, France, October 11 - 16, 2009). CODES+ISSS '09

[55] Gross, D., and Harris, C.M. Fundamentals of queueing theory. John Wiley & Sons, Inc. New York, NY, USA. 1985

[56] Gschwind, M., Altman, E., Sathaye, S., Ledak, P., and Appenzeller., D. 2000. Dynamic and transparent binary translation. *IEEE Computer*, Vol. 33, No. 3, pp.54-59.

[57] Gupta, S., Dutt, N.D., Gupta, R.K., and Nicolau, A. 2003. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. VLSI.

[58] Gupta, S., and G. Demicheli 1991. VULCAN - A System for High-Level Partitioning of Synchronous Digital Circuits. Technical Report

[59] Hansen, M.C. Yalcin, H. and J.P Hayes, 1999. Unveiling the ISCAS-85 benchmarks: A Case Study in Reverse Engineering . IEEE Design and Test in Computers. Vol. 12, Issue 3

[60] Hapsim. http://www.helmix.at/hapsim

[61] Harchol-Balter, M. and Downey, A. B. 1997. Exploiting process lifetime distributions for dynamic load balancing. ACM Trans. Comput. Syst. 15, 3 (Aug. 1997), 253-285

[62] Hariri, A., Rastegar, R., Zamani, M. S., and Meybodi, M. R. 2005. Parallel hardware implementation of cellular learning automata based evolutionary computing (CLA-EC) on FPGA. FCCM, pp. 311-314.

[63] Haulbelt, C., Teich , J., Richter, K. and Ernst R. 2002. System design for flexibility. Design, Automation, and Test in Europe (DATE).

[64] Henkel, J. 1999. A low power hardware/software partitioning approach for core-based embedded systems. DAC, pp. 122-127

[65] He, C., Lu, M., and Sun, C. 2004. Accelerating seismic migration using FPGA-based coprocessor platform. FCCM, pp. 207-216.

[66] He, C., Zhao, W., and Lu, M. 2005. Time domain numerical simulation for transient waves on reconfigurable coprocessor platform. FCCM, pp. 127-136.

[67] Hennessy, J. and Patterson, D. Computer Architecture – A Quantitative Approach. Morgan Kaufman Publishers. 3rd edition. 1996

[68] Hezel, S., Kugel, A., Männer, R., and Gavrila, D. M. 2002. FPGA-Based Template Matching Using Distance Transforms. (September 22 - 24, 2002). FCCM.

[69] Hoare, C.A. 1961. Algorithm 64: Quicksort. Commun. ACM 4, 7 (Jul. 1961).

[70] Hodge, H. Hinton, H.S, and Lightner, M. Virtual Circuit Laboratory. ASEE. American Society for Engineering Education. 2000

[71] Horta, E. L., Lockwood, J. W., Taylor, D. E., and Parlour, D. 2002. Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In *Proceedings of the 39th Annual Design Automation Conference* (New Orleans, Louisiana, USA, June 10 - 14, 2002). DAC '02. ACM, New York, NY, 343-348.

[72] Huang, C., and Vahid, F. Dynamic Coprocessor Management for FPGA-Enhanced Compute Platforms. CASES 2008.

[73] Huang, C. and Vahid, F. Dynamic Transmuting Coprocessors. IEEE/ACM Design Automation Conference. DAC. July 2009.

[74] Huang, Z. and Ercegovac, M. D. 2001. FPGA Implementation of Pipelined On-Line Scheme for 3-D Vector Normalization. (April 29 - May 02, 2001). FCCM

[75] Images Scientific Instruments. http://imagesco.com

[76] Impulse CoDeveloper. http://www.impulsec.com/

[77] Intel QuickAssist Technology http://www.intel.com/technology/magazine/45nm/quickassist-0507.htm

[78] Ishfaq Ahmad , Arif Ghafoor , Kishan Mehrotra, Performance prediction of distributed load balancing on multicomputer systems, Proceedings of the 1991 ACM/IEEE conference on Supercomputing,

[79] IUPS Physiome Project. http://www.physiome.org.nz/

[80] James-Roxby, P., Brebner, G., and Bemmann, D. 2004. Time-Critical Software Deceleration in an FCCM. (April 20 - 23, 2004). FCCM

[81] James-Roxby, P. B. and Downs, D. J. 2001. An Efficient Content-Addressable Memory Implementation Using Dynamic Routing. (April 29 - May 02, 2001). FCCM

[82] Jefferson, D. and Reiher, P. 1991. Supercritical speedup. In Proceedings of the 24th Annual Symposium on Simulation Annual Simulation Symposium. IEEE 159-168

[83] Kazi, I. H., Chen, H. H., Stanley, B., and Lilja, D. J. 2000. Techniques for obtaining high performance in Java programs. ACM Comput. Surv. 32, 3 (Sep. 2000), 213-240

[84] Kim, H. and Smith, J. E. 2003. Dynamic binary translation for accumulator-oriented architectures. In Proceedings of the international Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (San Francisco, California, March 23 - 26, 2003).

[85] Kopetz, H. and Ochsenreiter, W. 1987. Clock synchronization in distributed real-time systems. IEEE Trans. Comput. 36, 8 (Aug. 1987), 933-940

[86] Krueger, S. D. and Seidel, P. 2004. Design of an on-line IEEE floating-point addition unit for FPGAs. FCCM, pp. 239-246.

[87] Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 7 (Jul. 1978), 558-56

[88] Lee, D., Luk, W., Villasenor, J., and Cheung, P. Y. 2003. A hardware Gaussian noise generator for channel code evaluation. FCCM.

[89] Lee, E. Computing Needs Time. Communications of the ACM. May 2009. Vol 52. Number 5.

[90] Lee, I. Davidson, S., and Wolfe, V. Motivating Time as a First-Class Entity. Technical Report MS-CIS-87-54. Department of Computer and Information Science. University of Pennsylvanis, Philadelphia, PA. Aug 1987

[91] Levine, B. A. and Schmit, H. H. 2003. Efficient Application Representation for HASTE: Hybrid Architectures with a Single, Transformable Executable. FCCM. IEEE Computer Society, Washington, DC, 101

[92] Levis, P. and Culler, D. 2002. Maté: a tiny virtual machine for sensor networks. SIGOPS Oper. Syst. Rev. 36, 5 (Dec. 2002), 85-95

[93] Lysecky, R., Vahid, F. and Tan, S. Dynamic FPGA Routing for Just-in-Time FPGA Compilation. Design Automation Conference (DAC), June 2004, pp. 954-959

[94] Lysecky R., Cotterell, S., and Vahid, F. A Fast On-Chip Profiler Memory. IEEE/ACM Design Automation Conference, June 2002, pp. 28-33

[95] Mahmoud, W. H., Haggard, R. L., and Abdelrahman, M. 2001. Hardware Implementation of Automated Sensor Self-Validation System for Cupola Furnaces (April 29 - May 02, 2001). FCCM.

[96] MedGadget Internet Journal. 2008. Supercomputer Creates Most Advanced Heart Model. http://medgadget.com/archives/2008/01/worlds_biggest_heart_model_simulated_1.html

[97] Message Passing Interface Standard. http://www.mcs.anl.gov/research/projects/mpi/

[98] Min-You Wu. On runtime parallel scheduling for processor load balancing, IEEE TPDS 1997

[99] Mitra, T. and Chiueh, T. 2002. An FPGA Implementation of Triangle Mesh Decompression. (September 22 - 24, 2002). FCCM.

[100] Moore, N., Conti, A., Leeser, M., and King, L. S. 2007. Writing Portable Applications that Dynamically Bind at Run Time to Reconfigurable Hardware. FCCM. IEEE Computer Society, Washington, DC, 229-238

[101] Moscola, J., Lockwood, J., Loui, R. P., and Pachos, M. 2003. Implementation of a Content-Scanning Module for an Internet Firewall (April 09 - 11, 2003). FCCM

[102] Moy, M., Maraninchi, F., and Maillet-Contoz, L. 2005. Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip. In Proceedings of the 5th ACM international Conference on Embedded Software (Jersey City, NJ, USA, September 18 - 22, 2005). EMSOFT '05. ACM, New York, NY, 317-324

[103] Naguib, Y. N. and Guindi, R. S. 2007. Speeding Up SystemC Simulation through Process Splitting. DATE

[104]Najjar, W., Böhm, W., Draper, B., Hammes, J., Rinker, R., Beveridge, R., Chawathe, M., and Ross, C. 2003. From algorithms to hardware -- a high-level language abstraction for reconfigurable computing. IEEE Computer, Vol. 36, Issue 8, August 2003, pp.63-69

[105]Nakamura, Y., Hosokawa, K., Kuroda, I., Yoshikawa, K., and Yoshimura, T. 2004. A fast hardware/software co-verification method for system-on-a-chip by using a C/C++ simulator and FPGA emulator with shared register communication. In Proceedings of the 41st Annual Conference on Design Automation (San Diego, CA, USA, June 07 - 11, 2004). DAC '04

[106]Nallatech. http://www.nallatech.com/

[107]NSR Physiome Project. http://nsr.bioeng.washington.edu/.

[108]Noguera, J., Badia, R.M. Dynamic run-time HW/SW scheduling techniques for reconfigurable architectures. CODES-ISSS, 2002

[109]OpenCollector. http://opencollector.org/news/Bitstream/

[110]Panel on "Programming Standards for FPGAs in High-Performance Computing Applications." Supercomputing Conference, 2005.

[111]Pérez, D. G., Mouchard, G., and Temam, O. 2004. A New Optimized Implemention of the SystemC Engine Using Acyclic Scheduling. DATE 2004

[112]Pimentel, J. and Tirat-Gefen, Y. 2006. Hardware Acceleration for Real-time Simulation of Physiological Systems. EMBS. pp 218-223

[113]Plessl, C. and Platzner, M. 2002. Custom Computing Machines for the Set Covering Problem (September 22 - 24, 2002). FCCM.

[114]Poletto, M. and Sarkar, V. 1999. Linear scan register allocation. ACM Trans. Program. Lang. Syst. 21, 5 (Sep. 1999), 895-913

[115]Ricks, K. G., Jackson, D. J., and Stapleton, W. A. 2005. An evaluation of the VME architecture for use in embedded systems education. SIGBED Rev. 2, 4 (Oct. 2005), 63-69.

[116]Rissa, T., Donlin, A., and Luk, W. 2005. Evaluation of SystemC Modelling of Reconfigurable Embedded Systems. In Proceedings of the Conference on Design, Automation and Test in Europe - Volume 3 (March 07 - 11, 2005). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 253-258

[117]Rosenblum, M. 2004. The Reincarnation of Virtual Machines. Queue 2, 5 (Jul. 2004), 34-40.

[118]Scrofano, R., Gokhale, M., Trouw, F., and Prasanna, V. K. 2006. Hardware/software approach to molecular dynamics on reconfigurable computers. FCCM, pp. 23-34.

[119]SGI Altix. http://www.sgi.com/products/servers/altix/

[120]Shin, H. and George, S. Impact of Axial Diffusion on Nitric Oxide Exchange in the Lungs. Journal of Applied Physiology. 2002

[121]Sidhu, R. and Prasanna, V. K. 2001. Fast Regular Expression Matching Using FPGAs. (April 29 - May 02, 2001). FCCM

[122]Sirowy, S., Givargis, T. and Vahid, F. Digitally-Bypassed Transducers: Interfacing Digital Mockups to Real-Time Medical Equipment. IEEE Engineering and Biology Society (EMBS). 2009. Minneapolis

[123]Sirowy, S. Sheldon, D., Givargis, T. and Vahid. F. Virtual Microcontrollers. SIGBED Review 2009.

[124]Smith, J. and Nair, R. Virtual Machines: Versatile Platforms for Systems and Processes. Morgan-Kaufman Publishers. 2005

[125]Sourdis, I. and Pnevmatikatos, D. 2004. Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching. (April 20 - 23, 2004). FCCM

[126]Srinivasan, V., Radhakrishnan, S., and Vemuri, R. 1998. Hardware/software partitioning with integrated hardware design space exploration. DATE, pp. 28-35

[127]Stark, R., Schmid, J, and Borger, E. Java and the Virtual Machine- Definition, Verification, and Validation. 2001

[128]Steiger, C., Walder, H., Platzner, M., and Thiele, L. 2003. Online Scheduling and Placement of Real-time Tasks to Partially Reconfigurable Devices. RTSS 2003

[129]Stitt, G., Vahid, F., McGregor, G., and Einloth, B. 2005. Hardware/software partitioning of software binaries: a case study of H.264 decode. CODES/ISSS, pp. 285-290

[130]Stitt, G., And F. Vahid. 2006. A Code Refinement methodology for performance-improved synthesis from C. ICCAD

[131]Stitt, G. and Vahid, F. Binary Synthesis. ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 12 No. 3, Aug 2007

[132]Sukhwani, B., Forin, A., and Pittman, R. N. 2008. Extensible On-Chip Peripherals. In *Proceedings of the 2008 Symposium on Application Specific Processors* (June 08 - 09, 2008). SASP. IEEE Computer Society, Washington, DC, 55-62.

[133]SystemC. http://www.systemc.org

[134]SystemC Synthesizable Subset. http://www.systemc.org

[135]Tawhai, M., and Ben-Tal, A. 2004. Multiscale Modeling for the Lung Physiome. Cardiovascular Engineering: An International Journal, Vol. 4, No. 1., March 2004. pp 19-26

[136]Tensilica Inc. http://www.tensilica.com/

[137]Thomas, D. B. and Luk, W. 2006. Efficient Hardware Generation of Random Variates with Arbitrary Distributions. (Fccm'06) - Volume 00 (April 24 - 26, 2006). FCCM

[138]Tripp, J. L., Mortveit, H. S., Hansson, A. A., and Gokhale, M. 2005. Metropolitan road traffic simulation on FPGAs. FCCM, pp. 117-126.

[139]Tsoi, K. H., Lee, K. H., and Leong, P. H. 2002. A Massively Parallel RC4 Key Search Engine. (September 22 - 24, 2002). FCCM.

[140]Tsoi, K. H., Leung, K. H., and Leong, P. H. 2003. Compact FPGA-based True and Pseudo Random Number Generators. (April 09 - 11, 2003). FCCM.

[141]Verilog Specification. http://www.verilog.com/VerilogBNF.html

[142]VHDL Specification Standard. http://www.vhdl.org/

[143]Villarreal, J., Park, A., Najjar, W. and Halstead R. Designing Modular Hardware Accelerators in C With ROCCC 2.0, in The 18th An. Int. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), Charlotte, NC, May 2010

[144]Virdes Development System. http://avoron.com/index.php

[145]Vuletic, M., Pozzi, L., and Ienne, P. 2004. Virtual Memory Window for a Portable Reconfigurable Cryptography Coprocessor. (April 20 - 23, 2004). FCCM

[146]Vuletic, M., Pozzi, L., and Ienne, P. 2005. Seamless Hardware-Software Integration in Reconfigurable Computing Systems. IEEE Des. Test 22, 2 (Mar. 2005), 102-113

[147]VmWare. http://www.vmware.com

[148]Wake, H. A. and Buell, D. A. 2003. Congruential Sieves on a Reconfigurable Computer. (April 09 - 11, 2003). FCCM

[149]Wang, Z. and Maurer, P. M. 1990. LECSIM: a Levelized event driven compiled logic simulation.). DAC '90

[150]Wang, X. and Nelson, B. E. 2003. Tradeoffs of Designing Floating-Point Division and Square Root on Virtex FPGAs. (April 09 - 11, 2003). FCCM

[151]Whitton, K., Hu, X. S., Yi, C. X., and Chen, D. Z. 2006. An FPGA Solution for Radiation Dose Calculation. (April 24 - 26, 2006). FCCM

[152]WindRiver Systems. http://www.windriver.com/

[153]Xen. http://www.xen.org

[154]Zhang, Y. and S.Q Zheng. 1995. Design and analysis of a systolic sorting architecture. SPDP. IEEE Computer Society, Washington, DC, 652

[155]Ziegler, H., So, B., Hall, M., and Diniz, P. C. 2002. Coarse-Grain Pipelining on Multiple FPGA Architectures (September 22 - 24, 2002). FCCM