Elaboration-time Synthesis of High-level Language Constructs in SystemC-based Microarchitectural Simulators

Zhuo Ruan, Kurtis Cahill, and David Penry
Department of Electrical and Computer Engineering
Brigham Young University
Provo, Utah 84602, U.S.A
{*zruan, grandk, dpenry*}*@et.byu.edu*

***Abstract*— Structural modeling serves as an efficient method for creating detailed microarchitectural models of complex microprocessors. High-level language constructs such as templates and object polymorphism are used to achieve a high degree of code reuse, thereby reducing development time. However, these modeling frameworks are currently too slow to evaluate future design of multicore microprocessors. The synthesis of portions of these models into hardware to form hybrid simulators promises to improve their speed substantially. Unfortunately, the high-level language constructs used in structural simulation frameworks are not typically synthesizable. One factor which limits their synthesis is that it is very difficult to determine statically what exactly the code and data to synthesize are. We propose an *elaboration-time synthesis* method for SystemC-based microarchitectural simulators. As part of the runtime environment of our infrastructure, the synthesis tool extracts architectural information after elaboration, binds dynamic information to a low-level intermediate representation (IR), and synthesizes the IR to VHDL. We show that this approach permits the synthesis of high-level language constructs which could not be easily synthesized before.**

I. Introduction

Structural simulation frameworks such as SystemC [1], Unisim [2], and Liberty Simulation Environment (LSE) [23] allow microarchitects to easily specify simulators in a concurrent and structural form which naturally and accurately mimics hardware (HW) behavior. The use of high-level language constructs (e.g. templates, object polymorphism, and dynamic casts) in structural simulators permits a high degree of code reuse, and thus allows architects to construct microarchitectural models quickly in software (SW).

While structural simulation frameworks improve the speed at which architects can create microarchitectural models, they are too slow to permit extensive exploration of complex future multicore systems. As means to accelerate these simulators, several researchers [5], [6], [17] have proposed to use FPGAs. These FPGA-based *hybrid simulators* contain a SW portion and a HW portion which communicate through an interface.

Unfortunately, designing hybrid simulators is difficult and time-consuming because it involves several manual steps: partitioning of the model, SW modification, HW design, and SW/HW interface design. We have previously proposed to automate these steps within a hybrid simulator development framework called the Simulator Partitioning Research Infrastructure (SPRI). SPRI automatically produces hybrid simulators from SystemC-based structural simulators [18]; SPRI generates VHDL code for the hardware portion, VHDL and C code for the HW/SW interface, and the modified SW portion of the simulator which uses the interface. SPRI shortens the design time of hybrid simulators from months to hours, allowing users to easily explore different hybrid simulators with different interfaces and partitionings [19].

An important problem which SPRI must solve is that high-level language constructs are not typically synthesizable. These constructs (e.g. virtual methods and templates) are important, however, for enhancing reuse in structural simulation frameworks. The problem in synthesizing these constructs arises because it is difficult to determine statically what exactly the code or data to synthesize are.

We propose to handle these high-level constructs by performing synthesis after elaboration is finished. This process is called *elaboration-time synthesis*. During a run of a given software structural simulator, SPRI extracts hierarchical information after elaboration, binds dynamic information to a low-level intermediate representation (IR), and synthesizes the IR to VHDL after resolving high-level language constructs. Our contributions are:

- providing the first attempt to synthesize the HW portion of a hybrid simulator automatically from a given SystemC-based microarchitectural simulator;
- proposing a new elaboration-time synthesis method for high-level language constructs (e.g. templates, object polymorphism) used in microarchitectural simulators;
- presenting the detailed analysis strategies and algorithms to aid the synthesis of microarchitectural models.

II. Related Work

The synthesis of microarchitectural models presented in this paper is a high-level synthesis dealing with SystemC constructs used to model microarchitectural simulators. The synthesis process includes translating unsynthesizable SystemC constructs to synthesizable ones and mapping synthesizable SystemC subsets directly to VHDL.

In the past two decades of high-level synthesis, researchers have spent much effort on improving the productivity of hardware design, using high-level languages to describe hardware behavior and then automatically synthesizing such

descriptions into HDL. SystemC appears to be the most widely-accepted solution for SW/HW codesign in both industry and academia. Public information about industrial SystemC synthesizers is limited; however, Cynthesizer and Catapult C support both behavioral and transaction-level modeling (TLM) [9], [22]. As shown in Table I, existing SystemC tools can be classified by their analysis approaches: 1) static analysis; 2) dynamic analysis; 3) hybrid analysis [15].

- **Tools Based on Static Analysis.** ParSyc and KaSCPar both generate an Abstract Syntax Tree (AST) from SystemC source code; ParSyc converts the AST into an intermediate representation using classes corresponding to constructs in SystemC code, while the AST used by KaSCPar is built with dedicated grammar rules for SystemC constructs; they obtain hierarchical information by parsing the AST [8], [10]. SC2V is written with LEX and YACC, and it only synthesizes the synthesizable subsets of SystemC into Verilog [4]. SCOOT is based on an existing C++ front-end. It is a model extractor and annotates a control flow graph with architectural information as an IR [3].
- **Tools Based on Dynamic Analysis.** Quiny uses an unmodified C++ compiler. It adopts a reflective approach which links in the Quiny library and returns expressions at runtime by overloading C++ operators [20]. This approach prevents Quiny from supporting language constructs which cannot be overloaded, e.g., selection operators. Another dynamic-analysis-based tool is DUST [13]. DUST extracts model structures and transactions by recording information of TLM constructs while running the simulation.
- **Tools Based on Hybrid Analysis.** Pinapa takes a hybrid approach in which GCC generates an AST by parsing SystemC source code. Pinapa creates the connection between the AST and the architectural information obtained after the elaboration phase; however, it has problems with templates and complex array index expressions, and does not have a VHDL backend [16]. [21] uses Pinapa to partition a SystemC program into several parallel blocks; each block is run in parallel on a separate processor core embedded in an FPGA.

The tools mentioned above successfully serve their application domains. However, none of them target the synthesis of SystemC-based microarchitectural simulators, and none of their application domains utilize high-level language constructs as frequently as do microarchitectural simulators.

III. Simulator Partitioning Research Infrastructure (SPRI)

SPRI automatically synthesizes hybrid microarchitectural simulators from structural simulation models. The SPRI synthesis flow for hybrid microarchitectural simulators is shown in Figure 1:

1) The partitioner splits the software model into a SW block and a HW block, guided by a user-provided partitioning specification.

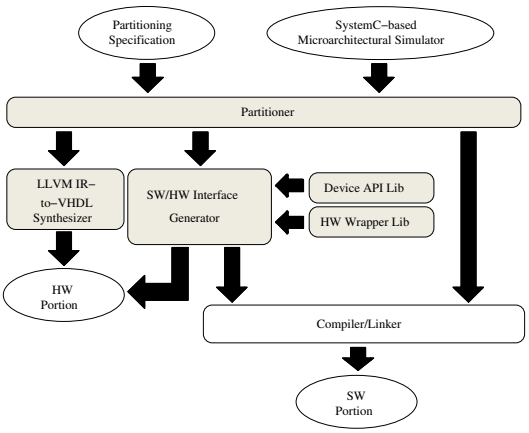

Fig. 1. SPRI Synthesis Flow for Hybrid Microarchitectural Simulators

2) The interface generator creates interface SW and HW libraries.
3) The partitioner modifies the SW block to call the interface SW library.
4) The HW block is synthesized into VHDL and then combined with the interface HW library.
5) The SW portion of the hybrid simulator is formed by linking the SW block with the interface SW library.
6) The HW portion of the hybrid simulator is synthesized and mapped to and FPGA using Xilinx ISE.

This paper focuses on the process of HW synthesis from SystemC to VHDL for generated hybrid microarchitectural simulators and specifically on how high-level language constructs are dealt with. One option for dealing with high-level language constructs is to keep them in SW and only partition SystemC synthesizable portions into HW. In this case, the synthesis would only need to handle SystemC synthesizable subsets; however, the resulting synthesizer would be unable to synthesize significant portions of the model if high-level language constructs are common in the model. We choose to support high-level constructs used in structural simulation models by detecting them, translating them, and replacing them. Thus SPRI can handle any purely structural partitioning selected by the user. Note that some partitionings may cause shared states and global variables to cross the SW/HW boundary; we will support such non-structural partitionings in the future.

IV. SystemC-to-VHDL Synthesis

The SPRI synthesizer takes software microarchitectural simulators written in SystemC as input. However, unlike other synthesizers, synthesis takes place after elaboration and the SPRI synthesizer does not parse SystemC source code. Elaboration-time partitioning removes the need to analyze template instantiation, object inheritance, and the creation of the simulation object hierarchy.

Elaboration-time synthesis requires that the model be synthesized *during a run of the simulator*. To enable runtime synthesis, we use LLVM [14]. LLVM is an extensible compiler framework which permits runtime code generation

Tool	Approach	Application	Open	C++ Coverage	HDL Backend
Academic Tools					
KaSCPar [10]	Static (Using a dedicated grammar)	Verification	Partially	Limited	No
ParSyc [8]		Synthesis	Unavailable	Unknown	Unknown
SC2V [4]		Synthesis	Yes	Weak	Yes (`sc_method`)
SCOOT [3]	Static (Based on existing C++ front-end)	Verification	No	Limited	No
Quiny [20]	Dynamic (Based on existing C++ front-end)	Synthesis, Analysis	Yes	Limited	Yes (`sc_method`, `sc_thread`)
Pinapa [16]	Hybrid	Verification, Analysis	Yes	Limited	No
DUST [13]	Dynamic	Introspection (TLM)	Yes	Unknown	No
Industrial Tools					
Synopsis [15]	Static	Synthesis, Verification	No	Unknown	Unknown
Catapult C (Mentor Graphics) [22]	Static	Synthesis, Verification	No	Unknown	Yes (TLM, Behavior)
Cynthesizer (Forte) [9]	Unknown	Synthesis, Verification	No	Unknown	Yes (TLM, Behavior)

TABLE I
EXISTING SYSTEMC ANALYSIS AND SYNTHESIS TOOLS IN ACADEMIA AND INDUSTRY

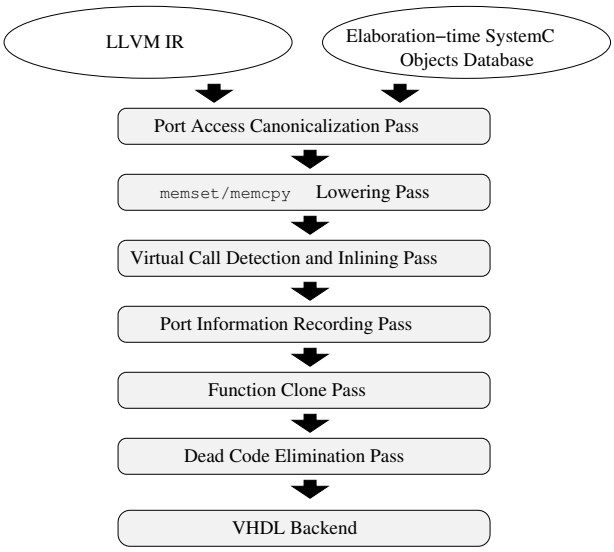

Fig. 2. VHDL Synthesis Flow

and the coexistence of both generated machine code and an analyzable and mutable internal representation (IR) of that code. The LLVM IR is a generic instruction-based representation in Static Single Assignment (SSA) form. Thus the SPRI synthesis flow compiles the SystemC source code to LLVM IR and then runs the compiled program through an LLVM JIT enhanced with our synthesis passes. There are six such passes, as well as a new code generator which produces VHDL instead of machine code. Figure 2 depicts the VHDL synthesis flow.

A. Creation of Elaboration-time SystemC Objects Database

The synthesis passes operate on a shared database of SystemC objects which were instantiated during elaboration. This database contains the module hierarchy, inheritance relations, and pointers to in-memory SystemC objects and data. We record the module hierarchy after elaboration. Instantiated SystemC objects are categorized by parsing the C++ run-time type information (RTTI) in terms of classes, instances, `sc_method`, `sc_cthread`, functions, ports, signals, clocks, and events. This parsing relies upon specific naming conventions for C++ mangled names and thus would need to be changed for different operating systems or compilers; we have attempted to isolate these changes to small portions of the code. Similarly, we detect hierarchical inheritance by looking at the structure of RTTI defined in the Application Binary Interface (ABI). We distinguish different kinds of SystemC processes because they are treated differently later in the VHDL backend: `sc_method` is equivalent to a VHDL process, while `sc_cthread` is synthesized into an FSM; we do not support `sc_thread` currently, but will in the future.

The SPRI partitioner also relies upon this database; therefore, the database is created before the partitioner runs. The partitioner informs the VHDL synthesizer of which objects must be synthesized.

B. LLVM IR Analysis/Manipulation Passes

The six LLVM IR analysis/manipulation passes run on SystemC functions represented in LLVM IR. They detect unsynthesizable constructs and transform them into statements that can be synthesized by the VHDL backend. Note that in the program examples that follow, `*this` points to the current `sc_module`; the `GetElementPtr` instruction computes a pointer to a field in an object.

- **Port Access Canonicalization Pass.** The Port Access Canonicalization Pass canonicalizes the method of access to values contained in SystemC ports and eases the later replacement process in the Function Clone Pass. It detects the special case in which an input port is assigned directly to an output port. The assignment appears as an LLVM call to the function formed by

overloading the "=" operator. The pass splits this special call into two calls: one call which loads the data from the input port and a second call which stores the data to the output port. Program 1 shows an example of this transformation; the original SystemC source was simply `port2 = port1`.

- **`memset/memcpy` Lowering Pass.** The LLVM front-end optimizes the movement of large data structures by using the `memset/memcpy` functions with constant size arguments. We filter them out by recognizing the function names and replace them with a sequence of LLVM `GetElementPtr`, `Load`, and `Store` instructions that set up or move data values at specific locations, as shown in Program 2.
- **Virtual Call Detection and Inlining Pass.** SystemC supports virtual SystemC methods (e.g. `sc_method`, `sc_cthread`, and `sc_thread`), virtual C++ methods, and templates. We handle these constructs in the following fashion:

  Virtual SystemC methods: The SystemC objects database contains pointers to the actual, concrete method which is intended, thus no additional analysis is needed.

  Virtual C++ methods: Virtual calls appear in the LLVM IR as a sequence of LLVM `GetElementPtr` and `Load` instructions corresponding to a traversal of the object and virtual table data structures followed by an indirect function call; the data structures are shown in Figure 3. For each indirect call, we search back in the IR to determine whether the function pointer was loaded from a virtual table, object field, or global pointer and then emulate the instructions which produce the function pointer.[1] This emulation is a form of *partial evaluation*. Once the function pointer is evaluated, we replace the indirect call with a direct call to that function and then use a standard LLVM function to inline the call. Program 3 shows the original LLVM IR for a virtual C++ method call followed by pseudocode for the steps which this pass takes to replace the virtual call. [12] uses a different approach to synthesize virtual methods by creating a switch statement to select possible method bodies; such an approach can be used to synthesize function pointers that are dynamically allocated.

  Templates: At elaboration-time, templates are nothing more than a naming convention for functions. We do not need to know the names of functions in order to look up their IR in LLVM; all we need is a C++ function pointer and these pointers are recorded in the locations previously described.
- **Port Information Recording Pass.** This analysis pass records the locations of SystemC ports in the LLVM IR and categorizes them in terms of input ports and output ports. It also assigns a unique ID to each port. We use unique port IDs in the argument replacement/renaming process of the Function Clone Pass to improve the readability of the generated VHDL code.

[1]We are able to resolve only indirect calls where the pointer is immutable after elaboration; in particular, function pointers which are sent through ports cannot be handled. We have yet to see a structural model which does this.

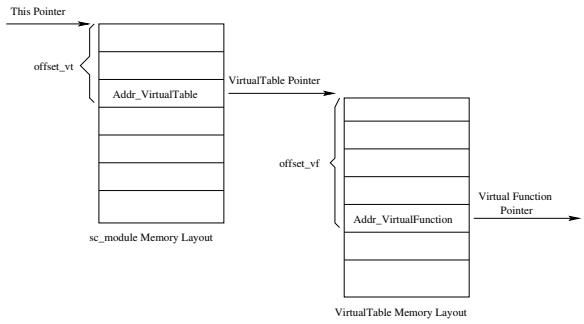

Fig. 3. Virtual Function Pointer Calculation

- **Function Clone Pass.** This pass collects all the methods that need synthesizing and creates new versions of the methods with ports turned into arguments. For each method, we first add new arguments to the method where each argument represents an LLVM `Call` instruction that reads or writes a SystemC port (input port or output port) contained in this function. We define these input-port arguments with primary data types but output-port arguments with pointer types. Second, we replace the uses of all these LLVM `Call` instructions with the function arguments we just added, insert LLVM `Load` or `Store` instructions to operate directly on the function arguments depending on whether they represent input ports or output ports, and then delete the original port-access `Call` instructions. Program 4 displays what a function looks like in LLVM IR before and after it runs through this pass: in Part 1, originally three separate calls are made to load in the values of `port_1` and `port_2` and then store the addition result to `port_3`; in Part 2, this pass simplifies this process by adding two arguments together and storing the result to the third argument.
- **Dead Code Elimination Pass.** We use a standard LLVM pass to erase dead LLVM instructions.

C. VHDL Backend

The VHDL backend synthesizes LLVM instructions to synthesizable VHDL statements. We utilize three strategies: 1) extracting states and translating SystemC `sc_method` and `sc_cthread`; 2) directly mapping synthesizable SystemC subsets to VHDL; 3) instantiating Xilinx IP cores. We directly map an `sc_method` to a VHDL process and instantiate Xilinx IP cores for non-VHDL operators such as signed multiplication/division. The implementation of a data array/structure relies on its size: we map small-size SystemC arrays/structures to VHDL array/record statements but instantiate Xilinx BlockRam IP for large-size ones. Mapping synthesizable SystemC subsets and instantiating Xilinx IP cores are straightforward; however, the synthesis of an FSM requires state extraction.

Program 1 Port Access Canonicalization Pass

Part 1 Original Pseudo LLVM IR:

```
tmp1 = GetElementPtr *this, port_1_offset,              // Get the address of the value of port_1
         sc_signal_in_if_offset;                           through the sc_signal interface
tmp2 = GetElementPtr *this, port_2_offset;              // Get the address of port_2
tmp3 = Call @sc_out<unsigned long>::operator=(          // Load the value from port_1 to port_2
         sc_out<unsigned int> *tmp1,
         sc_port<sc_signal_in_if<bool>,
         1, sc_one_or_more_bound> *tmp2);
```

Part 2 Changes to:

```
tmp1 = GetElementPtr *this, port_1_offset;              // Get the address of port_1
tmp2 = GetElementPtr *this, port_2_offset;              // Get the address of port_2
tmp3 = Call @sc_in<unsigned long>::operator(*tmp1);     // Load the value from port_1
tmp4 = Call @sc_out<unsigned long>::operator=(          // Assign that value to port_2
         *tmp2,*tmp3);
```

Program 2 memset/memcpy Lowering Pass

Part 1 Original Pseudo LLVM IR:

```
tmp1 = GetElementPtr *this, memory_offset;              // Get starting address for memory initialization
tmp2 = BitCast i32* tmp1 to i8*;                        // Pointer cast
Call void @llvm.memset.i32(i8* tmp2,                    // Call memset to initialize 2 memory locations
  InitialV=0, num=2);                                      starting at the tmp2 address with 0
```

Part 2 Changes to:

```
tmp1 = GetElementPtr *this, memory_offset;              // Get starting address for memory initialization
Store InitialV, *tmp1;                                  // Store the initial value
tmp2 = GetElementPtr *this, memory_offset+1;            // Get the next address
Store InitialV, *tmp2;                                  // Store the initial value
```

Program 3 Virtual Call Inlining Pass

Part 1 Pseudo LLVM IR:

```
tmp1 = GetElementPtr *this, vtable_offset;              // Get address of the pointer to the virtual table
tmp2 = Load tmp1;                                       // Load in the pointer to the virtual table
tmp3 = GetElementPtr **tmp2, vfunc_offset;              // Get the address of the virtual function pointer
tmp4 = Load tmp3;                                       // Load in the virtual function pointer
tmp5 = BitCast(...) *tmp4 to (module*)*;                // Cast function pointer for the function call
Call @tmp5(*this);                                      // Call the virtual function
```

Part 2 Pseudo C++ Code for partial evaluation:

```
char **vt_ptr = reinterpret_cast<char**> (              // Get vtable address
  reinterpret_cast<long>(sc_object) +
  vtable_offset);
void *vt = *vt_ptr;                                     // Load the vtable pointer
char ** vf_ptr = reinterpret_cast<char**>(              // Get the virtual function address
  reinterpret_cast<long>(vt) +
  vfunction_offset);
void *vf = *vf_ptr;                                     // Load the virtual function pointer
LLVM::Function *llvm_vf = LLVM::LookupFunction(vf);     // Use an LLVM standard method to look up
                                                           the actually-called function
LLVM::CallInst* CI_dir = CreateCallInst(llvm_vf);       // Create a direct call instruction
LLVM::CallInst.Replace(CI_dir, CI_indir);               // Replace the original indirect call with the new
                                                           direct call
LLVM::InlineFunction(llvm_vf);                          // Use an LLVM standard method to inline
                                                           the corresponding LLVM IR of the virtual function
```

Program 4 Function Clone Pass

Part 1 Original Pseudo LLVM IR:

```
define void @func_name(*this) {
  tmp1 = GetElementPtr *this, port_1_offset;            // Get address of port_1
  tmp2 = GetElementPtr *this, port_2_offset;            // Get address of port_2
  tmp3 = Call @sc_in<unsigned long>::operator(*tmp1);   // Load the value from port_1
  tmp4 = Call @sc_in<unsigned long>::operator(*tmp2);   // Load the value from port_2
  tmp5 = Add tmp3, tmp4;                                // Add the two values
  tmp6 = GetElementPtr *this, port_3_offset;            // Get address of port_3
  tmp7 = Call @sc_out<unsigned long>::operator=(        // Store the add result to port_3
           *tmp6,*tmp5);
}
```

Part 2 Changes to:

```
define void @func_name(*this, port_1, port_2, *port_3) { // port_1, port_2 are input; port_3 is output
  tmp1 = Add port_1, port_2;                            // Add the values of port_1 and port_2
  Store tmp1, *port_3;                                  // Store the result of the add operation to port_3
}
```

SystemC	LLVM	VHDL
class, class inheritance, `sc_method`, `sc_cthread` virtual method, virtual function call (indirect call), template	LLVM function + virtual table	entity, process, FSM
port, `sc_module` member variable, process/method local variable	LLVM module information	VHDL port, signal, variable
bit-wise, logical and arithmetic operations (except mod, div, and signed multiply)	LLVM bit-wise, logical, and arithmetic instructions	VHDL bit-wise, logical, and arithmetic operations (except mod, /, signed *)
Left/right shift	LLVM shift instruction	Combinational logic for zeros adding/deleting
Conditional statement (if else, switch, selection operator)	LLVM branch/switch instruction	Conditional statement (if else, elsif, case-when)
Bounded loop / Unbounded Loop	LLVM loop	VHDL for loop (combinational logic) / FSM
SystemC data type (including 1-D, 2-D array, structure, structure array, statically-allocated pointer)	LLVM type or pointer type	VHDL data type

TABLE II
SYSTEMC TO LLVM AND THEN LLVM TO VHDL CONVERSIONS

- **State Identification.**
  All non-local variables of SystemC processes are part of the simulator's state. This state does not entirely correspond to the target model's state. However, because a hybrid simulator is *not* a prototype of the system, we do not need to distinguish between the two. Therefore, the simulator's state is treated as though it is the target's state and is thus synthesized as a signal. To preserve LLVM's sequential semantics as the simulator's state is updated, all of the VHDL processes we generate first read all of their input signals into variables. The processes then operate only upon the variables. At the end of each process, all of the output signals are assigned. Although the code produced is verbose, the VHDL-to-logic synthesizer later optimizes away variables which are not actually part of the state. The simulator's state is initialized in the hardware; the initial values are located at fixed places of system memory after elaboration and they can be read directly from memory at synthesis time by applying partial evaluation.
- **FSM Extraction.** In SystemC, an FSM can be specified using either `sc_method` or `sc_cthread`. The `sc_method`-based FSM belongs to the synthesizable subset of SystemC and can be directly mapped to VHDL. However an FSM with `sc_cthread` and `wait` statements in the presence of control-flow statements is troublesome and must be interpreted as an FSM in our VHDL backend. We support three forms of control flow in `sc_cthread` FSMs: `wait` statements inside nested `if-else` conditionals, `wait` statements inside `switch` statements, and `wait` statements inside nested loops. The FSM synthesis is a process that extracts a state diagram. In this diagram, each node is a `wait` statement which is referred to as a state, and each edge represents an inter-state transition. An example is displayed in Figure 4. Applying a similar method as in [7], we implement our FSM synthesis in four steps: 1) identifying `wait` as a state; 2) traversing all the `wait` statements directly reachable from the current state; 3) interpreting the path from one `wait` to a directly-reachable `wait` as a state transition; and 4) evaluating the code between two `wait` statements before the transition is taken.

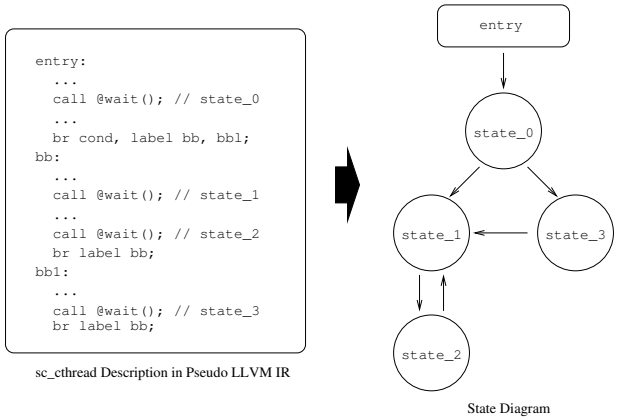

Fig. 4. Example of FSM Creation

V. CASE STUDY

Currently, the SPRI synthesizer supports all of the SystemC synthesizable subset and the additional C/C++ constructs shown in Table II. To verify its correctness, we use SPRI to synthesize two SystemC-based microarchitectural simulators. We test the SPRI synthesizer on a DRC 1000 system with a dual-core AMD Opteron-275 CPU running at 2.1 GHz with 2 GB of system memory and a Xilinx XC4VLX60-11 FPGA as the coprocessor. We use Xilinx ISE 8.2 to synthesize the SPRI-generated VHDL code for the FPGA.

We present the characteristics of these two SystemC simulators before and after synthesis in terms of lines of code, maximum FPGA frequency, FPGA resource utilization, synthesis time from SystemC to VHDL, and synthesis time from VHDL to gates.

Measurement	Microlib DLX Model
Lines of Source Code / VHDL Generated	2200 / 5311
Max FPGA Frequency (MHz)	125
FPGA Resource Utilization (Slices % / RAM %)	27 / 30
SystemC to VHDL Time (min)	<1
VHDL to Gates Synthesis Time (min)	7

TABLE III
DLX SYNTHESIS RESULTS

- **DLX Microarchitectural Simulator.** The first simulator synthesized is a Microlib fine-grain microarchitectural simulator modeling a five-stage, in-order pipeline implementing the DLX instruction set [11]. This simulator is composed of 23 module instances that describe registers and combinational logic with `sc_method`. The SystemC constructs used by the DLX model include 1-D port arrays, 2-D variable arrays, 1-D structure arrays, local pointers, left/right shift operators, modulo operators, bit-wise/arithmetic operators, initialization functions, bounded loops, conditional statements, and template classes. In order to verify the detection of class inheritance and the synthesis of virtual methods, we modified the simulator so that the data memory and instruction memory are inherited from a shared memory class but their behaviors are specified differently within the virtual methods. We synthesize a single hybrid simulator with all components in HW; the experimental results are shown in Table III.
- **RISC Microarchitectural Simulator.**
  The second simulator is the RISC microprocessor model example in the reference SystemC 2.2 package. It has a higher abstraction level than the Microlib DLX model. The RISC simulator describes a standard five-stage pipeline with each stage modeled by a `sc_cthread` FSM. It is composed of ten components: nine `sc_cthread` FSMs and one `sc_method` module (the PIC module). The three forms of `sc_cthread` FSMs discussed in Section IV-C are all included in this model. Additionally, it uses 1-D arrays, array pointers, bit-wise/arithmetic operators, initialization functions, unbounded loops, wait statements, conditional statements, switch statements, selection operators, signed multiplications, and signed divisions. We generate ten different hybrid simulators; each simulator has one component in HW. The entire model cannot be moved to the FPGA due to resource constraints. For each generated RISC hybrid simulator, we list the characteristics of its HW portion in Table IV. Note that the period/frequency constraint does not apply to the PIC unit because it is combinational logic.
  The SPRI synthesizer instantiates Xilinx IP cores for signed multiplications and divisions in the EXEC, FLOATING, and MMXU modules. It also generates different code for data arrays based on their sizes. For example, a 500×32-bit array in the ICACHE module is synthesized by using look-up tables (LUTs) which occupy many FPGA slices; however, the three 4000×32-bit arrays in the DCACHE module are replaced with Xilinx BlockRam IP cores to save the FPGA slice resource.

From these experiments we make several observations. First, we have demonstrated what our tool can successfully synthesize. All SystemC synthesizable subsets (e.g. bit-wise/arithmetic operators, shift operators, selection operators, and conditional statements) are supported. We map each `sc_method` directly to a VHDL process and transform each `sc_cthread` to an FSM; `sc_thread` will be supported in the future. We can also handle general C/C++ constructs such as statically-allocated pointers, unbounded loops, virtual methods, class inheritance, and templates. Second, we have shown that it is likely to be much more efficient to synthesize hybrid simulators from software microarchitectural simulators than to manually implement them.[2] Third, we noticed that the equivalent VHDL models generated by the SPRI synthesizer occupy many more lines of code than the original SystemC models. This, however, does not affect the performance of the generated VHDL models because over 80 percent of the extra code is created for the initialization of state arrays and memory blocks.

In addition to the immediate results, we observed a need for future exploration. The SPRI synthesizer instantiates Xilinx signed multiplier, divider, and BlockRam IPs to replace signed multiplication/division operators and data arrays. Such a replacement introduces multi-FPGA-cycle delay to any state in VHDL that contains these operations; however, these are assumed to be single-cycle operations in SystemC. The disparity in cycles can break communication timing among interconnected components in HW. We currently work around this problem by forcing each component in HW to be synchronized with SW in every target simulation cycle through handshake signals. However, such interface design results in a heavy synchronous-communication cost; thus we have not achieved a significant speedup (in fact, less than 2) even when we move all components of a SW simulator into HW, because a target simulation clock remains in SW and all HW operations have to be synchronized with it [19]. Thus, it will be necessary to use asynchronous communication techniques; we plan to use the theory of Latency-insensitive Bounded Dataflow Networks (Li-BDNs) to create Li-BDNs [24]. Li-BDNs allow different components or instances to execute for different numbers of cycles without affecting correctness and thereby enable them to be hooked up arbitrarily.

VI. CONCLUSIONS AND FUTURE WORK

Structural modeling eases the design of microarchitectural simulators by using high-level language constructs to increase model reuse. Hybrid simulators increase the

[2]The FPGA placing and routing may require several hours to complete, depending on the complexity of the simulators — still a vast improvement over what usually takes months or years to do manually!

Measurement	RISC Unit Moved to HW									
	DECODE	FETCH	BIOS	PAGING	EXEC	FLOATING	MMXU	ICACHE	PIC	DCACHE
Lines of Source Code / VHDL Generated	800 / 1840	147 / 477	80 / 4116	97 / 136	177 / 572	160 / 275	232 / 591	110 / 670	50 / 220	115 / 12115
Max FPGA Frequency (MHz)	155	230	171	407	248	113	327	187	N/A	157
FPGA Resource Utilization (Slices % / RAM % / DSP %)	6 / 0 / 0	1 / 0 / 0	3 / 30 / 0	1 / 0 / 0	6 / 0 / 4	5 / 0 / 4	2 / 0 / 4	50 / 0 / 0	1 / 0 / 0	6 / 90 / 0
SystemC to VHDL Time (min)	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
VHDL to Gates Synthesis Time (min)	3	<1	5	<1	1.5	1.5	4	8	<1	7

TABLE IV
RISC SYNTHESIS RESULTS

speed of these structural simulators. In order to avoid time-consuming manual implementation of such simulators, we provide an efficient alternative by automatically synthesizing SystemC-based simulators into hybrid ones. We use an elaboration-time synthesis method because resolving high-level constructs and detecting the module hierarchy require access to instantiated SystemC objects stored in memory. Our synthesis tool performs LLVM IR analysis and manipulation for high-level language constructs as part of the SPRI runtime environment. The near future work will concentrate on extending SPRI to support the transformation to Li-BDNs and speeding up SPRI-produced hybrid simulators. We plan to release the complete infrastructure in 2011.

VII. ACKNOWLEDGMENTS

This work has been supported by National Science Foundation grant CCF-1017004. We would also like to acknowledge Daniel Gracia Pérez, Giles Mouchard, and Olivier Temam for providing access to the Microlib DLX model.

REFERENCES

[1] *IEEE Std 1666-2005: IEEE Standard SystemC Language Reference Manual*. IEEE, 2005.

[2] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. Penry, O. Temam, and N. Vachharajani, "UNISIM: An open simulation environment and library for complex architecture design and collaborative development," *IEEE Computer Architecture Letters*, vol. 6, no. 2, pp. 45–48, July–December 2007.

[3] N. Blanc, D. Kroening, and N. Sharygina, "Scoot: A tool for the analysis of SystemC models," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 467–470, 2008.

[4] J. Castillo, P. Huerto, and J. I. Martinez, "An open-source tool for SystemC to Verilog automatic translation," *Latin American Applied Research*, vol. 37, pp. 53–58, 2007.

[5] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. H. Reinhart, D. E. Johnson, and Z. Xu, "The FAST methodology for high-speed SoC/computer simulation," in *Proceedings of the 2007 IEEE/ACM International Conference on Computer-aided Design*, 2007, pp. 295–302.

[6] E. S. Chung, J. C. Hoe, and B. Falsafi, "ProtoFlex: Co-simulation for component-wise FPGA emulator development," in *Proceedings of the 2nd Annual Workshop on Architecture Research using FPGA Platforms*, 2006.

[7] C. Cote and Z. Zilic, "Automated SystemC to VHDL translation in hardware/software codesign," in *Proceedings of the 9th International Conference on Electronics, Circuits and Systems*, vol. 2, 2002, pp. 717 – 720.

[8] R. Drechsler, G. Fey, C. Genz, and D. Grobe, "SyCE:an integrated environment for system design in SystemC," in *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping*, 2005, pp. 258–260.

[9] "Cynthesizer closes the ESL-to-silicon gap," Forte Design Systems, 2010. [Online]. Available: http://www.forteds.com/products/cynthesizer.asp

[10] "Kascpar – Karlsruhe SystemC parser suite," FZI – Microelectronic System Design, 2006. [Online]. Available: http://www.fzi.de/downloads/sim/archives/kascpar-documentation.pdf

[11] D. Gracia Pérez, G. Mouchard, and O. Temam, "A new optimized implementation of the SystemC engine using acyclic scheduling," in *Proceedings of the Design Automation and Test in Europe Conference*, February 2004, pp. 552–557.

[12] E. Grimpe and F. Oppenheimer, "Extending the SystemC synthesis subset by object-oriented features," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2003, pp. 25–30.

[13] W. Klingauf and M. Geffken, "Design structure analysis and transaction recording in SystemC designs: A minimal-intrusive approach," in *Proceedings of the Forum on Specification and Design Languages*, 2006, pp. 169–177.

[14] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.

[15] K. Marquet, M. Moy, and B. Karkare, "A theoretical and experimental review of SystemC front-ends," Verimag Research Center, Tech. Rep. TR-2010-4, 2010.

[16] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip," in *Proceedings of the 5th ACM International Conference on Embedded Software*, 2005, pp. 317–324.

[17] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors, "Exploiting parallelism and structure to accelerate the simulation of chip multi-processors," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, February 2006, pp. 29–40.

[18] D. A. Penry, Z. Ruan, and K. Rehme, "An infrastructure for HW/SW partitioning and synthesis of architectural simulators," in *2nd Workshop on Architectural Research Prototyping*, 2007.

[19] Z. Ruan and D. A. Penry, "Partitioning and synthesis for hybrid architectural simulators," in *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems*, 2010.

[20] T. Schubert and W. Nebel, "The QUINY SystemC front end: Self-synthesising designs," *Advances in Design and Specification Languages for Embedded Systems*, pp. 93–109, 2007.

[21] S. Sirowy, B. Miller, and F. Vahid, "Portable SystemC-on-a-chip," in *Proceedings of the 7th IEEE/ACM International Conference on Hardware/software Codesign and System Synthesis*, 2009, pp. 21–30.

[22] S. Sundralingram, "SystemC modeling, synthesis, and verification in CATAPULT C," Mentor Graphics, Tech. Rep., February 2010.

[23] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August, "Microarchitectural exploration with Liberty," in *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, November 2002, pp. 271–282.

[24] M. Vijayaraghavan and Arvind, "Bounded dataflow networks and latency-insensitive circuits," in *Proceedings of the 7th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, July 2009.