Center for Embedded Computer Systems
University of California, Irvine

A Hybrid Instruction Set Simulator for System Level Design

Yitao Guo, Rainer Doemer

Technical Report CECS-10-06
June 11, 2010

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{yitaog,doemer}@cecs.uci.edu
http://www.cecs.uci.edu

A Hybrid Instruction Set Simulator for System Level Design

Yitao Guo, Rainer Doemer

Technical Report CECS-10-06
June 11, 2010

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{yitaog,doemer}@cecs.uci.edu
http://www.cecs.uci.edu

Abstract

Validation is an essential step in System Level Design (SLD) for Multiprocessor System on Chip (MPSoC). Traditional Instruction Set Simulators (ISS) are often either slow (interpretive ISS) or unable to handle accurate multiprocessor simulation (static or dynamically compiled ISS). In this technical report, we propose a hybrid simulation scheme [10] which combines interpreted and static compiled ISS. The proposed ISS is free to execute a target function either natively or in interpreted mode. With the aid of System Level Description Languages (SLDL) like SpecC/SystemC, the designer using proposed ISS is able to differentiate the computation portion and the communication portion of the target code. By executing the computation intensive code on the host natively and the communication portion in interpreted mode, the proposed ISS is able to speed up the simulation significantly while maintaining acceptable accuracy and

support for multiprocessor simulation. We have implemented the proposed scheme based on SWARM [6] [14] [12] simulator and have conducted experiments with several real-life designs. Our test results show that the proposed ISS provides significant speedup in simulation time and maintains low error in timing estimation.

Contents

List of Figures

List of Tables

A Hybrid Instruction Set Simulator for System Level Design

Y. Guo, R. Doemer
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA
login@cecs.uci.edu
http://www.cecs.uci.edu

Abstract

Validation is an essential step in System Level Design (SLD) for Multiprocessor System on Chip (MPSoC). Traditional Instruction Set Simulators (ISS) are often either slow (interpretive ISS) or unable to handle accurate multiprocessor simulation (static or dynamically compiled ISS). In this technical report, we propose a hybrid simulation scheme [10] which combines interpreted and static compiled ISS. The proposed ISS is free to execute a target function either natively or in interpreted mode. With the aid of System Level Description Languages (SLDL) like SpecC/SystemC, the designer using proposed ISS is able to differentiate the computation portion and the communication portion of the target code. By executing the computation intensive code on the host natively and the communication portion in interpreted mode, the proposed ISS is able to speed up the simulation significantly while maintaining acceptable accuracy and support for multiprocessor simulation. We have implemented the proposed scheme based on SWARM [6] [14] [12] simulator and have conducted experiments with several real-life designs. Our test results show that the proposed ISS provides significant speedup in simulation time and maintains low error in timing estimation.

1 Introduction

1.1 System Level Design Methodology

System Level Design (SLD) reduces the design complexity of Multi-Processor System on Chip (MPSoC) and enables the designer to explore different software/hardware partitions at an early stage.

The growing complexity of SoC requires higher level of abstraction. Figure 1 [9] shows that, as the abstraction level goes down, the design complexity grows exponentially. Due to cost, efficiency, power consumption and real time requirements of embedded systems, design space needs to be explored on different abstraction levels. System Level Description Languages (SLDL) model software and hardware for embedded systems on different abstraction levels.

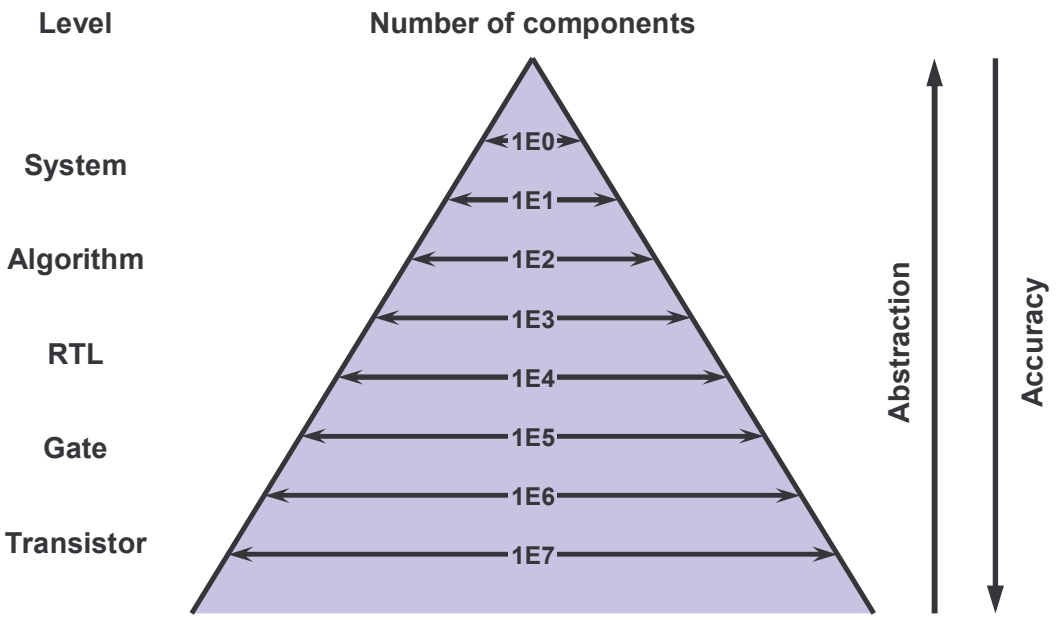

Figure 1: Design Complexity and Abstraction Level [9]

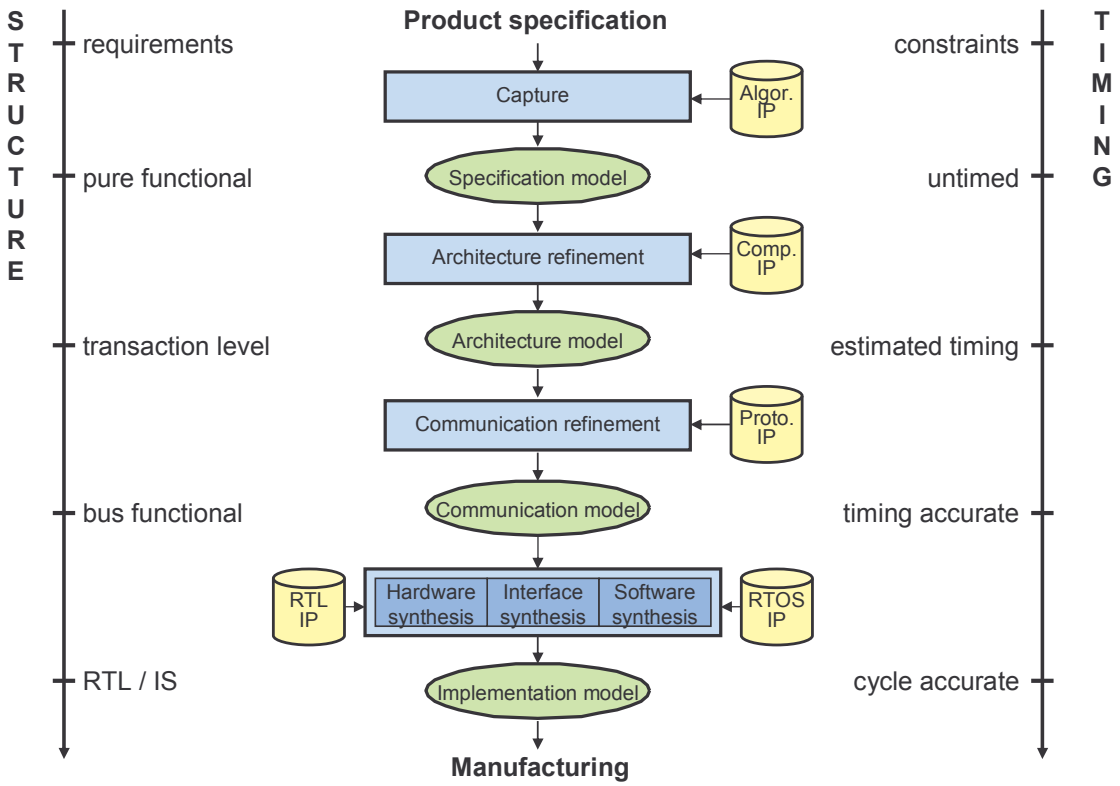

Figure 2: Top-down System Level Design Methodology [9]

1.1.1 Top-down Design Methodology

Figure 2 [9] shows a top-down work flow of system level design methodology. On each abstraction level, the flow is able to refine HW/SW for different design choices. The product specification is captured to generate the specification model which contains the pure function of the product. Computation and communication of the specification model are organized in behaviors and channels. A profiling tool can be used to estimate the execution time of each behavior for a given input. After architecture refinement, behaviors in specification model are mapped to different processors. At this stage, a Transaction Level Model (TLM) is generated to approximate the communication and the computation. Architecture independent profiling tools could ensure the estimation fidelity at this stage of design exploration, but TLM is unable to provide accurate performance estimation for timing of computation and communication. Communication refinement is used to map abstract channels to concrete communication protocol and hardware. After communication refinement, a Bus Functional Model (BFM) is generated. In order to get pin-accurate and cycle-accurate simulation result, hardware synthesis and software synthesis are applied to the bus functional model. An Instruction Set Simulator (ISS) is plugged in to provide cycle-accurate estimation of computation

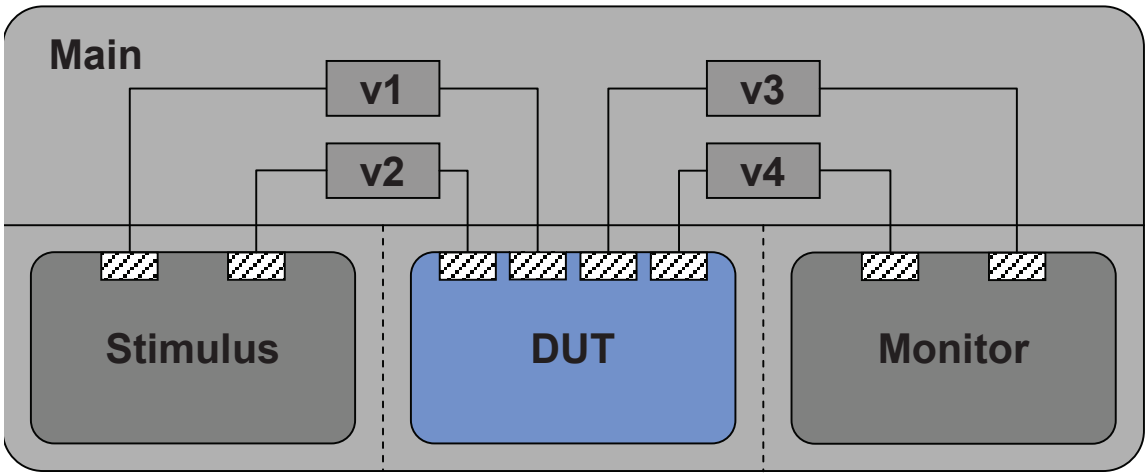

Figure 3: Functional Validation [9]

functions running on processors.

1.1.2 Validation

Accurate and fast performance estimation is crucial in SLD. On each level of SLD, functional and performance validations are required for the verification of the design. Different simulators are used in the validation process to simulate software and hardware behaviors.

Functional Functional validation verifies the correctness of the execution model. Figure 3 shows a common set up for functional validation. The stimulus produces input which is processed by Device Under Test (DUT), and the monitor verifies the result produced by the DUT.

Performance Performance validation estimates the performance for hardware and software. Different simulators are used to provide performance estimation in different models. The simulators are coordinated by a Discrete Event Simulator provided by the SLDL library.

1.2 Design Models at Different Levels of Abstraction

As shown in Section 1.1, the complexity of design and validation grows exponentially as the abstraction level goes down. In this section, we will discuss modeling embedded systems on different abstraction levels as shown in Figure 4.

1.2.1 Pure Functional Model

The pure functional model only simulates the functionality of the application and estimates the performance for pure computation. An architecture independent profiling tool is used in architecture

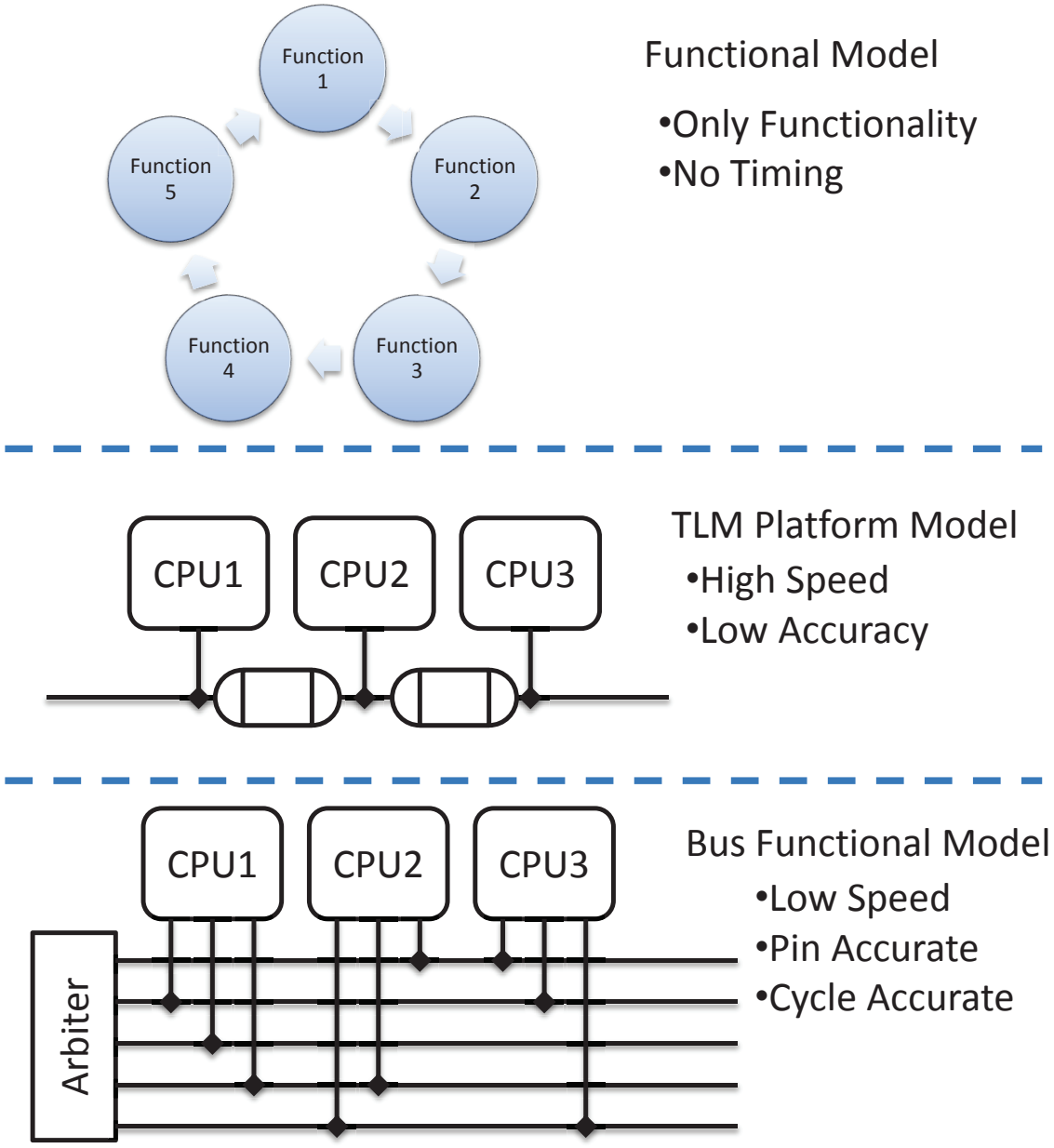

Figure 4: System Level Design Models at Different Abstraction Levels

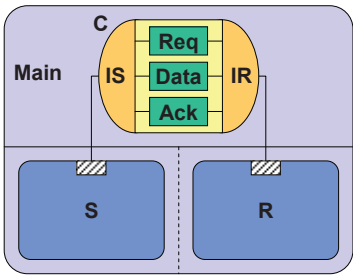

Figure 5: Transaction Level Model [9]

refinement to compare different algorithm mappings. The estimated timing is very inaccurate in terms of cycle count but accurate enough to show fidelity for comparing different algorithm partitions on multiple processors. The pure functional model takes the shortest simulation time, but cannot provide either estimation of actual execution time of certain architecture or the estimation of communications between individual processors.

1.2.2 The Transaction Level Model (TLM) Platform Model

Transaction Level Model models the communications between processors with abstract interface without implementation details of communication protocol. It encapsulates the communications in channels and simulates the exchange of data and the order of data transaction. TLM can be used to simulate the communication on abstraction level and verify the correctness of communication. The system-level designer will have to apply detailed communication protocol information to the abstract channels in following design stages. Thus, TLM is not capable of providing cycle-accurate bus transaction timings.

The TLM Platform Model simulates both the computation on individual processors and the communication among the processors. It provides cycle-approximate estimation for both communication and computation. The TLM Platform Model is not capable of simulating bus transactions cycle-by-cycle, thus it cannot provide accurate estimation of communication.

1.2.3 Bus Functional Model

The bus functional model provides a cycle-accurate estimation of communications on the bus. But inside each processor, a cycle-approximate simulator is used to provide performance estimation. Bus transactions are simulated by RTL model, thus BFM provides cycle-accurate and pin-accurate simulation for communications. Bus functional model takes substantially longer simulation time than TLM, but it provides a much more accurate estimation of communication.

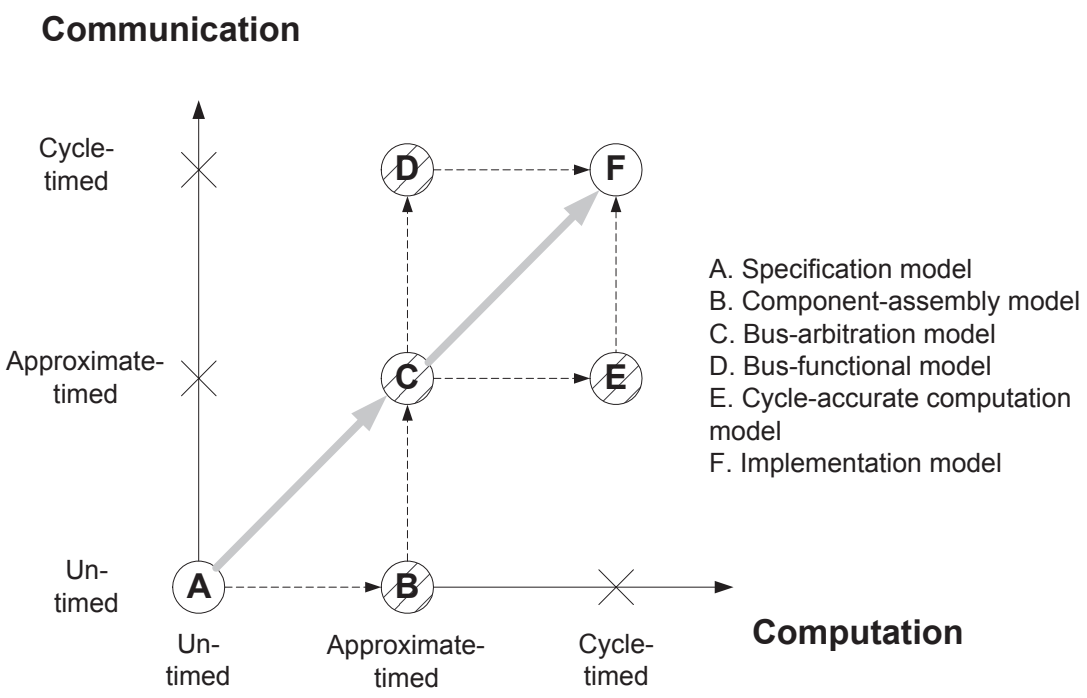

Figure 6: System Modeling Graph [5]

1.2.4 Implementation Model

The implementation model provides a cycle-accurate estimation for software and a pin-accurate estimation for hardware. The software code is synthesized and loaded by an interpreted ISS which is plugged back into the implementation model to provide cycle-accurate timing estimation. Bus transactions are simulated by RTL model, thus it provides pin-accurate result. The implementation model provides cycle-accurate and pin-accurate simulation for both communication and computation. Due to the complexity of implementation model simulation, it takes the longest simulation time.

1.2.5 Comparison of Different Models

The trade off between simulation accuracy and simulation time has been thoroughly discussed by Cai and Gajski [5]. As shown in 6 [5], for both communication and computation, there are three degrees of time accuracy: un-timed, approximate-timed and cycle-timed, which corresponds to no cycle count, cycle approximate and cycle accurate in this technical report. Each model described in the previous sections is represented by a point in the figure (A, B, C, D, F). In the SLD work flow, as the design flow approaches implementation model (point F), the simulation time increases

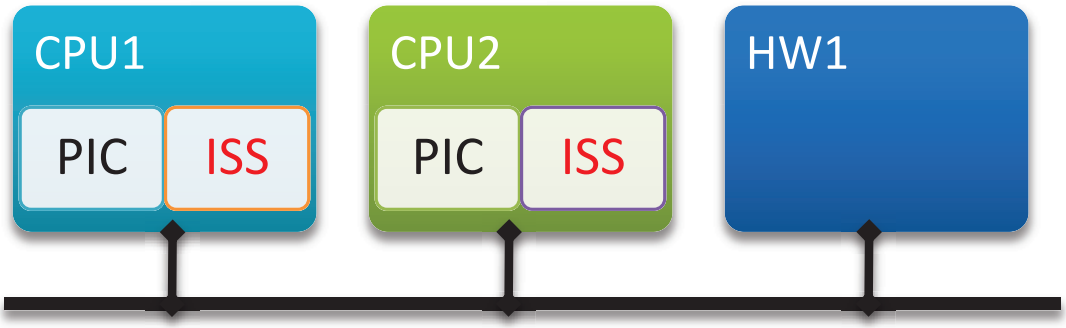

Figure 7: ISS in Bus Functional Model

exponentially. To achieve high accuracy and short simulation time, we proposed a hybrid model between D and F, which simulates communication functions in cycle-accurate mode and computation functions in cycle-approximate.

1.2.6 Proposed Hybrid Model

The proposed hybrid model combines the TLM Platform Model and the Bus Functional Model. Instead of plugging in an interpreted ISS which provides cycle-accurate timing estimation in the implementation model, we plug in a hybrid ISS which provides cycle-approximate timing estimation.

In SLDL (such as SystemC [3], SpecC [8]), computation is organized into behaviors, and communication is organized into channels. In TLM, a channel is an abstract description of communication, which can be implemented into different buses or protocols in BFM. It describes both the communication protocol and the interface with behaviors. When a channel is instantiated, the interface code will be generated and plugged back into the behaviors [9]. Therefore, SLDL defines a clear boundary between computation and communication in synthesized target source code.

Figure 7 shows the bus functional model of an embedded system. The system contains 2 CPUs and a custom hardware. All processors are connected to a common bus. The ISS is plugged into the CPU to provide performance estimation in terms of cycle count. Synthesized target source is loaded in corresponding ISS. Traditionally, an interpreted ISS is plugged into the processor to provide pin-accurate and cycle-accurate simulation. A centralized simulation engine coordinates the simulation. The simulation engine drives the interpretive ISS in each processor cycle by cycle. By the end of each bus cycle, the simulation engine will perform bus transactions thus the communication between different processors will be simulated. If the proposed ISS is plugged in, only communication code will be executed cycle-by-cycle, which will result in a pin-accurate and cycle-approximate simulation.

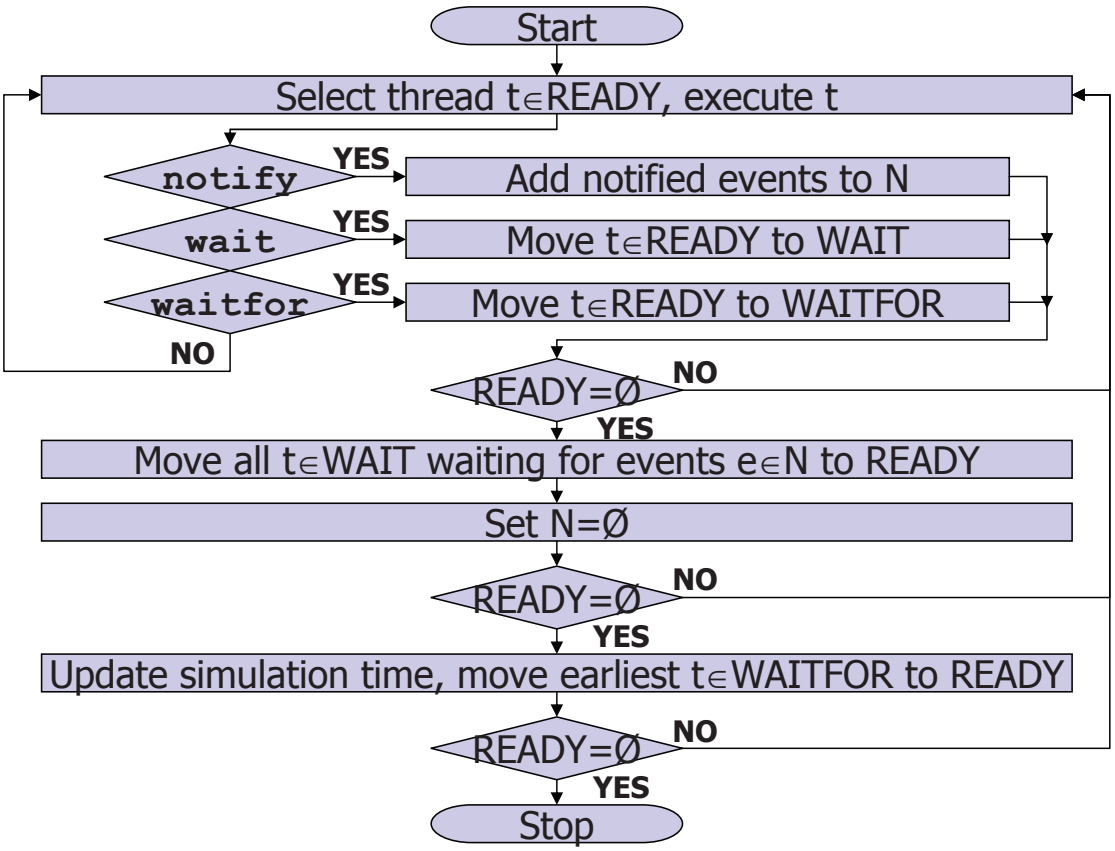

Figure 8: Discrete Event Simulation Model [9]

1.3 Discrete Event Simulation

In Discrete Event (DE) Simulation, each discrete event contains an instant in time and a change of system state. The simulation engine applies the discrete events in chronological order. In the context of SLD, events are changes of internal state of a processor or communications between processors.

As Figure 8 shows, threads are organized in READY queue, WAIT queue and WAITFOR queue. The WAITFOR queue is a priority queue sorted by the wait time of threads. Initially all threads are put into the READY queue. The simulation engine picks a thread from the READY queue and executes the thread. The thread will either notify an event or be moved to WAIT queue or WAITFOR queue. When the ready queue is empty, the notified threads in the WAIT queue are moved to the READY queue. If the READY queue is still empty, the first thread in the WAITFOR queue will be selected, and simulation time will advance to the time that the thread waits for. If the READY queue is still empty, either the simulation ends or a deadlock occurs.

1.3.1 Simulators

Hardware/Software simulators are employed by the Discrete Event simulation engine to provide performance estimation / functional simulation for different hardware. In this section, we review some common simulators for hardware/software.

Hardware Simulator

VHDL/Verilog VHDL [2]/Verilog [1] are hardware description languages that model the behavior, structure, functional, and physical properties of hardware. They are also used for synthesis on different abstraction levels and simulate the behavior of hardware with a Discrete Event simulation engine.

Software Simulator

Profiler A profiler simulates the execution of a program by sampling the function calls at constant time intervals. With the statistics provided by the profiler, the program designer can easily discover any bottlenecks of the program and optimize the code.

Instruction Set Simulator An Instruction Set Simulator (ISS) simulates the internal state of a processor when executing a input binary code. The ISS fetches, decodes and executes each instruction like a real processor and updates the processor's internal state. With the help the ISS, we can accurately estimate the processor cycle count of executing a given input binary code. Other statics like memory hierarchy statistics can also be retrieved by an ISS.

1.3.2 SpecC/SystemC Simulator

The System Level Description Language employs a Discrete Event simulator. All hardware units are treated as threads in Figure 8. In different models, hardware units will be synthesized with different IPs.

1.4 Instruction Set Simulator

Instruction Set Simulator emulates the behavior of a processor. An ISS models both the function and internal states of a processor. Generally, the internal states include the registers, memory, pipeline and cache.

1.4.1 Classification

Traditional instruction set simulators can be categorized into three classes: Interpreted, Static Compiled and Dynamically Compiled.

Interpreted ISS An interpreted ISS executes the target binary instruction by instruction. In each iteration, the interpreted ISS fetches, decodes and executes the instruction. The interpreted ISS's are slow due to their interpreted nature, but provide very detailed estimation of cache, pipeline and memory model. The interpreted ISS's are very flexible. They provide estimations for various purposes.

Static Compiled ISS A static compiled ISS compiles the target code into host binary. Some approaches compile from the target source and insert timing estimation on source level, while other approaches map the instructions and registers of the target architecture to the host architecture and translate the target binary to host binary. The static compiled ISS is very fast in terms of simulation time. But it cannot provide accurate estimation for communication since the target code is executed in "one shot" without interrupt.

Dynamically compiled ISS A dynamically compiled ISS is a combination of static compiled and interpreted ISS. A dynamically compiled ISS can behave like an interpreted ISS which executes the binary the target binary instruction by instruction. It could also compiles part of the target code to host binary. Some approaches profile the target code and translate the most commonly used target code segment into host binary, while other uses Just In Time (JIT) compilation technique to compile the target code into host binary on the fly. Dynamically compiled ISS provides moderate accuracy in terms of performance estimation and relatively high simulation speed. With proper modifications, dynamically compiled ISS can be used to provide estimation for communication.

1.5 Related Work

A lot of work has been done to speed up the ISS simulation. Some approaches employ compiled ISS by mapping the target architecture to the native architecture and translate the target binary into native binary on assembly level [16]. Some approaches employ Just-In-Time (JIT) compilation and compile the most frequently executed target code into native code at run-time [7]. Some approaches compile part of the simulation code into dynamic link libraries, and load the libraries whenever native execution is needed [7]. But none of these approaches is aware of the separation of communication and computation in the source code which could be defined by a SLDL as described in section 1.2.6. And these approaches are designed under different assumptions of a partqicular application, but none of which is optimized for performance estimation of SLD.

A novel Timed TLM has been proposed [11] to tackle early stage system level design space exploration. It is fast, accurate and retargetable. But it cannot be categorized into ISS and cannot be adapted in traditional SLD performance estimation.

Other hybrid ISS approaches have also been proposed before [13]. But the native executed functions are selected according to an algorithm to minimize the energy estimation error. The overhead of global variable access in native functions is substantial. This approach is also not optimized for simulation for Multi-Processor System on Chip (MPSoC).

The proposed ISS takes advantage of the awareness of computation and communication boundary of SLDL. It provides a simple and efficient approach to provide performance estimation for

SLD and accurate cycle timing for communication, which is not optimized by any of the existing compiled ISS. With accurate cycle timing of communication, the proposed approach could be easily adapted into SLDL simulation engine which supports MPSoC.

The rest of this technical report is organized as follows: Chapter 2 will introduce the idea of the proposed ISS approach and the general process of code generation. Chapter 3 will give a detailed explanation of code generation process. Chapter 4 will give three examples and present some experimental results. Chapter 5 concludes the technical report and discusses future work.

2 Work Flow

In this chapter, we will describe our proposed hybrid ISS approach in general and compare different design choices of realizing our design specifications. First, we will describe how an ISS is adapted to SLD context. Then we are going to discuss the challenges of proposed hybrid ISS approach. Finally, we will describe the general work flow of our proposed approach.

2.1 Instruction Set Simulator (ISS) in System Level Design (SLD) Context

In order to accurately simulate a Bus-Functional Model (BFM), the design flow requires cycle-accurate simulation for communications between processors. For communication intensive applications, an interpreted ISS can be used to provide cycle-accurate simulation for all stages. However, many real life applications are computation intensive, in which case execution of computation code takes most of the simulation time. Thus simulating the computation code at the higher level in compiled mode will drastically shorten the simulation time. That is the idea of our hybrid ISS approach.

Traditionally, an ISS is integrated in the processor in BFM model by use of a wrapper. The wrapper is a module written in SLDL that communicates with the underlying Discrete Event simulation engine and drives the ISS simulation. Every time the wrapper code is executed, the ISS will be driven to step one cycle ahead. Then the wrapper performs I/O transactions via the processor bus and checks/sets for interruption with the Programmable Interruption Controller (PIC). Finally, the wrapper issues a *wait_for* request to notify the simulation engine to execute the wrapper again after exactly one processor cycle. For example, as shown in Figure 2.1 in the traditional model, the wrapper first calls *iss.init()* to initialize the state of the ISS. Then the wrapper enters an infinite loop. Every time the underlying Discrete Event (DE) simulation engines resumes the execution of the wrapper, the wrapper will first call *iss.cycle()* to drive the ISS step one cycle. Then *iss.read()* and *iss.write()* are called to perform I/O transactions. Finally the wrapper calls *wait_for* function to notify the DE to resume the execution of the wrapper after one CPU cycle's simulation time.

In our hybrid ISS approach, instead of driving the ISS exactly one cycle every time, a *step()* function is used to drive the ISS to either step one cycle in interpreted mode or step several cycles in compiled mode. The cycle count that the ISS actually stepped is then recorded by *cycle_cnt*. After performing bus transactions, the wrapper notifies the simulation engine to execute the wrapper code again after *cycle_cnt* processor cycles. For example, as shown in Figure 2.1 in the hybrid model, the wrapper first calls *hybrid_iss.init()* to initialize the state of the ISS. Then the wrapper enters

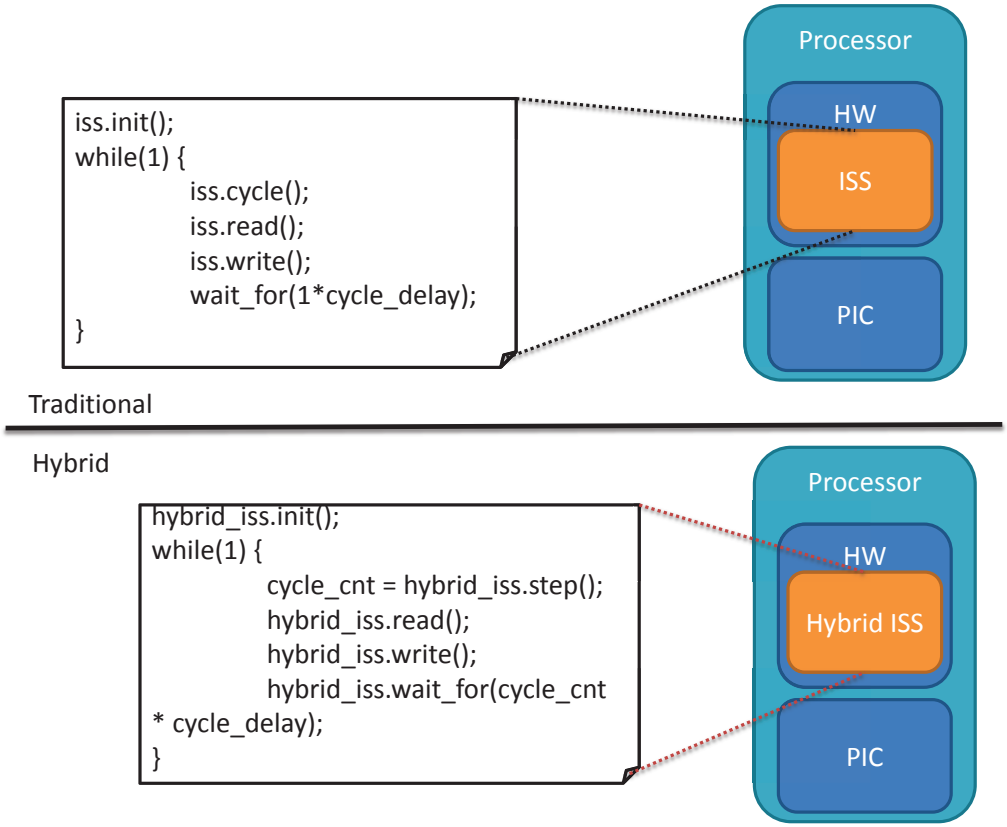

Figure 9: Traditional and Hybrid ISS in BFM

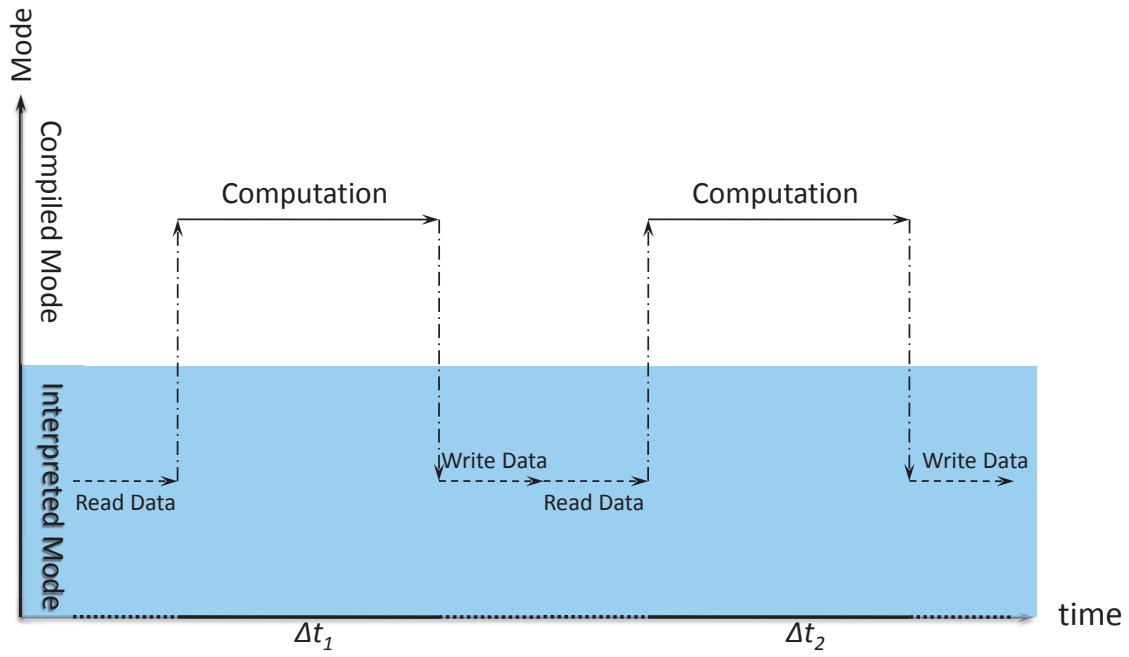

Figure 10: Execution Mode Switch with Proposed Hybrid Approach

an infinite loop. Every time the underlying Discrete Event (DE) simulation engines resumes the execution of the wrapper, the wrapper will first call *hybrid_iss.step()* to drive the ISS to execute one step. The number of the actual CPU cycle executed depends on the mode of the ISS. If it is in interpreted mode, exactly one cycle will be executed. If it is in higher level compiled mode, the computation function will be executed in "one shot" and the corresponding cycle count will be written to *cycle_cnt*. Then *iss.read()* and *iss.write()* are called to perform I/O transactions. Finally the wrapper calls *wait_for* function to notify the DE to resume the execution of the wrapper after *cycle_cnt* CPU cycles' simulation time.

2.2 Hybrid ISS Approach

The proposed hybrid approach employs two execution modes: Compiled Mode and Interpreted Mode. A program will execute as Figure 10 shows. The execution will jump back and forth between interpreted and compiled mode. In Interpreted Mode, the code is executed cycle-by-cycle, which is cycle accurate. In Compiled Mode, the code is executed in "one shot". Timing is annotated at the end of the computation, which is cycle approximate. In our implementation, the computation code can be executed either in Compiled Mode to speed up the simulation or in Interpreted Mode to provide performance estimation related information. Communication code (Read Data and Write Data in Figure 10) is executed in Interpreted Mode since there could be external I/O operations or bus transactions which need synchronization. By the end of each clock cycle, bus transactions

are simulated by the simulation engine of SLDL. This way, we can speed up the simulation without sacrificing the accurate timing for communication. The proposed ISS is cycle-accurate during communication and cycle-approximate during computation.

The major challenge of hybrid ISS approach is to coordinate the execution of target code and native code, which should:

- Ensure the correctness of the execution.
- Make a good trade-off between simulation accuracy and simulation speed.

In this section, we will discuss different approaches to ensure correctness and make a comparison in terms of execution efficiency, implementation complexity, portability and simulation accuracy.

2.2.1 Design Choice: Assembly Level Translation vs. Source Level Compilation

To maintain adaptability and efficiency, traditional compiled ISS approaches convert target code to host binary on assembly level [16]. It is a universal, fast but "dirty" solution, which could handle execution mode switch on instruction level but is barely retargetable. The translation on assembly level is a great choice for compiled simulation. But there could be many problems adapting the scheme to hybrid simulation in System Level Design context, since we need to switch the execution between host and target.

- A lot of code needs to be rewritten for different target architectures. Mapping one instruction set to another could be a lot of work if the designer wants to explore design options on different architectures on different host platforms.
- Relocating the target binary code could be a problem in our hybrid scheme. Although a code generation tool can be used to maintain a symbol table and handle the relocation.

The problem with assembly level translation is that a lot of the translation work has already been implemented by assembler and compiler. Thus, it would be much more convenient if we manipulate the target code on source level and let the compiler perform the low level operations. A switch between the compiled mode and interpreted mode can be realized by using a *system call*. The *system call* in the context of our proposed ISS approach, is similar to but not exactly the same as the system call of an OS. It is triggered by a *trap* instruction and handled by interrupt handlers in the ISS. We also need to make sure the behavior of the program is properly synchronized on binary level. The major challenge of this approach is to coordinate execution of compiled and interpreted mode and make sure the variables on the host side and the target side are properly synchronized.

2.2.2 Execution Mode Switch

In the proposed approach, execution mode switch is needed only on the computation and communication boundaries. With the ability to separate computation and communication codes of SLDL, we are able to encapsulate all computation code blocks in separate functions. Thus we can restrain the

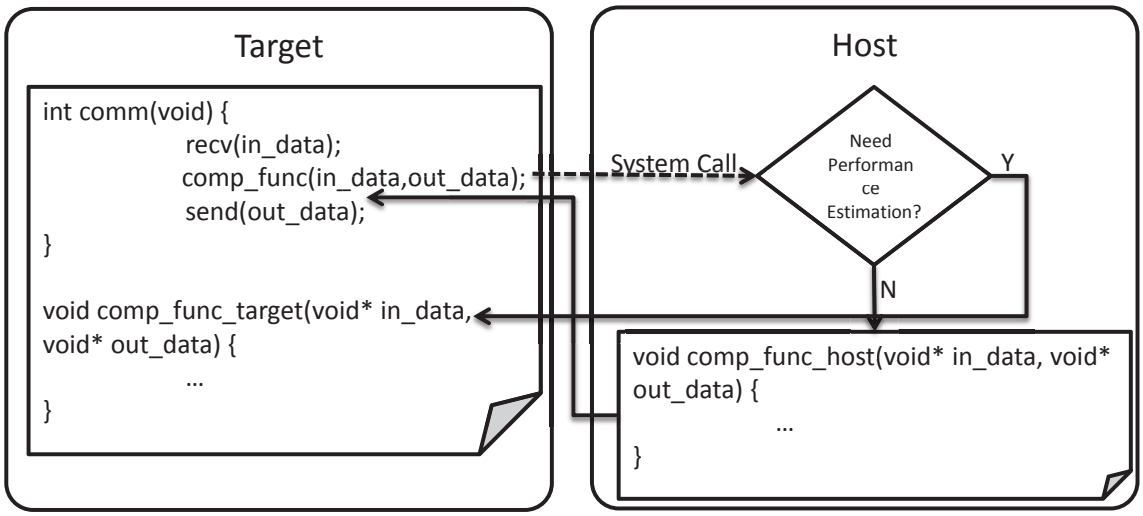

Figure 11: Execution Mode Switch

execution mode switch on function boundaries. Due to limited number of execution mode switches, the overhead of such encapsulation is small. With predefined context switch position in source code, we can achieve the mode switch by manipulating the target code on C source level, which is highly retargetable. The only thing that needs to be changed for different targets is Application Binary Interface (ABI).

In execution mode switch, the modified target code is compiled and loaded by a modified interpreted ISS and the program is executed in interpreted mode inside the ISS. The computation functions in the target code are replaced by a series of system calls. Copies of all computation functions are compiled and linked with the proposed ISS with proper modification. When the interpreted simulator encounters a computation function, the system call will pass the control over the host ISS. The host ISS will decide whether to execute the corresponding computation function natively for fast simulation or interpretively to collect information for performance estimation. If the function is executed interpretively, cycle count used by the function will be recorded along with profiling results for future performance estimation. If the function is executed natively, parameters will be retrieved from the target stack according to the target ABI. After the native execution, the return value is written back to target stack. Cycle count of the ISS is updated, and a notification will be sent to the simulation engine.

For example, Figure 11 shows the simplified code segment of execution mode switch. The *comm()* function wraps the major execution path on the target side. The target first calls a communication function *recv()* to receive the data to be processed. The address of the data is then passed to computation function *comp_func()*. When the Program Counter (PC) hits *comp_func()*, a system call is triggered. The host ISS takes over the execution. It will decide whether to execute the function natively or in interpreted mode. If the function is executed on the host side, the parameters of

the function will be acquired from the target register/memory according to the target ABI. Then a wrapper on the host side execute the computation function with the parameters acquired. After the function is executed, the return value of the function is then written back to the target memory/register by the wrapper. Then the execution of the target is resumed, and the output data is written to *out_data*. The target calls the *send()* function to send the data via processor bus. The actual bus transaction will be taken care of by the wrapper mentioned in Section 2.1.

2.2.3 Memory Synchronization

In the proposed approach, special consideration needs to be paid to memory synchronization which ensures that the variable values on the host side and the target side are consistent.

Endianness Endianness is an important issue if two processors of different endianness are accessing the same memory region. To ensure the correctness of read and write, the ISS will convert a memory word from host endian to target endian when it is loading memory to register and convert the endian back when writing back to the memory. This way, the computation function on the host side will not need to worry about the endianness of the target. Considering that we put computation intensive functions that are likely to access memory more frequently on the host, it makes sense to let the target take care of the endianness conversion to reduce the run-time overhead.

Global Variables Unlike assembly level translation, whose resulting code is executed in the same memory region as the interpretive target, the source level code generation approach we employed in the proposed hybrid ISS has an additional task. We need to perform resource synchronization manually since the target and host codes are executed in different contexts. With all communication code excluded from the host functions, the host code will not perform any I/O operations, the only synchronization needed is the synchronization between the on-chip memories and the host memory. Such memory synchronizations should be considered for three types of variables: global variables, dynamic allocated memory, and local variables.

It is possible to synchronize the value of global variables at the beginning and the end of the computation functions. The problem is that the overhead might be too high. To find out which part of memory has been modified, all memory accesses need to be replaced with a special function by the code generation tool, so that statistics of all memory accesses can be generated. It will not be a considerable overhead for the target side since it is executed in interpreted mode. But for the host side, if all memory accesses are replaced by a function call, the overhead will be significant. Another possibility is to synchronize all global variables no matter if they have been modified or not. If switching between interpreted mode and compiled mode is very frequent, this could also create a large overhead.

Considering that the endianness problem is taken care of by the simulator, the host can read or write the target memory freely. The only trick is to make sure that the host is writing to the correct addresses. That leads to our solution of tackling synchronization of global variables.

For global and static variables, the target address of each variable can be retrieved from the symbol table after target code compilation. The symbol table will be converted to a header file and

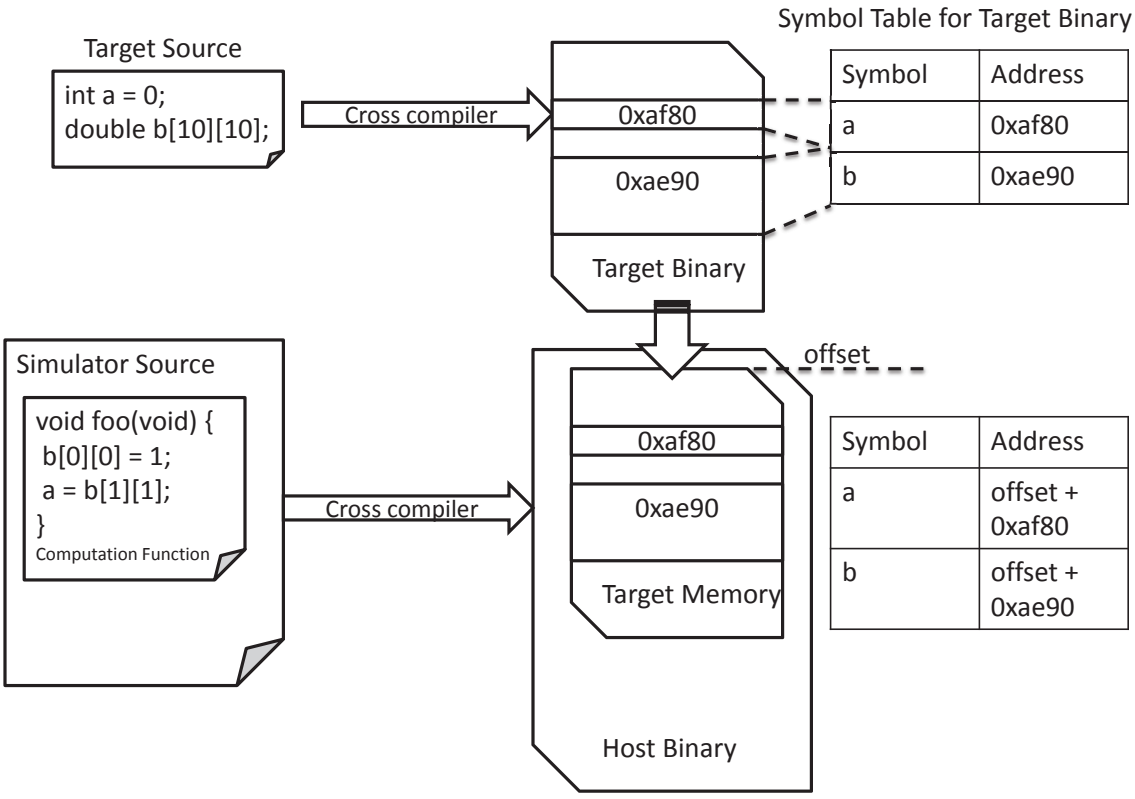

Figure 12: Global Variable Synchronization

compiled with ISS source code to instrument the global variables. When a function is accessing a global variable in native execution mode, it will first look up in the symbol table for the address. With the MMU function in the interpretive ISS, the actual address of a variable can easily be calculated. The resulting address is an offset in the target memory region. Therefore, the host function will be able to access the global variables directly from the target memory.

For example, Figure 12 shows a simplified scenario. After the the target source is compiled, each global variable corresponds to a target address (*a* - 0xaf80, *b* - 0xae90 in the example). At run-time the target binary is loaded by the simulator binary. If there is no operating system involved and the target binary is not relocated, the address of the global variables in the simulator will just be its original address plus an offset, which marks the starting address of the simulator target memory. When the computation source on the host refers to that global variable, it should go to the original address of the variable plus the offset. If the target binary is loaded by a relocatable loader, simply changing the value of *offset* will produce the correct address. If an operating system that supports virtual memory is involved, the virtual address of the variable should be fed into a function which

converts it to its corresponding run-time physical memory address. Then an offset should be added to the translated address to produce the correct address.

Pointers Synchronization of pointers is similar to synchronization of global variables. An offset should be added to the actual pointer value at run-time. The offset is determined by the loading scheme as described in the previous section. If the binary is loaded without relocation, the starting address of the target memory is used as the offset. If the target binary is relocated during loading, a run-time relocation offset should be added to the previous offset. If an operating system with virtual address space is involved, the simulator should retrieve the actual physical address from the MMU of the processor.

For all functions with pointer type parameters, the pointer value is converted to a host memory address by adding the offset and optionally going through a MMU. For all functions with pointer type return value, the pointer value is converted to target target address by subtracting the offset and optionally going through a MMU. With pointer type parameters and return value conversion, all pointer type variables in native execution scope are referring to native address space. This also avoids the pointer analysis problem, which is proved to be difficult at compile time [15].

Dynamically Allocated Memory For dynamically allocated memory, a customized version of *malloc()* function can be implemented and linked with the ISS source to cooperate with the interpreted *malloc()* function to ensure that all dynamically allocated memory segments are actually allocated in the simulation target memory region and all internal data structures for *malloc()* function are synchronized.

C memory management routines use static global data structures to maintain the used and free memory. A simplified *malloc()* is implemented on the target side to provide dynamic memory management. Lists of free memory chunks and occupied memory chunks are stored as static global variables in the target memory. With the same technique that we used in acquiring the address of global variables, we can access the memory management data structures in the target memory. To use the data structure, we need to use the same algorithm as the memory management routines on the target side. But all addresses in the target management data structure need to be converted to host memory addresses for allocation operations and the host memory address should be converted back to the target memory addresses for de-allocate operations.

Another problem is that code on the host other than the computation functions could also invoke memory management routines. We need to restrict the use of customized version of malloc function inside the computation function scope. This can be done by adding a prefix to customized memory allocation routines and rename the function calls in the computation functions.

Local Variables For local variables, no synchronization is needed under the assumption that the computation code is executed in one shot. Lack of memory synchronization for local variables will not affect the correctness of the program. But cache statistics will not be accurate if the host code frequently accesses the target memory. Additional cache statistics could be generated by inserting run-time statistics code whenever target memory is accessed through a pointer, but it will greatly affect the performance of the simulator.

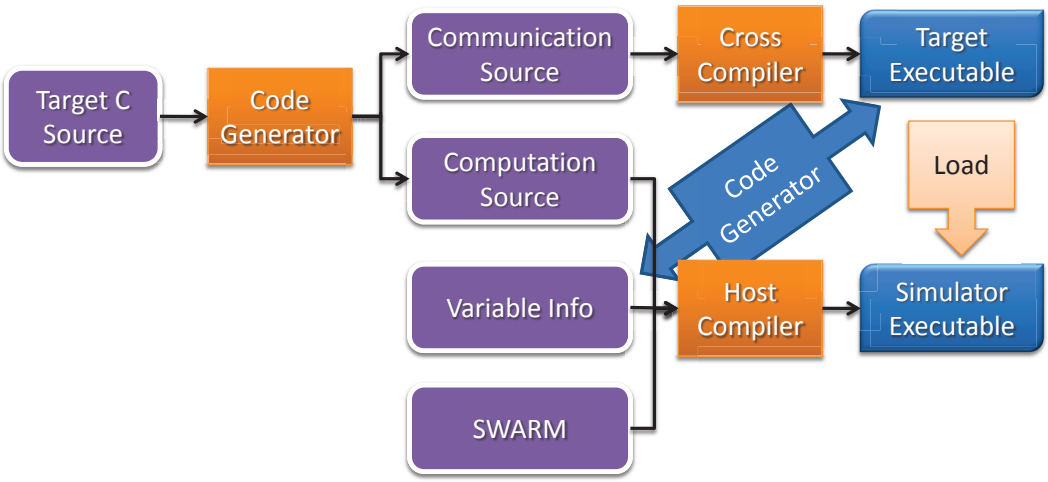

Figure 13: Work Flow Overview

2.3 Work Flow Overview

To achieve the specifications in the previous sections, the target code is processed by a code generation tool. The tool strips the computation code from the target source and generates the computation code for the host. Computation functions in the target source are then replaced by the system call stubs. The computation code and corresponding auxiliary functions are then complied on the host and linked with an interpreted ISS. The resulting host simulator loads the modified target binary to perform simulation.

As Figure 13 shows, the target C/SpecC source is fed to our code generation tool. With the information of communication and computation function provided by the SLD tool, the code generation tool will strip the source of computation functions from the target source as well as their dependencies to produce communication source. On the target side, the communication source is then compiled by a cross compiler to produce the final target executable. The executable is then used as an input for our code generation tool to provide a symbol table which contains the addresses for all global variables. Our code generation tool then produces code for the computation functions to access the global variables. On the host side, the computation functions and their dependencies stripped from the target source along with the global information and *typedef*/*struct* are compiled into host binary. The binary is then linked with the SWARM library to produce the final simulator binary.

On higher level, the SLDL invokes a wrapper for the simulator binary (described in Section 2.1) to drive the simulation. Bus transactions and I/O operations are simulated by the wrapper in the SLD context.

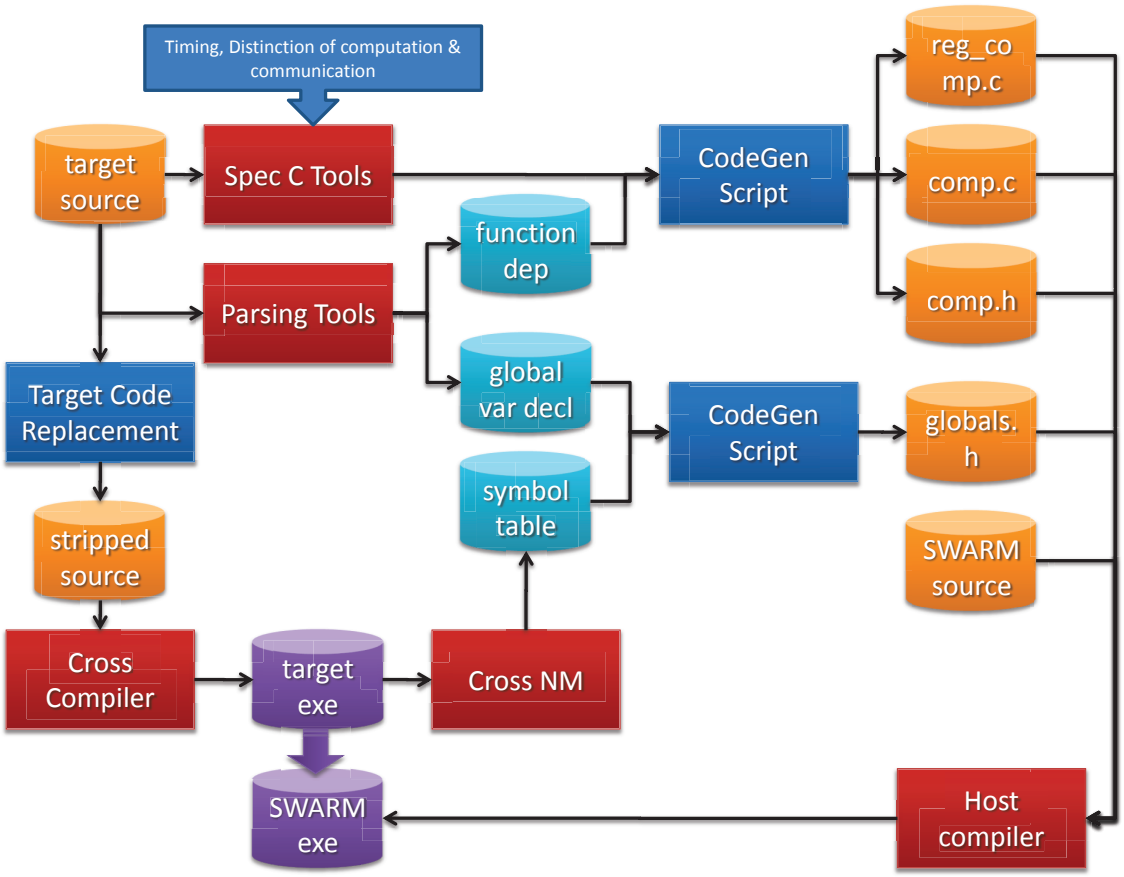

Figure 14: Code Generation Work Flow

3 Implementation

3.1 Compilation Work Flow

We built a code generation tool to achieve the proposed approach. Our code generation tool takes C/SpecC code as input, replaces the computation functions in the target source and generates the source for native execution. The tool also generates code for target to host memory synchronization. Performance estimation is conducted at run-time and timing synchronization is provided by the host source. Finally, the host source is linked with an interpretive ISS and the target binary is loaded by the hybrid ISS.

Figure 3.1 shows the work flow of the code generation tool. According to the discussion in the previous chapter, the target source is first processed by the parsing tools. Function dependencies, complex custom types and global variables definitions are retrieved from the target source and stored in separate files. On the target side, the target source goes through a target source replacement tool,

all computation functions will be copied to the host source and renamed in the target source. The computation functions in the target source are then replaced by a series of labels with the same name as the computation functions. A system call (trap instruction) is implemented under the label so that whenever the target side executes the computation function, an interrupt will be triggered and the control will be handed over to the host simulator.

The stripped source then goes through a cross compiler to produce the final target binary. The target binary is fed into the *nm* tool to produce a symbol table which contains the addresses of global variables. The symbol table and the global variable declarations will be fed into our code generation tool to produce code that accesses the global variables in the correct addresses in target memory, and store them in comp.h. Any *struct* definitions, *enum* definitions and *typedef* definitions are also copied to comp.h.

The code generation tool derives a new simulator class (*MyArm*) from base ISS class *SWARM* in our code generation tool. Computation functions are member functions of the derived class. With function dependency information, our code generation tool copies all computation function along with their dependencies into the derived class and stores them in comp.cpp. To invoke the computation functions correctly, an interrupt handler should be generated for each computation function. The handler marshals the parameters of the function call and invokes the corresponding computation function. The interrupt handler also returns the value from the computation function to the target according to the target's Abstract Binary Interface (ABI).

The generated host source files (comp.h, comp.c and globals.h) are compiled with the native compiler and linked with the SWARM library to form the proposed ISS. The target binary is loaded by the proposed ISS binary during the simulation.

3.2 Target Code Generation

The computation function calls are replaced by system calls. After compilation of the target code, the address of global and static variables can be fetched from the target binary. Such address information should be fed back to the code generation tool for host code generation.

3.2.1 Target Functions

In the proposed approach, the computation functions are renamed. A system call with a unique id is generated to replace the original function call. When the function call occurs, the host is responsible for retrieving the parameters and save the return value according to the ABI specification of the target compiler. Detailed information is described in host interrupt handler generation in Section 3.3.2.

As shown in Figure 3.2, *dct()* is the computation function in the target source. It consists of three functions *preshift(), chendct()* and *bound()*. It is first renamed to *interpreted_dct()* to prevent the target from calling the function. Function names in the symbol table are essentially the same as assembly labels. A global assembly label with the same name as the computation function is generated. The label is implemented as a trap instruction (*swi 0x800012*). Each computation function corresponds to a unique id (*0x800012*) and each unique id corresponds to an interrupt handler in the ISS's interrupt vector table. On assembly level, when a computation function is

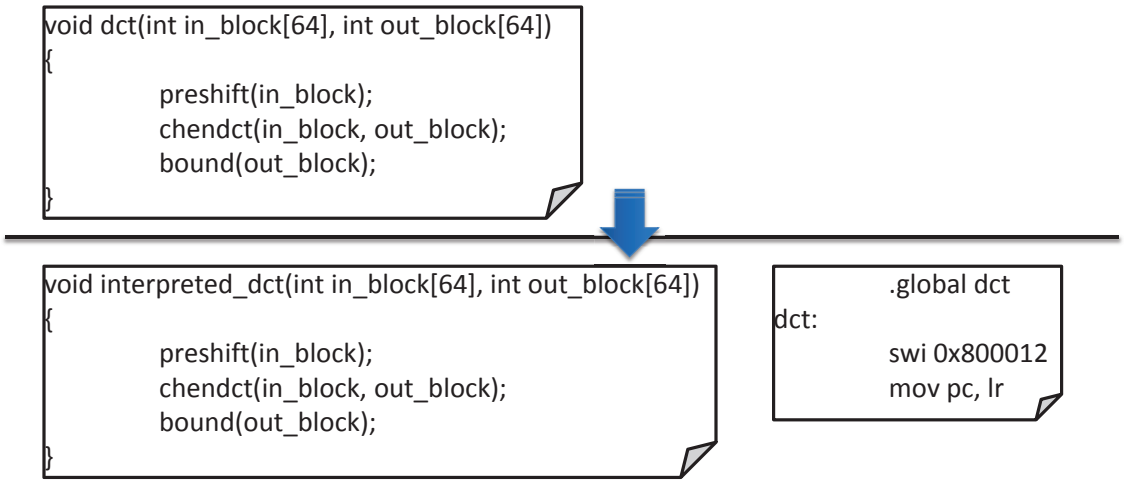

Figure 15: Target Functions

called, parameters for the computation function will be stored in registers or pushed to the stack which depends on the ABI specification of the architecture. Then the corresponding interrupt is triggered. The interrupt handler on the host side will be responsible for retrieving the parameters, invokes the compiled version of the computation function and returns the value to the target.

3.2.2 Target Global Variables

The addresses of global and static variables are needed by the host code generation. Thus, after the compilation of the modified target source, the symbol table is retrieved from the target binary by *nm*. The addresses along with the symbol names are fed back to the code generator.

As shown in Figure 16, the target source is compiled by the cross compiler and the generated binary is processed by *nm* to produce a symbol table. The symbol table produced by name and the definitions of the global variable retrieved by the code scanning tool are fed to the host code generation tool to form the definition of target global variables. The rules of the conversion are described in Section 3.3.2.

3.3 Host Code Generation

The host code generation consists of two parts: function code and global variable definition code. The computation function code along with their auxiliary functions are inserted into a hybrid ISS class (*MyArm*) derived from the original interpreted ISS (*SWARM*).

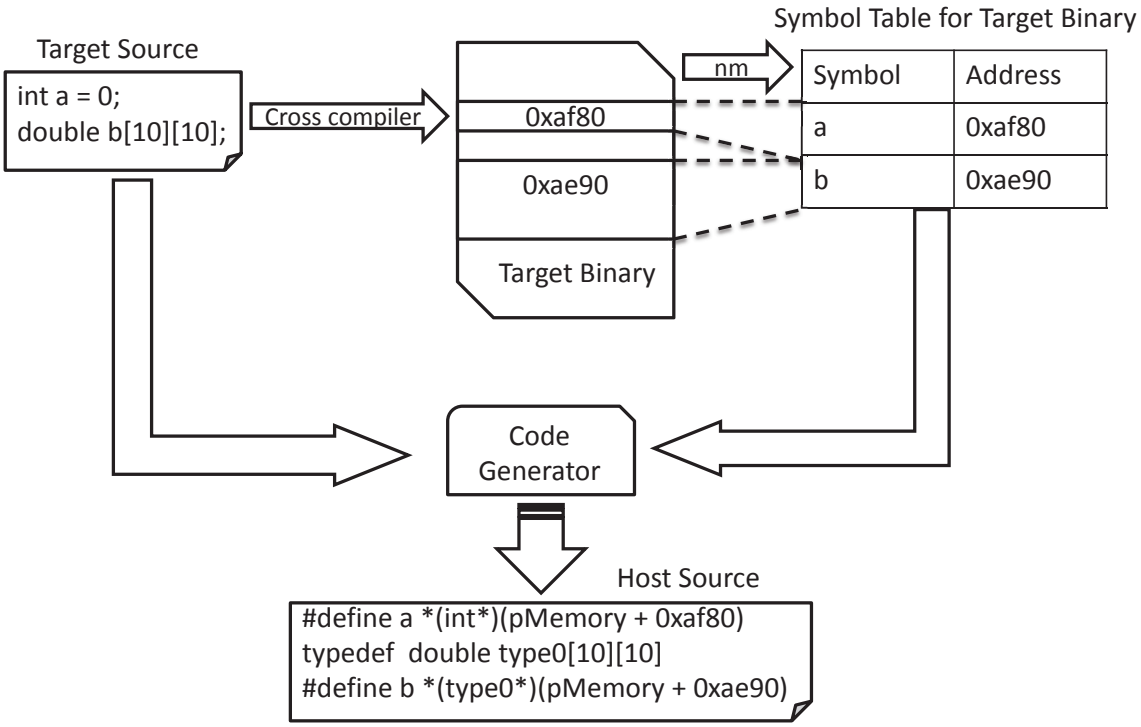

Figure 16: Target Global Variable

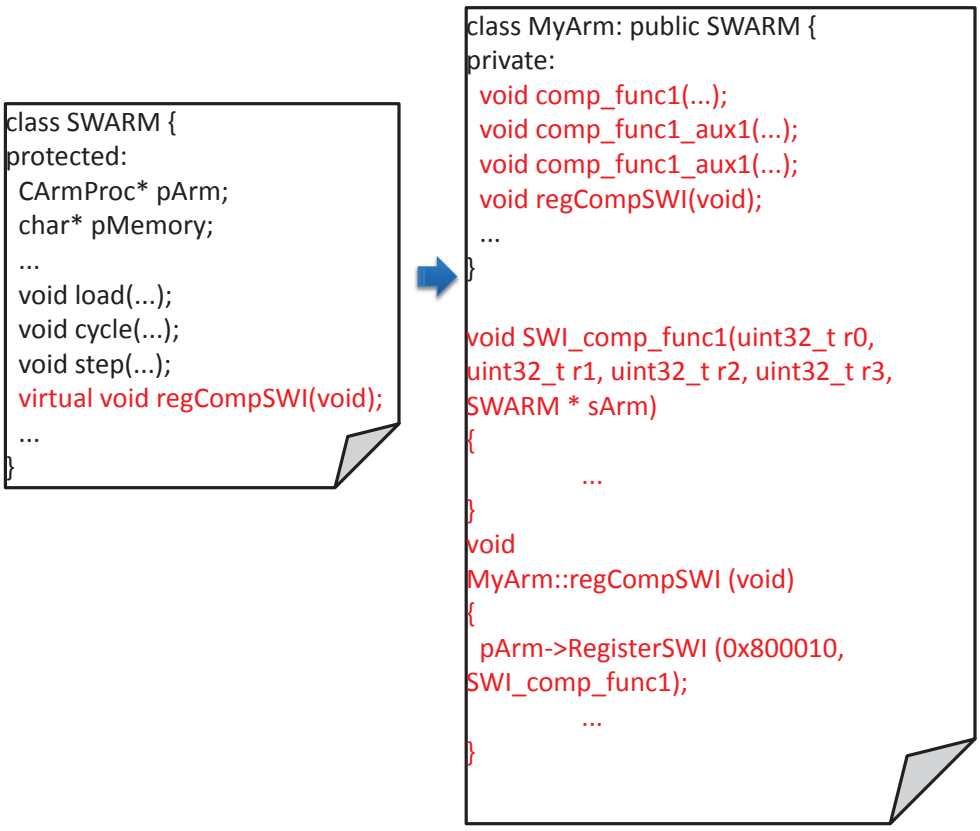

Figure 17: Host Code Generation Overview

3.3.1 Compiled Computation Function Generation

Hybrid ISS Structure As shown in Figure 17, the proposed hybrid ISS (*MyArm*) is constructed based on the interpreted counterpart (*SWARM*). It is derived from the interpreted ISS. The *SWARM* base consists of two major components, a processor core (*CArmProc* pArm*) and its main memory (*char* pMemory*). The *SWARM* class also provides functions for SLDL wrapper to control the simulation (*load(), cycle(), step()*). It also contains a virtual function *regCompSWI()* to register computation function interrupt handlers.

In the derived hybrid ISS class (*MyArm*), all computation functions (*comp_func1()*) and their auxiliary functions (*comp_func1_aux1()* and *comp_func1_aux2()*) are member functions of the hybrid ISS. For each computation function, an interrupt handler function (*SWI_comp_func1()*) is generated to retrieve the parameters of the computation function and save the return value. *regCompSWI()* is implemented to register such interrupt handlers to the hybrid ISS during the initialization.

The interpreted ISS is compiled into a library, and the generated hybrid class is linked with the library to generate the final simulator binary. The design of the hierarchy mainly deals with the presence of several instances of ISS in a multiprocessor simulation. Each processor will have its own SLDL wrapper function to invoke the simulator binary source and all instances share the same base interpreted ISS code. The same function or global variables could appear on different processors, but as long as they are in different hybrid ISS instances, there will not be any conflicts between the simulators. The design also makes it easier to define global variables offsets (described in Section 3.3.2).

Dependencies for Computation Functions A source scanning tool (*silentbob*) is used to generate the dependencies for computation functions. After the scanning the target source, the dependency lists for all computation functions are generated. Our code generation tool will then merge the dependency list together and make sure that there are no duplicates. All source code of the functions in the list is copied to the host code as member functions of the hybrid ISS class. A scope identifier (*MyArm::*) will be added to each function definition. Declarations of the functions are also copied into the declaration of *MyArm* class in the header file.

For example, Figure 18 shows dependency generation for function *dct()*. *dct()* depends on function *preshift()*, *chen_dct()* and *bound()*. The source code of all four functions are copied to the host source. A scope identifier (*MyArm::*) is added to the definition of each function. Additionally, a *SWI_dct()* function is generated as interrupt handler that invokes the *dct()* function. Declarations of all these functions are added to the class definition of *MyArm* in the host header file. The next section will explain the generation of the interrupt handler function.

Interrupt Handlers for Computation Functions An interrupt handler function is generated for each computation function to retrieve the parameters and save the return values. The interrupt handler is architecture dependent. In our example, we build our hybrid ISS on top of an ARM ISS interpreted simulator called SWARM. The interrupt handler should conform the Abstraction Binary Interface (ABI) of the target compiler in order to retrieve the parameters and save the return values correctly.

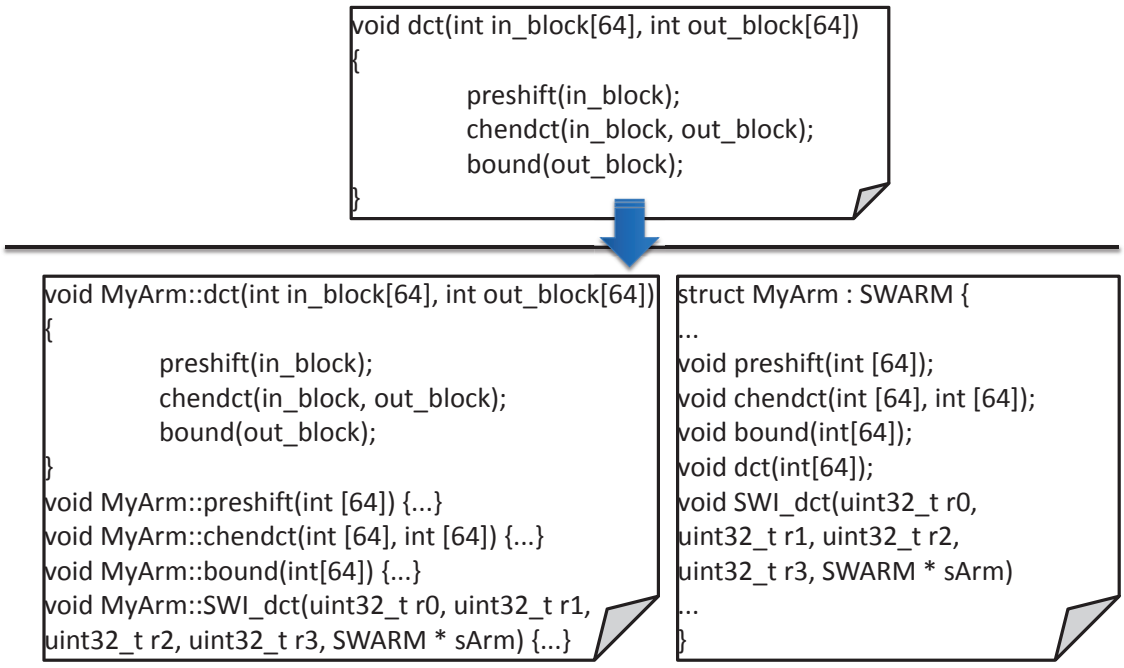

Figure 18: Host Interrupt Dependency Generation

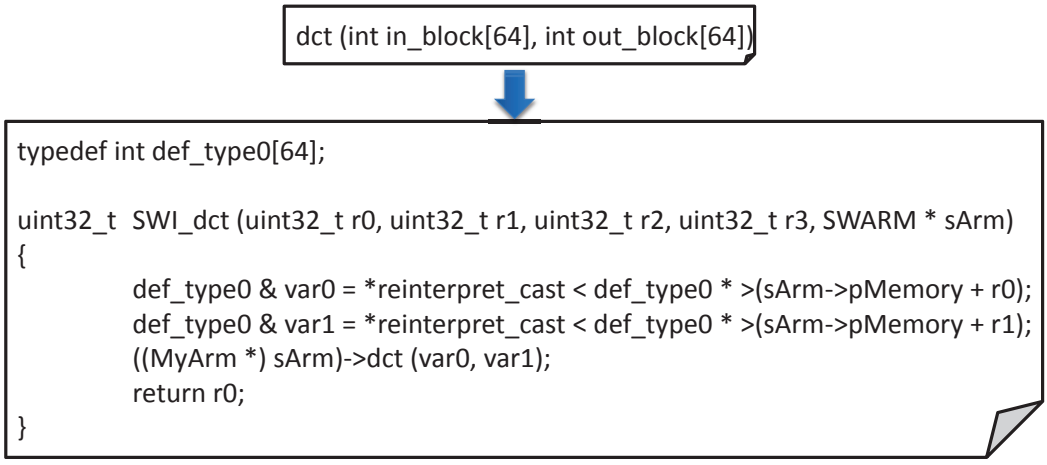

Figure 19: Host Interrupt Handler

In ARM ABI specification, the parameters are stored in registers *r0 - r3*. The return value is stored back in *r0*. Thus the only information we need to make the function call is the function prototype which provides all parameter types and the return type.

The prototype of the interrupt handler is:
*uint32_t SWI_func (uint32_t r0, uint32_t r1, uint32_t r2, uint32_t r3, SWARM * sArm);*

- *r0 - r4* stores values of *r0 - r4* register of the processor, which stores the value of the first four parameters of the function.
- *sArm* is a pointer to an instance of the SWARM ISS.

For each parameter in the function parameter list, a variable of the parameter type is declared. The variables are assigned the values of the corresponding register with proper type conversion. The parameter type conversion follows the following rules:

- If parameter is of a simple type (*int*, *char*, *double*), a simple explicit conversion will be used.
- If the parameter is of a pointer type, an offset will be added to the pointer value. The calculation of the offset is the same as the calculation of address offset for global variables.
- If the parameter is of an array type (*int [64][64]*), a *typedef* statement will be generated to define *new_type*. The variable will be accessed with **(new_type*)(offset + address)* (as shown in Figure 19).
- If the parameter is of a complex type (*struct*, *union* . . .)), the address will be fetched from the register. A proper offset will be added to the address. The actual content of the complex type variable will be fetched from the stack.
- For the ARM processor, if there are more than four parameters in the function parameter list, parameters other than the first four will be fetched from the process stack.

For example, as shown in Figure 19, function *SWI_dct()* is generated for function *dct()*. The *SWI_dct()* function has five parameters, r0 - r2 stores register values and sArm is a pointer to the simulator object. The simulator object contains a pointer to the processor core (*sArm* → *pArm*) and a pointer to the target memory (*sArm→pMemory*).

The code generator parses the function's parameter list and converts the value of the corresponding register to the parameter type. For array types (*int[64]* in *dct()* function), a *typedef* statement is generated and a reinterpreted conversion is used to interpret the value correctly. Since such types are pointers to target memory, an offset of the target memory should be added (*sArm→pMemory*). The parameters are passed in as a reference to the corresponding parameter type.

In this example, the function does not return anything. If the function returns a value, the value will be converted to *uint32_t* and stored in *r0*. If the function returns a pointer, an offset (*sArm→pMemory*) will be subtracted from the return value. If the function returns a complex type, a variable of the complex type will be declared to store the value. The value of the complex variable will be placed on top of the process stack.

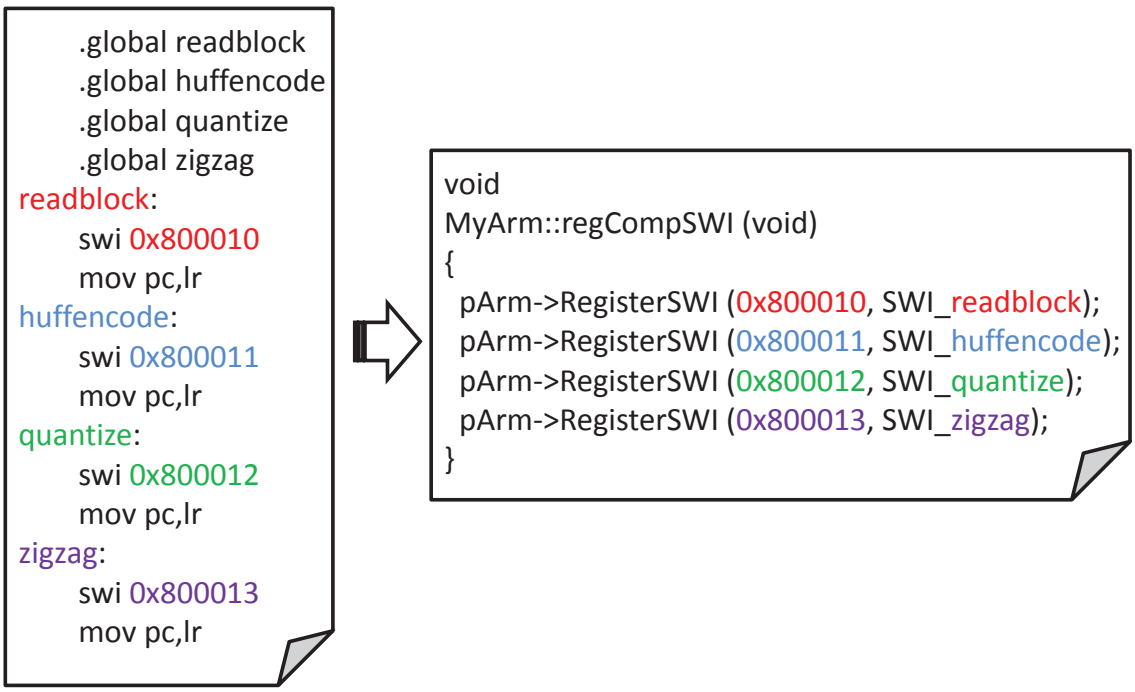

Figure 20: Computation Interrupt Registration

Registering for Computation Interrupt Handlers For each computation function, an id number is associated with the function name (e.g. *0x800012* in Figure 3.2). A vector table is stored in the processor core (*CArmProc*) to record the entry point of each interrupt handler function. To register the computation interrupt function in the vector table, the virtual function *regCompSWI()* needs to be implemented in the hybrid ISS class (*MyArm*).

For example, as shown in Figure 20, all four computation functions (*readblock(), huffencode(), quantize()* and *zigzag()*) are registered to the processor core (*pArm*) with a interrupt handler register function (*RegisterSWI()*) provided by the processor core. The *regCompSWI()* function will be executed in the constructor of the SWARM class, so that when the program is loaded each label for the computation function will correspond to a unique interrupt id in the processor core.

3.3.2 Memory Synchronization

Pointer Analysis It has been shown that in C language, general pointer analysis at compile time is impossible, the value of pointers at run-time is unpredictable [13]. Thus we avoid such compile time pointer analysis by mapping pointers / global variables directly to target memory. Given the fact that the ISS takes care of the byte order conversion, the host is able to access the target memory freely without worrying about the endianness.

Custom Types To compile the computation functions on the host, information about custom types in the target source is collected. The following custom types need to be subtracted from the target source code.

- *typedef* declarations
- *struct* declarations
- *union* declarations
- *enum* declarations
- *m*acro definitions

These declarations need to be copied to the header file of the host source.

Global Variables As discussed in Section 2.2.3, synchronizing the values of global variables would consume a lot of simulation time. Thus we choose to mandate the global variables in the host source to point to the corresponding addresses in the target memory, we can use the code generator to replace every occurrence of the global variable with a specialized code segment that calculates the address of the corresponding variable in the target memory. Such replacements can be easily achieved with macro definitions. The variable name is defined as a macro that de-references a pointer of the variable type. The value of the pointer is an offset plus the target address of the variable in the symbol table. The replacement is conducted by the pre-processor instead of our code generator.

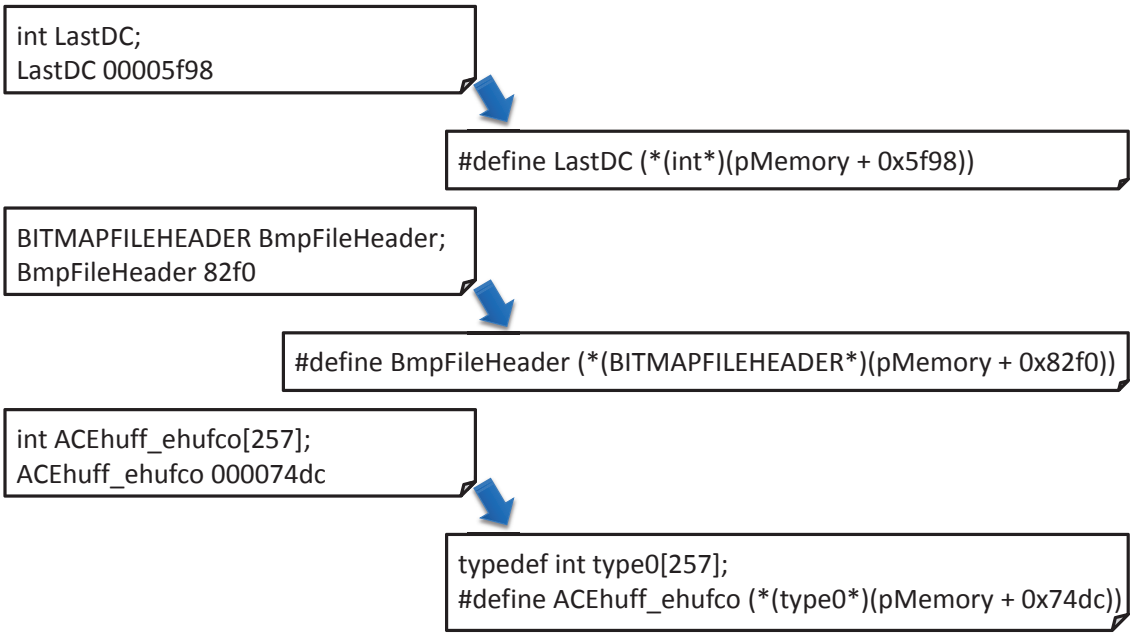

Figure 21: Global Variable Substitution

Macro Definition Convention In order to have the preprocessor replace the global variable names without changing the original code manually, we use the follow macro definition convention.

*#define var_name *(type*)(offset + target_address)*

- *var_name* is the name of the global variable
- *type* is the type of the global variable
- *offset* is the offset for the global variable in the host context
- *target_address* is the target address of the global variable retrieved from the symbol table of the target binary.

Address Binding The addresses of global variables are retrieved from the symbol table of the target binary. An offset (*pMemory*) should be added to the address retrieved from the symbol table (as shown in Figure 16) to construct the correct address. Since the global variables are only used in computation functions which are in the hybrid ISS class (*MyArm*) scope, *pMemory* will be recognized. With the presence of multiple instances of hybrid simulator, *pMemory* will refer to different scopes to resolve the conflict.

For example, as shown in Figure 21 on the left side, each variable name is associated with a target address after compilation of the target source. On the right side, a macro definition for the

variable is constituted of the its address in the target binary plus an offset. In this simplified example, no relocatable loader or operating system that supports virtual memory is involved. If a relocatable loader is involved, a function that calculates the relocated address should be added to the offset. If an operating system with virtual memory is involved, the virtual address should be converted to physical address by the MMU function of the simulator.

Type Mapping For simple types like *int*, *char* or *double* or user defined types (*struct*, *union*, *enum* and *typedef*), we can use the type name directly in type mapping.

For array types (*int [257]* in the example in Figure 21), especially for multidimensional arrays. We need to define new types in order to perceive the variables' value correctly.

For example, in Figure 21, a *typedef* is defined for type *int [257]* so that the same type of macro definition can be used for array types.

Local Variables Since the computation code is executed in "one shot" without interrupt, there is no need to synchronize general local variables. But for static local variables, they need to be synchronized the same way as global variables. Since the static variables are only accessible inside the function scope, we need to convert all static local variables to global variables. Such conversion might results in naming conflicts. Thus we should add a prefix to all static local variables before the conversion. The prefix should be unique between functions, thus the function names are used as prefixes for local variables. Since we don't change the target source in our code generator, such naming conversion is safe in terms of preserving the local static variables' original access scope.

Dynamic Allocated Memory Dynamic allocated memory can be synchronized by coordinating the *malloc()* and *free()* function on both target and host side. On the host side, all invocations of memory allocation functions are replaced by a custom function (with an additional *swi_* prefix) to perform memory allocation and de-allocation with the record in the target memory.

As described in Section 2.2.3, whenever a allocation or de-allocation function is called on the host side, the host memory management function should allocate/de-allocate memory in the target memory region and do the book keeping in the data structure in target memory. The following code shows the data structure used by the target side to keep track of used and free memory chunks.

```
typedef struct HNTAG
{
  struct HNTAG* pNextAddr;
  struct HNTAG* pNextSize;
  struct HNTAG* pPrevAddr;
  struct HNTAG* pPrevSize;

  void*         pHole;
  uint32_t      nLength;
} HEAP_NODE;
```

```
typedef struct HTAG
{
  HEAP_NODE* pAddrList;
  HEAP_NODE* pSizeList;
  HEAP_NODE* pUsedList;
} HEAP;

static HEAP* __heap;
```

On the target side, the memory management functions use a simple double linked list implementation to store free memory chunk list and used memory chunk list. The *HEAP_NODE* represents a node in the linked list. It contains the starting memory address (*pHole*), the length (*nLength*) of the memory chunk. Optionally the nodes can be ordered by address (with *pPrevAddr* and *pNextAddr*) and/or by size (with *pPrevSize* and *pNextSize*). The free list is ordered by both size and address. The used list is ordered only with address. A global static variable *__heap* is used to store the entry point of the data structure.

The target address of *__heap* variable can be retrieved at compile time. Thus after the target is initialized, we can access the memory management data structure in the target memory by accessing the value of *__heap*. Whenever we read from *__heap*, the target memory addresses should be converted to host memory addresses (as described in retrieving pointer and global variable addresses from the target). Whenever we write to the *__heap*, the host memory addresses should be converted to target memory addresses (as described in function return value conversion). As long as we can ensure the correctness of addresses between target and host side. The implementation of memory management functions is a trivial issue. The memory management functions can be simply converted from its target counterpart.

To used the customized version of memory management functions on the host side, we need to add a prefix to all invocations of memory management functions in the computation functions and their dependencies. So that the customized version of memory management function will not confuse other functions in the host source.

3.3.3 Technicalities in Merging the Target C Source

Merging the target source from different C source files could result in conflicts in the host source code. Naming conflicts, header re-expansion and dependency problems could result in compilation error of the host source. In this section, we briefly discuss a few caveats in merging the target source into a single host source file.

Expanding Macro Definitions Macro expansion could be a big problem for merging different source files. Because different source files could use different definitions for the same macro name. Originally the codes are in different file scope when they are merged together, some of the macro definitions might not get properly expanded. For example, for the following code:

`define1.h:`

```
#define THRESHOLD 100

define2.h:
#define THRESHOLD 1000
```

The code generator could be confused by the two definitions and finally one of the macros could be wrongfully expanded in the host source.

Another problem with macro definition is that if there is conditional macro definitions, the code generator could also get confused. For example, consider the following code:

```
#ifdef WIN32
int main(int argc, char** argv) {
#else
int main(void) {
#endif
```

The *WIN32* version of the code takes parameters from the command line, while the embedded version of the code takes no parameters. The code generator will be confused parsing the function prototype, since the macro definitions are passed as compiler parameters.

To solve these conflicts, we need to expand the macro definitions before the code generation instead of copying the macro definitions directly to the generated source. Before stripping the computation source, we can first use the preprocessor with the *CFLAGS* parameter in the *Makefile* to eliminate these conflicts.

***static* Declarations** Many embedded system applications are implemented by C language. To restrict the accessing scope in C language, the *static* keyword is often used. In our code generation, we have to get rid of the *static* keyword in the function declaration to make computation functions a general member of the *MyArm* class. But simply removing the *static* keyword could result in naming conflicts. For example, considering the following code:

```
file1.c:
static void print(int a) {
printf("%d\n", a);
}

file2.c:
static void print(int a) {
printf("%X\n", a);
}
```

The two versions of *print()* function will result in conflict for our code generator. The solution is similar to resolving static local variables. We simply add the file name as a prefix to the function name (*file1_print()* and *file2_print()*) and replace all occurrences of the functions in the corresponding file with the new name.

Global Variable Name Shadowing In C language, if a local variable has the same name as the global variable, the global variable is shadowed in the local scope. But in our approach of synchronizing global variable, we use macro definition. The source code will first be processed by the preprocessor and the local variables that have the same name as the global variable will be replaced by the address of the global variable. The compiler would complain about the definition of the local variable. For example, for the following code segment.

```
int a = 10; // => #define a *(int*)(pMemory + 0xaf80)
void foo(void) {
int a = 5;
...
}
```

After code generation, the code will be converted to:

```
void MyArm::foo(void) {
int *(int*)(pMemory + 0xaf80) = 5;
}
```

which will result in a compile error. To solve this problem, we need to rename the local variables that have the same name as any of the global identifier (global variable names, user defined structs ...).

***struct* Dependencies** Our code generator extracts the *struct* and *typedef* definition from the target source code in undetermined order. This could cause dependency problems. For example, for the following definitions:

```
struct s2{
int a;
struct s1 b;
}
struct s1{
enum_type c;
}
typedef enum {RED, YELLOW, BLUE} enum_type;
```

The definition of *s2* depends on the definition of *s1*, and the definition of *s1* depends on the definition of *enum_type*. With definition order shown in the example, the compiler will not be able to resolve the definitions of *s1* and *s2*. To solve this problem, a directed acyclic graph should be constructed for nested types. Then, the code generator can simply traverse the graph to produce the type definitions in the correct order.

Figure 22: Image Processor

4 Experimental Results

We have tested several real world applications with interpreted, native and hybrid simulation. For interpreted simulation, a modified SWARM simulator is used, and all the code is executed completely in interpreted mode. SWARM [6] is an interpreted ISS for ARM processor. It is modified by Schirner and Sachdeva [14] to be adapted in SLD environment. Then Kim [12] modified the SWARM to support for multiprocessor simulation. Our modified SWARM is based on Kim's version. We consider the cycle count of the interpretive case to be accurate. Error rates of the other cases are calculated based on the cycle count of the interpreted mode. For native simulation, the target code is compiled with a native compiler which has no cycle estimation in this case. For Hybrid simulation, the proposed hybrid ISS is used.

4.1 Image Processor

As shown in Figure 22, the image processor reads an image and performs several transformations on the image. In our test cases, a 192 by 144 image is used as input. At each stage, the output of the previous stage is used as input. The image will be first transformed into black and white image, then transformed to night view, then horizontally mirrored, then horizontally flipped, then vertically flipped and finally blurred.

Each stage is considered to be a computation function. In our test cases, we choose different stages to run in compiled mode. The speedup is calculated based on the complete interpreted mode (test case 1). The cycle count in complete interpreted mode (test case 1) is considered to be accurate. The error for each test case is calculated based on the cycle count of the complete interpreted mode.

In Table 1, we show the experimental results for simulation time of the image processor. A computation function is marked as "C" if it is chosen to run in compiled mode or "I" if it is chosen to run in interpreted mode. For each test case, we measure the simulation time (Time(s)) to calculate the simulation speedup.

The accuracy for each test case is shown in Table 2. The ideal estimation reuses the performance estimation from the interpreted mode, which is the best performance estimation in theory. In real life, a profiling tool will be used to produce the cycle count estimation for each computation function. And the final accuracy depends on the accuracy of the profiling tool, which is beyond the scope of our work.

As shown in Table 1 and Table 2, if all functions are chosen to run in compiled mode, the simulation gains a speedup of 3659X while still maintains a low cycle count error of 4.15% for ideal performance estimation. Since SWARM provides a complete simulation of ARM memory

Table 1: Tests Cases and Speedup for Image Processor

id	Modules in Compiled Mode						Time(s)	Speedup
	B & W	Night View	HMirror	HFlip	VFlip	Blur		
1	I	I	I	I	I	I	497.727	1.00
2	I	C	C	C	C	C	16.909	29.44
3	C	I	C	C	C	C	4.074	122.17
4	C	C	I	C	C	C	7.421	67.07
5	C	C	C	I	C	C	13.045	38.15
6	C	C	C	C	I	C	13.404	37.13
7	C	C	C	C	C	I	439.66	1.13
8	C	C	C	C	C	C	0.136	3659.76

Table 2: Tests Cases and Accuracy for Image Processor

id	ISS Cycles	Ideal Estimation		
		Est	Total	Error(%)
1	32245709	0	32245709	0.00
2	1513616	29392849	30906465	4.15
3	328202	30578362	30906564	4.15
4	406755	30496778	30903533	4.16
5	1321022	29581835	30902857	4.16
6	1539454	29364228	30903682	4.16
7	25838940	5068768	30907708	4.15
8	10162	30896564	30906726	4.15

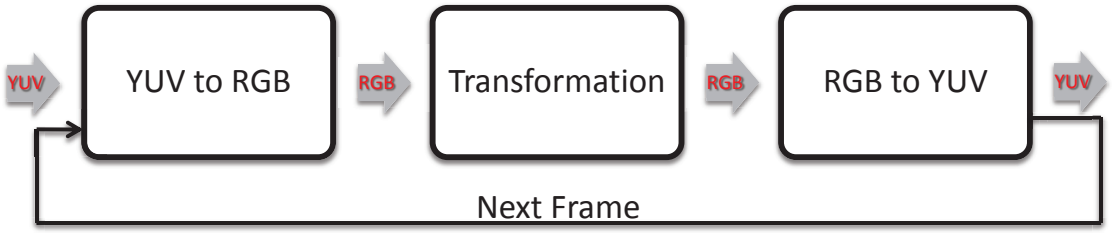

Figure 23: YUV Converter

hierarchy, the simulation speed of memory operations in SWARM is extremely slow. That is the major cause of the drastic speedup in the hybrid scheme. Most of the computation functions are matrix operations, which involve a lot of memory access. We can get the same conclusion from the *Blur* function. The *Blur* function consists only 80% of the cycle count in complete interpreted mode. But the simulation time consumes by *Blur* is close to 88%. Because *Blur* function accesses memory more frequently.

4.2 YUV Converter

As shown in Figure 23, the YUV converter [4] loads a YUV video stream, converts a YUV frame into RGB frame, performs transformations on the RGB frame, converts the modified RGB frame into YUV frame and finally saves the output YUV frame to an output video stream. In our test cases, a 352 by 288 YUV video stream is used as input. The converter picks every other frame starting from the 21st frame in reversed order, performs transformation on the frame and saves the the modified frame to output stream. Thus, the output stream is the 2X playback of the input stream in reversed order with transformation.

Each operation (*YUV to RGB*, *Negative*, and *RGB to YUV*) is considered to be a computation function. In our test cases, we choose different operations to run in compiled mode. The speedup is calculated based on the complete interpreted mode (test case 1). The cycle count in complete interpreted mode (test case 1) is considered to be accurate. The error for each test case is calculated based on the cycle count of the complete interpreted mode.

In Table 3, we show the experimental results for simulation time of the image encoder. A computation function is marked as “C” if it is chosen to run in compiled mode or “I” if it is chosen to run in interpreted mode. For each test case, we measure the simulation time (Time(s)) to calculate the simulation speedup.

The accuracy for each test case is shown in Table 4. In terms of accuracy, we use two ways of performance estimation. Auto estimation estimates the cycle count of computation functions based on the cycle count from interpreted mode in the first iteration of the execution. Ideal estimation reuses the performance estimation from the interpreted mode, which is the best performance estimation in theory. Since the cycle count for each computation function in all iterations are the same, the auto estimation is the same as ideal estimation. In real life, a profiling tool will be used to

Table 3: Tests Cases and Speedup for YUV Converter

id	Modules in Compiled Mode			Time(s)	Speedup
	YUV to RGB	Negative	RGB to YUV		
1	I	I	I	715.006	1.00
2	I	C	C	2.528	282.83
3	C	I	C	350.576	2.04
4	C	C	I	472.875	1.51
5	C	C	C	2.306	310.06

Table 4: Tests Cases and Accuracy for YUV Converter

id	ISS Cycles	Auto Estimation			Ideal Estimation		
		Est	Total	Error(%)	Est	Total	Error (%)
1	52219066	0	52219066	0.00	0	52219066	0.00
2	205710	52013932	52219642	0.00	52013932	52219642	0.00
3	28437689	23783376	52221065	0.00	23783376	52221065	0.00
4	23957735	28263876	52221611	0.00	28263876	52221611	0.00
5	100334	52030592	52130926	0.17	52030592	52130926	0.17

produce the cycle count estimation for each computation function. And the final accuracy depends on the accuracy of the profiling tool, which is beyond the scope of our work. We want to show the upper bound and lower bound of accuracy with different estimation tools in our experimental results.

As shown in Table 3 Table 4, if all functions are chosen to run in compiled mode, the simulation gains a speedup of 310X while still maintains a low cycle count error of 0.17% for ideal performance estimation. The computation functions in YUV converter also access memory very frequently. We can draw the same conclusion as for image processor by comparing *Negative* and *RGB to YUV* function. *Negative* is more computation intensive while *RGB to YUV* is more memory intensive. The latter takes 26% less cycle count than the former, and 35% more simulation time than the former. The *YUV to RGB* function takes very short simulation time since it invokes *fread* function which is actually running on the host. Thus, the memory access in the simulator is avoided.

4.3 JPEG Encoder

As shown in Figure 24, a JPEG encoder is composed of five major operations: *readblock*, *dct*, *quantize*, *zigzag* and *huffman*. A picture block of 256 bytes from a BMP picture is read by *readblock* and goes through *dct*, *quantize*, *zigzag* and *huffman* to produce the final JPEG image. At each stage, the output of the previous stage is used as input. In our example, we use the JPEG encoder to encode a 166 by 96 mono BMP image. We choose to run different stages in compiled mode. For

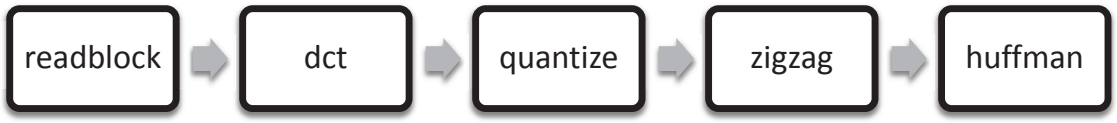

Figure 24: JPEG Encoder

Table 5: Tests Cases and Speedup for Image Processor

id	Modules in Compiled Mode					Time(s)	Speedup
	read	dct	quan	zigzag	huff		
1	I	I	I	I	I	56.280	1.00
2	C	I	I	I	I	54.119	1.08
3	C	C	I	I	I	28.007	2.09
4	C	C	C	I	I	18.032	3.25
5	C	C	C	C	I	14.615	4.01
6	C	C	C	I	C	3.903	15.01
7	C	C	I	C	C	11.286	5.19
8	C	I	C	C	C	27.689	2.12
9	C	C	C	C	C	0.437	134.10

each test, we measure the simulation time and the cycle count produced by the hybrid ISS. The speedup is calculated based on the complete interpreted mode (test case 1). The cycle count in complete interpreted mode (test case 1) is considered to be accurate. The error for each test case is calculated based on the cycle count of the complete interpreted mode.

In Table 5, we show the experimental results of JPEG encoder. A computation function is marked as "C" if it is chosen to run in compiled mode or "I" if it is chosen to run in interpreted mode. For each test case, we measure the simulation time (Time(s)) to calculate the simulation speedup.

The accuracy for each test case is shown in Table 6. In terms of accuracy, we use two ways of performance estimation. Auto estimation estimates the cycle count of computation functions based on the cycle count from interpreted mode in the first iteration of the execution. Ideal estimation reuses the performance estimation from the interpreted mode, which is the best performance estimation in theory. In real life, a profiling tool will be used to produce the cycle count estimation for each computation function. And the final accuracy depends on the accuracy of the profiling tool, which is beyond the scope of our work. We want to show the upper bound and lower bound of accuracy with different estimation tools in our experimental results. In the test case, the estimation of *huffencode* in auto estimation is very inaccurate (the first iteration of *huffencode* takes three times as many cycles as average, since it sets up the encoding tables in its first iteration), but better

Table 6: Tests Cases and Accuracy for Jpeg Encoder

id	ISS Cycles	Auto Estimation			Ideal Estimation		
		Est	Total	Error(%)	Est	Total	Error (%)
1	4259525	0	4259525	0.00	0	4259525	0.00
2	3667270	601360	4268630	0.01	600989	4268259	0.00
3	1990576	2293540	4284116	0.37	2298315	4288891	0.48
4	1307698	3001300	4308998	0.95	2994785	4302483	0.80
5	1093127	3235480	4328607	1.41	3228965	4322092	1.26
6	290805	7917820	8208625	92.31	4021976	4312781	1.04
7	753095	7444240	8197335	92.05	3559686	4312781	1.04
8	1757823	6459820	8217643	92.52	2558830	4316653	1.13
9	66235	8152000	8218235	92.54	4256156	4322391	1.26

performance estimation tools will greatly improve the results.

As shown in Table 5 Table 6, if all functions are chosen to run in compiled mode, the simulation gains a speedup of 134X while still maintains a low cycle count error of 1.26% for ideal performance estimation. The JPEG encoder is more computation intensive and less memory intensive than the previous two examples. Thus, the speedup is significantly less the previous two examples. The more computation functions are chosen to run in compiled mode, the larger the speedup is and the greater the cycle error is. For individual functions, we can see from the table that:

- *dct* takes almost half of the simulation time. But the cycle count of *dct* is similar to *huffencode*. The performance difference could be caused by the architecture difference between Pentinum and ARM.
- *huffencode* produces the largest cycle count error when it is run in compiled mode considering that the cycle estimation for *huffencode* is very inaccurate.
- *quantize* and *zigzag* produces the same cycle count error when they are run in compiled mode since the cycle count difference between iterations of the two functions is very small. But running *quantize* in compiled mode results in a larger speedup in simulation time since *quantize* takes longer computation time.

From the experimental results of the JPEG encoder, we can conclude that for computation intensive applications like JPEG encoder, the proposed hybrid ISS approach drastically speeds up the simulation while maintains a low cycle count error.

4.4 Limitations

The proposed hybrid ISS provides accurate performance estimation for computation intensive applications. For communication intensive applications, traditional interpreted ISS should be used

to provide cycle-accurate timing between communication and computation. For moderate applications, the user is free to choose whether to run a computation function in interpreted mode or compiled mode. By targeting a particular operating system, which is listed in Future Work, we can better estimate the timing of communication in the middle of a computation function.

5 Conclusion and Future Work

In this technical report, we proposed a new hybrid ISS approach for performance estimation of embedded systems in System Level Design context. The proposed hybrid ISS approach simulates computation functions in compiled mode and communication functions in interpreted mode. The proposed hybrid ISS approach drastically speeds up the simulation for computation intensive applications while still maintains a low cycle count error.

Traditionally, in System Level Design context, the simulation is either cycle approximate (like in Transaction Level Model or Bus Function Model) or completely cycle-accurate and pin-accurate (like in Implementation Model). The former produces an inaccurate performance estimation, while the latter takes a substantial longer simulation time. In the proposed approach, by simulating the computation functions in cycle-approximate mode and maintains cycle-accurate and pin-accurate simulation for communication between processors, we allow the user to explore the design space in between. In our proposed approach, the user is free to choose whether to run a function in interpreted mode or compiled mode. Thus, the user will have the freedom to make a trade off between simulation speed and simulation accuracy in their own design.

In this technical report, we also discussed different approaches of code generation and proposed our code generation approach on C source level. The proposed code generation approach is flexible, retargetable and adaptable.

To realize our design specification, we proposed our schemes of code generation, execution mode switch and context synchronization technique. These techniques are highly adaptable and easy to implement.

To tackle the switch between compiled mode and interpreted mode, we proposed our execution mode switch scheme. A system call is used to switch from interpreted mode execution to compiled mode execution. The interrupt handler generated by our code generator takes care of the marshaling of parameters and return values. The scheme can applied to various architectures with only small modifications according to the Abstract Binary Interface of the target architecture.

To synchronize the context between compiled and interpreted mode, we proposed our memory synchronization scheme. We synchronize the global/static variables and pointers by accessing the variables in target memory directly. We synchronize dynamic allocated memory by implementing our customized version of memory management functions on the host side and make the customized memory management functions operate on data structures in the target memory. We avoid the synchronization of local variables by execution the computation functions in one shot on host side.

In conclusion, our proposed hybrid ISS approach makes a good trade off between simulation speed and simulation accuracy. We provide the user the freedom to run functions in compiled mode or interpreted mode in the context of System Level Design. Our code generation is easy to implement and highly adaptable.

5.1 Future Work

Our proposed approach is not able to handle communications in the middle of the execution of computation functions, since computation functions are executed in compiled mode without interrupt. For computation intensive applications, cycle count error caused by this delay of communication handling is negligible. But for communication intensive applications, such errors could greatly affect the accuracy. We need to apply some modifications to our proposed approach to reduce such errors. For modern OS, communication is usually handled by device drivers in the background. The driver code and the OS kernel code are all considered to be communication code in our context. Our goal is to execute some of these communication code "in the middle" of computation function.

In interpreted simulation, the interruption caused by communications will not change the execution path of computation functions. It will only delay the execution of computation functions. Based on this observation, we can simply "interrupt" the clock for computation without actually interrupting the execution. Thus, the execution of computation functions will not be changed. The only thing to be changed is the ISS wrapper (see Section 2.1) in SLD context. In the presence of OS, the wrapper is executed whenever an OS timer interruption is triggered. Thus, all tasks except the computation functions will be executed in interpreted mode. If a computation function is executed, the cycle count advancement caused by the computation function will not be applied immediately. Instead, a portion of the advancement will be applied whenever the OS executes the computation function for a unit time slice.

Such modifications should be applied to the wrapper based on a specific OS. The scheduler of OS also needs to be modified so that the wrapper can access the scheduler's internal data structure.

6 ACKNOWLEDGMENT

This work has been supported in part by funding from the National Science Foundation (NSF) under research grant NSF Award #0747523. The authors thank the NSF for the valuable support. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Verilog. http://www.verilog.org/, 2006. EDA Industry Working Groups.

[2] VHDL. http://www.vhdl.org/, 2006. EDA Industry Working Groups.

[3] SystemC. http://www.systemc.org, 2010. Open SystemC Initiative (OSCI).

[4] Timothy Bohr and Rainer Doemer. A flexible video stream converter. Technical Report Technical Report CECS-08-13, Center for Embedded Computer Systems, University of California, Irvine, 2008.

[5] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM.

[6] Michael Dales. SWARM, ARM instruction simulator. http://www.cl.cam.ac.uk/~mwd24/phd/swarm.html, Feburary 2003.

[7] Joseph D'Errico and Wei Qin. Constructing portable compiled instruction-set simulators: an adl-driven approach. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 112–117, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[8] Rainer Doemer, Andreas Gerstlauer, and Daniel Gajski. *SpecC Language Reference Manual*. SpecC Technology Open Consortium, 2.0 edition, December 2002.

[9] Andreas Gerstlauer, Rainer Doemer, Junyu Peng, and Daniel Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[10] Yitao Guo. A hybrid instruction set simulator for system level design. Master's thesis, University of California, Irvine, 2010.

[11] Yonghyun Hwang, Samar Abdi, and Daniel Gajski. Cycle-approximate retargetable performance estimation at the transaction level. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 3–8, New York, NY, USA, 2008. ACM.

[12] Kyoung Park Kim and Rainer Doemer. Design exploration using multiple arm instruction set simulators. Technical Report Technical Report CECS-09-08, Center for Embedded Computer Systems, University of California, Irvine, 2009.

[13] Anish Muttreja, Anand Raghunathan, Srivaths Ravi, and Niraj K. Jha. Hybrid simulation for embedded software energy estimation. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 23–26, New York, NY, USA, 2005. ACM.

[14] Gunar Schirner, Gautam Sachdeva, Andreas Gerstlauer, and Rainer Doemer. Modeling, simulation and synthesis in an embedded software design flow for an arm processor, May 2006.

[15] Suan Hsi Yong and Susan Horwitz. Pointer-range analysis. In *In SAS04, number 3148 in LNCS*, pages 133–148. Springer, 2004.

[16] Jianwen Zhu and Daniel Gajski. A retargetable, ultra-fast instruction set simulator. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 62, New York, NY, USA, 1999. ACM.