Accelerating Mixed-Abstraction SystemC Models on Multi-Core CPUs and GPUs

by

Anirudh Mohan Kaushik

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2014

Author's Declaration

Statement of Contributions:

Chapter 4:

Contributor	Contribution
Kaushik, A. M.	Manusript writing and systemc-clang tool design.
Patel, H. D.	Manuscript editing, aided in formalizing the definitions of Suspension-CFG and Suspension Labeled Transition System, and aided in initial systemc-clang tool design.

Abstract

Functional verification is a critical part in the hardware design process cycle, and it contributes for nearly two-thirds of the overall development time. With increasing complexity of hardware designs and shrinking time-to-market constraints, the time and resources spent on functional verification has increased considerably. To mitigate the increasing cost of functional verification, research and academia have been engaged in proposing techniques for improving the simulation of hardware designs, which is a key technique used in the functional verification process. However, the proposed techniques for accelerating the simulation of hardware designs do not leverage the performance benefits offered by multiprocessors/multi-core and heterogeneous processors available today. With the growing ubiquity of powerful heterogeneous computing systems, which integrate multi-processor/multi-core systems with heterogeneous processors such as GPUs, it is important to utilize these computing systems to address the functional verification bottleneck. In this thesis, I propose a technique for accelerating SystemC simulations across multi-core CPUs and GPUs. In particular, I focus on accelerating simulation of SystemC models that are described at both the Register-Transfer Level (RTL) and Transaction Level (TL) abstractions. The main contributions of this thesis are:

1. A methodology for accelerating the simulation of mixed abstraction SystemC models defined at the RTL and TL abstractions on multi-core CPUs and GPUs.

2. An open-source static framework for parsing, analyzing, and performing source-to-source translation of identified portions of a SystemC model for execution on multi-core CPUs and GPUs.

Acknowledgements

I am extremely grateful to my supervisor Prof. Hiren D. Patel for his guidance, support, and dedication throughout this thesis and during my graduate studies. His training and enthusiasm towards addressing challenging problems has had a tremendous positive effect in my life.

I thank my thesis committee members, Prof. Mark Aagaard and Prof. Derek Rayside for reviewing this thesis and for their constructive feedback.

I feel extremely privileged and humbled to have interacted with some amazing instructors, academics, and fellow students during my time here at UWaterloo. I sincerely thank you all for your wisdom and inspiration. I thank my colleagues at the CAESR lab Dan Wang, Zhuoran Yin (Jerry) and Mohamed Hassan for their knowledge and support during my graduate studies. Special thanks to my fellow Waterluvians : Siddharth Subramaniam, Bharathwaj Raghunathan, and Hemant Saxena for always obliging to go to Burrito Boyz.

I thank my VIT buddies for their love, support and wonderful memories: Srinivas Suryanarayan, Aakash Bothra, Navnit Narayan Das, Shenin Shyamkumar, Sanket Jaiswal, Krishnendu Piplai, Aditya Sarathy, Akshay Shrivastava, Pooja Premkumar, and Pratiksha M. Sharma.

To my cousin Srikanth Vaidyanathan, and his family (Geeta, Vrushabh, and Kaushika) - thank you for being a home away from home.

Finally, my heartfelt thanks to my parents for their unconditional love and support. They are the sole reason for my success, happiness, and making my dreams come true.

Dedication

This thesis is dedicated to my parents Mrs. Revathy Mohan and Mr. M. V. Mohan. This thesis would not be possible without their constant love, support, and encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The process of functional verification in the hardware design cycle is a crucial and a time consuming component. It is responsible for nearly two-thirds of the overall design cycle time, and it is widely acknowledged as one of the bottlenecks in the design cycle [1]. However, the increasing complexity of designs due to consumer demands and stringent time-to-market constraints have further exacerbated the resources and time put into the functional verification process. Figure 1.1 highlights the trends in design complexity and designer effort required to realize these complex designs. The design productivity gap in Figure 1.1 represents the disparity between what designs can be conceived based on the rate of number of transistors that can be packed in a single chip and designs that can actually be realized based on the available resources. As observed, this design productivity gap is wide and continues to widen.

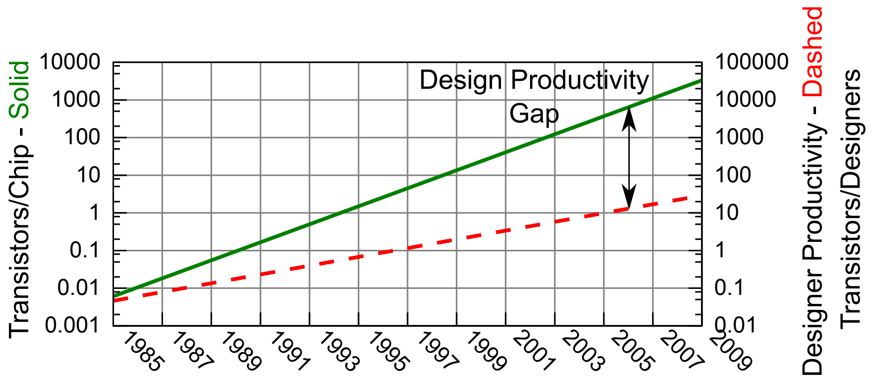


Figure 1.1: Design productivity gap trend [2].

Fortunately, academia and industry have proposed approaches to mitigate this widening gap. For example, the introduction of register-transfer level (RTL) abstraction over gate-level schematics was a welcomed relief to hardware designers. Although RTL is currently the defacto abstraction level for specifying synthesizable designs, the level of detail involved in specifying designs and the cycle accurate simulation associated with this abstraction level results in a time-consuming design process. This has led to the introduction of higher abstraction levels for specifying designs, such as Transaction Level (TL) to accelerate the design process. While these efforts have helped improve designer productivity, the effort needed to validate designs continues to grow due to incresing complexity of hardware designs.

SystemC [3] is a system level design language (SLDL) that suports modeling designs at both the RTL and TL abstractions. It has been approved by the IEEE Standards Association as IEEE 1666-2005 [4]. Its growing popularity among industries and academia is primarily attributed to its open-source nature. It is adopted and supported by many electronic design automation (EDA) industries in their design flow. SystemC is a collection of C/C++ classes that provide a notion of time, hardware concurrency, and support for hardware data-types. This has helped popularize its use among academia and industries as C/C++ is a familiar and commonly used programming language and requires a standard C++ compiler. In addition, the underlying language infrastructure, extensions, and object-oriented programming paradigm of C/C++ further extends the usability of SystemC over traditional SLDLs such as Verilog and VHDL. Although SystemC provides the ability to model hardware designs at the RTL and TL abstraction levels, it still struggles to deliver on its promise for faster design cycle times. This is because the SystemC's reference implementation is a single threaded implementation designed for uni-processor systems. The reference implementation employs co-operative multi-threading to simulate concurrent execution of hardware processes. With the current trend in computing shifting from high frequency uni-processors to multi-cores and multi-processors due to the latter's performance gains as a result of higher throughput, there has been considerable interest in parallelizing the SystemC scheduler [5, 6, 7, 8, 9, 10, 11]. Moreover, heterogeneous computing systems that combine multi-processor and multi-core systems with specialized hardware capable of accelerating specific types of computations are becoming increasingly ubiquitous due to their performance benefits. Graphics Processing Unit (GPU) is an example of a specialized processor capable of accelerating parallel computations such as graphics rendering in current heterogeneous computing systems. Initially, GPUs were exclusively used for performing graphics rendering. However, with the advent of proprietary and open-source frameworks for programming on GPUs, it is possible to perform compute intensive tasks that were previously executed on general processors on the GPU. The technique of performing compute intensive tasks on the GPU that are traditionally handled on general purpose processors is termed as General Purpose Computing on Graphics Processing Units (GPGPU). The performance benefits of heterogeneous systems with GPUs have contributed to its ubiquitousness in industry and academia. As computing systems shift from consisting of a single processor to multiple heterogeneous processors, applications need to be re-designed in order to leverage the parallelism provided. This works well for the case of simulation of hardware designs as hardware designs are inherently parallel.

The idea of using GPUs for logic simulation is not new, and has been studied in [12, 13, 14] that focus on parallelizing gate-level simulations. With respect to accelerating hardware designs described in SystemC, [15] and [16] have proposed techniques for accelerating SystemC RTL designs using GPUs. However, these works execute simple code structures on the GPU cores, and therefore do not take into consideration the effect of the control flow of a program on its execution time on the GPU. The control flow of a program is important when considering its execution on the GPU device as not all programs can be made to execute faster on the GPU. This is because, GPUs are designed for executing a single instruction on multiple data, and are devoid of micro-architectural features such as branch predictors and out-of-order processing that are otherwise common in general processors. Moreover, with the apparent benefits of designing models at the TL abstraction over RTL, and the ability to design models at the TL abstraction level using SystemC, it is important to consider accelerating mixed-abstraction (RTL and TL) models using these heterogeneous computing systems.

In this thesis, I focus on accelerating the simulation of mixed-abstraction SystemC models using heterogeneous systems with multi-core CPUs and GPUs, in particular NVIDIA GPUs. The motivation behind using SystemC is based on its growing rise in its adoption due to its free and open-source simulation and modeling framework. To this end, I develop an open-source static analysis framework called systemc-clang that parses SystemC RTL and TL models and extracts the structural and behavioral information present in the model. The mapping of the execution of a SystemC model across multi-core CPUs and GPUs, and the translation necessary for GPU execution are implemented as plugins to systemc-clang. My approach shows promising results, and with GPU architecture specific optimizations and performance based extensions to the SystemC scheduler, this technique can yield further improvements.

1.1 Main Contributions

The main contributions of this thesis are:

1. Improving the methodology proposed by Sinha et al. [17] for accelerating mixed-abstraction SystemC simulations defined at the RTL and TL abstractions across multi-core CPUs and GPUs.
    - Propose an intermediate representation (IR) to capture the behavioral semantics of SystemC processes and algorithms to generate the IR.

- Automate identification of a mapping of data-parallel computations in the input SystemC model across multi-core CPUs and GPUs that results in faster execution time of the SystemC model.

2. systemc-clang[18]: An extensible open-source framework for parsing and analyzing SystemC models

    - Develop an analysis framework for SystemC models that parses and collects relevant information regarding the structure and behavior of SystemC models.
    - Provide a convenient method for developing extensions to systemc-clang for further analysis of SystemC models using plugins.
    - Automate source-to-source translation of SystemC models to execute on multi-core CPUs and GPUs. Source-to-Source translation takes into account memory transfer overhead between CPU and GPU and therefore, implements an optimization that reduces memory transfer overhead between CPU and GPU.

3. Experimentally validate the efficiency of systemc-clang using SystemC RTL benchmarks from the S2CBench suite [19] and eight TLM models, and evaluate the proposed method of accelerating SystemC simulations against previous works using multi-core CPUs and GPUs.

1.2 Organization of Thesis

This thesis is organized as follows:

Chapter 2 provides a brief background on the SystemC simulation kernel and the constructs provided to describe hardware models. I also provide an overview of the TLM style of modeling hardware designs using SystemC. Towards the end of the chapter, I introduce the GPU architecture and the CUDA programming framework for executing general purpose applications on NVIDIA GPUs.

Chapter 3 discusses previous work related to accelerating hardware design simulation in general and SystemC simulations across multi-core CPUs and GPUs. I also discuss previous efforts on distributing computation across multi-core CPUs and GPUs for general purpose applications.

Chapter 4 introduces systemc-clang, the open-source static analysis framework for SystemC RTL and TL models developed in this thesis. In this chapter, I discuss the tool's implementation and features for extensibility in the form of systemc-clang plugins. I then discuss in detail the structural and behavioral IRs maintained by systemc-clang, and the transformations involved in generating the behavioral IR.

Chapter 5 discusses about the proposed methodology for accelerating mixed-abstraction SystemC models across multi-core CPUs and GPUs. I discuss in detail about the SCuitable plugin responsible for modifying the behavioral IR to generate a mapping of the SystemC model's execution across the processors in the heterogeneous platform. I also discuss about the translation plugin CUDA-Gen, which is responsible for translating the identified portions of the SystemC model for execution on the GPU to CUDA.

Chapter 6 compares the previous approaches for accelerating SystemC RTL simulations on GPUs with the proposed approach using the S2CBench benchmark suite. I also evaluate the proposed approach on four TLM models. In addition, I also compare the efficiency of systemc-clang based on its ability to extract the structural and behavioral information present in the input SystemC model.

Chapter 7 concludes this thesis by enumerating possible future extensions to this thesis and providing a summary of the work carried out in this thesis.

Chapter 2

Background

In this chapter I describe the SystemC simulation kernel and the architecture of GPUs. I also describe the CUDA programming model for NVIDIA GPUs with a simple example to illustrate the execution of parallel computations on NVIDIA GPUs.

2.1 SystemC

SystemC [3] is a widely used SLDL that consists of C/C++ classes that provide the ability to model hardware and software. There are five major extensions that SystemC provides to model hardware [20]:

- A notion of time
- Support for hardware data-types
- Module hierarchy and organization
- Concurrency
- Communication between different modules and processes

Using examples wherever needed, I describe these constructs and their implementation in SystemC.

A notion of time
SystemC provides a notion of time through the sc_time class. It consists of an enumerated type that provides a range of time resolutions such as SC_NS to denote nano-second resolution of time and SC_PS to denote pico-second resolution of time. For models requiring clocks, SystemC provides a class called sc_clock through which the clock cycle, duty period and resolution of the clock can be set.

Support for hardware data-types
In addition to the rich data-types provided by C/C++, SystemC also allows users to define arbitrary widths for integer and fixed-point data-types. For example, sc_uint⟨5⟩ is an unsigned integer data-type with a bit width of five. An important SystemC data-type that models real hardware is the sc_logic data-type that represent the uninitialized value 'X', high impedance value 'Z' and the boolean values '0' and '1'. The SystemC data-type classes also provide convenient functions such as range and bit select operations to operate on selected bits.

Module hierarchy and organization
Typical hardware designs consist of several hardware blocks interconnected through some communication interface, with each hardware block performing some function. In SystemC, these hardware blocks are defined as modules. SystemC modules are defined using the SC_MODULE class. The communication interfaces such as ports and signals, and the functional behavior of the module are defined in the module.

Communication between different modules and processes
Hardware processes communicate through wires, signals, and ports. SystemC provides two ways of communication between processes and modules: channels and events. Inter-module communication is carried out through channels while communication between processes are carried out by events. SystemC provides three types of channels: primitive, hierarchical, and evaluate-update channels. Primitive channels such as sc_fifo and sc_mutex are simple channels that are designed for fast prototyping and simulation. A hierarchical channel is a channel that contains processes and other channels. Evaluate-update channels model real-world electronic signals that operate on the delta-cycle paradigm. The sc_signal and sc_buffer are examples of evaluate-update channels.

Concurrency
Hardware designs are inherently parallel. However, traditional hardware simulators that run on single processor systems employ techniques to provide the illusion of concurrent execution. The SystemC simulation framework is no exception to this and employs a run-time scheduler that manages the scheduling and synchronization of concurrent hardware

processes. It uses a discrete-event (DE) scheduler that decides when to switch between concurrent processes in response to events. I provide a more in-depth description of the discrete-event scheduler in Section 2.1.2.

2.1.1 Overview of the Building Blocks of SystemC

Figure 2.1 is a SystemC model of a D flip-flop. I shall highlight the modules, processes, and module hierarchy for this example to give the reader a better understanding of the SystemC code structure.

```
#include "systemc.h"

SC_MODULE(dff) {
  sc_in<bool> clk;
  sc_in<bool> i_data;
  sc_out<bool> o_data;

  void dflipflop(){
    while(true) {
      wait();
      o_data.write(i_data.read());
    }
  }

  SC_CTOR(dff) {
    SC_THREAD(dflipflop);
    sensitive<<clk.pos();
  }
};
```

Figure 2.1: A SystemC description of D-Flip Flop.

Modules: As mentioned earlier, the modules represent hardware blocks of the SystemC model. In this example, dff is the module name. The module consists of the ports clk, i_data, and o_data. Ports clk and i_data are input boolean ports denoted by sc_in⟨bool⟩ and port o_data is the output port denoted by sc_out⟨bool⟩.

Processes: SystemC processes describe the implementation of the module. A module may have more than one SystemC process. For this particular example, dflipflop is the SystemC process. There are three different types of SystemC processes namely SC_METHOD, SC_THREAD, and SC_CTHREAD. The type of the process and its sensitivity list is defined in the constructor of the module. A SC_METHOD process is a SystemC process that is called by the SystemC scheduler whenever a signal changes on its sensitivity list. It cannot be suspended and resumed. A SC_THREAD on the other hand, is a SystemC process that can be suspended and resumed through wait() calls and event notifications. For this example, the process dflipflop is a SC_THREAD process that is sensitive to the positive edge of the clock. Notice that the dFlipFlop is suspended for every iteration of the while loop using a wait() call. An SC_CTHREAD is a variant of the SC_THREAD SystemC process, and is used to model clocked processes.

Events: Since SystemC utilizes a DE simulation paradigm, events are an important aspect for the scheduler. Events allow triggering suspended processes through a notification mechanism. SystemC provides three types of events: immediate, delta, and timed events. Immediate events are triggered immediately and executed in the same delta-cycle before any update to the signals. Delta events are caused due to change in the signals and are processed in the next delta-cycle. Timed events are triggered after a certain time has elapsed.

2.1.2 SystemC Simulation Kernel

The SystemC simulation kernel is responsible for the scheduling and synchronization of SystemC processes. Figure 2.2 describes the execution semantics of the scheduler. The execution of the scheduler can be split into two major phases: **Elaboration Phase** and **Execution Phase**. The **Elaboration Phase** builds the architecture of the SystemC model and establishes the module hierarchy by resolving the port bindings between modules. The **Elaboration Phase** is followed by the **Execution Phase**, which performs the simulation of the model. The **Execution Phase** is split into three phases: **Initialization**, **Evaluation**, and **Update** phases. The scheduler maintains a queue of ready-to-run processes and a queue of suspended processes that is updated during the simulation. The simulation begins with the **Initialization Phase** wherein all processes present in the model are placed in the ready-to-run queue. In the **Evaluation Phase**, processes from the ready-to-run queue are picked in an unspecified order and are executed either till the end of the process or on encountering wait() call. Suspended processes are placed in the suspended queue. During this phase, a process's execution might trigger some processes in the suspended queue to be available for execution. Therefore, these processes are placed back into the ready-to-run queue. The simulation kernel remains in this state until there are no more processes in the ready-to-run queue. When the ready-to-run queue is empty, the simulation kernel enters into the **Update** phase in which signal values are updated. At this point, change in signal values

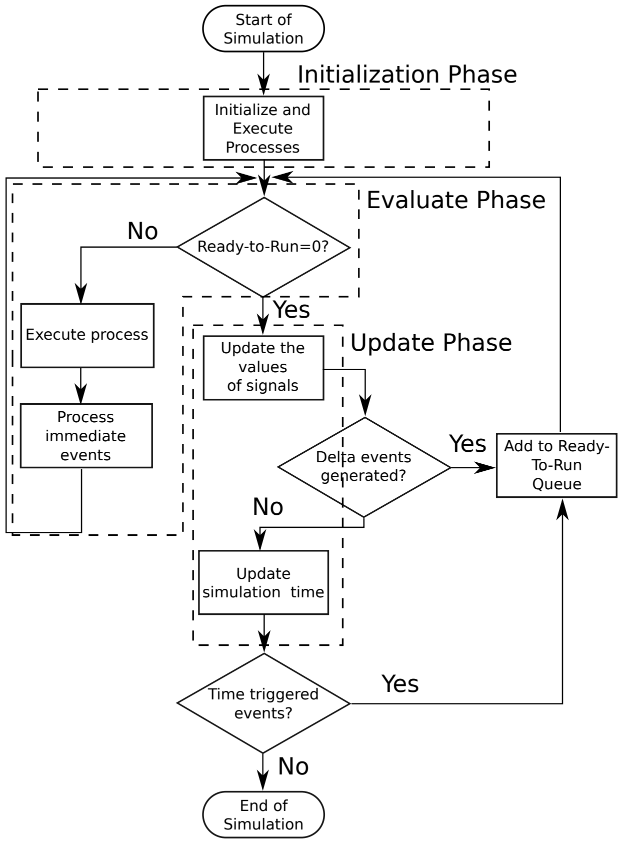


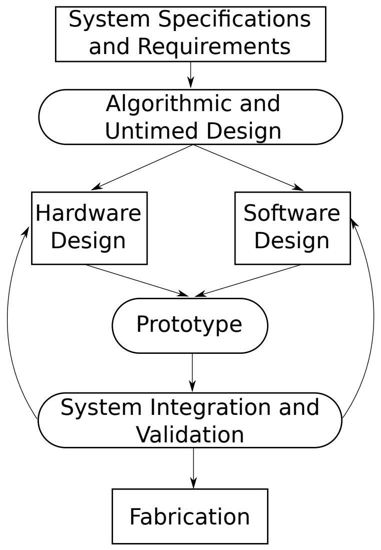
Figure 2.2: OSCI SystemC Simulation Kernel.

may trigger activation of some suspended processes. This is also known as a delta-cycle. The re-activation of processes due to the update of signal values causes the kernel to perform another round of evaluation. When no delta events are generated by the **Evaluation Phase** and **Update Phase**, the scheduler scans the suspended queue of processes, and updates the simulation time to the nearest time of activation. Simulation time is advanced to the earliest time of activation and the scheduler re-enters the **Evaluation Phase**. When there are no more processes in the suspended queue and the ready-to-run queue or the specified simulation time has been met, the kernel exits and performs any necessary cleanup.

2.1.3 Transaction Level Modeling (TLM) in SystemC

The conventional design process for System-on-Chip (SoC) designs requires defining hardware designs at the RTL abstraction level, which has proved detrimental to the design cycle with rising complexity and shrinking time-to-market constraints. This is because of the level of detail and cycle accuracy involved at this abstraction level. Figure 2.3a illustrates the conventional SoC design flow, which begins with the system requirements and specifications that includes a purely algorithmic and untimed model of the design. The flow then splits into two paths: hardware development and software development. The hardware development process involves designing the system at the cycle-accurate RTL abstraction level using a hardware descriptive language such as VHDL or Verilog. Necessary functional verification, synthesis, and timing analysis are carried out on the RTL design resulting in a prototype that is ready to be fabricated in a foundry. Meanwhile the software development process proceeds independently without having any knowledge about the hardware design development. Therefore, the validation of the software is delayed as it has to wait for a hardware prototype in order to deploy the software. This in turn is dependent on how quickly the hardware development process can roll out the prototype. Any bugs or modifications observed when integrating the software with the prototype results in a reiteration of the entire process leading to an expensive and time consuming design process. With shrinking time-to-market constraints, the SoC design flow has undergone modifications to provide a hardware/software co-simulation framework to speed up the design flow process and mitigate the high cost associated with the conventional design flow. This is shown in Figure 2.3b.

The Transaction Level Modeling (TLM) abstraction is a level of abstraction above RTL that is intended to improve hardware simulation by relaxing cycle accuracy and at the same time, serve as an appropriate platform for hardware/software verification. As shown in Figure 2.3b, the TLM model serves as a reference model between the hardware and software development process resulting in early and accurate verification and architectural exploration. The concept behind TLM designs is to abstract the cycle accurate details of communication between hardware modules and instead, use function calls for intermodule communication. In TLM designs, modules are categorized as initiators and targets. Initiators are modules that generate transactions to be sent over some channel connecting other modules, and targets are modules that accept the transactions generated by initiators. Interconnects act as both initiators and targets but do not modify the contents of the transaction passing through. The OSCI TLM 2.0 [22] standard provides three primary core interfaces for communicating transactions: transport interface, debug interface and direct memory interface. The transport interfaces are the primary interfaces connecting initiators and targets. It provides the blocking and non-blocking interfaces that provide different coding styles for TLM designs. The blocking transport interface is intended to capture the loosely-timed model (LT) of TLM designs. It is implemented as a virtual function


System Specifications and Requirements
Algorithmic and Untimed Design
Hardware Design
Software Design
Prototype
System Integration and Validation
Fabrication
(a) Conventional SoC Design Flow.

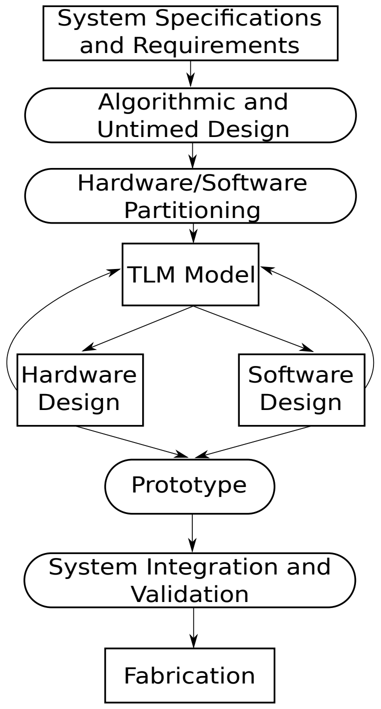

System Specifications and Requirements
Algorithmic and Untimed Design
Hardware/Software Partitioning
TLM Model
Hardware Design
Software Design
Prototype
System Integration and Validation
Fabrication
(b) Enhanced SoC Design Flow with TLM.

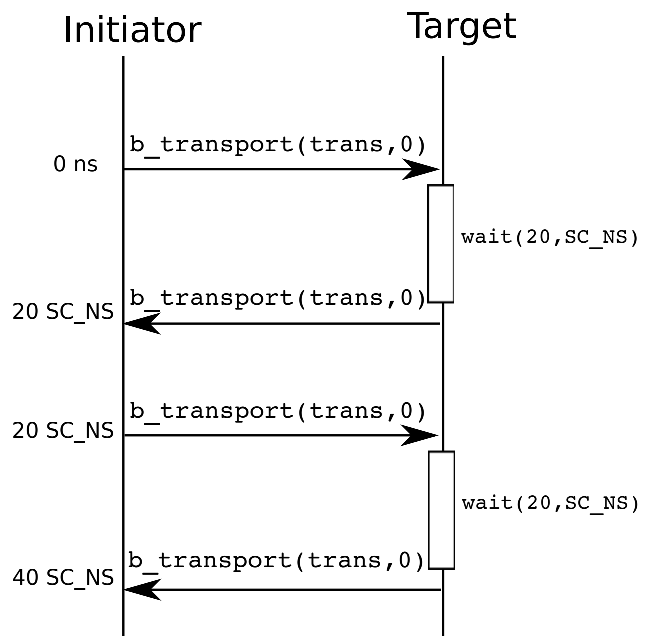
Figure 2.3: SoC Design Flow [21].

in the blocking transport interface class, and it is used to model transactions between initiators and targets as a single function call. It takes as arguments a reference to the transaction object and a timing delay, and does not have a return value. Figure 2.4 shows an example of a user defined blocking transport function call and the associated message sequence chart for the transport interface. A message sequence chart illustrates the interaction between the initiator and target using the transport interface. When the transport interface is called by the initiator, the method performs some operation on the transaction object trans, and then synchronizes with the SystemC scheduler via a wait() call.

However, frequent wait() calls result in high switching context overhead between the SystemC processes and the SystemC scheduler. Therefore, in order to reduce the context switching between SystemC processes and the SystemC scheduler to realize faster simulation, the blocking transport allows initiators to run ahead of the simulation time and synchronize with the SystemC scheduler when necessary. This is termed as temporal decoupling. In such a scenario, the ini-

```
void b_transport(TRANS &trans, sc_time& t_delay) {
  wait(20, SC_NS);
  perform_operation(trans);
}
```

(a) Blocking Transport Interface Method Definition.



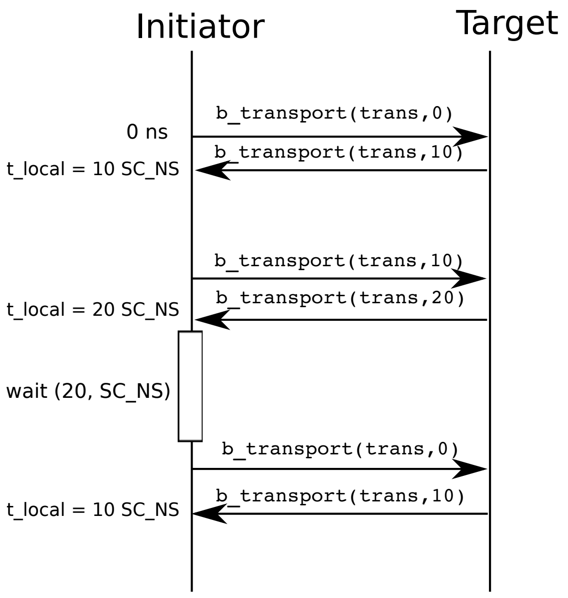
(b) Message Sequence Chart.

Figure 2.4: Example of a Blocking Transport Interface Method calling a wait().

tiator continues to accumulate its local time until it yields to the SystemC scheduler through a wait() call. Figure 2.5 illustrates the concept of temporal decoupling. The blocking transport increments the timing delay t_delay associated with the transport by 20 SC_NS whenever this blocking method is called. It continues to accumulate time until an explicit synchronization with the SystemC scheduler occurs at which time, the local time of the initiator is reset. A potential problem with temporal decoupling is that the initiator can continue to execute without advancing simulation time and thereby, prevent the SystemC scheduler from executing other processes. The TLM-2.0 library provides a time quantum class that ensures other concurrent processes are allowed to be scheduled by the SystemC scheduler. This is done by ensuring that the decoupled initiator synchronizes with the SystemC scheduler and resumes only when the next time quantum is reached.

```
void b_transport(TRANS &trans, sc_time &t_delay) {
  sc_time inc(10, SC_NS);
  t_delay = t_delay + inc;
  if(t_delay == sc_time(20, SC_NS)) {
    wait(20, SC_NS);
    t_delay = SC_ZERO_TIME;
  }
  perform_operation(trans);
}
```

(a) Blocking Transport Interface Method Definition.



(b) Message Sequence Chart.

Figure 2.5: Example of a Temporally Decoupled Blocking Transport Interface Method.

On the other hand, the non-blocking transport interface is intended to capture the approximately-timed model (AT) of TLM designs. AT models are useful for architectural analysis as they contain sufficient timing information. The transaction between the initiator and target is broken down into several phases with each phase having a timing constraint. The phases of the transaction adhere to some underlying communication protocol. The non-blocking transport interface takes as arguments a reference to the transaction, a phase argument, and a timing delay, and it returns an enumerated type that conveys whether the transaction object or timing delay has been modified by the target. The non-blocking interface uses a forward connection from initiator to target and a backward connection from target to initiator. Similar to the blocking transport, it is implemented as a virtual function in the abstract non-blocking transport interface class. The

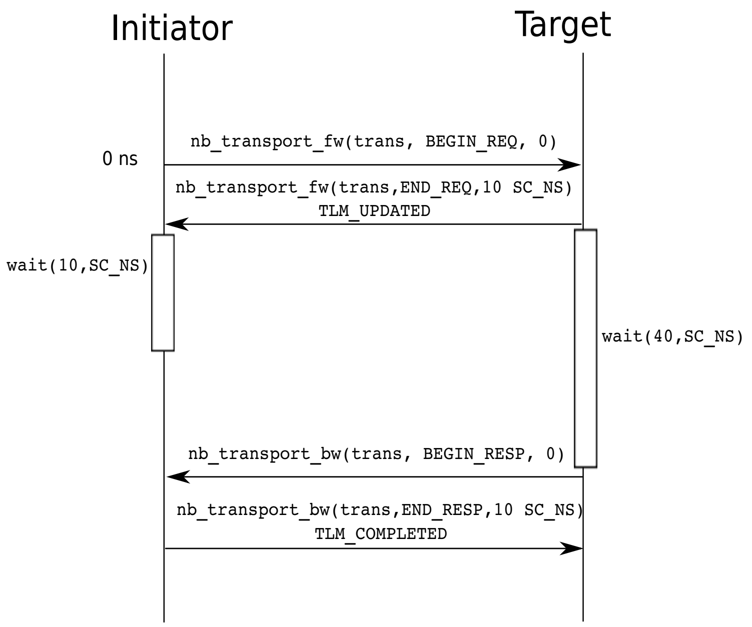
```
tlm::sync_enum target::nb_transport_fw(TRANS &trans, tlm_phase & phase,
    sc_time &delay) {

  perform_operation(trans);
  delay = sc_time(10, SC_NS);
  // Notify target module
  target_event.notify(40, SC_NS);
  phase = END_REQ;
  return TLM_UPDATED;
}

tlm_sync_enum initiator::nb_transport_bw(TRANS &trans, tlm_phase &phase,
    sc_time &delay) {

  if(phase != BEGIN_RESP) {
  // ERROR
  }
  perform_operation(trans);
  phase = END_RESP;
  return TLM_COMPLETED;
}
```

(a) Non-Blocking Transport Interface Method Definition.


Initiator
Target
nb_transport_fw(trans, BEGIN_REQ, 0)
0 ns
nb_transport_fw(trans,END_REQ,10 SC_NS)
TLM_UPDATED
wait(10,SC_NS)
wait(40,SC_NS)
nb_transport_bw(trans, BEGIN_RESP, 0)
nb_transport_bw(trans,END_RESP,10 SC_NS)
TLM_COMPLETED

(b) Message Sequence Chart.

Figure 2.6: Example of a Non-Blocking Transport Interface Method.

possible return values are TLM_ACCEPTED, TLM_UPDATED, and TLM_COMPLETED wherein TLM_ACCEPTED denotes that the target has not modified the parameters of the transaction object and timing delay, TLM_UPDATED denotes that the target has modified the contents of the transaction object or incremented the timing delay, and TLM_COMPLETED denotes the end of the transaction. The phases of the base protocol are BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP. Figure 2.6 shows an example of a forward and backward non-blocking transport interface methods that are implemented in the target and initiator respectively, along with the message sequence chart. The thread process in the initiator begins the transaction by first setting the attributes of the transaction object and invoking the nb_transport_fw method with the initial values of the phase and timing delay set to BEGIN_RESP and 0 SC_NS respectively. As shown in Figure 2.6, the nb_transport_fw method is defined in the target module. It performs some operation on the transaction object and modifies the phase and timing delay to END_RESP and 10 SC_NS respectively. It also schedules the activation of the event target_event on which the target module is currently waiting on. Since the target has modified the transaction object, phase, and timing delay, it returns TLM_UPDATED on the return path. The return path is simply the return value of the function. The nb_transport_fw interface notifies an event that the target is sensitive to, which communicates to the initiator via the nb_transport_bw path with phase set to BEGIN_RESP. On return to the initiator, the initiator suspends itself for a duration of 10 SC_NS. The definition of the backward transport interface is defined in the initiator, which terminates the transaction by modifying the phase argument to END_RESP and returning TLM_COMPLETED.

The Direct Memory Interface (DMI) is a core interface that allows initiators to directly access a target's memory area without having to rely on the transport interface. The Debug Interface is a convenient interface that allows performing transactions to the target via the transport interface without having to perform any synchronizations with the SystemC scheduler. The OSCI TLM-2.0 [22] provides a data structure aimed at modeling memory mapped buses. This data structure is termed as the generic payload, and it consists of fields such as read/write command, address, data, and burst width. In order to provide interoperability between TLM designs, the generic payload is usually the transaction object that is communicated between initiators and targets via the interfaces.

2.2 Graphics Processing Units

Graphics Processing Units (GPUs) are massively parallel processors capable of performing simultaneous integer and floating-point computations. The need for GPUs and its architectural design has been shaped by the demand to perform multiple floating-point calculations per video frame for the gaming industry. This has resulted in GPU designs to primarily dedicate their silicon area towards computational units resulting in packing more simpler compute cores in the given area. This architectural difference supports the capability of GPUs to perform simultaneous compute operations in less time over traditional CPUs. However, the GPU architecture cannot be considered as a panacea design for all types of computations and therefore, there are types of computations that are better suited on the CPU due to the presence of additional micro-architectural features. In the next sections, I shall present a complete overview of the GPU architecture, and discuss about the CUDA programming model for NVIDIA GPUs.

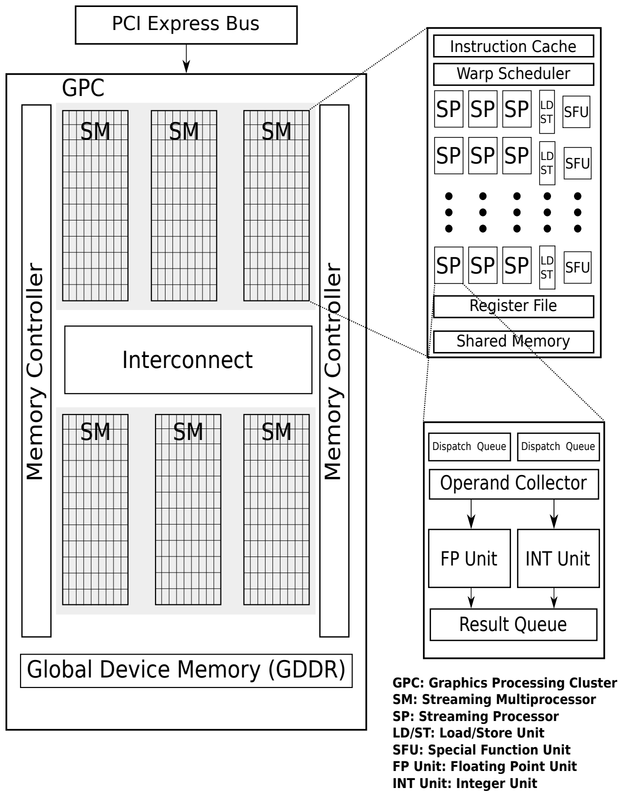


Figure 2.7: Generic NVIDIA GPU Architecture.

2.2.1 Architecture

Figure 2.7 illustrates the architecture of a generic NVIDIA GPU. The architecture can be broken down to three levels: Graphics Processor Clusters (GPCs), Streaming Multiprocessors (SMs), and Streaming Processors (SPs). The GPCs house a number of SMs (four SMs in each GPC in

Figure 2.7), and each GPC is connected to other GPCs via some interconnect. The SMs in turn house a number of SPs. Over the years, the number of SPs packed into each SM for different generations of GPU designs has increased consistently resulting in more throughput. A SM consists of a thread issue unit that issues threads in an in-order manner, special function units, and an on-chip shared memory. Usually, the number of threads that a single SM can manage is about 1024 threads. A SP is a very simple core that consists of a floating-point unit and an integer unit. The simplicity of the SP is key to packing more compute cores for accelerating parallel computations.

2.2.2 CUDA Programming Model

The CUDA programming model is an extension to the C language with support for single instruction multiple data (SIMD) style of parallel computation and synchronization. It is a proprietary framework by NVIDIA and applicable for NVIDIA GPUs. Figure 2.8 illustrates a vector addition example written using CUDA with which I shall use to describe the phases of the CUDA programming model and highlight some important extensions.

The CPU is responsible for initializing and allocating memory for the data structures required for computation on the GPU device. The cudaMalloc calls that are denoted in Lines 24-26 are extensions to the C malloc functions, and initialize the data structures in the global memory of the GPU. The cudaMemCpy calls in Lines 28-29 transfer data between the CPU memory region and the designated GPU global memory region. The final argument to the cudaMemCpy call indicates the direction of transfer. An argument of the form cudaMemcpyHostToDevice indicates the direction of transfer from the host (CPU) to the device(GPU), and cudaMemcpyDeviceToHost represents the direction of transfer from device (GPU) to host (CPU).

The function **vecAdd** is the data-parallel computation executed on the GPU, and it is launched with a certain thread hierarchy as shown in Lines 31-32. The thread hierarchy is composed of blocks and grids. Thread blocks are three-dimensional arrangement of threads, and grids are two-dimensional arrangement of blocks. For the example shown, the function **vecAdd** is launched with 2 thread blocks each with 512 threads. On launching **vecAdd** on the GPU, control is transfered to the GPU for execution. Since the CUDA programming model follows a SIMD style of parallel computation, the function **vecAdd**, which is defined in Lines 6-10, is replicated for each thread and each thread accesses an exclusive portion of the data structures using the thread identifier as index. Threads within a thread block are bunched into groups of 32 threads called warps. All threads in a warp execute in lock-step on the same instruction but on different data. Thread divergence within a warp due to thread identifier dependent conditions results in serialization of warp execution. The synchronization of threads within a thread block is carried

```
#include <cuda_runtime.h>
#include <cuda.h>

#define SIZE 1024

__global__ void vecAdd(int *gpu_a, int *gpu_b, int *gpu_out) {
  unsigned int i = threadIdx.x ;
  gpu_out[i] = gpu_a[i] + gpu_b[i];
  __syncthreads();
}

int main(int argc, char *argv[]) {
  int *a;
  int *b;
  int *gpu_a;
  int *gpu_b;
  int *out_sum;
  int *gpu_sum;

  a = (int*)malloc(SIZE*sizeof(int));
  b = (int*)malloc(SIZE*sizeof(int));
  out_sum = (int*)malloc(SIZE*sizeof(int));

  cudaMalloc((void**)&gpu_a, sizeof(int)*SIZE);
  cudaMalloc((void**)&gpu_b, sizeof(int)*SIZE);
  cudaMalloc((void**)&gpu_sum, sizeof(int)*SIZE);

  cudaMemcpy(gpu_a, a, SIZE*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_b, b, SIZE*sizeof(int), cudaMemcpyHostToDevice);

  dim3 grid(2, 1, 1);
  dim3 block (512, 1, 1);

  vecAdd<<<grid, block>>>(gpu_a, gpu_b, gpu_sum);
  cudaDeviceSynchronize();
  cudaMemcpy(out_sum, gpu_sum, SIZE*sizeof(int), cudaMemcpyDeviceToHost);
  cudaFree(gpu_a);  cudaFree(gpu_b);  cudaFree(gpu_sum);
  free(a); free(b); free(sum);
  return 0;
}
```

Figure 2.8: Vector addition code in CUDA.

out using the __syncthreads() function. On completion of **vecAdd** on the GPU, control returns back to the CPU and the result of the computation is copied back to the host memory region using the cudaMemCpy call.

The CUDA memory model consists of different types of memories that have varying impact on the code's execution time on the GPU. The global memory is a read/write memory that is common to all threads executing across the GPU. The latency of transferring data to the global memory via the slow PCI express bus is a bottleneck in most GPU applications. This is cause for concern as in some cases it overshadows the fast computation time that the GPU provides. I revert to this point when I discuss about the proposed technique for accelerating SystemC simulations, and I refer the reader to [23] for an interesting analysis on the contribution of memory copy overhead towards execution time of applications on GPUs. Registers and shared memory are on-chip memories. Each thread has its own set of the registers, and is typically used to hold the thread's frequently used data. Shared memory is allocated to each thread block, and is a convenient way for threads within a thread block to perform coordinated access to data as it is much faster compared to accessing global memory. A typical use case for shared memory is to bring in a chunk of data used frequently from the global memory to the shared memory and perform computations to this shared block.

Chapter 3

Related Work

In this section, I shall introduce previous related works aimed at improving simuation of hardware designs and the tools that aid in analysing, parsing, and representing SystemC designs for performance analysis and verification purposes. I also discuss previous work aimed at distributing computations across CPUs and GPUs for general purpose applications.

3.1 Parallel Discrete Event Simulation

There have been several research efforts that propose modifications to the SystemC kernel to leverage the parallelism offered by multi-processor and multi-core systems. The execution of discrete event simulation kernel on parallel processors is termed as parallel discrete event simulation (PDES). There are two categories of approaches towards PDES: conservative and opportunistic. Conservative approaches process events in a causal order, and ensure that no causality violations occur by constructing algorithms to determine when an event is safe to process. Shumacher et al. [5], Chandran et al. [7], and Chopard et al. [11] propose conservative approaches for parallel simulation of synchronous SystemC models on CPUs. The proposed effort by Chopard et al. [11] places the SystemC scheduler on one node, and synchronizes the processes executing on other nodes at every delta-cycle. To maintain a consistent view of simulation time, a master node is made responsible to collect timed events and notifying all processes when a timed event is processed. Schumacher et al. [5] propose a synchronous parallel SystemC scheduler that executes ready-to-run processes on available cores. Before entering the update phase, the parallel processes are synchronized, and the channel updates happen in a sequential manner. Chandran et al. [7] propose a partitioning and grouping of SystemC processes in addition to parallelizing the simulation kernel of SystemC. On the other hand, Mello et al. [8] and Jones et al. [10] propose opportunistic methods for accelerating the simulation of loosely timed TL models. However Mello et al. [8] do not support timed and immediate notifications, which prohibits mixed-abstraction simulation, and re-use of existing TL models using other SystemC events. Jones et al. [10] enable parallel simulation of mixed abstraction RTL and TL models, however they require users to explicitly describe temporal constraints to limit temporal decoupling. A more recent work by Moy [24] proposes a programming model that allows tasks to be spread across time thereby improving the opportunity for parallel simulation of SystemC TL loosely timed (LT) models. The key observation made by Moy is that traditional TLM code models the time taken by a computation by performing the computation and then calling a wait() statement where the timing argument to wait() call represents the time taken by the computation. This hampers the opportunity for extracting parallelism in TLM models as unlike in RTL models where a clock tick can place multiple processes in the ready-to-run queue of the SystemC scheduler, SystemC TL models deal with quantitative time. Hence, Moy proposes a SystemC construct of the form sc_during that takes as input a sequence of computation code blocks and a timing parameter. The sc_during construct spreads the computation across the time mentioned in the timing parameter. By spreading the computation across time rather than performing the computation and then suspending the process for a specific time, more opportunities for parallel execution of loosely timed TLM models can be obtained as overlapping computations in time can be carried out in parallel provided these computations are independent. For a detailed analysis on the origins of PDES and some of the efforts related to PDES, I refer the reader to [25].

Recent works on accelerating SpecC simulations have combined parallelizing the simulation kernel and discovering opportunities for out-of-order execution [26], [27] and [28]. These efforts rely on sophisticated compiler techniques to statically analyze SpecC models and detect timing and data hazards on channels, variables, and ports.

In this proposed effort, I use the parallel SystemC scheduler developed by Sinha et al.[17] that is similar to the one proposed by Schumacher [5] with support for TLM models.

3.2 Accelerating SystemC models using GPUs

Nanjundappa et al. [15] parallelize the simulation of SystemC RTL models on GPUs. Their work is the first effort to harness the potential of GPUs as platforms for parallelizing hardware simulations. They propose a methodology to translate SystemC RTL descriptions into CUDA kernels that preserve the original DE semantics. Their method places each SystemC process on a distinct CUDA thread belonging to a different warp, in order to minimize the adverse effect of thread divergence. To model the suspension and activation of SC_THREADS, they convert an SC_THREAD description to a switch-case statement with synchronization barriers. Vinco et al. [16], propose an alternative method for simulating SystemC RTL models on GPU. Their work proposes a process duplication mechanism to improve opportunities for parallel execution and reduce the number of synchronization barriers compared to the approach by Nanjundappa et al. [15]. The process duplication mechanism results in the creation of independent data-flows that can be executed on separate SMs of the GPU. For SystemC models with a large number of concurrent SystemC processes, the above approaches work well as these concurrent SystemC processes can be executed on distinct GPU cores [15] or on distinct multiprocessors [16]. However, these approaches cannot be used to accelerate simulation of SystemC TL models that allow event and variable timed SystemC wait() calls. This is because current GPUs cannot suspend their execution and resume execution from the next instruction. Realizing the need to support accelerating SystemC models defined at abstraction levels above the RTL abstraction level, Sinha et al. [17] propose a CUDA SystemC library to facilitate handling of immediate, timed, and delta wait() calls, and notifications thereby simulating both RTL and TL models. Their methodology generates wrapper functions for processes identified for GPU execution through user-guided hints, which is responsible for communicating events across concurrent processes. However, their approach does not take into consideration the program characteristics of the SystemC processes that may or may not be beneficial for GPU execution. Moreover, their approach suffers from frequent memory transfers across the CPU and GPU in the presence of multiple SystemC scheduler calls (wait() and notify()). This is because the SystemC scheduler calls are handled by the SystemC scheduler resident on the CPU and therefore, the state of the GPU execution has to be saved and restored on encountering a SystemC scheduler call. In this thesis, I work on modifying Sinha et al. [17] approach to take into account of this bottleneck of memory transfer across CPU and GPU, and relieve the effort involved in identifying processes suitable for CPU and GPU execution by automating this process. I refer the reader to [29] and [30] for a thorough investigation on the recent works related to accelerating SystemC simulations on GPUs.

3.3 SystemC Front-Ends

As SystemC is a SLDL that is based on C/C++, it poses a challenge to designers responsible for building automation and analysis tools to assist in the design and verification process. This is because analyzing SystemC models requires the ability to parse, represent, and comprehend SystemC models. Realizing the need for SystemC front-ends and the unavailability of a reference SystemC front-end, there are several open-source and closed-source front-ends available. These front-ends can be broadly classified into two categories: static and dynamic front-ends. Static front-ends use static methods to extract the structural and behavior information of SystemC models and represent the parsed information in an intermediate representation. SystemPerl [31], KaSCPar [32], Scoot [33], and SystemCXML [34] are examples of static SystemC front-ends. SystemPerl [31] generates Perl scripts to parse and extract structural information from an input SystemC model. SystemCXML [34] is another static SystemC front-end that uses the Doxygen documentation tool to construct an intermediate representation in XML format that captures the structural information of the SystemC input design. Static parsers that utilize compiler analysis for parsing the SystemC design include ParSysC [35], Scoot [33], Pinapa [36], and PinaVM [37]. ParSysC [35] utilizes the Purdue Compiler Construction Tool Set (PCCTS), which is a set of tools designed for creating compilers, to extract structural information of SystemC designs. Scoot [33] is another static SystemC front-end based on the GCC compiler that extracts structural information from SystemC models and provides extensions for type-checking and code re-synthesis for faster simulations. On the other hand, dynamic parsers such as PinaVM [37] and Pinapa [36] observe that the entire structure of the SystemC model cannot be constructed from static analysis due to dynamically instantiated modules. Therefore, such parsers extract the hierarchy of the SystemC model at run-time by executing the elaboration phase of the SystemC scheduler. PinaVM [37] is a revamped version of Pinapa [36] that utilizes the LLVM framework. PinaVM [37] extracts additional information of the architecture of the design by analyzing the bit code generated by the LLVM just-in-time compiler (JIT). Table 3.1 summarizes the capabilities of available SystemC front-ends. For a more in-depth analysis of the available SystemC front-ends, I refer the reader to [38].

While there are merits in both categories of SystemC front-ends, I believe that a front-end that distributes the analysis of SystemC models between static and dynamic approaches is beneficial. It is important to note that none of the previous approaches parse SystemC models described above the RTL abstraction level such as TL abstraction level. Therefore, I have focused on developing an open-source static framework for parsing SystemC models defined at the RTL and TLM levels of abstraction that I hope to integrate with the available dynamic front-ends such as PinaVM [37]. To my knowledge, HIFSuite [39] supports parsing of SystemC TL models. However, HIFSuite is a commercial and closed-source tool, and I believe that an open-source tool is essential to the community and to the tool's development.

3.4 Mapping Applications Across CPUs and GPUs

With the rising popularity of heterogeneous computing platforms with GPUs, there have been numerous research efforts to distribute the execution of an application on a heterogeneous computing system. Luk et al. [40] propose an automatic adaptive mapping framework that is built on top of threading libraries. It maintains a history of the runtime of an application on multi-core

SystemC front-end	Type of front-end	Abstraction Level Supported	Technique
SystemPerl [31]	Static	RTL	Uses Perl scripts to extract structural information.
SystemCXML [34]	Static	RTL	Extracts structural information using Doxygen and generates an intermediate representation of SystemC model.
ParSysC [35]	Static	RTL	Uses the Purdue Compiler Construction Tool Set (PCCTS) for parsing SystemC models.
Scoot [33]	Static	RTL	Uses the GCC compiler for parsing SystemC models and supports type-checking and code re-synthesis for faster simulation.
Pinapa [36]	Dynamic	RTL	Uses the GCC compiler and executes the elaboration phase of the SystemC scheduler for constructing module hierarchy.
PinaVM [37]	Dynamic	RTL	Similar to Pinapa but uses the LLVM compiler instead.

Table 3.1: Summary of SystemC front-ends.

CPUs and GPUs by performing a training run of the application. Using the profile history, they develop curve fitting equations to determine a near optimal mapping for the same application with different data sets. A more recent work by Wang et al. [41] propose a scheduling scheme that is a hybrid between sampling scheduling and splitting scheduling. They break the execution of the application into several phases and determine a stable partitioning when the variances of the performance ration of two consecutive executions are small. However, these works do not address the issue of GPU contention due to concurrent GPU kernel invocations for a heterogeneous architecture where a GPU is shared by many CPUs. Gregg et al. [42], address the above by querying the device during runtime to determine whether the device is utilized and then making a decision based on the profiling time scaled by a contention factor. The authors of [43] use machine learning techniques to extract an optimal mapping of OpenCL kernels on CPU-GPU based systems in the presence of GPU contention. A recent work by Gohli et al. [44] propose a static technique that derives a mapping of an application across processors in a heterogeneous computing system by using Monte Carlo search trees. Diop et al. [45] propose a framework for executing OpenCL kernels across a GPU cluster using meta-functions that abstract the underlying architecture of the cluster and, thereby allowing the programmer to focus on improving the algorithm of the application. In this thesis, I focus on developing a straightforward static approach for determining a mapping of the execution of the SystemC model based on the profile times of the data-parallel computations considered for GPU execution.

Chapter 4

systemc-clang: A Framework for Static Analysis of SystemC RTL and TL Models

This chapter presents systemc-clang, an extensible open-source framework for analyzing SystemC models defined at the RTL and TL abstraction levels.

4.1 systemc-clang Tool Flow

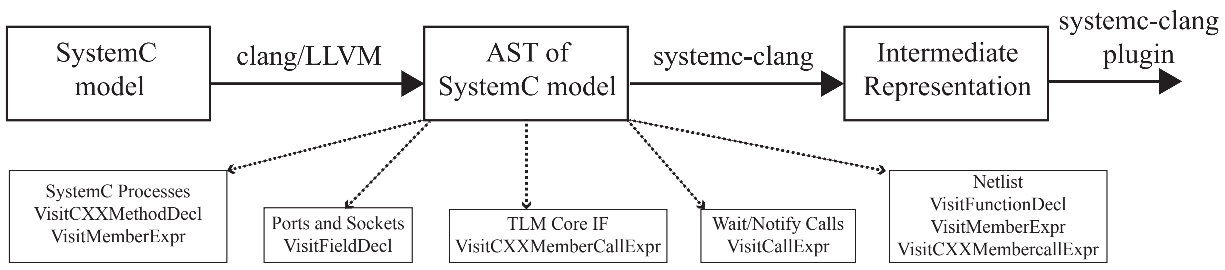

SystemC model
clang/LLVM
AST of SystemC model
systemc-clang
Intermediate Representation
systemc-clang plugin
SystemC Processes VisitCXXMethodDecl VisitMemberExpr
Ports and Sockets VisitFieldDecl
TLM Core IF VisitCXXMemberCallExpr
Wait/Notify Calls VisitCallExpr
Netlist VisitFunctionDecl VisitMemberExpr VisitCXXMembercallExpr

Figure 4.1: systemc-clang Tool Flow [46].

Figure 4.1 shows the tool flow of systemc-clang[18]. The input to systemc-clang is the Abstract Syntax Tree (AST) of the SystemC model generated by the LLVM/clang [47] framework. To extract the necessary structural and behavioral information present in the input SystemC model, systemc-clang dispatches a number of virtual functions of the form VisitNode(Node*) to visit AST nodes of interest. For example, VisitCXXMethodDecl(CXXMethodDecl*) is a virtual method that visits methods of a C/C++ class, which is used by systemc-clang to extract

SystemC processes constituting a SystemC module. The structural and behavioral information extracted by systemc-clang are stored as intermediate representations (IRs) that can be queried and analyzed to extract further information. To query the IRs, systemc-clang provides application programming interfaces (APIs) that can be used by systemc-clang plugins to perform further analysis on the parsed information. Before going into the details of the tool, I introduce the basic classes and features of clang that are utilized by systemc-clang.

clang constructs	Description
CXXRecordDecl	Derived from base class Decl. Represents a C/C++ class or structure.
CXXMethodDecl	Derived from base class Decl. Represents a method of a C/C++ class.
MemberExpr	Derived from base class Stmt. Represents a member of a C/C++ class or structure.
CallExpr	Derived from base class Stmt. Represents a call to a function.
PointerType	Derived from base class Type. Represents pointer declarators.
VisitMemberExpr	Visits AST node of type MemberExpr.
VisitCallExpr	Visits AST node of type CallExpr.
VisitCXXMethodDecl	Visits AST node of type CXXMethodDecl.
VisitCXXRecordDecl	Visits AST node of type CXXRecordDecl.

Table 4.1: Fundamental clang classes and methods.

4.2 clang Basics

The abstract syntax tree (AST) is the primary input to systemc-clang that is parsed to build the IRs and carry out analysis. The ASTContext class holds all the information about the AST for a translation unit, which is a basic unit of compilation. It provides the entry point to the AST via the function getTranslationUnitDecl(). To traverse the AST obtained from the getTranslationUnitDecl() function, clang provides two AST visitor classes: ASTVisitor and RecursiveASTVisitor. An AST visitor is a class that traverses the AST. The difference between the ASTVisitor and RecursiveASTVisitor class is that the RecursiveASTVisitor class performs a pre-order depth-first traversal of the AST in a recursive manner. systemc-clang uses the RecursiveASTVisitor class for traversing the AST as this class is a public API, and it is has been subjected to many improvements by the clang community. The RecursiveASTVisitor class internally uses the ASTVisitor class. The RecursiveASTVisitor class provides virtual functions of the form VisitNode(Node*), that visits nodes of type Node. For instance, VisitCXXMethodDecl(CXXMethodDecl*) is a virtual function that visits AST nodes of type CXXMethodDecl. The return type associated with the visit functions determines whether to continue traversing the AST or not. A return type of true will continue traversing the AST.

The base classes of clang [48] are Stmt, Decl, and Type. The Stmt class represents a C/C++ statement. It is a base class for many derived classes such as CompoundStmt, which represents a group of statements and IfStmt, which represents an if/then statement. The Decl class represents a C/C++ declaration of a variable, function, class, structure etc. Classes such as FunctionDecl that represents a function declaration and CXXRecordDecl that represents a C/C++ class/struct/union declaration are derived from the Decl base class. The Type class represents C/C++ types. Classes such as PointerType and ArrayType that represent pointer declarators and array declarators respectively are derived classes of the Type class. Table 4.1 summarizes the basic and derived clang classes along with some additional virtual functions provided by RecursiveASTVisitor class.

4.3 Extraction of Structural and Behavioral Information

In this section, I highlight some of the structural information that is extracted by systemc-clang using examples wherever necessary. In the following subsections I describe the extraction methodology of systemc-clang by highlighting some examples of the structural information extracted. Table 4.2 tabulates all the information systemc-clang extracts from a given SystemC model.

Information Type	Properties
Modules	Module Name
Processes	Process Type, sensitivity, properties such as payload information and wait() calls
Ports	Port Name, Port Access Type, Data Type
Events	Event Name
Class Members	Member Name, Member Type
Sockets	Socket Name, Protocol Type, Data bus Width, Socket Type, Callback Methods
wait() Calls	Type of wait(), Duration of wait(), Event Name
Event Notification Calls	Type of event notifications, Duration of event notifications, and Event Name.
Local Variables	Variable Name, Variable Type
TLM Payload	Payload Pointer, Command, Address, Data Pointer, Data Length, Streaming Width, Byte Enable Pointer, DMI Allowed, Initial Response Status, Extensions
TLM Core Interface	Socket Name, Interface Type, Payload Pointer, Delay Argument, Phase Argument Starting address, Ending Address

Table 4.2: Summary of information extracted by systemc-clang from a SystemC model.

4.3.1 Extraction of SystemC modules

Modules are the basic building blocks of a SystemC design. Figure 4.2a, shows a partial AST dump of a SystemC module declaration, and the FindSCModule class implemented in systemc-clang. Relevant information in the AST and in the class definition are highlighted. Since the modules are defined as SC_MODULE, which is a C/C++ class, the FindSCModule class uses the VisitCXXRecordDecl(CXXRecordDecl*) to identify SC_MODULE declarations. To identify SC_MODULES, systemc-clang iterates over an identified CXXRecordDecl node and checks if its base class type is sc_core::sc_module. It can be observed from the highlighted AST shown in Figure 4.2a that the AST node type of SC_MODULE(dff) is CXXReordDecl. On identifying the SC_MODULE, the name of the module is extracted using the getIdentifier() function. Figure 4.2b shows the FindSCModule class definition of systemc-clang responsible for identifying the SC_MODULES.

```
CXXRecordDecl 0x4893f10 struct dff
|-public '::sc_core::sc_module':'class sc_core::sc_module'
|-CXXRecordDecl 0x48940d0 struct dff
|-FieldDecl 0x4894220 clk 'sc_in<_Bool>':'class sc_core::sc_in<_Bool>'
|-FieldDecl 0x4894330 i_data 'sc_in<_Bool>':'class sc_core::sc_in<_Bool>'
|-FieldDecl 0x489fcc0 o_data 'sc_out<_Bool>':'class sc_core::sc_out<_Bool>'
```

(a) SC_MODULE AST.

```
class FindModule:public RecursiveASTVisitor<FindModule>{
  public:
    FindModule(CXXRecordDecl*, llvm::raw_ostream& );
    ~FindModule();
    // Typedefs declarations

    //Virtual methods from RecursiveASTVisitor class
    virtual bool VisitCXXRecordDecl(CXXRecordDecl *decl);

    // Member declarations
   private:
    CXXRecordDecl* _decl;
    string _moduleName;
    bool _isSystemCModule;
}
```

```
bool FindModule::VisitCXXRecordDecl(CXXRecordDecl *d) {
  for (CXXRecordDecl::base_class_iterator
       bi = _decl->bases_begin(),
       be = _decl->bases_end();
       bi != be; ++bi) {
     QualType q = bi->getType();
     string baseName = q.getAsString();
     if (baseName == "::sc_core::sc_module"){
       _isSystemCModule = true;
       IdentifierInfo *info = _decl->getIdentifier();
       if (info != NULL) {
         _moduleName = info->getNameStart();
       }
     }
  }
  if(_isSystemCModule == false) {
    return false;
  }
  return false;
}
```

(b) FindModule class in systemc-clang.

Figure 4.2: Extraction of SystemC modules by systemc-clang.

4.3.2 Extraction of SystemC ports

systemc-clang extracts information regarding the direction and data-type of ports. For instance, for a port of the form sc_in⟨bool⟩, systemc-clang will report it to be an input port of boolean type. Figure 4.3a shows the AST for port declarations inside a SystemC module. To extract information regarding the ports, systemc-clang dispatches the VisitFieldDecl(FieldDecl*) function to traverse the AST node representing port declarations as shown in Figure 4.3b. The portType data type holds the mapping of the port name with the data type. The three portTypes represent input, output, and inout ports. From the AST shown, the ports clk, i_data, and o_data are identified. TLM-2.0 sockets are identified in a similar manner by traversing AST nodes of type FieldDecl.

4.3.3 Extraction of TLM-2.0 core interfaces

Since the TLM-2.0 core interfaces are implemented as user-overridable functions that belong to an abstract interface class, systemc-clang identifies the TLM-2.0 core interfaces used by inspecting AST nodes of type CXXMemberCallExpr. For the identified transport interface methods, the

```
FieldDecl 0x59eb020 clk 'sc_in<_Bool>':'class sc_core::sc_in<_Bool>'
FieldDecl 0x59eb130 i_data 'sc_in<_Bool>':'class sc_core::sc_in<_Bool>'
FieldDecl 0x59f6b30 o_data 'sc_out<_Bool>':'class sc_core::sc_out<_Bool>'
```

(a) SystemC Ports AST.

```
class FindPorts: public RecursiveASTVisitor<FindPorts> {
  public:
    FindPorts(CXXRecordDecl*, llvm::raw_ostream&);
    ~FindPorts();

    // Typedef Declarations

    // Virtual methods from RecursiveASTVisitor class
    virtual bool VisitFieldDecl(FieldDecl*);

    // Member Declarations
   private:
    portType _inPorts;
    portType _outPorts;
    portType _inoutPorts;
  };
}
```

```
bool FindPorts::VisitFieldDecl(FieldDecl* fd) {
  QualType q = fd->getType();
  string fname;
  if (IdentifierInfo *info = fd->getIdentifier()) {
    fname = info->getNameStart();
  }

  const Type* tp = q.getTypePtr();
  FindTemplateTypes* te = new FindTemplateTypes();
  te->Enumerate(tp);

  argVectorType args = te->getTemplateArgumentsType();
  argVectorType::iterator ait = args.begin();

  if (args.size() == 0) {
    return true;
  }
  if (ait->first == "sc_in")  {
    _inPorts.insert(kvType(fname, te));
  }
  else if (ait->first == "sc_out") {
    _outPorts.insert(kvType(fname, te));
  }
  else if (ait->first == "sc_inout") {
    _inoutPorts.insert(kvType(fname, te));
  }
  return true;
}
```

(b) FindPorts class in systemc-clang.

Figure 4.3: Extraction of SystemC ports by systemc-clang.

transaction object, timing delay, and phase argument is identified. Figure 4.4a and 4.4b show the AST node of the non-blocking forward transport interface and the FindCoreInterface class definition respectively. The arguments to the transport interface method such as the transaction object, timing delay, and protocol phase are extracted for each transport interface method.

4.3.4 Extraction of SystemC processes

SystemC processes are identified by inspecting the constructor of the SystemC module as the SystemC process type and sensitivity list for the process are defined in the constructor of the module. Figure 4.5a shows the partial AST of the D-flip flop SystemC module constructor. It can be observed that the SystemC process dFlipFlop is defined as a SystemC thread process, and it is sensitive to the rising edge of the clock. The constructor of the SystemC module is obtained by traversing AST nodes of CXXMethodDecl and typecasting to CXXConstructorDecl type, which represents a C++ constructor. Since SystemC processes are member functions of the

```
CXXMemberCallExpr 0x5e59168 'enum tlm::tlm_sync_enum'
|-MemberExpr 0x5e59088 '<bound member function type>' ->nb_transport_fw 0x5d473a0
| `-ImplicitCastExpr 0x5e591a8 'class tlm::tlm_fw_nonblocking_transport_if <class tlm::
    tlm_generic_payload, class tlm::tlm_phase>
     *' <UncheckedDerivedToBase (virtual tlm_fw_nonblocking_transport_if)>
|    `-CXXOperatorCallExpr 0x5e59048 'class tlm::tlm_fw_transport_if<struct tlm::tlm_base_protocol_types
    > *'
|      |-ImplicitCastExpr 0x5e59030 'class tlm::tlm_fw_transport_if<struct tlm::tlm_base_protocol_types>
     *(*)(void)' <FunctionToPointerDecay>
|      | `-DeclRefExpr 0x5e59008 'class tlm::tlm_fw_transport_if<struct tlm::tlm_base_protocol_types> *(
    void)' lvalue CXXMethod 0x5dd4ca0
          'operator->' 'class tlm::tlm_fw_transport_if<struct tlm::tlm_base_protocol_types> *(void)'
|      `-ImplicitCastExpr 0x5e58fd0 'class sc_core::sc_port_b<class tlm::tlm_fw_transport_if<struct tlm
    ::tlm_base_protocol_types> >'
        lvalue <UncheckedDerivedToBase (tlm_initiator_socket -> tlm_base_initiator_socket -> sc_port ->
            sc_port_b)>
|        `-MemberExpr 0x5e58fa0 'tlm_utils::simple_initiator_socket<AT_typeB_initiator, 32>':
          'class tlm_utils::simple_initiator_socket<struct AT_typeB_initiator, 32, struct tlm::
             tlm_base_protocol_types>' lvalue ->socket0x5e48940
|          `-CXXThisExpr 0x5e58f88 'struct AT_typeB_initiator *' this
|-UnaryOperator 0x5e590f8 'tlm::tlm_generic_payload':'class tlm::tlm_generic_payload' lvalue prefix '*'
| `-ImplicitCastExpr 0x5e590e0 'tlm::tlm_generic_payload *' <LValueToRValue>
|    `-DeclRefExpr 0x5e590b8 'tlm::tlm_generic_payload *' lvalue Var 0x5e4e6f0 'trans' 'tlm::
    tlm_generic_payload *'
|-DeclRefExpr 0x5e59118 'tlm::tlm_phase':'class tlm::tlm_phase' lvalue Var 0x5e4e7b0 'phase' 'tlm::
    tlm_phase':'class tlm::tlm_phase'
`-DeclRefExpr 0x5e59140 'class sc_core::sc_time' lvalue Var 0x5e4e860 'delay' 'class sc_core::sc_time'
```

(a) SystemC Non-Blocking Transport AST.

```
class FindCoreInterfaceType: public RecursiveASTVisitor<FindCoreInterfaceType>{
  public:
    FindCoreInterfaceType(CXXMethodDecl*, llvm::raw_ostream&);
    ~FindCoreInterfaceType();

     // Typedefs Declarations

     // Virtual methods from RecursiveASTVisitor class
     virtual bool VisitCXXMemberCallExpr (CXXMemberCallExpr*);

     // Member Declarations
   private:
     CXXMethodDecl *_d;
  };
 }
```

```
bool FindCoreInterfaceType::VisitCXXMemberCallExpr(CXXMemberCallExpr *ce) {
  if(ce->getMethodDecl()->getNameAsString() == "b_transport" ||
   ce->getMethodDecl()->getNameAsString() == "get_direct_mem_ptr" ||
   ce->getMethodDecl()->getNameAsString()=="invalidate_direct_mem_ptr") {
    transportType = ce->getMethodDecl()->getNameAsString();
    if(CXXOperatorCallExpr *co = dyn_cast<CXXOperatorCallExpr>
     (ce->getImplicitObjectArgument()->IgnoreImpCasts())) {
      FindArgument fa(co->getArg(0)->IgnoreImpCasts());
      socketName = fa.getArgumentName();
    }
    // Extract payload and delay arguments
    Properties *p = new Properties(transportType, payloadName, delay);
    _socketTransportMap.insert(socketTransportPairType(socketName, p));
  }
  else if (ce->getMethodDecl()->getNameAsString() == "nb_transport_fw" ||
    ce->getMethodDecl()->getNameAsString() == "nb_transport_bw") {

    transportType = ce->getMethodDecl()->getNameAsString();
    if(CXXOperatorCallExpr *co = dyn_cast<CXXOperatorCallExpr>
     (ce->getImplicitObjectArgument()->IgnoreImpCasts())) {
      FindArgument fa(co->getArg(0)->IgnoreImpCasts());
      socketName = fa.getArgumentName();
    }
    //Extract payload, delay, and phase arguments
    Properties *p = new Properties(transportType, payloadName, phase, delay);
    _socketTransportMap.insert(socketTransportPairType(socketName, p));
  }
  return true;
}
```

(b) FindCoreInterface class in systemc-clang.

Figure 4.4: Extraction of TLM-2.0 Core Interfaces by systemc-clang.

```
-CXXMemberCallExpr 0x57df548 'class sc_core::sc_process_handle'
|-MemberExpr 0x57df2c8 '<bound member function type>' ->create_thread_process 0x4d735f0
| `-CallExpr 0x57df2a0 'class sc_core::sc_simcontext *'
|   `-ImplicitCastExpr 0x57df288 'class sc_core::sc_simcontext *(*)(void)'
     <FunctionToPointerDecay>
|     `-DeclRefExpr 0x57df250 'class sc_core::sc_simcontext *(void)' lvalue Function 0x4d9dd00
        'sc_get_curr_simcontext' 'class sc_core::sc_simcontext *(void)'
|-ImplicitCastExpr 0x57df598 'const char *' <ArrayToPointerDecay>
| `-StringLiteral 0x57df2f8 'const char [10]' lvalue "dFlipFlop"
|-CXXBoolLiteralExpr 0x57df330 '_Bool' false

`-CXXOperatorCallExpr 0x57e06a0 'class sc_core::sc_sensitive' lvalue
  |-ImplicitCastExpr 0x57e0688 'class sc_core::sc_sensitive &(*)(class sc_core::sc_event_finder &)'
   <FunctionToPointerDecay>
  | `-DeclRefExpr 0x57e0630 'class sc_core::sc_sensitive &(class sc_core::sc_event_finder &)'
     lvalue CXXMethod 0x4c7c150
     'operator<<' 'class sc_core::sc_sensitive &(class sc_core::sc_event_finder &)'
  |-MemberExpr 0x57e0440 'class sc_core::sc_sensitive' lvalue ->sensitive 0x4e44bd0
  | `-ImplicitCastExpr 0x57e0420 'class sc_core::sc_module *' <UncheckedDerivedToBase (sc_module)>
  |   `-CXXThisExpr 0x57e0408 'struct dff *' this
  `-CXXMemberCallExpr 0x57e04e8 'class sc_core::sc_event_finder' lvalue
    `-MemberExpr 0x57e04b8 '<bound member function type>' .pos 0x4faf000
      `-ImplicitCastExpr 0x57e0510 'const class sc_core::sc_in<_Bool>' lvalue <NoOp>
        `-MemberExpr 0x57e0488 'sc_in<_Bool>':'class sc_core::sc_in<_Bool>' lvalue ->clk 0x57d2930
          `-CXXThisExpr 0x57e0470 'struct dff *' this
```

(a) SC_MODULE constructor AST.

```
class FindEntryFunctions: public RecursiveASTVisitor<FindEntryFunctions> {
 public:
   FindEntryFunctions(CXXRecordDecl* d,llvm::raw_ostream& os);
   ~FindEntryFunctions();
   // Typedefs Declarations

   // Virtual methods from RecursiveASTVisitor class
   virtual bool VisitStringLiteral(StringLiteral *l);
   virtual bool VisitCXXMethodDecl(CXXMethodDecl *d);
   virtual bool VisitMemberExpr(MemberExpr *e);

   // Member Declarations
 private:
   CXXRecordDecl *_d;
   CXXMethodDecl* _entryMethodDecl;
   Stmt* _constructorStmt;
 };
}
```

```
bool FindEntryFunctions::VisitMemberExpr(MemberExpr *e) {
  switch (pass) {
  case 2: {
    string memberName = e->getMemberDecl()->getNameAsString();
    if (memberName == "create_method_process") {
      _procType = METHOD;
    }
    else if (memberName == "create_thread_process") {
      _procType = THREAD;
    }
    else if (memberName == "create_cthread_process") {
      _procType = CTHREAD;
    }
    break;
  }
  default: break;
  }
  return true;
}
```

```
bool FindEntryFunctions::VisitStringLiteral(StringLiteral *s) {
  switch(pass) {
  case 2: {
    _entryName = s->getString();
    ef = new EntryFunctionContainer();
   // Set process name, type, and
   // constructor body for ef
    if(_procType != 0) {
      _entryFunctions.push_back(ef);
    }
    break;
  }
  case 3: {
    break;
  }
  default: break;
  }
  return true;
}
```

```
bool FindEntryFunctions::VisitCXXMethodDecl(CXXMethodDecl *md) {
  _otherFunctions.push_back(md);
  switch(pass) {
  case 1: {
    if (CXXConstructorDecl* cd = dyn_cast<CXXConstructorDecl>(md)) {
      const FunctionDecl* fd = NULL;
      cd->getBody(fd);
      if (cd->hasBody()) {
         _constructorStmt = cd->getBody();
      }
    }
    break;
  }
  case 2: {
    break;
  }
  // Cases to extract additional information
  default: break;
 }
  return true;
}
```

(b) FindEntryFunction class in systemc-clang.

Figure 4.5: Extraction of SystemC processes by systemc-clang.

SystemC class sc_module, the processes are identified by traversing the node of CXXMember-CallExpr and retrieving their member declarators to check whether the process is a SC_METHOD, SC_CTHREAD or SC_THREAD. The process type is identified by retrieving the member declaration of the SystemC process, and comparing with the strings create_thread_process for SC_THREADS, create_method_process for SC_METHOD, and create_cthread_process for SC_CTHREADS. Figure 4.5b shows the FindEntryFunction class of systemc-clang responsible for extracting SystemC processes.

4.3.5 Extraction of TLM-2.0 generic payload attributes

The TLM-2.0 generic payload is a data structure with attributes that reflect memory-mapped bus protocols. It is used as the transaction object for communication between initiators and targets in TLM-2.0 designs. Based on the underlying communication protocol, the initiators and targets modify the attributes of the generic payload. The attributes of the generic payload are set using member functions of the payload class such as set_command() to set read/write command, set_address() to specify the starting address of the target memory map, and set_response_status() to specify the status of the transaction protocol. Figure 4.6a shows the AST node of the generic payload data structure, and Figure 4.6b shows the class definition for the FindPayloadCharacteristics class of systemc-clang. The AST nodes of two member functions of the generic payload class namely set_command and set_address are shown in Figure 4.6a.

4.3.6 Extraction of suspension and notification calls

The calls to SystemC scheduler via wait() and notify() are extracted from each identified SystemC process by traversing AST nodes of type CallExpr, which represents a C/C++ function call. The type of wait() and notify() call such as delta, timed, or delayed are identified by inspecting the arguments passed to the wait() and notify() calls. Figures 4.7a and 4.7b illustrates the AST node description of different wait() and notify() calls and the FindWaitNotify class definition of systemc-clang. The information regarding the wait() and notify() calls along with the sensitivity list extracted are the building blocks towards building the behavioral intermediate representation.

4.3.7 Extraction of netlist

On extracting all the SystemC modules present in the input SystemC model and the architectural components for each module, the top level module that contains the module instantiations and

```
CXXMemberCallExpr 0x4f9ca88 'void'
|-MemberExpr 0x4f9ca30 '<bound member function type>' ->set_command 0x4c36310
| `-ImplicitCastExpr 0x4f9ca18 'tlm::tlm_generic_payload *' <LValueToRValue>
|   `-DeclRefExpr 0x4f9c9f0 'tlm::tlm_generic_payload *' lvalue Var 0x4f9b580
      'trans' 'tlm::tlm_generic_payload *'
`-ImplicitCastExpr 0x4f9cab8 'tlm::tlm_command':'enum tlm::tlm_command' <LValueToRValue>
  `-DeclRefExpr 0x4f9ca60 'tlm::tlm_command':'enum tlm::tlm_command'
    lvalue Var 0x4f9bb10 'cmd' 'tlm::tlm_command':'enum tlm::tlm_command'

CXXMemberCallExpr 0x4f9cb68 'void'
|-MemberExpr 0x4f9cb10 '<bound member function type>' ->set_address 0x4c365d0
| `-ImplicitCastExpr 0x4f9caf8 'tlm::tlm_generic_payload *' <LValueToRValue>
|   `-DeclRefExpr 0x4f9cad0 'tlm::tlm_generic_payload *' lvalue Var 0x4f9b580
      'trans' 'tlm::tlm_generic_payload *'
`-ImplicitCastExpr 0x4f9cbb0 'sc_dt::uint64':'unsigned long long' <IntegralCast>
  `-ImplicitCastExpr 0x4f9cb98 'int' <LValueToRValue>
  `-DeclRefExpr 0x4f9cb40 'int' lvalue Var 0x4f9b910 'adr' 'int'
```

(a) TLM-2.0 generic payload AST.

```
bool FindPayloadCharacteristics::VisitCXXMemberCallExpr(CXXMemberCallExpr *ce) {
  if(MemberExpr *me = dyn_cast<MemberExpr>(ce->getCallee()->IgnoreImpCasts())) {
    if(DeclRefExpr *de = dyn_cast<DeclRefExpr> (me->getBase()->IgnoreImpCasts())) {
      Payload *p = new Payload(_command, _address, _dataPtr, _dataLength,
       _streamingWidth, _byteEnablePtr, _DMIAllowed, _responseStatus);
    }
  }
  if(ce->getMethodDecl()->getNameAsString() == "set_command") {
    FindArgument fa(ce->getArg(0)->IgnoreImpCasts());
    _command = fa.getArgumentName();
  }
  else if (ce->getMethodDecl()->getNameAsString() == "set_address") {
    FindArgument fa(ce->getArg(0)->IgnoreImpCasts());
    _address = fa.getArgumentName();
  }
  // Test for all member functions in the generic payload class
  return true;
}
```

```
class FindPayloadCharacteristics: public RecursiveASTVisitor<FindPayloadCharacteristics> {
  public:

    FindPayloadCharacteristics(CXXMethodDecl*, llvm::raw_ostream&);
    ~FindPayloadCharacteristics();
    // Typedefs Declarations

    // Virtual methods from RecursiveASTVisitor class
    virtual bool VisitCXXMemberCallExpr(CXXMemberCallExpr *ce);

    // Member Declarations
  private:
    CXXMethodDecl* _d;
  };
}
```

(b) FindPayloadCharacteristics class in systemc-clang.

Figure 4.6: Extraction of TLM-2.0 Core generic payload characteristics by systemc-clang.

```
CXXMemberCallExpr 0x424ad60 ’void’
‘-MemberExpr 0x424ad30 ’<bound member function type>’ .notify 0x38d26a0
  ‘-DeclRefExpr 0x424ac90 ’class sc_core::sc_event’ lvalue Var 0x423e620 ’ev’ ’class sc_core::sc_event’

CXXMemberCallExpr 0x424aea0 ’void’
|-MemberExpr 0x424ae70 ’<bound member function type>’ .notify 0x3905f20
| ‘-DeclRefExpr 0x424ad88 ’class sc_core::sc_event’ lvalue Var 0x423e620 ’ev’ ’class sc_core::sc_event’
|-ImplicitCastExpr 0x424aed8 ’double’ <IntegralToFloating>
| ‘-IntegerLiteral 0x424ae28 ’int’ 6
‘-DeclRefExpr 0x424ae48 ’enum sc_core::sc_time_unit’ EnumConstant 0x3797550 ’SC_NS’ ’enum sc_core::
    sc_time_unit’
```

```
CXXMemberCallExpr 0x424b060 ’void’
|-MemberExpr 0x424b030 ’<bound member function type>’ ->wait 0x38ada60
| ‘-ImplicitCastExpr 0x424b098 ’class sc_core::sc_module *’ <UncheckedDerivedToBase (sc_module)>
|   ‘-CXXThisExpr 0x424b018 ’struct dff *’ this
|-ImplicitCastExpr 0x424b0b8 ’double’ <IntegralToFloating>
| ‘-IntegerLiteral 0x424afd0 ’int’ 10
‘-DeclRefExpr 0x424aff0 ’enum sc_core::sc_time_unit’ EnumConstant 0x3797550 ’SC_NS’ ’enum sc_core::
    sc_time_unit’

CXXMemberCallExpr 0x424b3c0 ’void’
|-MemberExpr 0x424b390 ’<bound member function type>’ ->wait 0x38ad460
| ‘-ImplicitCastExpr 0x424b3f0 ’class sc_core::sc_module *’ <UncheckedDerivedToBase (sc_module)>
|   ‘-CXXThisExpr 0x424b378 ’struct dff *’ this
‘-ImplicitCastExpr 0x424b410 ’const class sc_core::sc_event’ lvalue <NoOp>
  ‘-DeclRefExpr 0x424b350 ’class sc_core::sc_event’ lvalue Var 0x423e620 ’ev’ ’class sc_core::sc_event’
```

(a) wait() and notify() calls AST.

```
class FindWaitNotify: public RecursiveASTVisitor<FindWaitNotify> {
  public:

    FindWaitNotify(CXXMethodDecl*, llvm::raw_ostream&);
    ~FindWaitNotify();
    // Typedefs Declarations

    // Virtual methods from RecursiveASTVisitor class
    virtual bool VisitCallExpr(CallExpr* expr);

    // Member Declarations
  private:
    CXXMethodDecl* _entryMethodDecl;
  };
}
```

```
bool FindWaitNoify::VisitCallExpr(CallExpr* e) {
  if (e->getDirectCallee()->getNameInfo().getAsString() == string("wait")) {
    // Extract wait call arguments such as wait duration and event.
  }
  if (e->getDirectCallee()->getNameInfo().getAsString() == string("notify")) { //
    // Extract notify call arguments such as notify duration and event.
  }
  return true;
}
```

(b) FindWaitNotify class in systemc-clang.

Figure 4.7: Extraction of wait() and notify() calls by systemc-clang.

```
CXXOperatorCallExpr 0x41bd110 'void'
|-ImplicitCastExpr 0x41bd0f8 'void (*)(const in_if_type &)' <FunctionToPointerDecay>
| `-DeclRefExpr 0x41bd070 'void (const in_if_type &)' lvalue CXXMethod 0x398a520 'operator()' 'void (
    const in_if_type &)'
|-MemberExpr 0x41bd000 'sc_in<_Bool>':'class sc_core::sc_in<_Bool>' lvalue .clk 0x41ae890
| `-DeclRefExpr 0x41bcfd8 'struct dff' lvalue Var 0x41bcd70 'dff1' 'struct dff'
`-ImplicitCastExpr 0x41bd188 'const in_if_type':'const class sc_core::sc_signal_in_if<_Bool>' lvalue <
    NoOp>
  `-ImplicitCastExpr 0x41bd158 'in_if_type':'class sc_core::sc_signal_in_if<_Bool>' lvalue <
      DerivedToBase (sc_signal -> sc_signal_inout_if -> sc_signal_in_if)>
    `-DeclRefExpr 0x41bd030 'class sc_core::sc_clock' lvalue Var 0x41bc840 'sig_clk' 'class sc_core::
        sc_clock'

`-ImplicitCastExpr 0x41bd2e8 'const in_if_type':'const class sc_core::sc_signal_in_if<_Bool>' lvalue <
    NoOp>
  `-ImplicitCastExpr 0x41bd2c0 'in_if_type':'class sc_core::sc_signal_in_if<_Bool>' lvalue <
      DerivedToBase (sc_signal_inout_if -> sc_signal_in_if)>
    `-DeclRefExpr 0x41bd1f8 'sc_signal<_Bool>':'class sc_core::sc_signal<_Bool, 0>' lvalue Var 0
        x41bcb60 'sig_i_data' 'sc_signal<_Bool>':'class sc_core::sc_signal<_Bool, 0>'

`-ImplicitCastExpr 0x41bd4a8 'class sc_core::sc_signal_inout_if<_Bool>':'class sc_core::
    sc_signal_inout_if<_Bool>' lvalue <DerivedToBase (sc_signal_inout_if)>
  `-DeclRefExpr 0x41bd358 'sc_signal<_Bool>':'class sc_core::sc_signal<_Bool, 0>' lvalue Var 0x41bccc0
      'sig_o_data' 'sc_signal<_Bool>':'class sc_core::sc_signal<_Bool, 0>'
```

(a) Port bindings AST.

```
class FindPortBinding: public RecursiveASTVisitor<FindPortBinding> {
  public:

    FindPortBinding(FunctionDecl*, llvm::raw_ostream&);
    ~FindPortBinding();

    // Typedef Declarations

    // Virtual methods from RecursiveASTVisitor Class
    virtual bool VisitCXXOperatorCallExpr(CXXOperatorCallExpr*);

    // Member Declarations
  private:
};
```

```
bool FindMemberExpr::VisitMemberExpr(MemberExpr* e) {
  DeclarationNameInfo di = e->getMemberNameInfo();
  _name = di.getAsString();
  Expr* base = e->getBase();

  if (DeclRefExpr* de = cast<DeclRefExpr>(base)) {
    DeclarationNameInfo di = de->getNameInfo();
    _moduleInstanceName = di.getAsString();
  }
  return true;
}

bool FindDeclRefExpr::VisitDeclRefExpr(DeclRefExpr* e) {
  DeclarationNameInfo di = e->getNameInfo();
  _name = di.getAsString();
  return true;
}
```

```
bool FindPortBinding::VisitCXXOperatorCallExpr(CXXOperatorCallExpr* e) {
  const Expr* arg0 = e->getArg(0);
  FindMemberExpr me(const_cast<Expr*>(arg0), _os);
  const Expr* arg1 = e->getArg(1);

  FindDeclRefExpr fe(const_cast<Expr*>(arg1), _os);

  PortBindContainer* pb = new PortBindContainer(me.getName(),
                           me.getModuleInstanceName(),
                           fe.getName(), e);
  return true;
}
```

(b) FindPortBinding class in systemc-clang.

Figure 4.8: Extraction of port bindings by systemc-clang.

port connections is identified. This is usually the sc_main() function. The sc_main() function is identified by identifying AST nodes of type FunctionDecl and comparing the name of the function with the string sc_main. Port connections in SystemC can be carried out either by name or by position. For the D Flip-Flop example in Section 2.1.1, dff dff1(clk,i_data,o_data) is an example of a port connection by position wherein the signals clk, i_data, and o_data should match the port declaration order in the module definition. On the other hand, dff1.clk(clk), dff1.i_data(i_data), dff1.o_data(o_data) is an example of a port connection by name. Figures 4.8a and 4.8b show the AST node for named port connections for the D-flip flop example and the class definition of FindPortBindings class of systemc-clang responsible for extracting the netlist. If dynamically instantiated modules are present in the SystemC module, systemc-clang cannot extract the entire module hierarchy as it relies on static analysis for extracting structural and behavioral information.

It is important to note that the time taken for systemc-clang to complete parsing the input SystemC module is proportional to the size of the AST. However, this can be mitigated by taking advantage of clang's precompiled headers that precompiles common set of headers into one precompiled header.

4.4 Intermediate Representation for Structural Information

The intermediate representation (IR) is a collection of classes that stores the parsed information with additional methods to query and display the AST node for the parsed information. It is maintained at three levels: global, module, and process level.

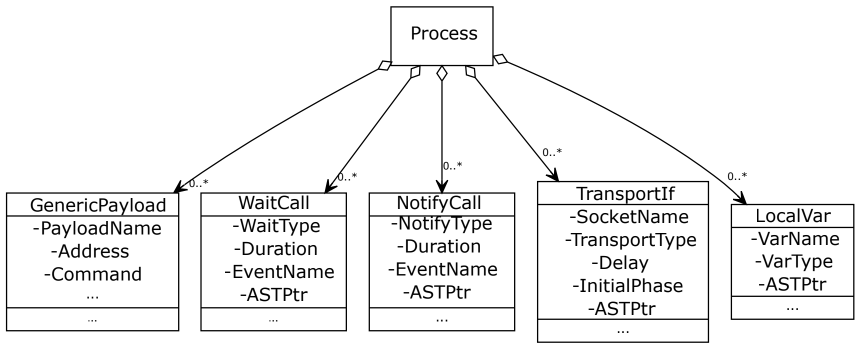


Figure 4.9: IR maintained at the process level [46].

Figure 4.9 shows the class diagram of the IR maintained at the process level. At this level, the sensitivity list, information regarding wait() and notify() calls, generic payload settings, and

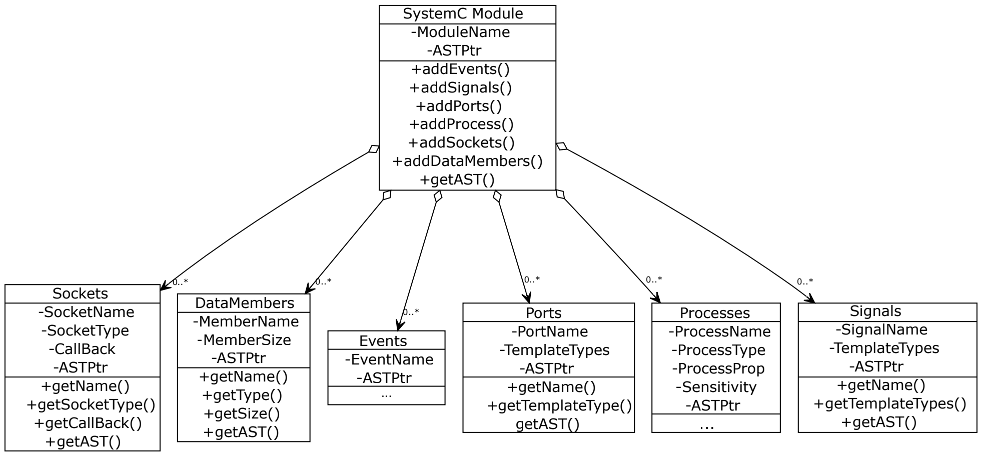


Figure 4.10: IR maintained at the module level [46].

local variables defined in the process are stored. At the module level, information regarding the core interfaces, properties of channels (ports, signals, and sockets) in terms of data types and data widths, processes, data members, and member functions are maintained. This is shown in Figure 4.10.

At the global level, the hierarchy of the design in terms of signal bindings between module instances and the specified simulation time are recorded as shown in Figure 4.11. The key advantage of the IR is that for each information parsed, a pointer to the AST node is also stored, which allows interested developers to perform further analysis of the SystemC model AST through the use of systemc-clang plugins.

4.5 Intermediate Representation for Behavioral Information

In this section, I shall describe the IR maintained for representing the behavioral information of a SystemC model. In the previous sections, I showed how the sensitivity list and calls to wait() and notify() are extracted by systemc-clang for a SystemC module. systemc-clang makes use of this information to construct the IR for behavioral information. It maintains the behavioral IR for each SystemC process present in the SystemC module. Figure 4.12a illustrates a simple SystemC process with some wait() and notify() calls. I use this example to list the transformations for building the behavioral IR.

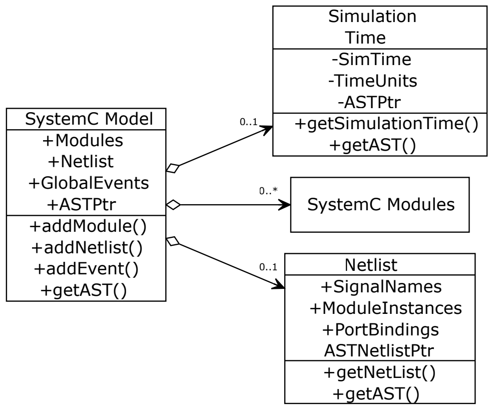


Figure 4.11: IR maintained at the global level [46].

4.5.1 Generation of Suspension-CFG

The first step towards generating the behavioral IR is transforming the control flow graph (CFG) of the SystemC process into a Suspension-CFG. Definition 1 provides a formal definition of a CFG of a program.

Definition 1. *A CFG of a program is a graph representation of the program $G = \langle V, E\rangle$ where the vertices of the graph V denote the basic blocks, and edges E between basic blocks represent the control flow between basic blocks. A basic block is a sequence of code statements with one entry point and one exit point.*

A Suspension-CFG retains the control flow of the CFG, but splits basic blocks containing wait() calls into multiple basic blocks with the wait() call in a separate basic block.

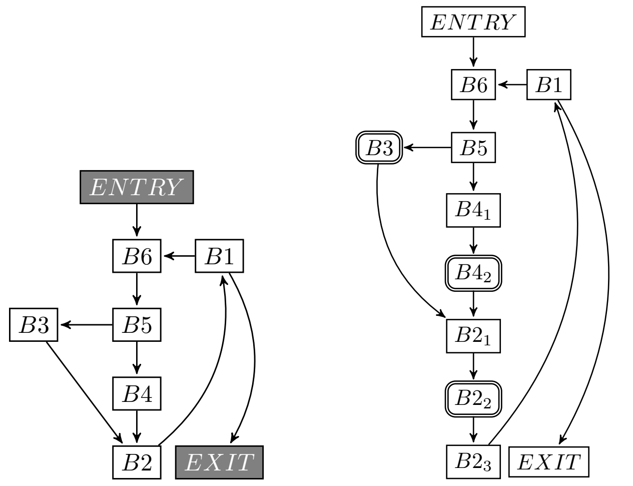
Definition 2 provides a formal definition of Suspension-CFG.

Definition 2. *A* Suspension-CFG *of a SystemC process is a control-flow graph $G = \langle V, E\rangle$ where the basic blocks can be of one of the following types: $T_V = \{ENTRY, EXIT, REG, SUS\}$ such that $\forall v \in V$, if $type(v) = SUS$ then $|S| = 1$ where S represents the sequence of statements present in a basic block of the* Suspension-CFG.

The basic blocks in a CFG are categorized into four groups: $ENTRY$, $EXIT$, SUS, and REG. $ENTRY$ and $EXIT$ blocks correspond to the first and last basic blocks of the CFG respectively. Basic blocks with wait() calls are broken down into a series of REG blocks and SUS blocks. A SUS block contains a single statement, which is the wait() call. The remaining

```
          void thread_process() {
            sc_uint<4> a;
B6,B1:      while(true) {
B5:           a = in.read();
              if(a > 5) {
B4:             out.write(4);
                wait(5, SC_NS);
              }
B3:           else {
                wait(2, SC_NS);
              }
B2:           ev1.notify(3, SC_NS);
              wait(ev2);
              out.write(1);
            }
          }
```

(a) SystemC process example.



(b) Control-Flow-Graph for example.

(c) Suspension-CFG for example.

Figure 4.12: Transformation of CFG to Suspension-CFG.

blocks are categorized as regular blocks or REG blocks. REG blocks include condition statements, loops, and general C/C++ statements. Figure 4.12b and 4.12c shows the construction of the Suspension-CFG of the SystemC process from its CFG. The CFG shown in Figure 4.12b is obtained from parsing the SystemC process shown in Figure 4.12a using clang. In Figure 4.12b, the calls to wait() reside in basic blocks **B4**, **B3**, and **B2**. Therefore, these blocks need to be split such that the wait() call resides in a separate basic block. Algorithm 1 describes the transformation from the CFG to Suspension-CFG. To aid in understanding the algorithm, Table 4.3 tabulates the functions used in Algorithm 1 and their functionality.

Function Name	Input	Output	Description of Functionality
hasWait	Basic block of CFG	{True, False}	Returns $true$ if the basic block of CFG has a wait call.
type	Basic block of CFG	{REG, ENTRY, EXIT}	Returns type of basic block.
splitVertex	Basic block of CFG	Sequence of basic blocks	Splits a basic block with a wait call into sequence of basic blocks.

Table 4.3: Description of functions used in Algorithm 1.

The algorithm begins by picking each vertex v in the CFG and constructing the sets E_{in} and E_{out}, which denote the set of incoming and outgoing edges at v. It then checks whether the vertex has a wait() call via the function $hasWait(v)$. If a wait() call exists, the algorithm

Algorithm 1: Construction of Suspension-CFG from CFG.

```
   /* Initial declarations                                          */
1  Let $G = \langle V_G, E_G \rangle$ be the CFG of the SystemC process.
2  Let $V \leftarrow V_G$ and $E \leftarrow E_G$.
3  foreach $v \in V_G$ do
4      Let $E_{in} \leftarrow \{(v_i, v_j) \in E : v_j = v\}$
5      Let $E_{out} \leftarrow \{(v_i, v_j) \in E : v_i = v\}$
6      if hasWait$(v) \wedge$ type$(v) = REG$ then
7          $[v_1, v_2, \ldots, v_k] \leftarrow$ splitVertex(v)
8          $E_{from} \leftarrow \{(v_i, v_1) : \forall (v_i, v_j) \in E_{in}\}$
9          $E_{split} \leftarrow \{(v_l, v_{l+1}) : \forall l \in [0, k-1]\}$
10         $E_{to} \leftarrow \{(v_k, v_j) : \forall (v_i, v_j) \in E_{out}\}$
11         $V' \leftarrow \{v_l : \forall l \in [0, k-1]\}$
12         $V \leftarrow V \cup V' - \{v\}$
13         $E \leftarrow E_{from} \cup E_{split} \cup E_{to} - E_{in} - E_{out}$
14     end
15 end
16 return $\langle V, E \rangle$
```

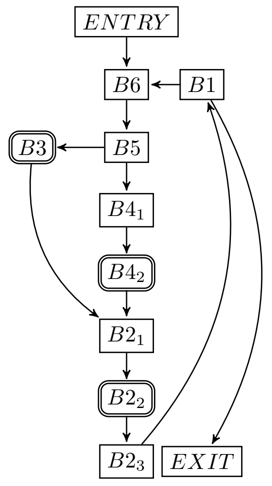
splits the vertex into k vertices. For each wait() call, $splitVertex(v)$ will generate a maximum of three additional vertices to denote the statements executed before the wait() call, statements executed after the wait() call, and the wait() call itself. All previous incoming edges to vertex v are now connected to the first vertex generated by the $splitVertex(v)$ and all outgoing edges are now connected to the last vertex generated by $splitVertex(v)$. Intermediate vertices are linked together as shown in Line 9. The original vertex and its incoming and outgoing edges are substituted by the set of vertices and the new edges formed. For SystemC processes that are SC_METHODS, the Suspension-CFG is the original CFG with an additional vertex that denotes a wait() call based on the sensitivity list. In other words, a SC_METHOD can be modeled as a SC_THREAD that suspends when signals in its sensitivity list do not change.

4.5.2 Generation of Suspension Labeled Transition System

The Suspension-CFG provides a syntactic representation of the SystemC process; however it does not elicit the execution semantics of the SystemC process. In particular, the execution semantics of the SystemC process include the paths formed by the basic blocks that execute between suspension points and the actions that trigger the execution of these paths between suspension points. Representing such semantics allows for better analysis, in particular for determining GPU executable code and easier source-to-source translation. The Suspension Labeled Transition System represents the behavior of SystemC processes, and it is modeled as a labeled transition system (LTS) with states indicating calls to wait() and transitions between states as code that is executed between wait() calls that are triggered based on some actions. Definition 3 formalizes the Suspension Labeled Transition System (SLTS).

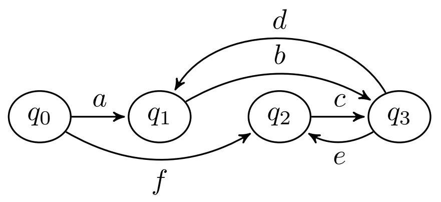
Definition 3. *A suspension labeled transition system (SLTS) of a SystemC process labeled over a finite set of symbols A is a 5-tuple $\langle Q, q_0, L, V, T\rangle$ where Q is the set of suspension states, $q_0 \in Q$ is the initial state, L is the set of actions that determine firing of transitions between suspension states, V is the set of all basic blocks in the* Suspension-CFG, *and T is the set of transitions. A SLTS of a SystemC process is constructed from the* Suspension-CFG *of the SystemC process denoted as $G = \langle V, E\rangle$.*

An action that triggers a transition between two suspension states is a boolean expression where the operators are relation and/or logical operators and the operands in the boolean expression are members of the finite set of symbols A. A transition $t \in T$ is a 4-tuple $\langle q_s, B, g, q_f\rangle$ where $q_s, q_f \in Q$ denote the start and final states of the transition respectively, B is a sequence of basic blocks executed between q_s and q_f denoted as $B = (v_0, v_1, ..., v_n)$ where $\forall 0 \leq i < n$ such that $n \in \mathbb{N}$: $v_i \in V$ and $\langle v_i, v_{i+1}\rangle \in E$, and $g \in L$ is the action that triggers the transition.



$$\begin{tikzcd}
 & ENTRY \arrow[d] & \\
 & B6 \arrow[d] & B1 \arrow[l] \arrow[dddddd, bend left] \\
B3 \arrow[dddr, bend right] & B5 \arrow[l] \arrow[d] & \\
 & B4_1 \arrow[d] & \\
 & B4_2 \arrow[d] & \\
 & B2_1 \arrow[d] & \\
 & B2_2 \arrow[d] & \\
 & B2_3 \arrow[uuuuuur, bend right] & EXIT
\end{tikzcd}$$

(a) Suspension-CFG for example in Figure 4.12a.



$$\begin{tikzcd}
q_0 \arrow[r, "a"] \arrow[rr, bend right, "f"'] & q_1 \arrow[rr, bend left, "b"] & q_2 \arrow[r, "c"] & q_3 \arrow[l, bend left, "e"] \arrow[ll, bend right, "d"']
\end{tikzcd}$$

$a: \quad \langle q_0, [B6, B5, B4_1], true \wedge (a > 5), q_1\rangle$
$b: \quad \langle q_1, [B2_1], true, q_3\rangle$
$c: \quad \langle q_2, [B2_1], true, q_3\rangle$
$d: \quad \langle q_3, [B2_3, B1, B6, B5, B4_1], true \wedge (a > 5), q_1\rangle$
$e: \quad \langle q_3, [B2_3, B1, B6, B5], true \wedge (a \leq 5), q_2\rangle$
$f: \quad \langle q_0, [B6, B5], true \wedge (a \leq 5), q_2\rangle$

(b) Suspension Labeled Transition System constructed from Suspension-CFG.

Figure 4.13: Transformation of Suspension-CFG to Suspension Labeled Transition System.

Figure 4.13b shows the Suspension Labeled Transition System for the example shown in Figure 4.12a. The Suspension Labeled Transition System is built from the corresponding Suspension-CFG generated. Algorithm 2 describes the transformation of the Suspension-CFG to Suspension Labeled Transition System. To aid in understanding Algorithm 2, Table 4.4 tabulates the functions used in Algorithm 2 and their functionality.

Function Name	Input	Output	Description of functionality
getEntryBlock	Suspension-CFG	Basic Block of Suspension-CFG	Returns the entry basic block of Suspension-CFG.
type	Basic block of Suspension-CFG	{REG, SUS, ENTRY, EXIT}	Returns type of basic block.
newState	Basic block of Suspension Labeled Transition System	State of Suspension Labeled Transition System	Creates a state of the Suspension Labeled Transition System.
symbolicPaths	Pair of basic blocks of Suspension-CFG that denote source and destination	Set of paths	Generates a set of all possible paths between source and destination basic blocks.
getState	Basic block of Suspension-CFG	State of Suspension Labeled Transition System	Returns state of Suspension Labeled Transition System.
hasCondition	Basic block of Suspension-CFG	{True, False}	Returns $true$ if basic block has condition for execution.
conjunctionAction	Sub-graph of Suspension-CFG	Conjunction of actions	Conjuncts the actions associated with the execution of the sub-graph.
getPostBlocks	Basic block of Suspension-CFG	Sequence of basic blocks	Returns the sequence of basic blocks executed after input basic block in Suspension-CFG.

Table 4.4: Description of functions used in Algorithm 2.

The input to the algorithm is the Suspension-CFG generated in Algorithm 1. The suspension blocks identified in the Suspension-CFG are first allocated states as shown in Line 3. For the example provided, the suspension blocks $B4_2$, $B3$, and $B2_2$ are annotated with the states q_1, q_2, and q_3 respectively. The initial state of the transition system is q_0. The sets L and T that denote the set of actions encountered and set of transitions between states respectively are initially set to empty. For a pair of suspension points v_1 and v_2, the algorithm first retrieves the set of all possible paths between v_1 and v_2. The problem of path enumeration is a tedious one, as the number of paths in a program increases exponentially in the size of the program. In this thesis, I do not employ heuristics to improve the path enumeration process. Consider the transition from suspension block $B2_2$ to $B4_2$. There exists only one possible path between these two

Algorithm 2: Construction of Suspension Labeled Transition System

```
    /* Initial Declarations                                                    */
 1  Let $v_{entry} \leftarrow \mathsf{getEntryBlock}(G)$
 2  Let $V_{SUS} \leftarrow \{v : \forall v \in V, type(v) = SUS\}$
 3  Let $Q \leftarrow \{q : \forall v \in V_{SUS} \cup \{v_{entry}\}, q \leftarrow \mathsf{newState}(v)\}$
 4  Let $T \leftarrow \emptyset$
 5  forall $v_1, v_2 \in V_{SUS} \cup \{v_{entry}\}$ do
 6      Let $P \leftarrow \mathsf{symbolicPaths}(v_1, v_2)$
 7      Let $q_s \leftarrow \mathsf{getState}(v_1), q_f \leftarrow \mathsf{getState}(v_2)$
 8      Let $B \leftarrow \emptyset$
 9      Let $P \leftarrow \emptyset$
10      forall $p \in P$ do
11          Let $G' \leftarrow \{v : \forall v \in \mathsf{blocks}(p), \mathsf{hasCondition}(v) = true\}$
12          $B \leftarrow \mathsf{getPostBlocks}(v_1)$
13          Let $g \leftarrow \mathsf{conjunctionAction}(G')$
14          $T \leftarrow T \cup \{\langle q_s, B, g, q_f \rangle\}$
15      end
16  end
17  return $\langle Q, q_0, L, V, T \rangle$
```

suspension points of the form [$B2_3$, $B1$, $B6$, $B5$, $B4_1$]. The states associated with the suspension blocks v_1 and v_2 are identified using getState. In this case q_s is q_3 and q_f is q_2. The algorithm begins constructing the Suspension Labeled Transition System by picking each possible path from P, and constructing a set of blocks G' that are predicated by some condition. The conditions across all these conditional blocks in a path are accumulated in the set g. For this transition to happen, the condition in basic block $B5$ should not evaluate to true. In other words, the condition $a > 5$ should evaluate to false for this transition. Therefore, the conjunction of the actions for this transition is $true \wedge (a \leq 5)$. Variables and constants encountered in this transition are accumulated in the set A. The basic blocks executed between the two suspension points in a path can be viewed as the execution of blocks occurring **after** the suspension point v_1 or **before** the suspension point v_2. Since, the algorithm begins with the entry node of the Suspension-CFG, the set B is constructed by inspecting the blocks executed after v_1. In this case the set B for this transition consists of the basic blocks [$B2_3$, $B1$, $B6$, $B4_1$]. The set of transitions is updated with the initial and final states of the transition denoted by q_s and q_f, the conjunction of actions g, and the basic blocks B.

The behavioral IR is key towards analyzing SystemC modules for performance. In the following chapter, I further extend the expressiveness of the Suspension Labeled Transition System to capture details necessary to decide a partitioning of a SystemC model for execution on multi-core CPUs and GPUs.

4.6 systemc-clang Plugins

systemc-clang provides a convenient feature for performing further analysis of SystemC models through plugins. Interested developers can further analyze the SystemC model by writing systemc-clang plugins that interact with the IRs and the information present in them. A systemc-clang plugin is created by inheriting the SystemCConsumer class, which is responsible for carrying out the parsing of the SystemC model and generating the IR. By inheriting the SystemCConsumer class, the plugin has access to the IR, and by overriding a virtual function provided by the parent class, the developer can perform further traversals and analysis on the AST nodes available in the IR. Currently, systemc-clang provides a performance analysis plugin for detecting opportunities for out-of-order execution in SystemC based on the method proposed by Chen et al. [26]. While their work focuses on SpecC, which is another SLDL similar to SystemC, the plugin is based for SystemC designs. This systemc-clang plugin augments the behavioral IR with information regarding the time-stamps at which transitions between suspension points are scheduled by the SystemC scheduler. This information can be easily extracted using the information of the wait() and notify() calls as these scheduler calls influence the simulation time. For my proposed technique, the analysis, partitioning, and translation of SystemC models for execution across multi-core CPUs and GPUs are implemented as systemc-clang plugins.

4.7 Summary of systemc-clang

systemc-clang is an open-source static framework for parsing and analyzing SystemC models defined at both the RTL and TL abstraction levels. It is based on the clang/LLVM framework. It extracts both the structural and behavioral information of an input SystemC model and stores the information extracted in appropriate IRs. In order to facilitate the interaction with the IRs for further analysis, systemc-clang provides a convenient feature in the form of systemc-clang plugins to query the IRs. systemc-clang is available for download and development at http://anikau31.github.io/systemc-clang/.

Chapter 5

Accelerating Simulation of Mixed-Abstraction SystemC Models on Multi-Core CPUs and GPUs

In this chapter, I present a technique for accelerating mixed-abstraction SystemC simulations across multi-core CPUs and GPUs. The proposed technique is implemented as systemc-clang plugins SCuitable and CUDA-Gen that generates a suitable mapping of SystemC processes for execution across multiple CPUs and GPUs, and translates the GPU identified portions to CUDA respectively.

5.1 SCuitable: A systemc-clang plugin for generating a mapping of a SystemC model for execution on multi-core CPUs and GPUs

SCuitable is a systemc-clang plugin that identifies a partitioning of a SystemC model for execution across multiple CPUs and GPUs. Since SCuitable is a systemc-clang plugin, it has access to the behavioral and structural IRs maintained by systemc-clang. In order to analyze the SystemC model for performance evaluation, SCuitable extends the behavioral IR maintained by systemc-clang. These extensions are described in the following sections.

5.1.1 Extensions to the Suspension Labeled Transition System

Previous efforts by Nanjundappa et al. [15] and Vinco et al. [16] utilized the GPU to accelerate SystemC simulations by executing concurrent SystemC processes on distinct threads and distinct dataflows on distinct thread-blocks, respectively. These approaches scale well with the number of concurrent processes due to the presence of a large number of compute cores on the GPU. However, the execution time of a program on the GPU is dependent on the program's control flow characteristics. For instance, a program that exhibits intensive control flow divergence or contains data dependent computations would perform slower on the GPU than on a single general purpose core. This is because a GPU core is a simple in-order core devoid of any sophisticated micro-architectural features such as branch predictors and re-order buffers otherwise present in general cores. Therefore, utilizing the GPU for executing data-parallel computations and multi-core CPUs for task parallelism is more beneficial over utilizing the GPU for just task parallelism as it addresses the execution of concurrent processes by issuing them on on multiple general purpose cores and execution of large data-parallel computations on the GPU. However, GPUs are shared resources and are treated as I/O devices. Since there could be a number of concurrent data-parallel portions suitable for GPU execution, choosing a subset of these data-parallel portions to execute on the GPU is critical towards improving the simulation time. This is because for a computing system with a single GPU, data-parallel computations scheduled for execution by concurrent CPU threads are executed serially on the shared GPU [1]. Hence, there is a need to identify transitions across all SystemC processes present in the input SystemC model that overlap in simulation time and identify the data-parallel computations contesting for the shared GPU resource across these overlapping transitions. Therefore, the extensions to the behavioral IR augment the transitions with the earliest possible time-stamp that the SystemC scheduler can schedule the transition for execution, and annotate the transitions with data-parallel computations.

Augmenting the Labeled Transition Suspension System with timing details

For a SystemC model with static lengths to wait() and notify() calls, it is possible to determine the earliest time-stamps at which the SystemC scheduler can schedule the transitions. This is because wait() calls are the only scheduler calls that advance simulation time. The time-stamp at which a transition is scheduled is denoted as a tuple of the form $t_{s,d} = \langle s, d \rangle$ where s denotes the

[1]The new Kepler GPUs from NVIDIA have more than one connection to the shared GPUs allowing simultaneous kernels invoked from multiple simultaneous CPUs to execute concurrently on the GPU. However, the experimentation and algorithms proposed in this thesis are for the Tesla generation of NVIDIA GPUs and applicable to the GPU generations before the Tesla generation.

simulation time and d denotes the delta-cycle at which transition is scheduled by the SystemC scheduler. Table 5.1 tabulates the different SystemC wait() calls and their effect on the simulation time.

Wait Type	Simulation Time Increment
wait(k, UNITS)	$\langle s+k, d\rangle$
wait(SC_ZERO_TIME)	$\langle s, d+1\rangle$
wait(event)	$\langle k, d\rangle$ for immediate notifications (event.notify()) $\langle s+j, d\rangle$ for timed notifications (event.notify(j, SC_NS)) $\langle s, d+1\rangle$ for delta notifications (event.notify(SC_ZERO_TIME))

Table 5.1: SystemC wait() calls, and effect on simulation time.

For a single SystemC thread process, transitions following a timed wait of the form wait(k, UNITS) will be scheduled at $s+k$ simulation cycles where s is the current simulation time prior to the wait() call. For an event based wait() call of the form wait(e), where e is an sc_event, the simulation time advance is determined by the event notification. For an immediate notification, transitions following the wait(e) will be scheduled by the scheduler at simulation time-stamp k where k is the time at which the transition containing the event notification is scheduled. For a timed notification, transitions following the wait(e) will be scheduled at simulation time-stamp $s+j$, where j denotes the time argument of the delayed notification, and s denotes the current simulation time. Transitions following delta wait() calls of the form wait(SC_ZERO_TIME), are resumed in subsequent delta-cycles that do not advance the simulation time.

Since the SystemC scheduler is responsible for synchronization across all processes, the ordering of wait() and notify() calls is important to avoid any livelock and deadlock situations. For example, Figure 5.1a shows a SystemC model with two threads with some calls to wait() and notify(). The simulation of such a model does not terminate because process p1 waits on an event that is never notified. Therefore, this process continues to wait on event $e1$, and therefore the instruction to stop the simulation sc_stop() is never executed. Figure 5.1b shows a SystemC process that terminates but functions incorrectly by prematurely ending the simulation as the SystemC scheduler cannot determine the next time advance. This is because both processes wait on events for which the notifications appear after the suspensions. Therefore, I focus on accelerating SystemC simulations that are functionally and temporally correct.

Algorithm 3 describes the algorithm for generating the Global Suspension Labeled Transition System. A Global Suspension Labeled Transition System binds all the Suspension Labeled Transition Systems present in a SystemC model, and annotates each transition across all Suspension Labeled Transition Systems with the earliest time the SystemC scheduler can schedule

```
#include "systemc.h"

SC_MODULE(module) {

  sc_event e1, e2;

  SC_CTOR(module) {
    SC_THREAD(p1);
    SC_THREAD(p2);
  }

  void p1() {
    while(true) {
      wait(SC_ZERO_TIME);
      wait(1, SC_NS);
      e2.notify();
      wait(5, SC_NS);
    }
  }

  void p2() {
    while(true) {
      wait(e1);
      wait(10, SC_NS);
      sc_stop();
    }
  }
};

int sc_main(int argc, char *argv[]) {

  module m("m");
  sc_start();

  return 0;
}
```

(a) SystemC livelock example.

```
#include "systemc.h"

SC_MODULE(module) {

  sc_event e1, e2;

  SC_CTOR(module) {
    SC_THREAD(p1);
    SC_THREAD(p2);
  }

  void p1() {
    while(true) {
      wait(SC_ZERO_TIME);
      wait(e1);
      e2.notify();
      wait(5, SC_NS);
    }
  }

  void p2() {
    while(true) {
      wait(e2);
      wait(10, SC_NS);
      e1.notify();
    }
  }
};

int sc_main(int argc, char *argv[]) {

  module m("m");
  sc_start();

  return 0;
}
```

(b) SystemC deadlock example.

Figure 5.1: Examples of incorrect SystemC code.

Algorithm 3: Construction of Global Suspension Labeled Transition System

```
/* Initial declarations                                                    */
Let $Q \leftarrow \emptyset, V \leftarrow \emptyset, EL \leftarrow \emptyset$
foreach $s \in S$ do
    foreach $t \in T$ do
        $t_{s,d} = \langle \infty, \infty \rangle$
    end
end
$Enqueue(Q', q_0), V \leftarrow V \cup \{q_0\}$
while $Q \neq \emptyset$ do
    $q \leftarrow Dequeue(Q')$
    foreach $s \in S$ do
        foreach $t \in T$ such that $q_s = q$ do
            Let $T'$ be the set of transitions such that $q_f = q$.
            Apply $f(q, t_{s,d}, T')$
            if $q_f \notin V$ then
                $Enqueue(Q', q_f), V \leftarrow V \cup \{q_f\}$
            end
            if $isEvent(q_s)$ then
                if $t(e) \in EL$ then
                    $\langle i, j \rangle = EL(e), t_{s,d} = \langle i, j \rangle$
                    if $q_f \notin V$ then
                        $Enqueue(Q', q_f), V \leftarrow V \cup \{q_f\}$
                    end
                end
                else
                    $Enqueue(Q', q)$
                end
            end
            if $t$ contains an event notification then
                $EL \leftarrow EL \cup \langle e, t_{s+k,d} \rangle$
            end
        end
    end
end
```

it. The methodology for determining the earliest time-stamps for execution is based on the technique proposed by Chen et al. [26] for SpecC designs. The algorithm uses a modified breadth-first search algorithm to update the transitions with timing information. It maintains a queue Q' that queues the states traversed in the Global Suspension Labeled Transition System, and the set V that holds all the visited states. It also maintains a set EL that holds the events notified in a transition along with the earliest time-stamp possible for the event notifications. The algorithm begins by first setting the time tuples for each transition across all Suspension Labeled Transition Systems present in the SystemC model to $\langle\infty,\infty\rangle$. Since all SystemC processes are placed in the ready-to-run queue by the SystemC scheduler at the initialization phase, I use q_0 to collectively denote the initial states for all SystemC processes. This initial state is the first state enqueued into Q'. The head of Q' is dequeued and compared with the starting state q_s of transitions across all Suspension Labeled Transition Systems. The state dequeued is denoted as q. A set T' is maintained by the algorithm that holds all the transitions that have q as the final state. In other words T' holds the set of transitions that are incoming to the state q. For the identified transitions and the type of wait() call, the function $TimeAdvance$ is applied, whose definition is given below.

$$f(q, t_{s,d}, T') = \begin{cases} \langle 0,0\rangle & : IsInitial(q) = True \\ min(t'_{s,d} \in T') + \langle k,0\rangle & : IsTimed(q) = True \\ min(t'_{s,d} \in T') + \langle 0,1\rangle & : IsDelta(q) = True \\ t_{s,d} & : otherwise \end{cases}$$

The functions $IsInitial(q)$, $IsTimed(q)$, and $IsDelta(q)$ return the type of wait() call of the suspension state q.

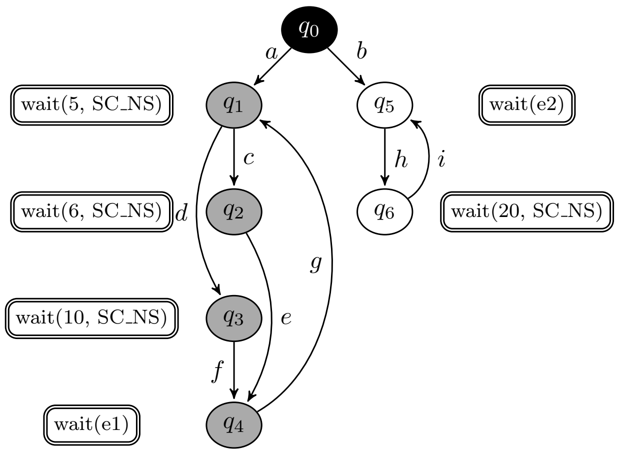
Figure 5.2 shows two SystemC processes and the Global Suspension Labeled Transition System assuming that these are the only two SystemC processes present in the SystemC model. The initial state q_0 is marked in black, the shaded states represent the states of SystemC process p1 and the unshaded states represent the states of SystemC process p2. The states are annotated with the wait() calls. When the state dequeued is the initial state q_0, the transitions exiting from this state are executed first and therefore, their $t_{s,d}$ is set to $\langle 0,0\rangle$. The transitions a and b from the initial state q_0 are annotated with time-stamp $\langle 0,0\rangle$. For timed wait() calls, $t_{s,d}$ is determined by adding k to the minimum time-stamp among all incoming transitions, where k is the duration of the wait() call. Since, SystemC provides a range of time resolutions, appropriate conversions between resolutions are carried out, which are not shown in the algorithm. In Figure 5.2, the state q_1 is a timed wait() call of duration $\langle 5, SC\_NS\rangle$. Since there is only one transition with $q_f = q_1$, the earliest time at which transitions c and d are scheduled is at 5 SC_NS.

For delta wait() calls of the form wait(SC_ZERO_TIME), the delta-cycle of the current simulation time is incremented by one. On calculating the earliest time at which the transition is scheduled by the SystemC scheduler, the state q_f, which is the final state of the transition, is

```
void p1() {
  while(true) {
    // Code Block 1
    wait(5, SC_NS);
    e2.notify(4, SC_NS);
    if(condition) {
      wait(6, SC_NS);
      // Code Block 2
    }
    else {
      wait(10, SC_NS);
      //Code Block 3
    }
    wait(e1);
    //Code Block 4
  }
}

void p2() {
  while(true) {
    wait(e2);
    //Code Block 5
    wait(20, SC_NS);
    //Code Block 6
    e1.notify();
    //Code Block 7
  }
}
```

(a) Example of 2 SystemC threads.



$a:$	$\langle q_0, ..., q_1, \langle 0,0\rangle\rangle$
$b:$	$\langle q_1, ..., q_5, \langle 0,0\rangle\rangle$
$c:$	$\langle q_1, ..., q_2, \langle 5,0\rangle\rangle$
$d:$	$\langle q_1, ..., q_3, \langle 5,0\rangle\rangle$
$e:$	$\langle q_2, ..., q_4, \langle 11,0\rangle\rangle$
$f:$	$\langle q_3, ..., q_4, \langle 15,0\rangle\rangle$
$g:$	$\langle q_4, ..., q_1, \langle 29,0\rangle\rangle$
$h:$	$\langle q_5, ..., q_6, \langle 9,0\rangle\rangle$
$i:$	$\langle q_6, ..., q_5, \langle 29,0\rangle\rangle$

(b) Global Suspension Labeled Transition System for the example.

Figure 5.2: Examples of SystemC threads and their corresponding timed Suspension Labeled Transition Systems.

enqueued into Q' and added into the set V. Notice that the piecewise function $f(q, t_{s,d})$ does not have handle the case of event based wait() calls. This is because the transition following an event wait() call is dependent upon the notification of the event. Therefore, for event wait() calls, a check for the event in the event list EL is done to extract the earliest time the notification for the event is scheduled. The event list EL holds the earliest notification timestamps of events as shown in lines 28-29. For immediate notifications of the form e.notify(), the event list is updated with the event and the current time-stamp of the transition. In other words, the k added to the simulation time is 0. For delayed notifications of the form e.notify(k, UNITS), the event list is updated with the event and current time-stamp of transition advanced by k units. The delayed notification on event $e2$ in Line 5 of Figure 5.2a is updated in the event list EL with the entry $\langle e2, 9\ SC\_NS\rangle$ as the transition that executes the notification is scheduled at $5\ SC\_NS$ and the delayed notification is scheduled $4\ SC\_NS$ later. Therefore, the transition h from q_s, which is waiting on event $e2$, is scheduled at $9\ SC\_NS$. If the event entry for an event based wait() call is not available in the event lit EL, the queue Q' is not updated with the final state for that transition. Instead, the same state is enqueued into the queue. For the state q_4 that waits on event $e1$, the time at which the transition g is scheduled cannot be determined without determining the scheduling time of transition i as this transition executes the notification of event $e2$. Therefore, in the execution of the algorithm for such an example, the algorithm will enqueue and dequeue state q_4 till the transition i updates the event list EL with the scheduling time of the event notification to event $e2$. The algorithm iterates till the queue Q' is empty.

Augmenting the Global Suspension Labeled Transition System with data-parallel computations

In order to identify the data-parallel segments that can be executed on the GPU, SCuitable requires the user to annotate the data-parallel segments. With the help of Polly [50], a clang tool for polyhedral optimizations, this can be automated as well; however, I intent to integrate this with SCuitable as future work. The identification of the data-parallel segments is carried out by annotating them with a C++ macro of the form GPU_EXEC (T, S) where T denotes the execution time of the data-parallel computation on a general purpose core, and S is the speedup of this data-parallel computation on the GPU. For instance, a macro of the form GPU_EXEC(54, 5) conveys that the data-parallel execution takes 54 ms to compute, and that the GPU execution of the same is 5 times faster. that the speedup is intended to convey the speedup of the computation on the GPU, and does not take into account the time taken for transferring the data between CPU and GPU. The data-parallel segments of a transition are a subset of the basic blocks of execution in the transition. For the identified data-parallel computations, SCuitable extracts the data-type and size of the variables involved. It calculates a measure of the time taken to transfer the data between the CPU and GPU by analyzing the variables read/written to in the data-parallel computation. Currently, SCuitable uses the data transfer latency measures for the NVIDIA C2075 Tesla GPU. However, these measures can be configurable to suit other platforms.

Currently, CUDA kernels executing on the GPU do not possess the ability to suspend and resume their execution. Therefore, in this approach, synchronization of SystemC processes using wait() and notify() calls are carried out by the SystemC scheduler residing on the CPU. Hence, a data-parallel execution marked for GPU execution with wait() and notify() calls must transfer control back to the CPU on encountering a scheduler call. To resume execution on the GPU after the scheduler call, the state of the GPU computation has to be transferred to the CPU prior to the scheduler call and transferred back to the GPU after the scheduler call. Therefore, the calculation of the data transfer overhead between the CPU and GPU for a data-parallel computation should take into account the effect of scheduler calls present in the data-parallel computation.

```
for (unsigned int i = 0; i<100; i++) {
  // Code Block 1
  for (unsigned int j = 0; j<100; j++) {
    // Code Block 2
    wait(1, SC_NS);
  }
}
```

(a) Code snippet with wait() call in inner loop.

```
for (unsigned int i = 0; i<100; i++) {
  // Code Block 1
  for (unsigned int j = 0; j<100; j++) {
    // Code Block 2
  }
  wait(1, SC_NS);
}
```

(b) Code snippet with wait() call in outer loop.

Figure 5.3: Position of wait() calls in loops.

Figure 5.3 illustrates two situations in which the wait() call is positioned at different loop levels. In Figure 5.3b, the wait() call is executed 100 times as the outer loop bound is 100. Therefore, to measure the latency of data transfer for the computation, the data transfer overhead between the CPU and GPU is scaled by 100. In Figure 5.3a, the wait() call is positioned in the inner loop. Since the inner loop is executed for a total of $100 * 100 = 10000$, the data transfer latency between the CPU and GPU is multiplied by 10000. The data transfer overhead and the execution time of all the data-parallel computations on the GPU determines the mapping of the data-parallel computations across multi-core CPUs and GPUs. To identify which data-parallel segment executes on the GPU, a boolean flag $isGPU$ is set to True/False for each data-parallel computation. When $isGPU$ is set to $True$, then the corresponding data-parallel segment is translated for execution on the GPU. The boolean flag is set by the $GPUMAP$ algorithm discussed in the Section 5.1.4. The definition for the Global Suspension Labeled Transition System for a SystemC model is provided in Definition 4.

Definition 4. *Let p_1 and p_2 be the SystemC processes present in a SystemC model. Let $SLTS_1 = \langle Q_1, q_{01}, L_1, V_1, T_1 \rangle$ be the SLTS that represents SystemC process p_1 and $SLTS_2 = \langle Q_2, q_{02}, L_2, V_2, T_2 \rangle$*

be the SLTS that represents SystemC process p_2. The Global Suspension Labeled Transition System *(GSLTS) of the SystemC model is given by the parallel composition $GSLTS = SLTS_1 || SLTS_2 = \langle Q, q_0, L, V, T \rangle$ where:*

> *$Q = Q'_1 \times Q'_2 \cup \{q_0\}$ is the set of suspension states such that $Q'_1 = Q_1 - \{q_{01}\}$ and $Q'_2 = Q_2 - \{q_{02}\}$.*
>
> *q_0 is the initial state.*
>
> *$L = L_1 \cup L_2$ is the set of actions that triggers transitions between suspension states.*
>
> *$V = V_1 \cup V_2$ is the set of basic blocks representing the SystemC model.*
>
> *$T = T'_1 \cup T'_2 \cup T'_0$ is the set of transitions such that $T'_1 = T_1 - T_{01}$ and $T'_2 = T_2 - T_{02}$. T_{0i} is the set of all transitions with start state q_{0i}. More formally, T_{0i} is denoted as $\{t = \langle q_{si}, B_i, g_i, q_{fi} \rangle \in T_i \mid q_{si} = q_{0i}\}$. Therefore, T'_i denotes the set of transitions that do not have q_{0i} as the initial state. T'_0 is the set of initial transitions with initial states $q_s = q_0$. More formally, T'_0 is denoted as $\{t'_0 \mid$ Let $t_i \in T_{0i}$ such that $t_i = \langle q_{si}, B_i, g_i, q_{fi} \rangle$, then $t'_{0i} = \langle q_0, B_i, g_i, q_{fi} \rangle\}$.*

A transition $t \in T$ is a 5 tuple $\langle q_s, \langle B, isGPU \rangle, g, t_{s,d}, q_f \rangle$ where q_s, B, g, and q_f have the same meaning introduced in Definition 3. $isGPU$ is a boolean flag that indicates whether the basic block B is identified for GPU execution, and $t_{s,d}$ denotes the earliest time-stamp at which the transition can be scheduled for execution by the SystemC scheduler. $t_{s,d}$ is represented as a tuple of the form $t_{s,d} = \langle s, d \rangle$ where s represents the simulation time and d denotes the delta cycle.

5.1.2 Assumptions and Requirements for SCuitable

Before describing the partitioning algorithm for executing a SystemC model across multiple-core CPUs and GPUs, I list down two assumptions and requirements necessary for SCuitable to generate a mapping.

- **Calls to wait() and notify() should be static.**
  By knowing the duration of wait() and notify() calls, the Global Suspension Labeled Transition System can be constructed, and therefore, overlapping data-parallel computations

contending for the shared GPU can be identified.

- **Static loop bounds for all data-parallel computations.**
  Loop bounds for all data-parallel computation should be static in order to calculate the effect of data transfer overhead between the CPU and GPU towards the total execution time.

5.1.3 Partitioning a SystemC model for execution on multi-core CPUs and GPUs

In the previous sections, I described the construction of the Global Suspension Labeled Transition System necessary for the performance analysis of SystemC models. These modifications are key in identifying overlapping transitions, and the data-parallel segments available for GPU execution in these overlapping transitions. Since the GPU is a shared resource, data-parallel segments scheduled simultaneously for execution on the GPU by multiple CPU threads results in serialized execution on the GPU. Therefore, the identification of a subset of concurrent data-parallel computations for GPU execution that balances the overhead of serial execution and the benefits of fast kernel execution on the GPU is necessary for realizing simulation gains using GPUs. The identification of the best selection of data-parallel segments to execute on the GPU can be considered as a 0-1 knapsack problem. For heterogeneous computing systems with more than one GPU, the identification process can be extended to a multiple 0-1 knapsack problem. In this work, I consider heterogeneous computing systems with single GPU systems. Before discussing the proposed technique for determining a partitioning of a SystemC module for execution on multi-core CPUs and GPUs, I provide a brief overview of the proposed technique.

Overview

Figure 5.4 provides an overview of the proposed parallel co-simulation of SystemC mixed-abstraction models across multi-core CPUs and GPUs. The co-simulation methodology can be divided into two phases: the analysis stage, and the translation and co-simulation stage.

Figure 5.5 illustrates the analysis stage of the proposed methodology. The input SystemC model consists of six SystemC processes labeled as sc_process a, sc_process b, sc_process c, sc_process d, and sc_process e. For the input SystemC model, the AST of the model is extracted using the clang/LLVM framework. systemc-clang extracts the necessary structural information

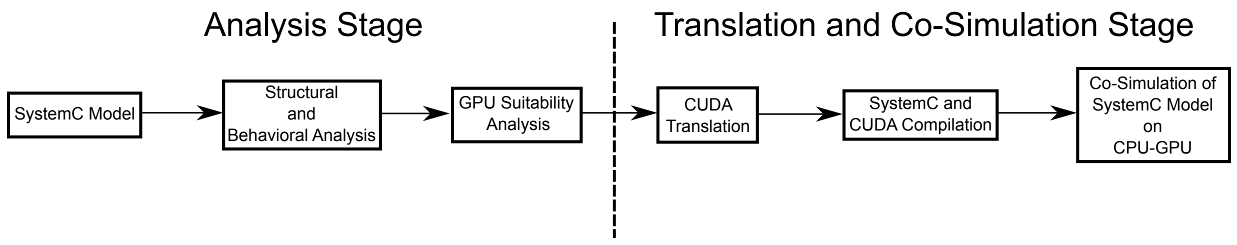

Analysis Stage
Translation and Co-Simulation Stage
SystemC Model
Structural and Behavioral Analysis
GPU Suitability Analysis
CUDA Translation
SystemC and CUDA Compilation
Co-Simulation of SystemC Model on CPU-GPU

Figure 5.4: Methodology for accelerating mixed-abstraction SystemC models.

and generates the structural and behavioral IRs. Using the behavioral IR, the systemc-clang plugin SCuitable generates a Global Suspension Labeled Transition System with the transitions augmented with data-parallel segments and earliest time-stamps that the SystemC scheduler can schedule them. Based on a set of rules and constraints, the data-parallel segments are analyzed for GPU execution benefit, and marked for GPU execution. The set of rules and constraints for deciding which data-parallel segments are executed on the GPU are described in Section 5.1.4. For the SystemC model example provided in Figure 5.5, assume that the SystemC processes sc_process a and sc_process b contain data-parallel segments suitable for GPU execution.

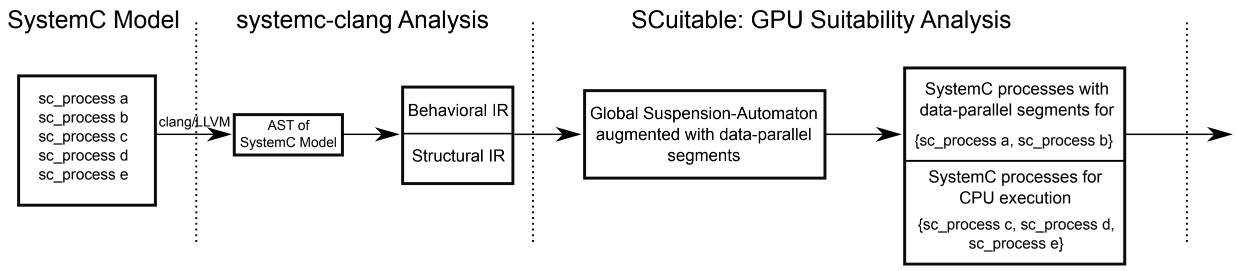

SystemC Model
systemc-clang Analysis
SCuitable: GPU Suitability Analysis
sc_process a
sc_process b
sc_process c
sc_process d
sc_process e
clang/LLVM
AST of SystemC Model
Behavioral IR
Structural IR
Global Suspension-Automaton augmented with data-parallel segments
SystemC processes with data-parallel segments for
{sc_process a, sc_process b}
SystemC processes for CPU execution
{sc_process c, sc_process d, sc_process e}

Figure 5.5: Analysis stage of proposed methodology.

Figure 5.6 illustrates the translation and co-simulation stage of the proposed methodology. For SystemC processes with data-parallel segments marked for GPU execution in the analysis stage, CUDA kernels for the identified data-parallel segments are generated by the systemc-clang plugin CUDA-Gen. The algorithm for CUDA-Gen is described in Section 5.2. Following the translation stage, the compilation stage compiles the SystemC model with the nvcc and g++ compilers to generate a single CPU-GPU binary that is executed on the heterogeneous platform. I use the parallel SystemC library developed by Sinha et al. [17] for compiling the SystemC model that launches ready-to-run processes on available multiple CPU cores present in the heterogeneous platform. The specifics of the parallel SystemC library is decribed in the next section.

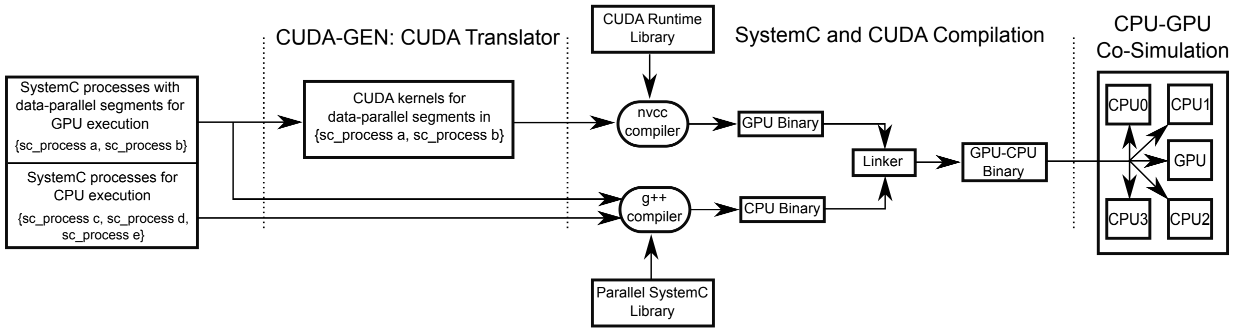

CUDA-GEN: CUDA Translator
CUDA Runtime Library
SystemC and CUDA Compilation
CPU-GPU Co-Simulation
SystemC processes with data-parallel segments for GPU execution
{sc_process a, sc_process b}
SystemC processes for CPU execution
{sc_process c, sc_process d, sc_process e}
CUDA kernels for data-parallel segments in {sc_process a, sc_process b}
nvcc compiler
g++ compiler
GPU Binary
CPU Binary
Linker
GPU-CPU Binary
CPU0
CPU1
GPU
CPU3
CPU2
Parallel SystemC Library

Figure 5.6: Translation and co-simulation stage of proposed methodology.

Parallel SystemC Scheduler

The sequential SystemC scheduler is parallelized similar to the one proposed by Schumaker et al. [5], with support for immediate and delayed wait() and notify() calls. The reference SystemC scheduler maintains a queue of ready-to-run processes. Processes from this queue can be picked by the SystemC scheduler in any order for execution. Since these processes are ready-to-run in the current simulation time, with the availability of multiple cores, these processes can be executed on different cores. Therefore, the parallel SystemC scheduler spawns ready-to-run processes on multiple cores. Figure 5.7 illustrates the parallel SystemC scheduler used. In order to maintain consistent behavior of the channels shared between modules, the evaluate-update phase of the parallel SystemC scheduler enforces a barrier synchronization, resulting in a serialized update to the channels. To support TL abstraction models, the parallel SystemC scheduler is extended to support immediate notifications. Processes activated by immediate notifications are scheduled to run in the same delta-cycle by the SystemC scheduler. Therefore, these new ready-to-run processes have to be added to the ready-to-run queue. The event notifications by processes are added to the shared event queue. To prevent any race conditions on the shared event queue, the parallel SystemC scheduler uses shared locks. Processes activated by immediate notifications are then executed in parallel on the available multiple cores. The parallel SystemC scheduler undergoes multiple iterations of executing ready-to-run processes on the available multiple cores until there are no more ready-to-run processes in the current delta-cycle. From here on, the parallel SystemC scheduler proceeds according to the reference implementation.

Simulation of data-parallel segments on the GPU

The identified data-parallel segments are translated into CUDA kernels for GPU execution. The execution of a CUDA kernel currently does not have the ability to suspend its execution and

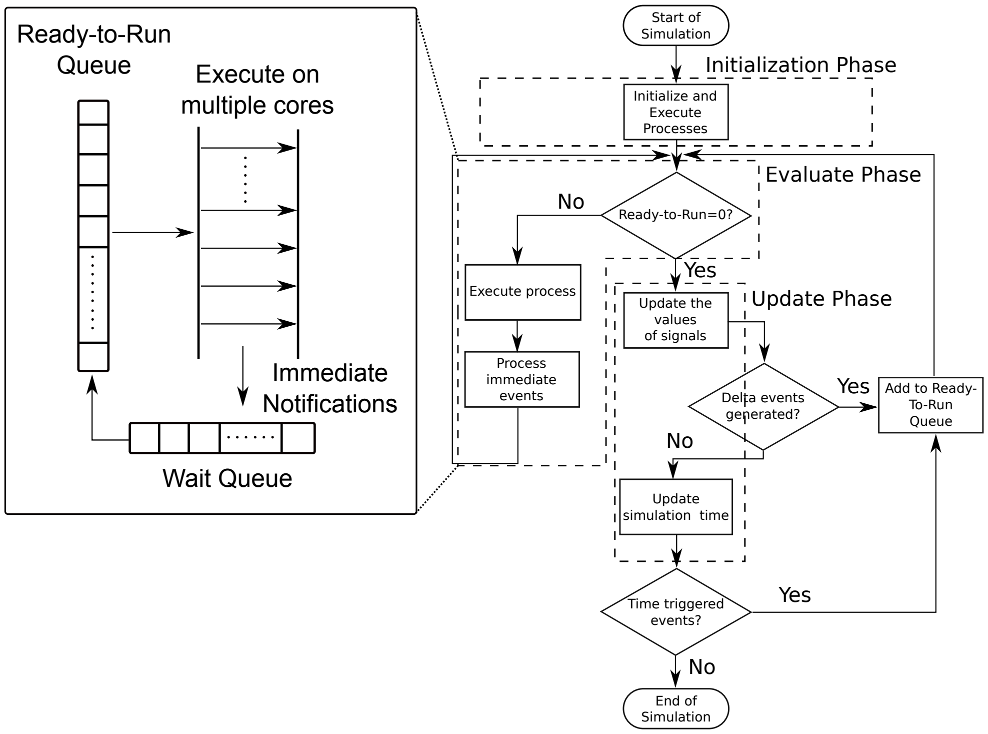


Figure 5.7: Parallel SystemC scheduler.

resume from the next instruction. Therefore, if the data-parallel segments contains wait() or notify() calls, the CUDA kernel transfers control back to the CPU as all the event scheduling and synchronization across concurrent SystemC processes are carried out by the SystemC scheduler, which executes on the CPU. CUDA kernels are asynchronously launched on the GPU. In other words, after a CUDA kernel is launched for execution on the GPU, control is immediately returned back to the CPU. However, in order to maintain consistent evaluation of channels across all concurrent SystemC processes, the kernel calls are made blocking such that control returns to the CPU only after the kernel has completed its operation. Hence, for a transition with a data-parallel segment marked for GPU execution, the suspension of the SystemC process is scheduled by the SystemC scheduler only after all computation in the transition is completed.

5.1.4 Problem Statement

The load of a processor at time-stamp $t_{s,d}$ can be defined as the number of data-parallel computations assigned to it. More formally, let D_i be the set of data-parallel computations assigned

Variable	Description
$c_{max} \in \mathbb{R}^+$	Maximum makespan value of concurrent data-parallel segments at time-stamp $t_{s,d}$.
$m \in \mathbb{N}$	Number of CPUs present in the system.
$n \in \mathbb{N}$	Number of GPUs present in the system.
$d \in \mathbb{N}$	Number of concurrent data-parallel segments at time-stamp $t_{s,d}$.
p_k	CPU processing time of data-parallel segment k
g_k	GPU processing time of data-parallel segment k
$x_{i,j} \in \{0,1\}$	Denotes whether data parallel segment $i \in [0,d]$ executes on processor $j \in [0, m+n]$

Table 5.2: Variables and constants used in ILP formulation of $GPUMAP$.

to processor i at time-stamp $t_{s,d}$. The load of processor i at time-stamp $t_{s,d}$ is calculated as $l_i = \sum_{d \in D_i} t(d)$, where $t(d)$ is the time taken to process data-parallel segment d on processor i. The makespan of the set of concurrent data-parallel segments at time-stamp $t_{s,d}$ is then defined as $c_{max} = max(l_i)\ \forall i \in M$ where M is the set of CPUs and GPUs present in the heterogeneous platform.

I denote the problem of identifying a mapping of a set of concurrent transitions at a time-stamp $t_{s,d}$ for execution across CPUs and GPUs as $GPUMAP$, and provide an integer linear programming (ILP) formulation for $GPUMAP$. Table 5.2 tabulates the variables and constants used in the formulation.

$$
\begin{aligned}
\text{minimize} \quad & c_{max} \\
\text{subject to} \quad & \sum_{j=1}^{m+n} x_{i,j} = 1, \forall i \in [1,d] \\
& d \leq m \\
& where\ c_{max} = max(\sum_{i=1}^{d}(x_{i,j}p_i + x_{i,k}g_i)) \forall j \in [1,m], \forall k \in [1,n]
\end{aligned}
$$

Constraint 1 ensures that each data-parallel segment i is assigned to one and only one CPU or GPU. As a result, a data-parallel segment cannot execute on more than one processor. Constraint

2 limits the number of concurrent data-parallel segments at time-stamp $t_{s,d}$ to be less than or equal to the number of CPUs available in the system. This implies that the algorithm for generating the mapping of concurrent data-parallel segments across CPUs and GPUs does not take into account the effect of context switching and load balancing between CPUs by the underlying OS.

The target here is to determine the optimal makespan value. Setting a constant makespan value does not provide the optimal mapping. This is because, different combinations of executions across CPUs and GPUs result in different makespan values. Therefore, a search for the makespan value resulting in an optimal mapping is required.

Algorithm 4 describes an approach that performs a search for an appropriate candidate makespan, and applies a greedy heuristic for determining which data-parallel computations execute on the GPU based on a candidate makespan that is assumed to be optimal. The greedy heuristic is identical to the greedy 0-1 knapsack algorithm that treats the GPU as a knapsack and the candidate makespan as the capacity of this knapsack. The algorithm first identifies concurrent transitions in the Global Suspension Labeled Transition System. Concurrent transitions are transitions that are scheduled by the SystemC scheduler at the same simulation time-stamp. These concurrent transitions are updated into the set CT. The identification of concurrent transitions can be done by picking a transition in the Global Suspension Labeled Transition System and comparing its earliest scheduled time-stamp with the earliest scheduled time-stamps of other transitions. For each collection of concurrent transitions $ct \in CT$, the algorithm next calculates a range of makespan values $\langle u, l \rangle$ from which a candidate makespan value is selected. Initially, the upper limit of this makespan range is set as the sum of the maximum processing times of all concurrent data-parallel computation in ct. Data-parallel computations in the collection of concurrent transitions ct are added to the set D'. A candidate makespan value c is determined by applying a binary search procedure on $\langle u, l \rangle$. The algorithm then calls the function MAP, which takes as input the candidate makespan c, the set of concurrent data-parallel computations D', the current minimum makespan value c', and the mapping $map = \langle D_{CPU}, D_{GPU} \rangle$. c' and map are initialized to u and $\langle \emptyset, \emptyset \rangle$ respectively. The function MAP generates a mapping of data-parallel computations across CPUs and GPUs based on the candidate makespan. Algorithm 5 describes the MAP function.

The MAP function is responsible for generating a mapping of concurrent data-parallel segments for execution across CPUs and GPUs. The MAP algorithm described in Algorithm 5 and the experimentation carried out in this thesis assumes a single GPU system or $n = 1$. However, it can be extended for multi-GPU systems. The MAP function first categorizes GPU only data-parallel computations and CPU only data-parallel computations based on the candidate makespan c. If the candidate makespan is greater than the GPU processing time of a data-parallel computation, that data-parallel computation is not considered for GPU execution. The data-parallel computations identified as candidates for GPU execution are arranged in non-increasing order

Algorithm 4: $GPUMAP$-Greedy algorithm.

```
   /* Initial declarations                                                    */
1  Let $GS$ denote the Global Suspension Labeled Transition System of the SystemC model.
2  Let $u$ and $l$ denote the upper and lower limits of the range $R$ respectively.
3  $GS' \leftarrow GS$
4  $CT \leftarrow \emptyset$
5  foreach $t \in GS'$ do
6      $ct \leftarrow \emptyset$
7      $ct \leftarrow ct \cup \{t\}$
8      Remove $t$ from $GS'$
9      foreach $t' \in GS'$ do
10         if $t_{s,d} = t'_{s,d}$ then
11             $ct \leftarrow ct \cup \{t'\}$
12             Remove $t'$ from $GS'$
13         end
14     end
15     $CT \leftarrow CT \cup \{ct\}$
16 end
17 $l = 0$
18 $u = 0$
19 foreach $ct \in CT$ do
20     $D' \leftarrow \emptyset$
21     $u = u + \sum_{d \in ct} max(p(d), g(d))$
22     $c' = u$
23     $map \leftarrow \langle \emptyset, \emptyset \rangle$
24     $D' \leftarrow D' \cup d \,, \forall d \in ct$
25     while $u \neq l$ do
26         $c = (u + l)/2$
27         $x = MAP(c, D', c', map)$
28         if $x = true$ then
29             $l = c'$
30         else
31             $u = c'$
32         end
33     end
34 end
```

Algorithm 5: $MAP(c, D', m, map)$ function

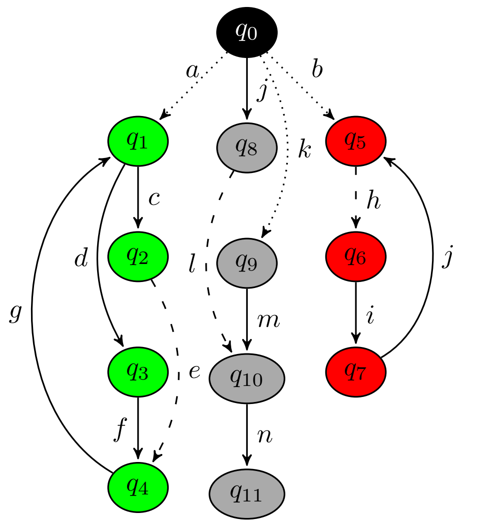
```
   /* Initial declarations                                                    */
1  Let $D_{GPU}$ represent the set of GPU mapped data-parallel computations and $D_{CPU}$ represent the
   set of CPU mapped data-parallel computations.
2  $D_{GPU} \leftarrow \emptyset$ , $D_{CPU} \leftarrow \emptyset$
3  foreach $d \in D'$ do
4      if $g(d) \leq c$ then
5          $isGPU(d) = True$
6      end
7  end
8  foreach $d \in D'$ do
9      if $g(d) > c$ then
10         $isGPU(d) = False$
11     end
12 end
13 Arrange $D'$ in order of non-increasing acceleration factor $\alpha(d) = p(d)/g(d)\ \forall d \in D'$ such that
   $isGPU(d) = True$
14 foreach $d \in D'$ such that $isGPU(d) = True$ do
15     if $g(d) < c$ then
16         $c = c - g(d)$
17         $D_{GPU} \leftarrow D_{GPU} \cup \{d\}$
18     end
19 end
20 foreach $d \in D' - D_{GPU}$ do
21     Schedule across CPUs to minimize load balance.
22     $isGPU(d) = False$
23     $D_{CPU} \leftarrow D_{CPU} \cup \{d\}$
24 end
25 if $c' > makespan(D_{CPU}, D_{GPU})$ then
26     $c' = makespan(D_{CPU}, D_{GPU})$
27     $map = \langle D_{CPU}, D_{GPU} \rangle$
28     Return $True$
29 end
30 Return $False$
```

of acceleration factors $\alpha(d) = p(d)/g(d)$. Arranging the data-parallel computations in non-increasing order improves the search strategy for determining an optimal makespan value and prioritizes data-parallel segments with maximum benefit from GPU execution to execute on the GPU. From the ordered list, each data-parallel computation is assigned to the GPU until the maximum candidate makespan c is reached. The remaining data-parallel computations are assigned to the CPUs using a simple load-balancing algorithm as all the CPUs are identical. The makespan for this mapping is then compared with the current minimum makespan value. If the makespan for the mapping is less than the minimum makespan value, the minimum makespan value and the mapping is updated with the current makespan value and the mapping, and the function returns $True$. If the makespan for the mapping is larger than the minimum makespan value, the minimum makespan value and mapping are not changed, and the function returns $False$. For a return value of $True$, the range is updated with the upper limit changed to the current makespan value. This is to probe for lower makespan values. On the other hand, a return value of $False$ indicates that the candidate makespan does not satisfy the constraints. In this case the lower limit of the range is updated with the current makespan value. Since for each collection of overlapping transitions at time-stamp $t_{s,d}$, the makespan is optimized to be as low as possible, the simulation time of the of the entire SystemC model is optimized for faster execution time.

To further augment the understanding of Algorithm 4 and 5, I describe the execution of Algorithm 4 and 5 with an example of a Global Suspension Labeled Transition System for a SystemC model that is shown in Figure 5.8 with some concurrent data-parallel segments annotated with CPU and GPU profiling times.

In Figure 5.8, transitions a, k, and b are scheduled for execution by the SystemC scheduler at the same time-stamp $t_{s,d}$, and have data-parallel segments that can execute on the GPU. Similarly transitions e, l, and h are transitions that are scheduled for execution at the same time-stamp $t'_{s,d}$ by the SystemC scheduler, and have data-parallel segments suitable for execution on the GPU. Using the transitions a, k, and b, I illustrate the execution of Algorithms 4 and 5, and arrive at a mapping of the data-parallel segments present in these transitions for execution across CPUs and GPUs that results in the fastest execution time for this set of concurrent transitions. Tables 5.3, 5.4, 5.5, and 5.6 tabulate the values of the candidate makespan value c, the makespan range, the current minimum mapping of data-parallel segments across CPUs and GPUs, and the current minimum makespan value c' associated with the current minimum mapping for different iterations of the execution of the Algorithms 4 and 5.

Algorithm 4 first calculates an artificial upper bound u, which is the sum of the maximum execution times of the data-parallel segments on CPUs and GPUs (Line 21). For the concurrent transition set $\{a, k, b\}$, this upper bound is $u = 145 + 70 + 30 = 245$. Therefore, the initial range from which the candidate makespan is determined is $[0, 245]$. The candidate makespan c is determined using a binary search, $c = (0 + 245)/2 = 122.5$ (Line 26). This candidate makespan



Transition	CPU (ms)	GPU (ms)	Acceleration Factor
a	145	18	2.5
k	50	70	0.7
b	25	30	0.8

Figure 5.8: Example of Global Suspension Labeled Transition System with profiling times of concurrent data-parallel segments.

value along with the initial empty mapping and minimum makespan value c', which is initially set to the upper bound, are passed as parameters to the function MAP (Line 27). The function MAP performs a pre-processing stage by determining data-parallel segments exclusive for CPU or GPU execution by comparing the candidate makespan value with their CPU and GPU execution times (Lines 3-12). For $c = 122.5$, data-parallel segment on transition a cannot be considered for CPU execution as its CPU execution time is greater than that of the candidate makespan value. Therefore, a has to be executed on the GPU. The remaining data-parallel segments on transitions b and c can be considered for execution on both CPU and GPU. Data-parallel segments identified for GPU execution are then arranged in non-increasing order of acceleration factor (Line 13). The order obtained is $a > b > k$. Treating the GPU as a knapsack and c as the capacity of the knapsack, it can be observed that executing all the data-parallel segments on the GPU results in a makespan value less than the candidate makespan value $(18 + 70 + 30 = 118 < 122.5)$ (Lines 14-19). This makespan value in addition is also less than the current minimum makespan value $c' = 245$. Therefore, the mapping and the makespan value are updated such that all data-parallel segments in the set of concurrent transitions $\{a, k, b\}$ are executed on the GPU, and the minimum makespan value is updated to 118 (Lines 25-28). The function MAP returns a boolean value of $true$ to $GPUMAP$ resulting in an update to the range. The upper value of the range is update to the new makespan value resulting in a new range $[0 - 118]$. Since the upper and lower ranges do not match, the algorithm continues to search for a new minimum candidate makespan value (Line 25). Table 5.4 tabulates the new range, candidate makespan value, current minimum mapping, and current minimum makespan value for this iteration of the algorithm. For the new candidate makespan value $c = 59$, it can be observed that data-parallel segment in transition b cannot be executed on the GPU. Hence, only a and c are eligible for GPU execution. For the candidate makespan $c = 59$, a mapping that executes data-parallel segments in transitions a and b on the GPU and k on the CPU results in a makespan of $max(18 + 30, 50) = 50$, which is less than $c = 59$. Moreover, this mapping's makespan value is also less than the current minimum makespan value $c' = 118$. Therefore, both the mapping and the minimum makespan values are updated with data-parallel segments a and b on GPU and k on CPU, and $c' = 50$. The MAP function returns $true$ and the range is updated with the upper range set to the current minimum makespan value of 50. For the new candidate makespan value of $c = (0 + 50)/2 = 25$, notice that the data-parallel segment in transition b can neither be executed on the CPU or on the GPU as its execution time is larger than c on both the CPU and GPU. Therefore, any mapping of the data-parallel segments in the transitions set $\{a, b, k\}$ will be more than the candidate makespan. Therefore, MAP will return $false$ for this candidate makespan, and the lower bound of the range is updated with 50 as the best possible mapping with the candidate makespan of 25 will result in a makespan of 50. The algorithm terminates at this point as the upper and lower bounds of the range are equal. The mapping stored as the current minimum mapping when the algorithm terminates is the best mapping possible with the fastest execution time.

5.2 CUDA-Gen: A systemc-clang plugin for translating identified GPU portions to CUDA

Once a mapping of data-parallel segments across concurrent transitions is determined, the translation of the identified GPU data-parallel segments to CUDA begins. The translation to CUDA is done by a systemc-clang plugin CUDA-Gen. The input to CUDA-Gen is the augmented Global Suspension Labeled Transition System with the $isGPU$ flag set appropriately for all data-parallel segmentsby $GPUMAP$. Algorithm 6 describes the translation algorithm.

The algorithm begins by first iterating through the Global Suspension Labeled Transition System of the input SystemC model, and identifying data-parallel computations set for GPU execution by SCuitable. Two sets RHS and LHS are maintained that hold the variables read from and written to in a data-parallel segment. As SystemC data-types and the generic-payload data-structure do not have associated CUDA data-types, these data-types need to be transformed into CUDA data-types. Conversion of SystemC data-types to CUDA data-types has been previously investigated by the authors of [51]. However, due to the closed-source nature of the CUDA data-type library proposed in [51], I use a straightforward approach for the data-type conversion. For

Parameter	Value
Makespan range	$[0 - 245]$
Candidate makespan value	$(0 + 245)/2 = 122.5$
Current minimum makespan value	245
Current minimum mapping	$GPU = \emptyset, CPU = \emptyset.$

Table 5.3: Initial value of parameters.

Parameter	Value
Makespan range	$[0 - 118]$
Candidate makespan value	$(0 + 118)/2 = 59$
Current minimum makespan value	118
Current minimum mapping	$GPU = \{a, b, k\}, CPU = \emptyset.$

Table 5.4: Parameters after first pass of Algorithms 4 and 5.

Parameter	Value
Makespan range	$[0 - 50]$
Candidate makespan value	$(0 + 50)/2 = 25$
Current minimum makespan value	50
Current minimum mapping	$GPU = \{a, b\}, CPU = \{k\}.$

Table 5.5: Parameters after second pass of Algorithms 4 and 5.

Parameter	Value
Makespan range	$[50 - 50]$
Candidate makespan value	N/A
Current minimum makespan value	50
Current minimum mapping	$GPU = \{a, b\}, CPU = \{k\}.$

Table 5.6: Parameters after final pass of Algorithms 4 and 5.

Algorithm 6: Translation of SystemC model into CUDA

```
/* Initial declarations */
1  Let $GS$ be the Global Suspension Labeled Transition System of the input SystemC model.
2  Let $LHS$ be the set of LHS assignment variables for a data-parallel segment.
3  Let $RHS$ be the set of RHS assignment variables for a data-parallel segment.
4  foreach $t \in GS$ do
5      foreach $d \in t$ such that $isGPU(d) = True$ do
6          $LHS \leftarrow \emptyset$
7          $RHS \leftarrow \emptyset$
8          $LHS \leftarrow LHS \cup \text{getLHSVariables}(d)$
9          $RHS \leftarrow RHS \cup \text{getRHSVariables}(d)$
10         ConvertDataType($LHS \cup RHS$)
11         GPUAllocate($LHS' \cup RHS'$)
12         CopyFromCPUToGPU($LHS' \cup RHS'$)
13         genThreadHierarchy()
14         genCUDAKernel()
15         genCUDASynchronize()
16         CopyFromGPUToCPU($LHS'$)
17     end
18 end
```

arbitrary data-widths such as sc_uint⟨6⟩, which represents an unsigned integer of data-width of 6 bits, the corresponding CUDA data type is an unsigned integer which is anded with 63. In other words, for arbitrary data-widths of width N, the corresponding CUDA datatype is anded with $2^N - 1$. CUDA-Gen currently does not support conversion of sc_logic data-types to CUDA. For the generic payload information, a similar GPU data-structure is maintained, and the fields of the generic payload information accessed in a data-parallel section identified for GPU execution are copied to the fields of the GPU data-structure version of the generic payload. The CUDA data-types are allocated on the GPU global memory via the GPUAllocate function. The algorithm transfers the data from the CPU to the GPU for all the variables read/written in the data-parallel segment. The thread hierarchy for the data-parallel segment is determined by investigating the loop bounds of all the nested loops in the data-parallel segments. For the NVIDIA Tesla C2075 GPU model, the number of threads that can be launched in a thread block is 512. If the collected loop bound in a data-parallel segment is less than 512, a single block with loop bound as the number of threads in a block is launched. For a collected loop bound greater than 512, the loop bound is rounded to the nearest multiple of 512, and the number of blocks is obtained by dividing the rounded number by 512. The kernel for the data-parallel segment is generated using the genCUDAKernel function. The kernel ensures that the loop bound constraints in the original data-parallel code are satisfied by preceding the main body of the kernel with a condition on the thread identifiers. In order to ensure consistency of the signals during the evaluate-update phase, the CPU thread invoking the GPU kernel is made to wait for the return of the kernel execution before proceeding to execute the remainder of the transition. The CUDA kernel synchronization with the CPU is handled by the genCUDASynchronize call. The variables updated in the data-parallel segment are copied back to their CPU counterparts using the CopyFromGPUToCPU function. Notice that the function only copies back variables that have been modified in the data-parallel segment. This is done by identifying variables read and written to in the data segment using the functions $getLHSVariables$ and $getRHSVariables$ as shown in Lines 9 and 10. This optimization helps in reducing the effect of the memory copy transfer overhead between the CPU and GPU as only necessary data is copied between the CPU and GPU. In Chapter 6, I highlight the effect of this optimization for the image processing TLM benchmarks.

Chapter 6

Results

In this chapter, I evaluate the efficiency of systemc-clang, and the speedups obtained from distributing the execution of a SystemC model over multi-core CPUs and GPUs. I also compare the proposed methodology with the previous works of Nanjundappa et al. [15] and Vinco et al. [16] for accelerating SystemC RTL designs.

6.1 Efficacy of systemc-clang

The efficiency of systemc-clang is measured by its ability to extract all the structural and behavioral information present in the input SystemC model. Currently, I measure systemc-clang's efficiency by manually comparing the SystemC model and the behavioral and structural information generated by systemc-clang. For instance, for an input SystemC model, I manually list the SystemC modules, the SystemC processes for each SystemC module, the signals and ports for each SystemC module, and the TLM-2.0 related information present. I compare the above with the report generated by systemc-clang and, measure how much structural and behavioral information systemc-clang extracts for the given model. I use the SystemC RTL benchmark suite S2CBench [19], a complete TLM model from Doulous [52], and five in-house developed TLM image processing models for measuring systemc-clang's efficiency.

6.1.1 S2CBench: A SystemC RTL Benchmark Suite

The S2CBench [19] benchmark suite consists of twelve benchmarks written in SystemC at the RTL abstraction level. The benchmarks present in this suite cover a wide variety of applications ranging from security to signal processing. For the SystemC RTL designs, I evaluate systemc-clang on its ability to extract all the structural and behavioral information information present in the SystemC model. Table 6.1 tabulates the benchmarks and the efficiency of systemc-clang in identifying and extracting the structural and behavioral information present in the model. The term SC in Table 6.1 is short for SystemC. I define efficiency as the number of SystemC constructs systemc-clang identifies over the number of constructs present in the SystemC model. The SystemC constructs considered for measuring the efficiency of systemc-clang for SystemC RTL designs are SystemC modules, processes, ports, signals, and SystemC scheduler calls (wait()).

Benchmark	# of SC modules	# of SC processes	# of SC ports	# of SC signals	# of wait() calls
ADPCM	2/2	3/3	8/8	3/3	7/7
AES Cipher	2/2	3/3	10/10	5/5	7/7
Decimation Filter	2/2	3/3	20/20	10/10	8/8
Disparity Estimation	2/2	3/3	32/32	15/15	15/15
FFT	2/2	3/3	20/20	10/10	13/13
FIR	2/2	3/3	10/10	5/5	7/7
Interpolation Filter	2/2	3/3	12/12	6/6	7/7
Kasumi Cipher	2/2	4/4	10/10	5/5	9/9
MD5C Cipher	2/2	3/3	16/16	8/8	11/15
Quicksort	2/2	3/3/	8/8	4/4	7/7
IDCT	2/2	3/3	18/18	9/9	1/3
Snow 3G Cipher	2/2	3/3	8/8	4/4	7/7
Sobel Filter	2/2	3/3	8/8	4/4	8/8

Table 6.1: Efficiency of systemc-clang on S2CBench.

The general structure of the benchmarks present in S2CBench consist of two modules, which represent the testbench and the main benchmark. The testbench module consists of two SystemC processes responsible for reading the input data and writing the output data. The main benchmark module contains processes responsible for performing the benchmark function. As observed from Table 6.1, systemc-clang identifies all the Systemc modules and processes present in all the S2CBench benchmarks. The data-types of the ports and signals are SystemC specific data-types such as sc_uint and sc_fixed. systemc-clang identifies all the signals and ports used in addition to the data-types for each benchmark efficiently. The behavioral information of a SystemC model is built using the wait() and notify() call information present in the model. For all benchmarks except MD5C Cipher and IDCT, the number of wait() calls present in the benchmarks match the number reported by systemc-clang. For the MD5C and IDCT benchmarks, some of the wait() calls are called from class methods that are not registered as SystemC processes in the SystemC module constructor. systemc-clang extracts wait() and notify() calls from methods registered as SystemC processes in the module's constructor. For such cases, systemc-clang can be configured to identify wait() and notify() calls in all methods of a SystemC module.

Table 6.2 tabulates the time taken by systemc-clang to build the structural and behavioral IR for the benchmarks present in the S2CBench benchmark suite.

Benchmark	Time taken to build AST (sec)	Time taken to build structural IR (sec)	Time taken to build behavioral IR (sec)
ADPCM	14.3	0.7	0.02
AES Cipher	13.3	0.7	0.01
Decimation Filter	37.5	1.5	0.04
Disparity Estimation	15.3	0.7	0.05
FFT	34.6	1.4	0.02
FIR	13.3	0.7	0.006
IDCT	16.3	0.7	0.006
Interpolation Filter	37.6	1.4	0.01
Kasumi Filter	15.3	0.7	0.02
MD5C Cipher	14.3	0.7	0.007
Quicksort	13.4	0.6	0.004
Snow 3G Cipher	15.3	0.7	0.008
Sobel Filter	13.3	0.7	0.008

Table 6.2: Time taken by systemc-clang to build structural and behavioral IRs for S2CBench benchmark suite.

6.1.2 A Complete AT TLM model from Doulous

Doulous is a consultancy company that provides training and resources for hardware designers. With regard to SystemC, Doulous plays an active role in developing tutorials and reference manuals for SystemC designs. The SystemC Language Reference Manual for TLM 2.0 standards [22] was developed by Doulous. To evaluate systemc-clang's ability to parse TLM-2.0 models, I use the complete approximately-timed model (AT) model [52] developed by Doulous. The AT model consists of 2 initiator types and 5 target types connected by an interconnect. Since the model is an approximately-timed model, the transport interface used is non-blocking. The initiator and target modules use the simple convenience sockets simple_initiator_socket and simple_target_socket respectively. The interconnect component uses the convenience multi_passthrough sockets. The payload communicated via the transport interfaces is the generic payload. systemc-clang extracts the attributes of the generic payload set by the initiator and target. Table 6.3 tabulates the structural and behavioral information extracted by systemc-clang for the AT example. For the AT

Property	Effectiveness of systemc-clang
# of SystemC modules	8/8
# of TLM 2.0 Sockets	9/9
# of TLM 2.0 Socket Register Callbacks	13/13
Identification of TLM-2.0 Generic Payload	☑
Identification of Core Interfaces	
Non-Blocking Forward Transport Interface	☑
Non-Blocking Backward Transport Interface	☑
Blocking Transport Interface	☑
DMI Interface	☑
Debug Interface	☑
# of wait() calls	4/4
# of notify() calls	11/11

Table 6.3: Efficiency of systemc-clang on complete AT model from Doulous.

model, systemc-clang correctly identifies the initiators, targets, and interconnect modules. The sockets and the underlying interfaces between initiators, targets and the interconnect are also correctly identified by systemc-clang. Note that although the input model is an approximately-timed model, and therefore, the transport interface used should be non-blocking, systemc-clang reports the presence of a blocking transport interface method definition. Since the AT example provided by Doulous is meant to serve as an example for interested developers, the interconnect sockets are registered with the blocking interface for developers to experiment with different coding styles and interfaces. Moreover, none of the targets or initiators communicate the payload information using the blocking transport interface. For this example, systemc-clang identified the wait() and notify() calls present in the model. However, the arguments to the wait() and notify() calls could not be determined statically, as the duration of the wait() and notify() calls are based on variables.

Table 6.4 tabulates the time taken by systemc-clang to build the structural and behavioral IRs for this AT model.

Time taken to build AST (sec)	Time taken to build structural IR (sec)	Time taken to build behavioral IR (sec)
26.8	1.2	0.01

Table 6.4: Time taken by systemc-clang to build structural and behavioral IRs for complete AT model.

6.1.3 TLM Image Processing Case Studies

I developed two image processing case studies at the TLM abstraction, and measure the efficiency of systemc-clang on these case studies. The two image processing case studies are a Canny edge detector, and a JPEG decoder. The Canny edge detector model is written in both the loosely-timed modeling style and the approximately-timed modeling style. The JPEG decoder is modeled using the loosely-timed modeling style. In addition, I also use WeiWei Chen's loosely-timed model of the JPEG encoder [53] to evaluate systemc-clang. Tables 6.5, 6.6, 6.7, and 6.8 tabulates the efficiency of systemc-clang on these image processing case studies.

The Canny edge detection model is a popular edge detection algorithm that uses a combination of filters to detect edges in an image. The algorithm first applies a Gaussian filter on the image to reduce the noise present in the image. It then calculates the gradient and direction for each pixel using a Sobel filter. The Sobel filter convolves the blurred image with two 3×3 kernels to extract the gradient intensity and direction in the horizontal and vertical directions. Once the gradient intensities and directions for each pixel are identified, a non-maximum suppression filter is applied that retains pixels on identified edges with maximum gradient values. To further remove noise, a hysteresis filter is applied on the detected images, which discards pixels below a certain threshold. The SystemC implementation of the Canny edge detection has four modules for each filter and two additional modules to handle reading data from input image and writing data to the output image. In the TLM sense, the module responsible for reading the values of input picture acts as the initiator, and the rest of the modules act as targets. The filters are implemented as SC_THREADS in each SystemC module. It can be observed from Tables 6.5 and 6.6 that systemc-clang identifies all the necessary SystemC constructs for building the structural and behavioral IR.

The JPEG file format is a widely used compressed image file format. I developed a JPEG decoder model using the LT style of modeling that takes as input a JPEG image and decompresses it into a BMP file. The stages of the JPEG decoder are entropy decoding, dequantizing, inverse DCT, and color reordering. In the SystemC implementation, these are implemented as normal class methods that are called by a SystemC process. Similar to the canny edge detection model, in the TLM sense, the initiator is the module responsible for reading the input picture and the

Property	Efficiency of systemc-clang
# SystemC Modules	6/6
# SystemC Processes	6/6
Identification of Non-Blocking Transport Interface	☑
Identification of Generic Payload	☑
# of wait() calls	6/6
# of notify() calls	0/0

Table 6.5: Efficiency of systemc-clang for AT Canny Edge Detection Model.

Property	Efficiency of systemc-clang
# SystemC Modules	6/6
# SystemC Processes	6/6
Identification of Blocking Transport Interface	☑
Identification of Generic Payload	☑
# of wait() calls	1/1
# of notify() calls	0/0

Table 6.6: Efficiency of systemc-clang for LT Canny Edge Detection Model.

target modules are the stages of the JPEG decoder. For this model, systemc-clang was able to identify all the necessary information for building the structural and behavioral IR.

The JPEG color encoder by WeiWei Chen [53] compresses BMP images to JPEG. It consists of the same set of steps as the JPEG decoder, but executed in reverse. The structure of the SystemC version of the JPEG color encoder treats each function of the encoder as a SystemC process implemented in its own SystemC module. Therefore, the number of SystemC processes and modules present in this case study is higher than my implementation of the JPEG decoder. From Table 6.7 it can be observed that systemc-clang identifies all the structural and behavioral information present in the JPEG encoder model.

Table 6.9 tabulates the time taken by systemc-clang to generate the structural and behavioral IRs for the above SystemC TLM image processing benchmarks.

Property	Efficiency of systemc-clang
# SystemC Modules	12/12
# SystemC Processes	7/7
Identification of Blocking Transport Interface	☑
# of wait() calls	4/4
# of notify() calls	0/0

Table 6.7: Efficiency of systemc-clang for LT JPEG Encoder Model.

Property	Efficiency of systemc-clang
# SystemC Modules	4/4
# SystemC Processes	4/4
Identification of Blocking Transport Interface	☑
# of wait() calls	1/1
# of notify() calls	0/0

Table 6.8: Efficiency of systemc-clang for LT JPEG Decoder Model.

6.2 Acceleration of SystemC RTL models

In this section, I compare previous related work by Nanjundappa at al. [15] and Vinco et al. [16] for accelerating SystemC RTL simulation using GPUs. Recall that the technique proposed by Nanjundappa et al. [15] converts each SystemC process present in the model into a state machine that is executed by a thread on the GPU. Since each process can have varying control flow, each process is executed by a thread in a different warp to eliminate the effect of thread divergence within a warp. On the other hand, Vinco et al. [16] propose generating independent data-flows from a SystemC model that are executed on separate streaming multiprocessors on the GPU. This reduces the overhead of frequent synchronization between threads at the cost of code duplication. For the remainder of the text, I refer the work of Nanjundappa et al. [15] as SCGPSim and that of Vinco et al. [16] as SAGA. To the best of my knowledge, these are the only two research efforts that propose utilizing the GPU for accelerating SystemC RTL simulations. I compare SCGPSim and SAGA with the proposed technique, SCuitable, using benchmarks from the S2CBench benchmark suite. Each benchmark is run for a million simulation cycles. The specifications of the target platform used in the experimentation is described in Table 6.10.

Benchmark	Time taken to build AST (sec)	Time taken to build structural IR (sec)	Time taken to build build behavioral IR (sec)
LT Canny Edge Model	13.4	0.6	0.04
AT Canny Edge Model	13.3	0.7	0.01
LT JPEG Decoder	13.2	0.8	0.02
LT JPEG Encoder	14.3	0.7	0.01

Table 6.9: Time taken by systemc-clang to build structural and behavioral IRs for SystemC TL image processing case studies.

Processor	Intel Xeon E5645
Number of cores	6
Hyper-Threading Enabled	No
Frequency	2.40 GHz
Operating System	Linux Ubuntu 12.04
GPU	Tesla C2075
Number of CUDA cores	448
Frequency of CUDA core	1.15 GHz
CUDA Driver Library Version	5.0

Table 6.10: Specifications of the heterogeneous platform used for experimentation.

6.2.1 3-stage Pipeline

The 3-stage pipeline is a simple SystemC example, in which the SystemC processes model the pipeline stages. The SystemC processes are implemented as SC_METHODS that are sensitive to the positive edge of a clock. A total of 5 SystemC processes are activated by the positive edge of the clock. Each process performs some arithmetic operation and the result of the operation is forwarded to the next stage in the pipeline. The computation involved in each SystemC process is minimal. Table 6.11 tabulates the execution time and speedups of the simulation with five concurrent processes using using the methods of SAGA, SCGPSim and SCuitable.

It can be observed that for five concurrent processes, none of the approaches performs better than the sequential simulation. Of the three, SCuitable does worse because the computation involved in each SystemC process is minimal, and therefore, the context switching overhead and synchronization of the concurrent threads dominates the total execution time. For SAGA and SCGPSim, the GPU utilization is too low to observe any simulation benefits. For 5 concurrent processes, SCGPSim launches 5 CUDA threads for each SystemC process, while SAGA

Processes	SystemC	SAGA		SCGPSim		SCuitable	
	Execution Time (sec)	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup
5	1	3	0.3	2.7	0.4	14	0.07

Table 6.11: Comparison of SAGA, SCGPSim, and SCuitable for 3-Stage Pipeline.

launches only one CUDA thread. For the SAGA approach, the entire 3-stage pipeline constitutes a data-flow. Therefore, multiple instances of the 3-stage pipeline are executed on distinct streaming multiprocessors on the GPU. Figure 6.1 shows the simulation benefits offered by the three approaches over the sequential simulation when the number of concurrent SystemC processes increases. It can be observed that both SAGA and SCGPSim provide a speedup of approximately 4 over sequential simulation while SCuitable does not show any simulation benefit over the sequential simulation. This is because, the effect of the thread synchronization and context switching between threads continues to dominate the execution time of the SystemC model resulting in sub-optimal speedups. SCGPSim and SAGA on the other hand accelerate the simulation of the 3-stage pipeline model as the number of compute cores available on the GPU device is abundant and no context switching takes place between the threads running on the different CUDA cores. Moreover, the effect of synchronization between CUDA threads is minimal due to the simple nature of the computation carried out by each thread.

6.2.2 AES Cipher

The Advanced Encryption Standard (AES) is a standardized encryption algorithm developed for encrypting military transactions. In each cycle, the algorithm works on a 4×4 matrix of bytes and a set of 128 bit keys. The number of transformations required to encrypt the input plain text is set to 10 in this SystemC implementation of the AES cipher.

In each iteration, a set of four operations is applied on the input plain text block: combining each element in the 4×4 matrix with a block of the key using a bitwise XOR operation, replacing each element with another using a look-up table, shifting the three lower rows for a certain number of steps (in this implementation, the number of steps is set to four), and a mixing operation that combines bytes in a column of the 4×4 matrix. In this AES implementation, these four operations are executed 10 times each in the same clock cycle. Table 6.12 tabulates the speedup of the AES simulation algorithm with 3 and 6 concurrent processes.

The amount of computation done by the SystemC process responsible for the algorithm is high compared to the other SystemC processes responsible for generating the stimulus and ac-

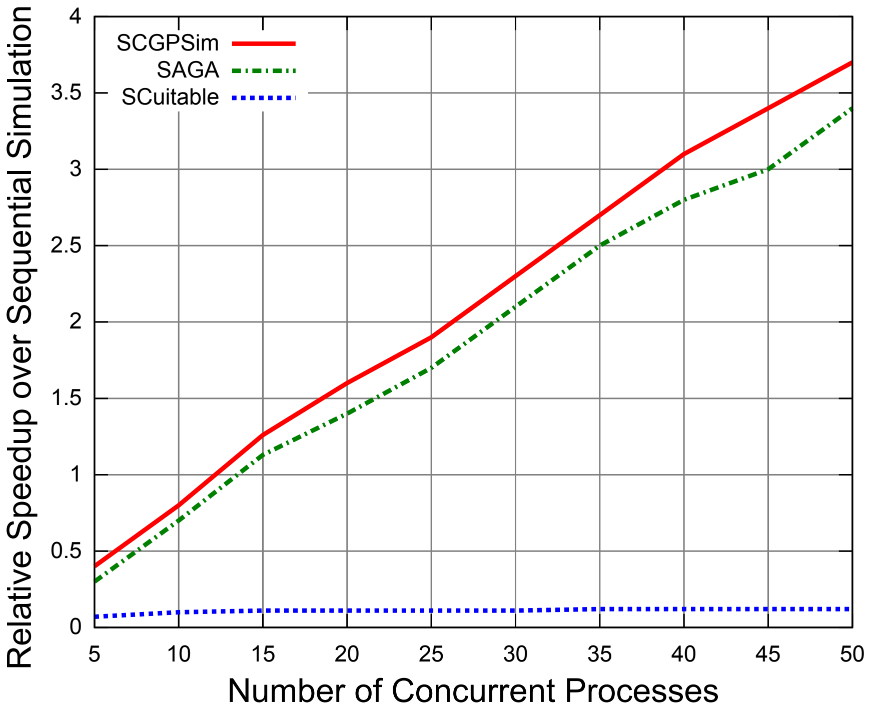


Figure 6.1: Comparison of relative speedups of simulations of SAGA, SCGPSim, and SCuitable over sequential SystemC simulation for 3-Stage Pipeline Model.

cumulating the final encoded text. In addition, the SystemC process responsible for the algorithm contains data-dependent conditionals making the control flow of the algorithm intricate. A CUDA core responsible for the algorithm will perform poorly compared to a CPU core due to its simple, in-order pipelined architecture. Therefore, both SAGA and SCGPSim perform poorly as both allocate a CUDA core to implement the algorithm. On the other hand, SCuitable executes the stimulus, display, and the cipher algorithm on distinct CPU cores, resulting in a higher speedup over SAGA and SCGPSim. As the cipher algorithm is placed on a separate CPU core, the execution of the algorithm can leverage the architectural features provided by the core. However, the GPU is not exploited by the AES cipher algorithm due to the unavailability of any beneficial data-parallel computations. For three and six concurrent processes, SCuitable provides a speedup of 2.3 and 3.3 respectively over the sequential simulator. Figure 6.2 illustrates the scalability of SAGA, SCGPSim, and SCuitable with increasing number of concurrent SystemC processes. The execution time of the AES cipher simulation using SAGA and SCGPSim remains constant as the number of compute cores is larger than the number of concurrent SystemC processes. Therefore, the benefits of SAGA and SCGPSim should be apparent when the

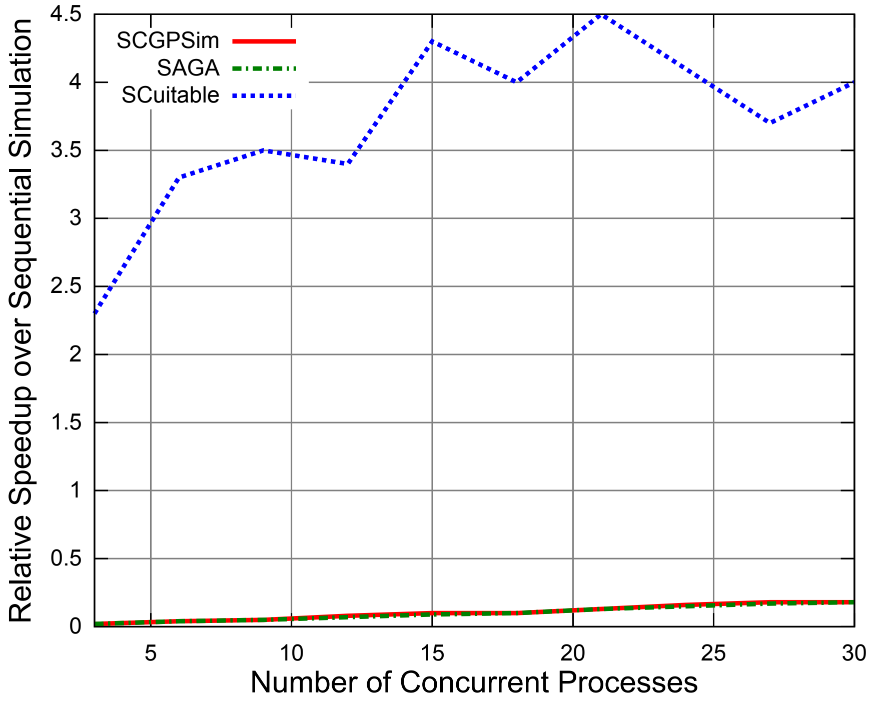


Figure 6.2: Comparison of relative speedups of simulations of SAGA, SCGPSim, and SCuitable over sequential SystemC simulation for AES Cipher Algorithm.

number of concurrent SystemC processes is sufficiently high. SCuitable demonstrates increasing speedups with increase in the number of concurrent SystemC processes. For an AES cipher model with 30 concurrent SystemC processes, SCuitable provides a speedup of 4 over sequential simulation. It is important to note that Nanjundappa et al. [15] use a pipelined AES cipher SystemC model to highlight the simulation benefits of SCGPSim. The AES SystemC model used in the experimentation is an unpipelined one.

6.2.3 Kasumi Cipher

The Kasumi Cipher is another block cipher used in the wireless communication domain, which works on 64-bit inputs and a block key size of 128 bits. Similar to the AES Cipher, the core of the Kasumi cipher algorithm is computationally intensive. Table 6.13 tabulates the relative speedup of the Kasumi cipher algorithm using the proposed techniques of SAGA, SCGPSim, and SCuitable. The table shows the speedup of the model with four concurrent processes. A similar trend is observed for the Kasumi Cipher, wherein SAGA and SCGPSim do poorly and do

Processes	SystemC	SAGA		SCGPSim		SCuitable	
	Execution Time (sec)	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup
3	34	1955	0.02	1929	0.02	15	2.3
6	69	1955	0.04	1929	0.04	21	3.3

Table 6.12: Comparison of SAGA, SCGPSim, and SCuitable for AES Cipher Algorithm

Processes	SystemC	SAGA		SCGPSim		SCuitable	
	Execution Time (sec)	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup
4	35	2700	0.01	2710	0.01	13	2.7

Table 6.13: Comparison of SAGA, SCGPSim, and SCuitable for Kasumi Cipher Algorithm.

not show any simulation benefits over sequential simulation. SCuitable on the other hand, shows a 2.7 speedup over the sequential SystemC simulator. The mapping returned by SCuitable for the Kasumi cipher model executes it across multi-core CPUs only due to the unavailability of any data-parallel segments for GPU execution. Figure 6.3 shows the relative speedup of SCGPSim, SAGA, and SCuitable as a function of the number of concurrent SystemC processes. Similar to the AES Cipher, the speedups of SAGA and SCGPSim are suboptimal because of the amount of computation offloaded to a CUDA thread. However, the benefits of SAGA and SCGPSim should become apparent on further increasing the number of concurrent SystemC processes. SCuitable shows good speedups with increase in number of concurrent SystemC processes over the sequential implementation of the algorithm as the computation is executed across multiple cores present. The highest speedup obtained by SCuitable over sequential simulation is around 4.3.

6.2.4 FIR Filter

The finite impulse response (FIR) filter is an important digital filter for digital signal processing (DSP) algorithms. The output of the filter is the weighted sum of the current and previous inputs. To increase the computation done per clock cycle, I increase the number of filter taps used for calculating the output. Tables 6.14, 6.15 and 6.16 tabulate the speedups of a 15-tap, 512-tap, and a 1024-tap FIR filter using the SAGA, SCGPSim, and SCuitable approach.

A n-tap filter performs n multiply-and-accumulate operations in a single cycle. For the 15-tap filter, all the approaches perform poorly with respect to the sequential simulation. In the

Processes	SystemC	SAGA		SCGPSim		SCuitable	
	Execution Time (sec)	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup
3	0.2	4.4	0.05	4	0.05	1	0.2
6	0.6	4.4	0.14	4	0.15	1.4	0.4

Table 6.14: Comparison of SAGA, SCGPSim, and SCuitable for 15-tap FIR Filter.

Processes	SystemC	SAGA		SCGPSim		SCuitable	
	Execution Time (sec)	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup
3	9	412	0.02	305	0.03	6	1.5
6	20	412	0.05	305	0.07	9	2.2

Table 6.15: Comparison of SAGA, SCGPSim, and SCuitable for 1024-tap FIR Filter.

Processes	SystemC	SAGA		SCGPSim		SCuitable	
	Execution Time (sec)	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup
3	17	826	0.02	609	0.03	11	1.5
6	35	826	0.04	609	0.06	14	2.5

Table 6.16: Comparison of SAGA, SCGPSim, and SCuitable for 2048-tap FIR Filter.

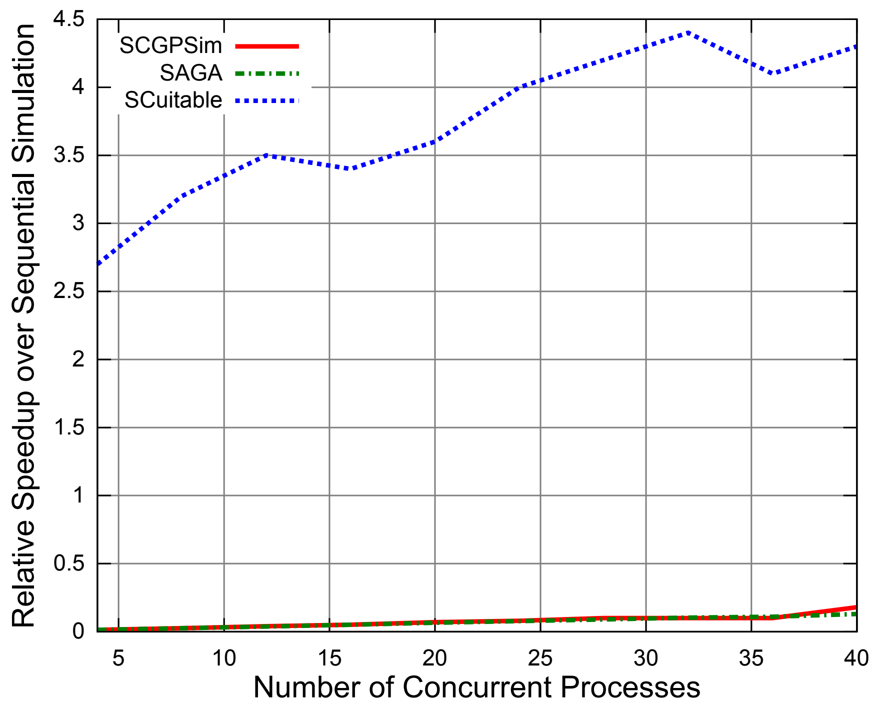


Figure 6.3: Comparison of relative speedups of simulations of SAGA, SCGPSim, and SCuitable over sequential SystemC simulation for Kausmi Cipher Algorithm.

case of SCuitable the overhead in spawning and synchronizing threads across multiple cores dominates the total execution time. Even on increasing in the number of concurrent processes, the simulation benefits of SCuitable do not come into effect as shown in Figure 6.4, which plots the relative speedups of the three approaches as the number of concurrent processes increases. Although the effect of context switching and load balancing do not contribute to the execution time of SCGPSim and SAGA approaches, both do not provide much simulation benefit due to the amount of computation done by a single compute core. However, for 1024-tap and 2048-tap FIR filters, SCuitable provides a speedup of 2 and 3 respectively over sequential simulation. This is because the amount of computation done by the processes is high, and therefore, the speedup benefit of distributing the computation across multiple CPUs diminishes the effect of context switching and thread spawning by the underlying operating system. With increase in the number of the concurrent processes, the simulation speedup offered by SCuitable increases to 4x for both types of filters. Although the number of threads executed on the CUDA core increases with the increase in the number of concurrent SystemC processes, SAGA and SCGPSim do not provide any simulation benefits when the number of concurrent SystemC processes is 50. This is because the intricate computation of the cipher algorithm assigned to a CUDA thread

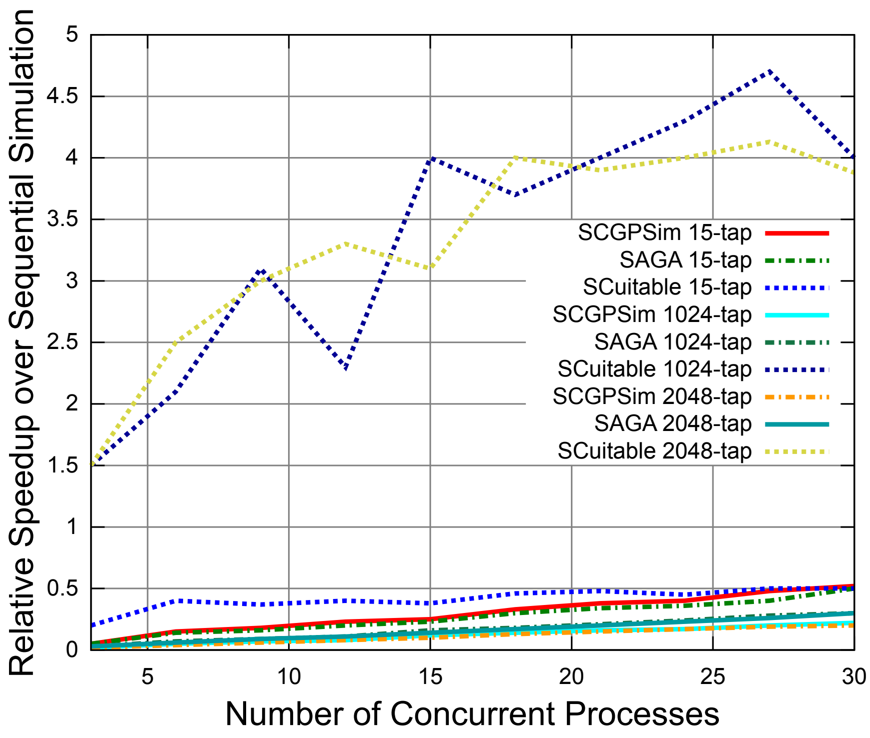


Figure 6.4: Comparison of relative speedups of simulations of SAGA, SCGPSim, and SCuitable over sequential SystemC simulation for different FIR filters.

overshadows the parallelism benefit. However, with further increase in the number of concurrent SystemC processes, the simulation benefits of SAGA and SCGPSim should be observed.

6.2.5 Sobel Filter

The Sobel Filter is used in image processing algorithms for edge detection. It consists of two 3$\times$3 operators that calculate the gradient and direction for each pixel. The Sobel filter algorithm is ideal for GPU execution as determining the gradient and direction for each pixel can be processed independently. Tables 6.17, 6.18, and 6.19 tabulate the speedups of the Sobel filtering algorithm for image sizes 128 $\times$ 128, 512$\times$512, and 1024$\times$1024 image dimensions respectively.

For each image dimension, the gradient and direction of an entire row of pixels is calculated in a single clock cycle. For the SCGPSim and SAGA approaches, the Sobel filter convolution computation is assigned to a single compute thread. As the image size increases, the convolution computation increases resulting in more work assigned to a CUDA thread. Therefore, both

Processes	SystemC	SAGA		SCGPSim		SCuitable	
	Execution Time (sec)	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup
3	8	253	0.03	232	0.03	8	1
6	17	253	0.07	232	0.07	18	0.9

Table 6.17: Comparison of SAGA, SCGPSim, and SCuitable for 128×128 Sobel Filter.

Processes	SystemC	SAGA		SCGPSim		SCuitable	
	Execution Time (sec)	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup
3	31	930	0.03	1015	0.03	15	2.1
6	64	930	0.07	1015	0.06	23	2.8

Table 6.18: Comparison of SAGA, SCGPSim, and SCuitable for 512×512 Sobel Filter.

SAGA and SCGPSim do not show good speedups across all image sizes. SCuitable on the other hand, maps the convolution computation to the GPU. Therefore, with increasing image sizes, the speedups obtained from SCuitable get better due to the abundant number of compute cores available for carrying out the convolution computation. For large images, SCuitable provides a speedup of 4 over sequential simulation. Figure 6.5 plots the relative speedup of the three approaches over sequential simulation with increasing number of concurrent SystemC processes. For clarity, Figure 6.6 plots only the SAGA and SCGPSim speedups with increase in number of concurrent SystemC processes. It can be observed that SCGPSim does better than SAGA with increasing number of SystemC processes as the number of threads launched for computation by SCGPSim is more than that of SAGA. For 30 concurrent processes, SAGA launches 10 threads while SCGPSim launches 30 threads for each of the 30 concurrent processes. SCuitable scales well with the number of concurrent SystemC processes; however its speedup is limited by the serialization of the CUDA kernel executions invoked by multiple simultaneous SystemC processes. For large images, SCuitable provides a speedup of 6.6 over sequential simulation when the number of concurrent processes is 30.

6.3 Acceleration of SystemC TLM models

In this section, I describe the simulation benefits SCuitable offers over the traditional uni-threaded SystemC scheduler for TLM designs. Since the parallel SystemC scheduler version is developed

Processes	SystemC	SAGA		SCGPSim		SCuitable	
	Execution Time (sec)	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup
3	60	2009	0.03	1864	0.03	18	3.3
6	124	2009	0.06	1864	0.07	31	4

Table 6.19: Comparison of SAGA, SCGPSim, and SCuitable for 1024×1024 Sobel Filter.

for SystemC-2.2, I modify the image processing case studies described in Section 6.1 to use TLM-1.0 constructs.

6.3.1 Canny Edge Detector

As described in Section 6.1, the Canny edge detection algorithm is a multi-stage edge detection algorithm. The stages of the algorithm are Gaussian blurring, Sobel filtering, non-maximum suppression, and hysteresis filtering. Each filter is implemented as a separate SystemC process. All of the above processes are suitable for GPU execution as these filters work on independent pixels. Tables 6.20 and 6.21 tabulate the execution times and speedup of the loosely-timed and approximately-timed models of the Canny edge detection algorithm respectively. The algorithm is executed on three different images of varying sizes. The sizes of the images used are 1024×1024, 2048×2048, and 4096×4096.

For the loosely-timed model, the filter operations on the image are performed one after the other. Each filter operation works on a sub-image and suspends itself for a duration of t time units by calling wait (t ,SC_NS). The amount of time that a filter suspends itself is adjusted based on the intensity of the computation. In this version of the Canny edge detection model, the Gaussian filter contains the most intensive computation and the non-maximum suppression filter has the least computation. Since the communication interface used is a blocking interface, each filter operation proceeds only when the channel connecting the modules is populated with data. For the LT model, SCuitable returns a mapping such that all the filter operations are executed on the GPU for all image sizes. From Table 6.20, the speedup obtained over sequential simulation is 4.5, 8.3, and 8.1 for small, medium, and large images respectively. To highlight the optimal mapping returned by SCuitable, Table 6.20 also tabulates the speedups obtained from other mappings.

For the AT model of the Canny edge detection algorithm, all the filter operations proceed in a pipelined fashion. For this AT model, SCuitable generates a mapping different from the LT model due to the presence of multiple data-parallel segments contending for the GPU at the same time. Across all different image dimensions, the Gaussian and Sobel filter operations are placed

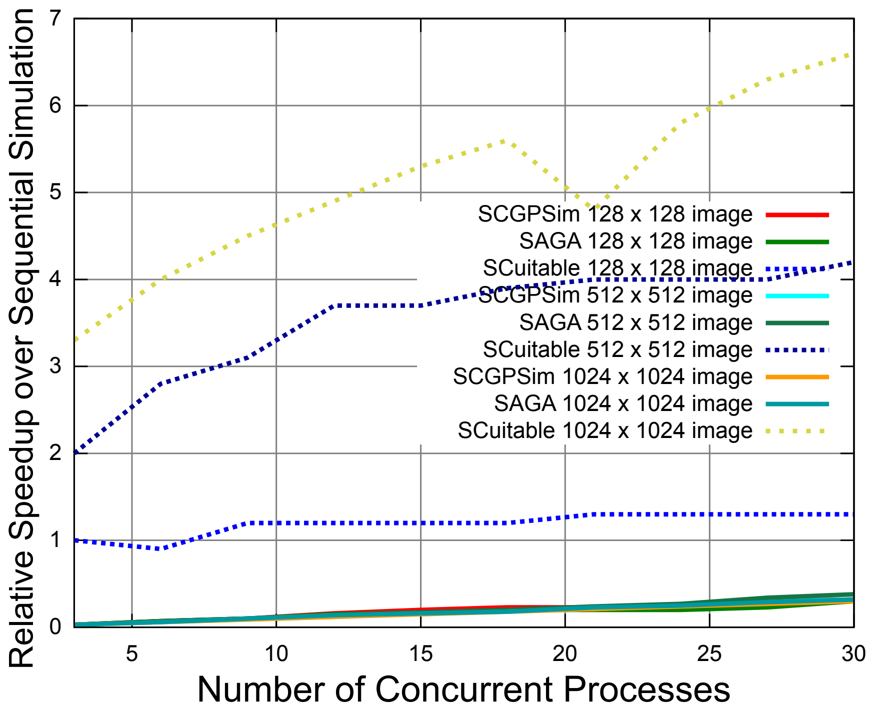


Figure 6.5: Comparison of relative speedups of simulations of SAGA, SCGPSim, and SCuitable over sequential SystemC simulation for Sobel filter on different image dimensions.

on the GPU and the non-maximum suppression and hysteresis filters are executed on the CPU. Table 6.21 tabulates the execution time and speedups of the approximately-timed model of the Canny edge detector. The mapping returned by SCuitable provides a speedup of 3.7, 4.1, and 4.9 over sequential simulation for small, medium, and large images. To highlight the optimality of the mapping returned by SCuitable, Table 6.21 also tabulates the speedups obtained for different mappings. As observed, the mapping returned by SCuitable provides the most benefit.

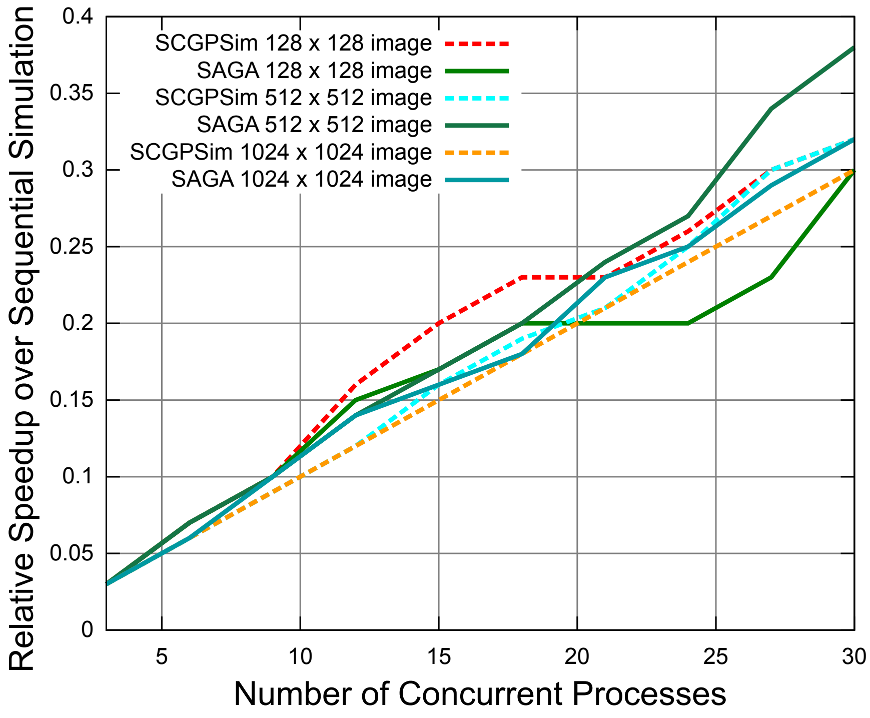


Figure 6.6: Comparison of relative speedups of simulations of SAGA, SCGPSim, and SCuitable over sequential SystemC simulation for different FIR filters.

6.3.2 JPEG Decoder

The JPEG Decoder takes as input a JPEG image and decompresses it into a BMP image. I develop two versions of the decoder: a color decoder and a black/white decoder. These two versions of the JPEG decoder are written in the LT modeling style. The stages of the decoder involve entropy decoding, dequantizing, inverse DCT, and color reordering. Tables 6.22 and 6.23 tabulate the speedup of the JPEG color and black/white decoders modeled in the loosely-timed coding style. The evaluation of the case study is carried out using three concurrent image streams of image dimensions 4506×3379, 1000×1000, and 200×200. The mapping returned by SCuitable places the dequantizing and color conversion stages of the JPEG decoder for execution on the GPU.

Table 6.22 tabulates the execution time and speedup of the LT JPEG color decoder model. The mapping returned by SCuitable provides a speedup of 9, 9.2, and 10.5 over sequential simulation for small, medium, and large image dimensions. To highlight the optimality of the mapping returned by SCuitable, Table 6.22 also tabulates the speedup of placing either the dequantizing

Image Dimensions	SystemC (sec)	SCuitable (sec)	SCuitable Speedup	Mapping 1 Speedup	Mapping 2 Speedup	Mapping 3 Speedup
Small	22	5	4.5	4.2	3.4	3
Medium	100	12	8.3	7.8	6.2	4.7
Large	480	59	8.1	7.5	5.2	3.7

Table 6.20: Execution times and speedup of LT model of Canny edge detection model.

Image Dimensions	SystemC (sec)	SCuitable (sec)	SCuitable Speedup	Mapping 1 Speedup	Mapping 2 Speedup	Mapping 3 Speedup
Small	26	6	4.3	1.9	2	3.7
Medium	100	21	4.5	2	2	4.1
Large	374	70	5.3	3.3	2	4.9

Table 6.21: Execution times and speedup of AT model of Canny edge detection model.

stage or the color conversion stage on the GPU shown as Mapping 1 and Mapping 2 respectively. Table 6.23 tabulates the execution time and speedup of the LT JPEG black/white decoder model. The mapping returned by SCuitable for this model is similar to that returned for the JPEG color decoder model. From Table 6.23, the speedup achieved by SCuitable over sequential simulation is 11 for small images, 9.5 for medium images, and 10.8 for large images. In comparison to the other possible mappings, the one SCuitable returns is optimal.

Effect of memory transfers between CPU-GPU

The slow PCI-Express bus connecting the CPU and GPU acts as an impediment to the execution time of applications running on the GPU. In order to mitigate the effect of slow memory transfer between the CPU and GPU, the CUDA-Gen plugin only copies data back to the CPU that have been modified in the data-parallel segments executed on the GPU. To highlight the effect of the data transfer overhead Figures 6.7, 6.8, 6.9, 6.10 show the contribution of the data transfer overhead towards the GPU execution time for the optimized and unoptimized versions of the TLM image processing case studies.

Figures 6.7 and 6.8 show the data transfer overhead for the unoptimized and optimized versions of the Canny edge detector model. In the unoptimized version of the Canny edge detection model, the contribution of the data transfer overhead to the GPU execution time for small, medium, and large image dimensions is 54%, 44%, and 40% respectively. The data transfer in the unoptimized version includes transferring read-only data from the GPU in addition to the

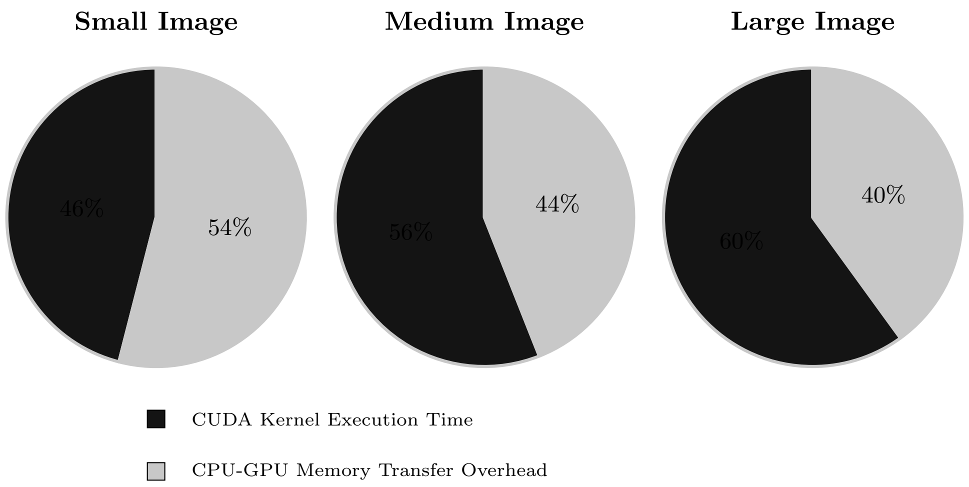


Figure 6.7: Data transfer overhead contribution to GPU execution in unoptimized Canny edge detection model.

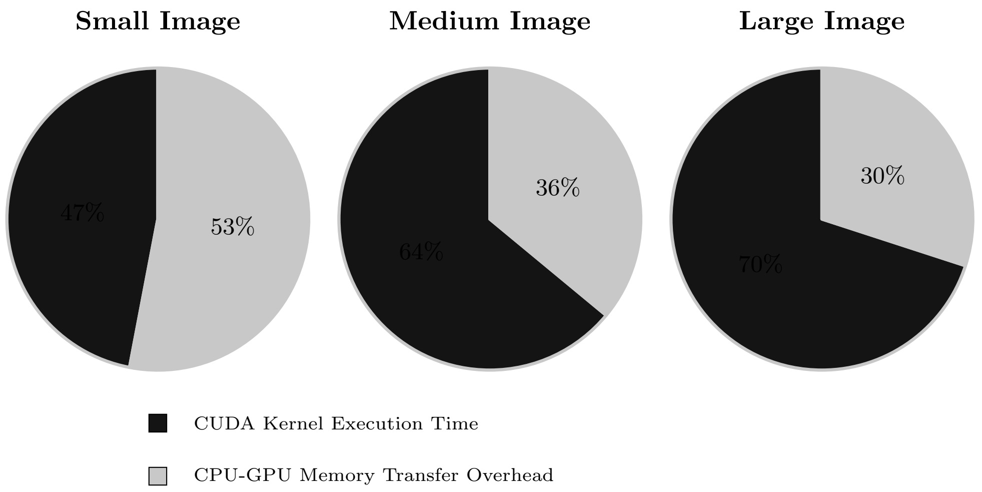


Figure 6.8: Data transfer overhead contribution to GPU execution in optimized Canny edge detection model.

Image Dimensions	SystemC (sec)	SCuitable (sec)	SCuitable Speedup	Mapping 1 Speedup	Mapping 2 Speedup
Small	45	5	9	7.5	1.7
Medium	130	14	9.3	7.4	1.8
Large	452	43	10.5	9.8	1.9

Table 6.22: Execution times and speedup of LT model of JPEG color decoder.

Image Dimensions	SystemC (sec)	SCuitable (sec)	SCuitable Speedup	Mapping 1 Speedup	Mapping 2 Speedup
Small	42	3	11	8.5	1.9
Medium	124	13	9.5	6	2.1
Large	420	42	10.8	7.8	1.5

Table 6.23: Execution times and speedup of LT model of JPEG black/white decoder.

modified data. On optimizing the model to copy only modified data back to the CPU, the contribution of the data transfer overhead reduces to 53%, 36%, and 30% for small, medium, and large images respectively. The performance benefits of the optimization has a drastic impact on the simulation of the Canny edge detection model for large and medium images.

Figures 6.9 and 6.10 show the contribution of the data transfer overhead to the GPU execution time for the unoptimized and optimized versions of the JPEG decoder model. In the unoptimized version of the model, the data transfer overhead contributes significantly to the total GPU execution time. For small, medium, and large images, the data transfer overhead contributes to 86%, 93%, and 96% of the total GPU execution time. However, optimizing the design to transfer only the modified data to the CPU results in a reduction of the contribution of the data transfer overhead to 90% and 74% for medium and large images respectively. For small images, the optimization does not affect the contribution of the data transfer overhead.

6.4 Summary

From the results, it can be observed that the type of computation executed on the GPU impacts the total execution time of the simulation of SystemC models. Due to the simple in-order pipelined architecture of a CUDA compute core, executing a code with intricate control-flow characteristics can perform worse on the GPU. SCGPSim and SAGA do not take into consideration the program characteristics beneficial for execution on the GPU. These methods use the

GPU for task parallelism by executing each SystemC process or data-flow on a CUDA compute core. For the above benchmarks, with the exception of the 3-stage pipeline, SAGA and SCGPSim do not provide simulation benefits due to the amount of computation carried out by a CUDA thread. However, it is important to note, that SCGPSim and SAGA should perform better for pipelined SystemC models. For instance, if the AES cipher SystemC model is modified such that the ten rounds required for encoding a plain-text key are pipelined, SCGPSim and SAGA should provide better simulation benefits as the number of SystemC processes increases,thereby increasing the opportunity to use more CUDA compute cores for computation. SCuitable on the other hand, decides on a mapping of th execution of a SystemC model based on the control flow characteristics. For all the benchmarks mentioned in the previous section with reasonable amount of computation, SCuitable provides simulation benefits over the sequential simulation. By utilizing the multiple CPUs for executing ready-to-run concurrent tasks, and the GPUs for performing data-parallel computations, both task parallelism and data-parallelism present in the SystemC model are exploited.

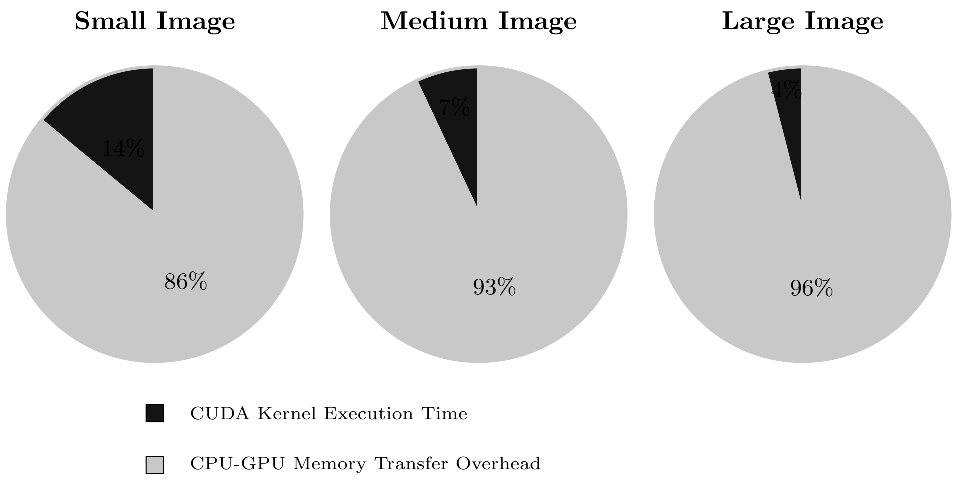


Figure 6.9: Data transfer overhead contribution to GPU execution in unoptimized JPEG decoder model.

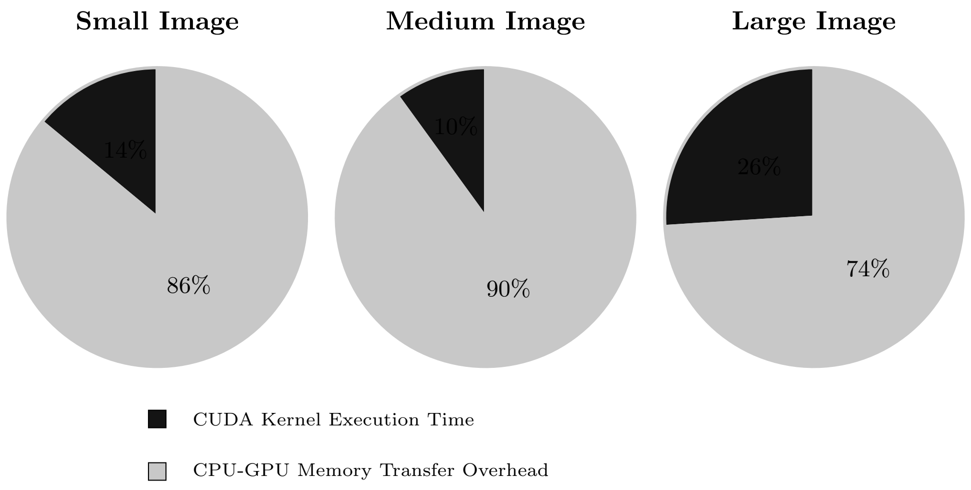


Figure 6.10: Data transfer overhead contribution to GPU execution in optimized JPEG decoder model.

Chapter 7

Future Work and Conclusions

In this chapter, I describe the potential future work extensions to this thesis, and provide a summary of the work carried out in this thesis.

7.1 Future Work

In this thesis, I investigated the use of multi-core CPUs and GPUs for accelerating mixed-abstraction SystemC simulations. The proposed method analyzes a SystemC model using the systemc-clang framework, and decides a mapping of the SystemC model for execution across multiple core CPUs and GPUs using a parallelized version of the SystemC scheduler. In the following subsections, I list some opportunities for future work using the framework developed in this thesis.

7.1.1 NVIDIA Kepler GPU Architectures

The new Kepler GPU architecture [54] provides a variety of interesting features in addition to an increase in the number of compute cores that can be exploited for further accelerating the simulation of SystemC models.

Dynamic Parallelism

Previous generations of GPUs including the one used in the experimentation for this thesis, do not allow CUDA threads on the GPU to spawn further CUDA threads. This results in an under

Image Size	SCGPSim			
	Tesla GPU		Kepler GPU	
	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup
Small	232	0.03	11	0.7
Medium	930	0.03	16	1.9
Large	1864	0.03	22	2.7

Image Size	SAGA			
	Tesla GPU		Kepler GPU	
	Execution Time (sec)	Speedup	Execution Time (sec)	Speedup
Small	253	0.03	16	0.5
Medium	1015	0.03	24	1.3
Large	2009	0.03	32	1.9

Table 7.1: Preliminary results of executing the SystemC model of Sobel filter on Kepler GPUs using SCGPSim and SAGA approaches.

utilization of the GPU device as not all compute cores are utilized for execution. The dynamic parallelism feature in the Kepler GPU architecture [55] allows CUDA threads executing on the GPU to spawn further threads for computation on the GPU. In other words, dynamic parallelism provides the ability to perform both task parallelism and data parallelism simultaneously. The previous methodologies of SCGPSim and SAGA use the GPU for carrying out task parallelism by allocating a CUDA thread for a SystemC process. In Section 6.2, it was observed that for the RTL benchmarks with reasonable amount of computation done per clock cycle, SAGA and SCGPSim performs poorly. In particular, for SystemC models that consist of data-parallel segments such as the Sobel filter, SAGA and SCGPSim do not utilize the GPU for exploiting the data parallelism present in the computation but allocate a single CUDA thread to handle the data-parallel computation. However, with dynamic parallelism, the CUDA thread responsible for the Sobel filter computation can now launch further CUDA threads to exploit the data parallel computation present on the GPU. Preliminary results for executing the SystemC model of the Sobel filter on the Kepler GPU using the approaches of SAGA and SCGPSim are shown in Table 7.1. The execution times and speedups tabulated in Table 7.1 reflect the benefits of combining the dynamic parallelism feature with the approaches of SAGA and SCGPSim.

It can be observed that the approaches of SAGA and SCGPSim do drastically better on the Kepler GPUs than on the Tesla GPUs as the data parallel section of the Sobel Filter handled by the CUDA thread can spawn further threads thereby parallelizing the data-parallel segment on the GPU. With dynamic parallelism, the speedup improvement of SCGPSim and SAGA over that of running on the Tesla GPU is 89% and 62% respectively. Therefore, one of the extensions I envision for this work is determining which technique (SCGPSim, SAGA, and SCuitable) to use based on the specifications of the heterogeneous computation platform provided. This would require integrating the algorithms and methodologies of SCGPSim and SAGA as plugins to systemc-clang and querying the specifics of the heterogeneous system before determining a mapping of the SystemC model for execution.

Memory Optimizations

In this thesis, I assume that the data-parallel segments used for computation on the GPU involve array sizes that can fit in the global memory of the GPU device. However, this may not always be true as it is quite likely that industrial grade SystemC models deal with data sizes larger than the allowable GPU global memory size. Note that previous methodologies of SCGPSim and SAGA also implicitly assume that the data set required for computation is within the limits of the global memory space available on the GPU. The problem of executing general models on GPUs that require more memory than that available on the GPU has been studied in [56]. I intend to integrate their approach and modify it for co-simulating SystemC models across multi-core CPUs and GPUs. Moreover, with the introduction of read-only data caches in the Kepler GPU architecture, constant references can be allocated on this cache resulting in faster access time. Therefore, there are opportunities for memory optimizations that utilize the GPU memory hierarchy in a much efficient way.

Concurrent Kernel Execution from Different CPU Threads

Previous generations of GPUs consist of only one connection between the cluster of CPUs and the shared GPU. This results in a serialized execution of the CUDA kernels that are simultaneously invoked by different CPU threads. Recall from Section 5.1.4 that the heuristic algorithm adopted by SCuitable for generating a mapping of SystemC model for execution across multiple CPU cores and GPUs takes into account of this serialization on the GPU. However, the Kepler architecture provides an improved feature termed Hyper-Q [55], that provides 32 connections between the cluster of CPUs and the shared GPU resulting in concurrent kernel execution on the GPU from different CPU cores. Using this feature improves the opportunity for executing more data-parallel segments on the GPU for SCuitable as different CPU cores with overlapping data-parallel segments can now utilize the GPU.

7.1.2 CUDA library for TLM-2.0 transport interfaces

The CUDA kernel invocation by the host can be made to resemble the operation of the TLM-2.0 transport interfaces. For instance, CUDA kernels by default are asynchronous in nature. In other words, once a CUDA kernel is invoked, the program control returns back to the host. This is similar to the non-blocking transport interface provided in the TLM-2.0 library wherein the initiator transports the payload information to the target and program control immediately returns back to the initiator as the non-blocking transport interface cannot contain wait() calls. Therefore, an asynchronous CUDA kernel invocation is similar to the non-blocking interface with the host as initiator and the GPU device as target. On the other hand, a CUDA kernel call can be made blocking by calling the cudaDeviceSynchronize() function that explicitly blocks the host from executing until the CUDA kernel has completed its computation on the GPU. Therefore, through the cudaDeviceSynchronize() call, a CUDA kernel invocation can be made to behave as TLM-2.0 transport interfaces. Combining the above with the CUDA library developed by Sinha et al. [17] for carrying out wait() and notify() calls on the GPU, a CUDA library for TLM-2.0 interfaces is possible. Any transport interface operations on the transaction payload that are data-parallel can be exploited on the GPU using such a library.

7.1.3 Support for APUs and OpenCL

The PCI-Express bus connecting the CPU and GPU is a bottleneck that affects the execution time of an application executing on the GPU. Towards the end of Section 6.3, I highlighted the contribution of the memory copy transfer overhead between the CPU and GPU across the PCI-Express bus to the total execution t ime on the GPU. It is quite possible that the benefit of the fast execution of the main kernel computation on the GPU is diminished by the data transfer overhead between the CPU and GPU. Therefore, there are variants of heterogeneous systems that combine the GPU and CPU on a single chip. Such heterogeneous systems are termed as Accelerated Performance Units (APUs). The AMD Fusion is an example of an APU. APUs resolve the issue of memory transfer overhead at the cost of fewer compute cores. There is almost negligible data transfer overhead as the memory is shared between the CPU and GPU. However, the number of compute cores available for execution is reduced compared to a standalone or dedicated graphics card. I intent to support this framework for APUs as well by modifying the optimization goals appropriately and developing a new systemc-clang translation plugin for the OpenCL programming framework, which is used to program the AMD APUs.

7.2 Conclusion

In this thesis, I investigate the use of multi-core CPUs and GPUs for co-simulating mixed-abstraction SystemC models. Previous approaches ([15] and [16]) proposed methods for accelerating RTL SystemC models by assigning each SystemC process to a CUDA thread or each independent data-flow to a CUDA thread block respectively. These efforts do not take into consideration the code structure of the program as CUDA cores are simple in nature and therefore, handle intensive control flow code poorly compared to CPUs. In this thesis, I work on improving the approach proposed by Sinha et al. [17], which also proposes accelerating TLM SystemC models, by identifying segments of the SystemC model can be executed on the GPU and multi-core CPUs. For this, I develop an open-source static framework called systemc-clang that analyzes RTL and TLM SystemC models, and systemc-clang plugins SCuitable and CUDA-Gen that are responsible for generating a mapping of data-parallel segments to be executed on the multi-core CPUs and GPUs and performing the source-to-source translation of the identified data-parallel segments to CUDA respectively. The SystemC scheduler used by the SCuitable framework is a parallelized SystemC scheduler with support for TLM SystemC models. For SystemC RTL designs with reasonable computation, SCuitable does better over the previous methods of SAGA and SCGPSim, and for TLM models, SCuitable provides a maximum speedup of 11 over sequential simulation.

References

[1] J. Bergeron, *Writing testbenches: functional verification of HDL models*. Kluwer Academic Publishers, 2000.

[2] T. Ziehe, “Software process improvement (spi) guidelines for improving software,” 1996.

[3] Open SystemC Initiative, “SystemC.” http://www.systemc.org.

[4] “IEEE Standard SystemC Language Reference Manual,” *IEEE Std 1666-2005*, 2006.

[5] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, “parSC: Synchronous Parallel SystemC Simulation on Multi-core Host Architectures,” in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS ’10, (New York, NY, USA), pp. 241–246, ACM, 2010.

[6] B. Chopard, P. Combes, and J. Zory, “A conservative approach to systemc parallelization,” in *Proceedings of the 6th international conference on Computational Science - Volume Part IV*, ICCS’06, (Berlin, Heidelberg), pp. 653–660, Springer-Verlag, 2006.

[7] E. P, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, “Parallelizing SystemC kernel for fast hardware simulation on SMP machines,” in *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS ’09, (Washington, DC, USA), pp. 80–87, IEEE Computer Society, 2009.

[8] A. Mello, I. Maia, A. Greiner, and F. Pecheux, “Parallel simulation of SystemC TLM 2.0 compliant MPSoC on SMP workstations,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE ’10, (3001 Leuven, Belgium, Belgium), pp. 606–609, European Design and Automation Association, 2010.

[9] P. Combes, E. Caron, F. Desprez, B. Chopard, and J. Zory, “Relaxing synchronization in a Parallel SystemC Kernel,” *Parallel and Distributed Processing with Applications, International Symposium on*, vol. 0, pp. 180–187, 2008.

[10] S. Jones, "Optimistic parallelisation of SystemC," tech. rep., Universite Joseph Fourier: MoSiG DEMIPS, 2011.

[11] B. Chopard, P. Combes, and J. Zory, "A Conservative Approach to SystemC Parallelization," in *Computational Science ICCS 2006* (V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra, eds.), vol. 3994 of *Lecture Notes in Computer Science*, pp. 653–660, Springer Berlin / Heidelberg, 2006.

[12] D. Chatterjee, A. DeOrio, and V. Bertacco, "Event-driven gate-level simulation with GP-GPUs," in *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, (New York, NY, USA), pp. 557–562, ACM, 2009.

[13] B. Wang, Y. Zhu, and Y. Deng, "Distributed time, conservative parallel logic simulation on GPUs," in *Proceedings of the 47th Design Automation Conference*, DAC '10, (New York, NY, USA), pp. 761–766, ACM, 2010.

[14] Y. Zhu, B. Wang, and Y. Deng, "Massively parallel logic simulation with GPUs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, pp. 29:1–29:20, June 2011.

[15] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla, "SCGPSim: a fast SystemC simulator on GPUs," in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10, (Piscataway, NJ, USA), pp. 149–154, IEEE Press, 2010.

[16] S. Vinco, D. Chatterjee, V. Bertacco, and F. Fummi, "SAGA: SystemC acceleration on GPU architectures," in *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, (New York, NY, USA), pp. 115–120, ACM, 2012.

[17] R. Sinha, A. Prakash, and H. D. Patel, "Parallel Simulation of Mixed-abstraction SystemC Models on GPUs and Multicore CPUs," in *proceedings of ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 455–460, IEEE, 7 2012.

[18] Anirudh Mohan Kaushik, "A SystemC Parser using clang." http://anikau31.github.io/systemc-clang.

[19] B. Schaffer, "S2cbench: Synthesizable SystemC Benchmark Suite." http://www.s2cbench.org/.

[20] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the ground up*. Springer, 2004.

[21] F. Ghenassia *et al.*, *Transaction-level modeling with SystemC*. Springer, 2005.

[22] O. S. Initiative, "OSCI TLM-2.0 user manual," *online] http://www. systemc. org*, 2008.

[23] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate CPU vs. GPU performance without the answer," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pp. 134–144, IEEE, 2011.

[24] M. Moy, "Parallel programming with SystemC for loosely timed models: A non-intrusive approach," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, (San Jose, CA, USA), pp. 9–14, EDA Consortium, 2013.

[25] R. M. Fujimoto, "Parallel discrete event simulation," *Commun. ACM*, vol. 33, pp. 30–53, Oct. 1990.

[26] W. Chen, X. Han, and R. Domer, "Out-of-order parallel simulation for esl design," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pp. 141–146, march 2012.

[27] R. Dömer, W. Chen, X. Han, and A. Gerstlauer, "Multi-core parallel simulation of system-level description languages," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, ASPDAC '11, (Piscataway, NJ, USA), pp. 311–316, IEEE Press, 2011.

[28] R. Dömer, W. Chen, X. Han, and A. Gerstlauer, "Multi-core parallel simulation of system-level description languages," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, ASPDAC '11, (Piscataway, NJ, USA), pp. 311–316, IEEE Press, 2011.

[29] M. Nanjundappa, A. Kaushik, H. Patel, and S. Shukla, "Accelerating SystemC simulations using GPUs," in *High Level Design Validation and Test Workshop (HLDVT), 2012 IEEE International*, pp. 132–139, Nov 2012.

[30] V. Bertacco, D. Chatterjee, N. Bombieri, F. Fummi, S. Vinco, A. Kaushik, and H. D. Patel, "On the use of GP-GPUs for accelerating compute-intensive EDA applications," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pp. 1357–1366, March 2013.

[31] W. Snyder, "SystemPerl-a perl library for SystemC," 2006.

[32] FZI Microelectronic System Design, "KaSCPar - Karlsruhe SystemC Parser Suite." http://www.fzi.de/index.php/de/component/content/article/238-ispe-sim/4350-sim-tools-kascpar-examples.

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

[34] D. Berner, J.-P. Talpin, H. D. Patel, D. Mathaikutty, and S. K. Shukla, “SystemCXML: An exstensible SystemC front end using XML.,” in *FDL*, pp. 405–409, Citeseer, 2005.

[35] R. Drechsler, G. Fey, C. Genz, and D. Grosse, “SyCE: an integrated environment for system design in SystemC,” in *Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on*, pp. 258 – 260, june 2005.

[36] 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*, pp. 317–324, ACM, 2005.

[37] K. Marquet and M. Moy, “Pinavm: a Systemc front-end based on an executable intermediate representation,” in *Proceedings of the tenth ACM international conference on Embedded software*, pp. 79–88, ACM, 2010.

[38] K. Marquet, M. Moy, and B. Karkare, “A theoretical and experimental review of SystemC front-ends,” 2010.

[39] N. Bombieri, G. D. Guglielmo, M. Ferrari, F. Fummi, G. Pravadelli, F. Stefanni, and A. Venturelli, “HIFsuite: tools for hdl code conversion and manipulation,” *EURASIP J. Embedded Syst.*, vol. 2010, pp. 4:1–4:20, Jan. 2010.

[40] C.-K. Luk, S. Hong, and H. Kim, “Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping,” in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 45–55, IEEE, 2009.

[41] Z. Wang, L. Zheng, Q. Chen, and M. Guo, “CAP: co-scheduling based on asymptotic profiling in CPU+GPU hybrid systems,” in *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM ’13, (New York, NY, USA), pp. 107–114, ACM, 2013.

[42] C. Gregg, J. Brantley, and K. Hazelwood, “Contention-aware scheduling of parallel code for heterogeneous systems,” in *2nd USENIX Workshop on Hot Topics in Parallelism*, HotPar, (Berkeley, CA), June 2010.

[43] D. Grewe and M. F. OBoyle, “A static task partitioning approach for heterogeneous systems using opencl,” in *Compiler Construction*, pp. 286–305, Springer, 2011.

[44] M. Goli, J. McCall, C. Brown, V. Janjic, and K. Hammond, "Mapping parallel programs to heterogeneous CPU/GPU architectures using a monte carlo tree search," in *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pp. 2932–2939, IEEE, 2013.

[45] T. Diop, S. Gurfinkel, J. Anderson, and N. E. Jerger, "Distcl: A framework for the distributed execution of OpenCL kernels," in *Proceedings of the 2013 IEEE 21st International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, MASCOTS '13, (Washington, DC, USA), pp. 556–566, IEEE Computer Society, 2013.

[46] A. Kaushik and H. D. Patel, "SystemC-clang: An open-source framework for analyzing mixed-abstraction SystemC models," in *Specification Design Languages (FDL), 2013 Forum on*, pp. 1–8, Sept 2013.

[47] LLVM Team, "The LLVM Compiler Infrastructure." http://www.llvm.org.

[48] clang Team, "clang: a C language family Frontend for LLVM." http://www.clang.llvm.org.

[49] N. Harrath and B. Monsuez, "SystemC Waiting State Automata," *Int. J. Crit. Comput.-Based Syst.*, vol. 3, pp. 60–95, Jan. 2012.

[50] LLVM Team, "The LLVM Compiler Infrastructure." http://polly.llvm.org/.

[51] N. Bombieri, F. Fummi, V. Guarnieri, F. Stefanni, and S. Vinco, "HDTLib: an efficient implementation of SystemC data types for fast simulation at different abstraction levels," *Design Automation for Embedded Systems*, vol. 16, no. 2, pp. 115–135, 2012.

[52] J. Aynsley, "A complete AT SystemC TLM example," 2009. https://www.doulos.com/knowhow/systemc/tlm2/at_example/.

[53] W Chen, "A JPEG Colour Encoder in SystemC." https://github.com/weiweichen/systemc-jpeg.

[54] NVIDIA, "Nvidia's next generation cuda compute architecture : Kepler gk110," 2012. http://www.nvidia.ca/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[55] NVIDIA, "CUDA Dynamic Parallelism Programming Guide," 2012. https://developer.nvidia.com/.

[56] N. Satish, N. Sundaram, and K. Keutzer, “Optimizing the use of GPU memory in applications with large data sets,” in *High Performance Computing (HiPC), 2009 International Conference on*, pp. 408–418, Dec 2009.

[57] S. Fine and A. Ziv, “Coverage directed test generation for functional verification using bayesian networks,” in *Design Automation Conference, 2003. Proceedings*, pp. 286–291, June 2003.