View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

© 2016 Warren Hargon Kemmerer

PARALLEL CODE-SPECIFIC CPU SIMULATION WITH DYNAMIC PHASE CONVERGENCE MODELING FOR HW/SW CO-DESIGN

BY

WARREN HARGON KEMMERER

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Deming Chen

ABSTRACT

While SystemC models provide a promising solution to the complex problem of HW/SW co-design within the system-on-chip paradigm, such requires a detailed annotation of transaction level energy and performance data within the model. While this data can be obtained through source code profiling of an application running on the target processor, accomplishing such when the target CPU hardware is not actively available typically requires time-consuming CPU simulation, which is often too slow to practically consider for large programs. Additionally, while the use of SystemC modeling with TLM 2.0 standard is widely adopted for the SoC modeling, the process of transforming C/C++ code to SystemC code with TLM 2.0 functionality remains non-trivial. Herein we propose an automated framework that:

1. Enables high speed code-specific CPU profiling support for both Sniper and gem5 using parallelized dynamic steady state phase convergence modeling, providing automatic annotation of energy and performance within source code.

2. Provides an automated C to SystemC TLM 2.0 code generation flow that utilizes the back-annotated source code to produce a SystemC module for seamless incorporation into the virtual prototype.

Maximum speedups obtained using Sniper and gem5 are 48.76x and 562x respectively, while average results obtained speedups of 31.5x and 323.1x. Sniper results maintain an average accuracy of 0.89% for latency and 0.10% for energy, while gem5 achieves average accuracies of 4.16% and 2.87% for latency and energy respectively.

To my wife, for her support and patience.

ACKNOWLEDGMENTS

This work was done in collaboration with Intel, with Taemin Kim and Andrey Ayupov as primary collaborators. We also acknowledge the collaboration of Louis Noel Pouchet, whose provided Polybench test suite serves as an integral part of our flow experimental results.

Elements of this research were performed in conjunction with a class project with Wei Zuo. This work was later submitted as a conference paper with coauthors Wei Zuo and Deming Chen. Within the context of this paper, my work has focused primarily on simulation speedup through convergence modeling while elements pertaining to back-annotated C to SystemC translation were performed by Wei Zuo. I have included the latter as it provides both context and completeness to the work.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS vi

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND AND RELATED WORK 3
2.1 Software Modeling Strategies 3
2.2 Simulation Tools 4
2.3 Simulation Modes 4
2.4 Transaction-level Modeling 6

CHAPTER 3 METHOD 8
3.1 Software Modeling 8
3.2 Energy vs. Power 9
3.3 Simulator Dependent ROI Division 9
3.4 Parallel ROI Profiling 10
3.5 Phase Convergence 12
3.6 Instruction Count Normalization 12
3.7 N-way Convergence Sampling 13
3.8 Thread Scheduling 13
3.9 Automated Back-Annotation 17
3.10 SystemC and TLM Generation 17
3.11 The Hierarchy of the Generated Code 18
3.12 The Code Generation Flow 19

CHAPTER 4 EXPERIMENTS AND RESULTS 21
4.1 Experimental Setup 21
4.2 Results 24

CHAPTER 5 CONCLUSION AND FUTURE WORK 33

REFERENCES 34

LIST OF ABBREVIATIONS

CPU	Central Processing Unit
ISA	Instruction Set Architecture
KVM	Kernel-based Virtual Machine
pFSA	Parallel Full Speed Ahead
ROI	Region of Interest
RTL	Register-Transfer Level
SoC	System on a Chip
TLM	Transaction Level Modeling

CHAPTER 1

INTRODUCTION

Determining an efficient partitioning between hardware and software generally requires accurate means of modeling the performance characteristics associated with running the customized software on the target platform [1]. While such modeling can occur at the RTL or even gate level, simulation times associated with these methodologies are often too slow due to the complexity associated with the low level design [2, 3]. For realistic applications, higher level SystemC models of hardware provide a promising solution through the abstraction of low level complexity while preserving the relative accuracy necessary to enable design decisions. However, running customized code on fully detailed CPU SystemC models incorporated into virtual prototypes (the high-level modeling stage of the SoC system before committing to a physical prototype) remains excessively time-consuming when considering large programs. A promising alternative involves the generation of code-specific SystemC modules which execute at host speed within the virtual prototype and require only relevant energy and performance information at target code locations. The process of extracting performance data from a lower, more accurate representation and returning this information for incorporation into the corresponding high-level model is known as *back-annotation*. Back-annotation enables a natively inaccurate higher level model to inherit the effects of details available only at the lower level, thereby resulting in the increased accuracy of the higher level model while maintaining a sufficient level of abstraction to enable relatively faster run-times. Current means of extracting performance and energy data for back-annotation generally involve a detailed CPU simulation of the source code, the entirety of which is often too slow for practical consideration for large programs.

We herein present dynamic phase convergence modeling in conjunction with parallelized region-of-interest (ROI) profiling as a technique for improving the efficiency of the performance extraction process within the context

of energy and performance extraction for SystemC modules. While several techniques already exist for achieving significant speedup with regard to code profiling, such generally operates at a granularity level that is inappropriate for the back-annotation process. Our technique specifically targets applications with iterative code constructs commonly considered for candidacy within custom accelerators, which coincidentally also require a significant majority of simulation time. We further provide a C/C++ to SystemC framework whereby the automatically back-annotated source code is ported into usable SystemC modules with TLM 2.0 support. These modules can be integrated into virtual prototypes providing an effective means of conducting early stage hardware/software co-design. In summary, our work contains following novel contributions:

1. An automated framework providing annotation of energy and performance within source code through code-specific CPU profiling support for both Sniper and gem5 using parallelized dynamic steady state phase convergence modeling attaining average speedups of 31.5x and 323.1x respectively.

2. An automated SystemC TLM 2.0 code generation framework to generate SystemC for the C code to enable seamless incorporation of the software modeling to the SoC virtual platform.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Software Modeling Strategies

We herein discuss several existing simulation strategies for the effective acquisition of energy and performance data.

Simpoint [4] utilizes advanced code analytics to achieve high level phase modeling that enables accurate estimation of aggregate performance statistics of large programs. While useful in providing complete run times of large programs, the methodology is not well suited in determining performance results within fine-grained sub-regions, as required for meaningful SystemC back annotation. Our target granularity is several orders of magnitude too small for our Simpoint application, noting specifically that the minimum interval of a single sample is generally larger than the entirety of the regions that we are attempting to classify.

Region-of-interest (ROI) code profiling [5, 6] acquires performance data associated with only a target region of code while preserving correctness relative to the entire program. Detailed simulation or performance estimation of the entire program is often unnecessary, especially in instances of hardware/software co-design in which regions such as variable instantiation and assignment can be ignored as only the computational region of code is under consideration. Region-of-interest code profiling relies on simulator capabilities for efficient fast forwarding and a cache warm-up, and is only effective as a standalone extraction method when the region is small relative to code preceding the region.

Sampling methodologies such as SMARTS [7] use statistical sampling over a provided range to characterize code regions. Within [8] the authors introduce a new hardware accelerator enabled means of quickly fast forwarding through non-meaningful regions of code. This drop in fast forwarding tim-

ing overhead enables the development of the pFSA sampling methodology, in which sequential code can be sampled in parallel at the expense of simulation redundancy, with the results aggregated into a single profile. However, results obtained remain sequential with respect to time while ultimate speedup is tightly bound by the degree of parallelism.

Our simulation methodology seeks to combine the techniques of pFSA simulation with dynamic phase modeling analysis through the use of parallelized non-adjacent region of interest profiling. Using active feedback, we seek to acquire simulation speedup through profiling only the select region-of-interest required by our model to achieve relative phase convergence.

2.2 Simulation Tools

Within the academic community, **Sniper** [5] and **gem5** [2] are widely adopted simulation tools. Sniper is a high-speed x86 simulator that provides modes of operation for fast-forwarding, cache-warming, and detailed profiling. Sniper is highly attractive due to its native support for iterative ROI simulation and incorporation of power characterization software (McPAT [9]). Gem5 offers cycle-accurate simulation for a wide range of ISAs and provides highly customizable hardware configurations. Gem5 also provides several simulation modes of operation, with a notable recent addition of hardware enabled fast forwarding [8]. Nevertheless, detailed profiling of large programs within both simulators remains a time-consuming process.

2.3 Simulation Modes

Both Sniper and gem5 offer several alternative simulation models that serve as a means whereby the simulation can quickly fast forward through the prefix code necessary to maintain program correctness. Sniper offers a functional only mode, with a speedup of 100x relative to detailed simulation, while also offering a cache only mode, with a speedup of 7x relative to detailed simulation. Switching between these modes is accomplished through simulator control scripts which interface with macro functions placed within the source code.

Within gem5, the options available for fast forwarding and cache warmup are noticeably more complex. Rather than selecting a simulation mode, the user specifies both a CPU model and a memory model. Traditionally available CPU models include Atomic, Timing-Simple, and Out-of-Order Detailed. The Out-of-Order Detailed CPU model is an accepted standard model for detailed CPU profiling within academia, providing cycle accurate results that have been verified against real world CPUs. Both Atomic CPU and Timing_Simple CPU models are lightweight in-order processor models; however, they provide different levels of complexity. Atomic CPU preserves functional correctness of code execution, but lacks timing profiling capabilities. Timing_Simple CPU processing timing profiling capabilities, but would result in significant error if used to profile a modern out-of-order processor, resulting in the latter model being more accurate than the former. Gem5 provides the means whereby these processor models can be swapped interchangeably at will, although control scripts might need to be modified to support the desired configurations.

Memory models can also be configured at various levels of complexity within gem5. The simplest form of memory, FastMem, is used solely for fast-forwarding, and assumes no cache structure in addition to instantaneous data transfer between the processor and main memory. While these assumptions are generally completely invalid for data profiling, this model provides a convenient means whereby program control flow can be correctly maintained. Speedup is achieved by removing complex bus and cache processes from the simulation paradigm while simultaneously skipping cycles in which the CPU would need to stall for memory accesses. As gem5 does provide the means whereby each cache level can be initialized, it would feasibly be possible to construct various levels of memory layouts, each of which provided a degree of speedup comparable to the level of complexity approximated into the design. However, in practice, memory configurations are most useful in an all or nothing state. For a given configuration, either all cache values can be ignored for fast speedup, or cache values are considered important and all values need to be monitored. This is inherent in the inter-dependencies of cache levels of the state of the other cache levels. Therefore, within the context of our simulations, we do not employ partial cache warming. Instead, we ensure that all applications of cache warm-up consider all cache levels.

Recent developments [8] introduce an alternative CPU model for fast

forward applications. This model, known as a kernel vitalization machine (KVM) CPU model, relies on hardware vitalization capabilities enabled in most modern server and desktop CPUs. Rather than simulating the entire CPU structure with associated memory or system functions, this model allocates a sub-section of the CPU and memory resources of the host computer, configuring it with the necessary parameters to enable direct translation between host and target platforms. After launching allocation of these resources, the simulator launches the target application for a designated number of instructions on the virtual machine. Once the virtual machine reaches the target instruction count, it returns the active state of the application to the simulator. At this point the simulator can switch CPU models to either a detailed or cache warming model all while inheriting the correct program state as determined by the virtual machine. It is important to note that this virtual machine is only useful in preserving processor state necessary for correct control flow (i.e. processor registers and memory values). No mechanism exists to enable the transfer of cache state between virtual machine and simulated CPU, and even if such existed, it would likely prove highly inaccurate as the virtual machine inherits cache properties of the host machine and cannot be arbitrarily set to a customized configuration. As a result, the caches inherited by the simulation CPU following a KVM execution are empty, and cache warming period is recommended to preserve at the very least correct low level cache functionality. Therefore, the application of KVM CPUs is limited to fast-forwarding, although the contribution of providing a means whereby fast forwarding can occur at near host speeds (approximately 19,000x faster than detailed simulation) cannot be understated.

2.4 Transaction-level Modeling

Transaction-level modeling (TLM) [10] is a popular method to model system communication. In TLM, the communication blocks are decoupled from computation details, thereby providing a uniform interface for different components in the system and transparently addressing the coherency issues caused by concurrent communication. TLM abstraction models capture the high-level data transaction and associated latency information while neglecting lower-level implementation details, and hence enable high simulation speed

with necessary accuracy for system performance modeling and architecture selection. In practice, TLM is usually implemented in SystemC. In system modeling, a TLM channel and communication block are generated for each component, in a manner that enables the component to communicate with other blocks while keeping the functionality transparent. To achieve full system integration within the context of such a SystemC model, any profiled CPU requires the generation of a TLM wrapper. However, by current standards, writing such a SystemC TLM wrapper that captures all possible CPU behavior remains non-trivial and requires tedious manual design effort.

Previous processor modeling frameworks primarily target the modeling of general purpose CPUs instead of targeted SoC applications. Since our goal is to design a processor modeling flow that can be used in an SoC platform, extending the software model with TLM wrappers becomes a necessity. Hence, within this work, we developed an automatic TLM block generator for the output of our software modeling, such that our back-annotated software models can be equipped with TLM wrappers, ready for direct integration into SystemC models used for hardware/software codesign.

CHAPTER 3

METHOD

Our overall framework contains two stages, the first of which is *Software modeling*. Given C code as input, this stage estimates the energy and latency associated with this application running on a target platform, and outputs back-annotated C code, which serves as the input of the second stage: *SystemC and TLM 2.0 code generation*. This stage wraps the C code into a SystemC module, which then can be directly plugged into a SystemC system modeling environment such as a virtual prototype.

3.1 Software Modeling

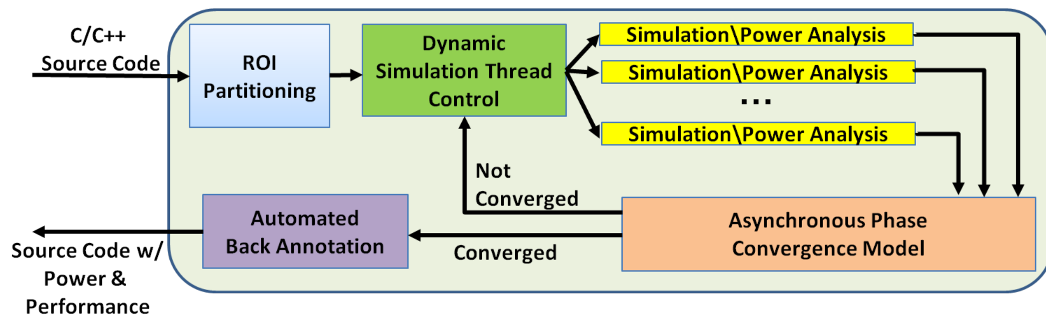

Figure 3.1: General flow for software modeling. Input source code is subdivided into multiple regions of interest that are dynamically simulated with results checked for steady state convergence. Results are back-annotated into source code.

An overview of our software modeling is shown in Fig 3.1. The user provides C source code with directives specifying the regions to be profiled. Code is then parsed and analyzed wherein each iterative region within the profiling area is further subdivided into multiple fine-grained regions of interest based on iteration scheme. Non iterative code blocks are arbitrarily

assigned to a single region. Regions are independently isolated and prepared for simulation in a manner that is dependent upon the simulator used. The simulations of multiple ROI occur in parallel with results passed through McPAT to enable power profiling. Simulation is managed using a dynamic simulation thread launcher, which operates based on feedback provided by an n-way convergence algorithm to dynamically determine the subset of ROI that must be run in order to determine phase boundaries. Once convergence is achieved, the simulation phase is complete and results are automatically back-annotated into the source code.

3.2 Energy vs. Power

We note that in general, our flow considers energy and latency for the back-annotation process as both variables represent quantities that can be directly accumulated during the runtime of the target application. While power can be derived for any region through the division of energy by latency, power cannot be accumulated using direct summation and it is therefore preferable to consider energy instead. However, we note that within our flow, we intermediately consider power instead of energy in circumstances where normalization with respect to latency is useful. Specifically, we perform all phase curve analysis using normalized quantities, and, as a result, phase curve analysis is performed with respect to power. Prior to the back-annotation process, any units of power are converted into energy through multiplication with the corresponding latency. Nevertheless, at any time power and energy can be derived from one another, as the latency of the corresponding region is always known.

3.3 Simulator Dependent ROI Division

As the input and control of each simulator is unique, we describe the process by which target regions are independently separated for both Sniper and gem5 environments.

Sniper: Sniper provides directives to specify target regions of interest within both purely sequential and iterative constructs. Utilizing these tools,

we generate a control script that initializes the simulation in fast-forward mode, switches to cache-only mode at a designated code location or iteration number, enters a detailed simulation at the target region of interest, and terminates upon completing the profile of the region of interest. For each region or interest, a custom source code is generated and compiled in conjunction with specific simulator commands that govern the behavior of the ROI. This source code in conjunction with control directives (in the form of a make file) is sufficient to enable independent parallel simulation of the target ROI.

Gem5: In order to isolate regions of interest while utilizing the hardware enabled fast forwarding, it is necessary to translate target locations within high level code directly into instruction counts. However, obtaining these instruction counts without contaminating the original program can prove challenging. We accomplish this by translating ROI directives into in-line assembly labels. After statically compiling and disassembling the code, we search for the designated labels as a means of obtaining the instruction address associated with all target code positions without modification to the compiled code.

To translate between instruction addresses and instruction counts we utilize PIN [11]. We generate a custom pintool that dynamically outputs the instruction count whenever the instruction addresses associated with the designated labels are executed on the host system. Combined with the code structural information obtained from parsing the source code, this is sufficient to enable a direct translation between the beginning and end of each ROI and their associated instruction counts.

As neither Sniper nor gem5 provides power data, power results are generated by McPAT. While exporting performance parameters from Sniper to McPAT is natively supported, such does not exist in gem5. Therefore, we utilize a script provided by R. Strong [12] to translate output statistics from gem5 to McPAT.

3.4 Parallel ROI Profiling

A key property of region-of-interest (ROI) code profiling is that each ROI can be simulated independent of all other ROI, even though the original code may be inherently sequential. While sequential code is highly dependent

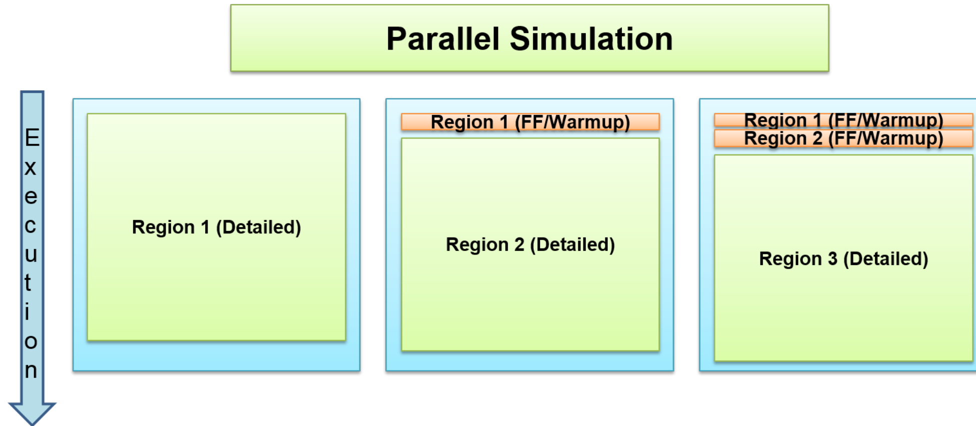

Figure 3.2: Illustration of 3 sequentially placed regions-of-interest running in parallel. Note that each ROI runs all preceding code in a fast-forward mode to preserve state.

upon code preceding regions (denoted as prefix code), ROI simulation satisfies this dependency by ensuring proper processor and cache states through preserving sufficient state during fast forward and warmup executions of all prefix code. This property of independence between ROI simulations enables the simulation of multiple ROI in parallel. Figure 3.2 illustrates the parallel execution of three sequentially positioned ROI. Note that the leftmost execution simulates the first ROI with no prefix code, the center execution simulates ROI 1 in fast forward/warmup mode before simulating ROI 2 in detailed mode, and the rightmost execution simulates both ROI 1 and ROI 2 in fast forward/warmup mode before simulating ROI 3 in detailed mode. Admittedly, this results in significant code execution redundancy; however, in simulation environments where the detailed simulation time is large and the fast forwarding execution time is several orders of magnitude faster than detailed simulation, this redundancy provides significant speedup. We note, however, that the upper bound of speedup is limited by the degree of parallelism available on the host machine. Recognizing that most users will be executing simulation with a finite degree of available parallel resources, in order to achieve a speedup greater than native ROI simulation, we have combined parallel ROI simulation with convergence modeling.

3.5 Phase Convergence

As a program executes, the dynamic state on the target platform can be categorized into phases. Significant phase changes can be caused by branch mispredictions, misses within the various caches, or changes within the general control flow. When considering the effects of these phases upon performance characteristics with respect to a dependent marker of code progression, such as number of instructions or number of sequentially placed ROI, this results in a mathematically describable phase function. At fine granularities this plot can appear highly volatile and periodic in nature, but at coarser granularities the resulting graph is generally smoother as periodic or volatile events become averaged over the larger interval and the resulting plot typically resembles a piecewise function. So long as the sampling required to determine the function is less than the overall ROI space, a significant speedup can be attained by approximating intermediate regions.

3.6 Instruction Count Normalization

In the general case, an iterative code structure need not have a constant number of instructions per iteration. This may be caused by iteration based control flow, or as commonly manifest in the benchmarks considered within our experiments, an inner loop structure with bounds dependent upon the outer loop iteration number. Additional inconsistencies in instructions per ROI may result when performing ROI sub-division, specifically in situations in which the number of iterations is not perfectly divisible by the number of ROI, causing some ROI to envelope more iterations than others. Whenever significant inconstancies in instruction counts per ROI occur, the resulting latency and power curves of our model within the ROI space can contain anomalous points that do not conform well to a linear piecewise function and are difficult to capture within convergence modeling.

In order to address these inconsistencies, we normalize latency per ROI relative to the number of instructions within each target region, with instruction counts obtained using our dynamic pin-tool profile. Convergence detection then utilizes the normalized latency curve in combination with the power curve. Note that we do not explicitly normalize power. Power is a

unit of Energy/Time, where Time is equivalent to Instructions/IPC*freq, where IPC is defined as Instruction per Cycle. Therefore, Power = Energy*IPC/(Instructions*freq), indicating that power is already normalized with respect to the number of instructions.

3.7 N-way Convergence Sampling

To describe n-way convergence modeling (where n is the number of concurrent processes), we first consider the simplest case of the single thread sampling. Here we first evaluate the upper and lower bounds of the provided region using a model to predict intermediary values. While convergence modeling is applicable to any arbitrary model, within the context of this thesis, we choose to consider a linear stepwise function. We then sample the midpoint of the region, comparing the result to our predicted values. We then update the model to reflect the data obtained through sampling the midpoint. Assuming our model remains inaccurate beyond a given threshold, there are 2 regions of unknown values, one on each side of the midpoint, and the process repeats. Per iteration, the size of the target interval decreases by a factor of 2. Given the number of ROI to be finite, the algorithm will terminate either when accuracy threshold is reached or interval size is 1. Since we define both as convergence, the algorithm is guaranteed to terminate successfully.

3.8 Thread Scheduling

In extending the algorithm to n-way convergence, we introduce a constant n parallel threads all executing on the given initial region. Parallel simulation requires the proper management of thread behavior on the host platform. In order to accomplish this, we consider the following thread scheduling techniques.

Naive Scheduling: In the naive implementation, threads are concurrently generated for each region-of-interest simulation and results are reordered following the completion of all processes. In the absence of convergence detection, this is the easiest algorithm to implement, as it involves little more

than a fork() instruction for each ROI. However, there are multiple shortcomings that make this unsuitable for our current design. First, the memory requirements become prohibitive when concurrently simulating large numbers of ROI. Second, operating systems attempting to service all threads generally rely on time-multiplexed context switching, which diminishes host processor performance, and results in all thread finishing at effectively the same time. Finally, without enforced ordering, implementing convergence detection on top of naive scheduling is self-defeating. If all threads finish execution at roughly the same time, then detecting convergence at the time when all threads complete is a degenerative case of convergence, and offers no benefit to complete simulation.

Batch Scheduling: In batch scheduling, regions of interest are executed into batches of n, where n is the degree of parallelism available on the host machine. The content of each batch is determined by the previous results of conservative n-way search algorithm, where excess resources are assigned to the candidate points of near future iterations. Upon the completed execution of a batch, results are reordered to preserve the correctness of the original algorithm. After the current model has been updated to reflect batch results, the next batch is generated and launched. This process continues until the entire model indicates convergence is achieved.

This algorithm offers several advantages over the naive implementation. First, the number of threads launched at any given time is fixed, thus limiting the amount of system memory required at any given time. Second, batch scheduling results in inherent ordering, as one batch is not launched until the previous batch has been completed. Third, thread-safe execution is easily achieved as the intermediate time between batch launches enables designated points at which the analysis for our n-way search can be performed. Fourth, batch scheduling ensures that the majority of ROI corresponding to converged phase regions never occupy system resources, as such threads will never be launched.

Our initial implementation of n-way search convergence modeling successfully employed the technique of batch scheduling. Global queues designated target threads with one region of interest per batch, while maintaining proper ordering between results for correct incorporation into the model. Unfortunately, as was discovered by our initial results and subsequent execution

profiling, batch scheduling also introduces inherent challenges that must be addressed to maximize runtime performance. Batch scheduling requires a high degree of thread synchronization, in which threads must wait for all other threads to reach a designated code position. As batch execution time is equivalent to the longest execution of any thread in the batch, load balancing between batches became key to maximizing resource consumption and containing overall runtime.

Unfortunately, within the context of ROI simulation, runtimes for disjoint ROI can vary widely, and in practice we found this variation could extend to several orders of magnitude. We attempted to address this by enforcing batches to contain ROI all from the same block of iterative code (in order to maximize similarity between ROI), but in practice this often proved insufficient to ensure proper balancing and resulted in wasted resources, often caused by edge cases and discrepancies in fast forwarding overhead present in considering ROI from opposite ends of the iteration space.

Algorithm 1: N-way convergence algorithm

```
1  Input: UncovergedRoI, tol
2  while UnconvergedRoI ≠ ∅ do
3  |   for t ∈ AvailableThreads do
4  |   |   minBlock ← find_least_populated_block(uncovergedRoI);
5  |   |   (ub, lb, mp) ← find_unconverged_interval(minBlock);
6  |   |   log_dep(t);
7  |   |   (simP, simL) ← detailed_simulation(mp);
8  |   |   while dep_complete(t) == false do
9  |   |   |   wait();
10 |   |   end
11 |   |   err ← calculate_err(simL, simP);
12 |   |   if err :≤ tol then
13 |   |   |   remove_unconverged(ub, lb);
14 |   |   end
15 |   |   update_model(ub, lb);
16 |   end
17 end
```

Asynchronous Dependency Thread Scheduling Our solution to the challenges associated with batch scheduling was the development of an asynchronous thread scheduler that tracked and maintained the dependencies

inherent within our n-way convergence algorithm. By employing this algorithm, we ensure that thread updates to the convergence model must only wait upon threads with which they have a direct dependency. This is accomplished using a dynamic array of mutex semaphores that protect individualized access to the convergence models and scheduling queues associated with each iterative code block. It eliminates the need for global synchronization and instead allows for independent threads of various execution time to run concurrently without negative impact. Each iterative ROI block is assigned a dependency list that ensures that the next dependent instruction may, with no wait, update the model, receive a new assignment, and immediately begin execution of the new assignment.

The pseudo code is shown in Algorithm 1. Given error tolerance tol and input set of all ROI, $UncovergedROI$, the algorithm outputs a converged model containing approximated power and latency for each ROI. We first identify the block within the unconverged ROI that currently contains the least number of assigned threads (line 4). We then request the next available interval within that block, identified by the upper and lower bounds (ub, lb), and receive an assignment to evaluate the interval midpoint mp (line 5). We log our dependency chain to preserve order correctness (line 6) and execute simulation to acquire power and latency (line 7). Upon return from simulation, we verify that all points within our dependency chain have also completed (lines 8-10). We then determine model error (line 11), use the error to determine and mark convergence (lines 12-14), and calculate new approximate values of power and latency using most recent simulation data, updating the model over the assigned interval (lines 15-17). The process repeats until no unconverged ROI remain.

As a further optimization, we recognize that while our asynchronous scheduling algorithm provides full resource utilization, it can also result in excess simulation in conditions where speculative assignments are provided to a region that converges soon thereafter. We address this by providing a means whereby model convergence can track and kill speculative simulation processes currently executing within the specified region. Special care is taken to ensure that while the simulation process is killed, the thread remains active and ready to receive a new assignment.

3.9 Automated Back-Annotation

Back-annotation of energy and latency data into the original source code is achieved through the automated generation of C++ header files. Results for each block of iterative ROI code are exported from our convergence model and assigned to designated arrays. Update functions are automatically incorporated into the original source code. The resulting source code when compiled will actively maintain current energy and latency of the system at ROI/iteration granularity.

3.10 SystemC and TLM Generation

As described in Sec 1, SystemC together with TLM 2.0 is a widely adopted methodology for SoC modeling and virtual prototyping. However, manually translating a C code into SystemC can be very tedious. Hence, to incorporate our software modeling output into the virtual platform automatically, we developed an automated SystemC and TLM 2.0 code generation framework, which takes C code as input, and directly transforms it into a component module written in SystemC equipped with TLM 2.0 communication channels and functions. In particular, it achieves three goals:

1. Enable seamless incorporation of the software model into virtual platform for effective system-level simulation and analysis.

2. Consider both communication and computation cost. The latency and energy associated with computation is derived from the back-annotated software model, while the communication cost is obtained from memory model and TLM channel.

3. Maintain the memory correctness in the SoC. In a real SoC, different components access memory and exchange data constantly. Therefore it is necessary to maintain the memory consistency among all the components. Since our original software model is a pure C code with no communication with the rest of the system, one major feature of our SystemC and TLM framework is to enable data movement between the component and the rest of the system.

3.11 The Hierarchy of the Generated Code

Figure 3.3 shows the hierarchy of the generated SystemC modules and simulation environment. We generate two components: The *Processor* module and the *Memory* module. The *Processor* is SystemC code embedded with the output of software modeling flow. The *Memory* module is a SystemC code used to model the memory behavior, annotated with latency and energy associated with memory accesses. These two modules are connected via communication payloads. This environment can be extended to incorporate additional components to form a more comprehensive system. The main contribution of our framework is to automatically generate the processor modeling, while the memory model is a separate (and standard) SystemC block which is used for simulation purpose; hence, we explain the processor model in detail.

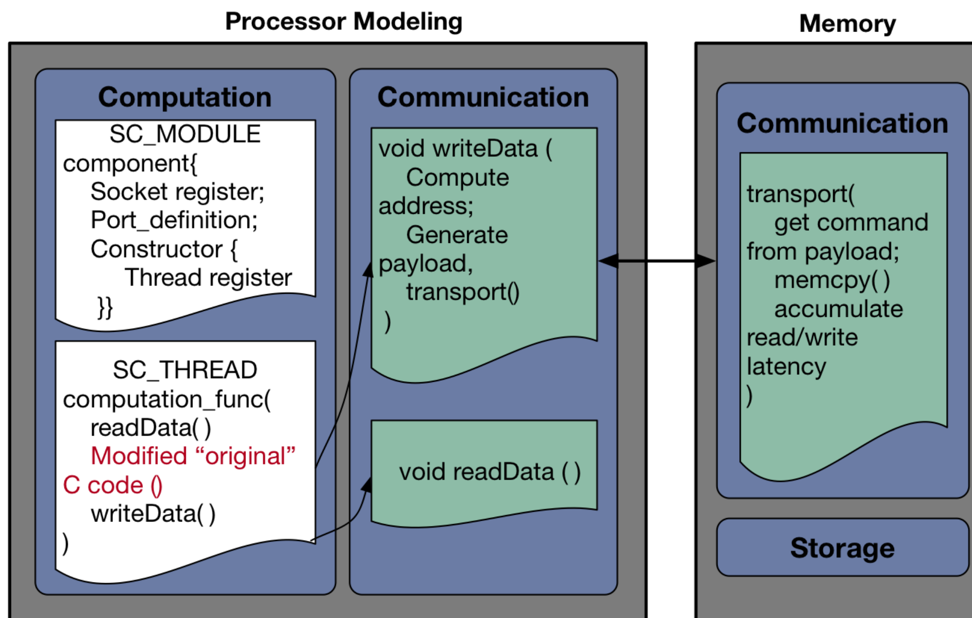

Figure 3.3: Generated SystemC TLM 2.0 platform

The *Processor* module is the SystemC code embedded with the output of our software modeling flow. The application of TLM 2.0 enables the separation of the computation and communication parts, as indicated in the figure. In the computation part there are two major pieces. The SC_MODULE() is a C++ class declaration which defines the interface and structure of the com-

ponent. It generates and registers the communication socket with the transport function, declares the member variables and methods, and declares the thread function in the constructor and registers it with the SystemC scheduler kernel. The SC_THREAD computation_func() is the main function; once the simulation starts, this function continues running until sc_stop() is called. The function body contains three parts: The read data() and writeData() are two communication functions, which are called before and after the computation to achieve atomic data movement. Note that once these two functions are called, the latency associated with computation part is collected and passed to these functions as a parameter, which is used in updating and synchronizing with the SystemC global scheduler. In the communication part, the framework creates two functions for data read and write, which are similar in structure. Considering writeData() as an example, it first generates the payload as a channel to pass necessary TLM 2.0 communication parameters to the target socket; then it generates the transport function, which directly binds to the socket and passes the payload parameters to the memory for communication.

3.12 The Code Generation Flow

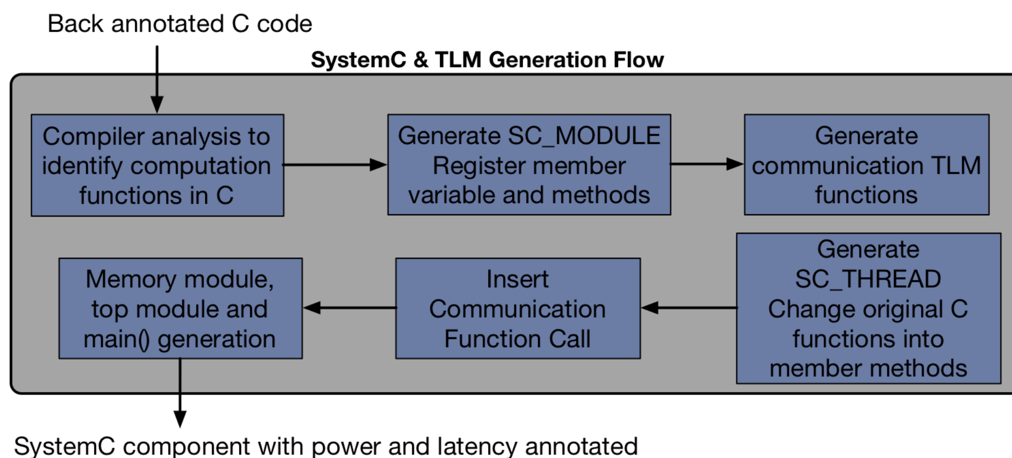

Figure 3.4: Overall flow of the code generation framework

With the hierarchy of the generated code illustrated, it is easier to demonstrate the code generation framework flow. Figure 3.4 shows the block dia-

gram. Given back-annotated C code as input, the first step in our flow utilizes compiler analysis to identify the function structures as well as the memory variables in the C code and transform them into C++ member functions and variables. Once this is completed, in the second step we generate the SC MODULE using the information acquired in step 1. We then generate the communication TLM functions, which can be done independent of the original C code and this function can be reused in different modules. The next step is to generate the body of the SC THREAD. Here we need to insert member methods which are transformed from the original C code and insert required SystemC syntax such as super while(1) loop and sc stop() function at the end of the function. The next step is to insert the communication function calls, and communicate with the memory module. Finally, the memory model, top module, and the main() functions are generated to enable the complete simulation environment. This step can also be done once and generated files can be reused for different applications. In addition, we also provide extensive scripts support, enabling the designer to perform all code generation steps as well as simulation by a single command.

CHAPTER 4

EXPERIMENTS AND RESULTS

Experimental results are collected on both Sniper and gem5. The primary purpose of our experimental results is simulation proof of compatibility with regard to our flow, the subsequent speedup and limited loss of accuracy.

4.1 Experimental Setup

Benchmarks We evaluate our described flow using benchmarks from the Polybench Suite [13]. The affine code structure offered within the Polybench suite is ideal for static ROI partitioning, although such is not required for our flow to operate correctly. Furthermore, we have targeted benchmark kernels within this suite, as custom accelerator modules for these kernels are available for incorporation into the virtual prototype environment [14].

Initially both gem5 and Sniper experiments were conducted using the four benchmarks of Atax, Correlation, Covariance, and Gemver. Results obtained provided sufficient demonstration of proof-of-concept application of convergence modeling within the context of both simulators. However, as results provided by these initial experiments indicated that gem5 provided greater potential application for our overall flow, additional verification tests were considered within the gem5 framework, further incorporating Gemm, Lu, and Jacobi-2d.

Characteristics of each benchmark considered are provided in Table 4.1. The column *Blocks* provides the number of iterative code blocks for each benchmark. The column *Depth* describes the iterative depth associated with each block. For example the simplest kernel, Atax, contains only a single iterative code block consisting of a double nested for loop. Correlation consists of four iterative code blocks, the first three of which are double nested for loops while the fourth consists of a triple nested for loop. Lu is a special

Table 4.1: A high-level description of code structure and data sets for all benchmarks considered in flow verification

Benchmark	**Blocks**	**Depth**	**Sniper Dataset**	**gem5 Dataset**
Atax	1	[2]	4000	8000
Correlation	4	[2,2,2,3]	1000	1000
Covariance	3	[2,2,3]	1000	2000
Gemver	4	[2,2,1,2]	4000	8000
Gemm	1	[3]	-	1024
Jacobi-2d	2	[2,2]	-	2000
Lu	1	[(2,3)]	-	1024

case, in which a single iterative code block contains both double and triple nested loops. Each benchmark dataset operates on an NxN matrix of variable size. The column *Sniper Dataset* lists the dataset size (N) associated with benchmarks used in conjunction with our Sniper experiments, while the column *gem5 Dataset* lists the dataset size (N) associate with benchmarks used in gem5.

Simulator Configuration Sniper: We maintain the default simulation environment inherent in the native Sniper, which instantiates a Xeon (Gainestown) duel-core processor running at 2.66 GHz. Although we recognize that this processor configuration was not originally designed to for SoC applications, it remains useful within our proof of concept verification. In addition, we specify the cache-warming phase of one iteration. Sniper experiments are conducted by utilizing 4-way parallelism on a host system with an Intel i7-4770K processor and 16 GB RAM.

Gem5: In using gem5, we likewise configure several simulation parameters, such as cache sizes, to their simulator default settings. We specify the CPU frequency to be 3.4 GHz, which is consistent with the McPAT template we obtained. Cache and functional warming are fixed at 3 million instructions. Gem5 experiments are conducted by utilizing 6-way parallelism on a host system with an AMD FX-6100 processor and 8 GB RAM.

Variations in ROI Granularity In order to demonstrate the variations in speedup associated with parallel ROI simulation with convergence modeling, we implement each benchmark with multiple levels of ROI granularity. For simplicity, in benchmarks with multiple blocks, we chose to limit our ex-

ploration space along a single dimension, meaning that modifications to the number of ROI blocks is applied universally to all code blocks. An exception is one code block within the Gemver benchmark that is only a single nested for loop, for which the total number of instructions and the corresponding simulation time are insufficient to consider for fine grain partitioning. For consistency, results are provided for all benchmarks at ROI granularities of 1, 50, 100, and 500. While it would be ideal to provide results for all possible granularity divisions within the iteration space, this would prohibitively increase our experimental runtime by several orders of magnitude. Thus we have primarily focused our experiments on variation of fine granularities, which is the target domain of both simulation parallelism and convergence modeling. In some cases we have included additional granularities to provide additional insight into general simulation trends.

Guided ROI Granularity Preliminary results indicated that ROI granularity has a dramatic effect on speedup, and in general, increasing the ROI granularity will result in increased speedup. However, in considering the results of Gemver as shown in Table 4.2 and Correlation as shown in Table 4.3, the increased granularity can be beneficial to a point, after which further increasing the granularity degrades the initial speedup. An analysis of the resulting phase curves indicates this slowdown can be primarily attributed to oscillations within the phase curve, which are graphically illustrated for Gemver within Fig 4.1 and Fig 4.2. The effects of this oscillation can be removed by clustering the alternating regions into groups, resulting in a smooth curve that converges quickly. However, defining the ROI granularity corresponding to this clustering may be too abstract for the common user. Given differences in benchmarks, including variations in problem size and iterative structures, we note while there is no one-size-fits-all granularity that is ideal across all benchmarks, our flow extracts enough information regarding the target application to produce an educated guess based on a minimum instruction threshold. We provide a simulation mode that automatically subdivides each iterative region into the maximum number of divisions, while then ensuring that each division maintains a minimal instruction count. This is made possible through the analysis of the dynamic host-profiling output that provides a detailed map of ROI to instruction count correlations. This is particularly useful in instances where instruction counts between ROI can

vary greatly, and results in a form of minimal load balances. We have chosen a minimum sub-division instruction count of 4,000,000 instructions, which corresponds roughly to the point at which periodicity corresponding to system events becomes prevalent within the resulting phase curves.

Variations in Warmup Within Sniper, warmup of iterative code can be instantiated by specifying the iteration number corresponding to the point in code at which the processor transitions between the fast forward and warmup states. Given the size of the iterative constructs considered within the provided benchmarks, sufficient accuracy was achieved using a warmup phase consisting of a single iteration. As the number of instructions associated with a single iteration can vary across benchmarks, this implies that, within Sniper, the number of instructions associated with each warmup phase is benchmark dependent. Gem5 provides the ability to control warmup duration at the instruction level, enabling perfect consistency between benchmarks. Therefore, within the gem5 environment, we further consider the effect of warmup variation across multiple benchmarks. As combining variations in ROI granularity and warmup duration results in a multidimensional problem, we have considered expanding the warmup variation only in the instance of ROI granularity 500. For all other gem5 evaluations, default warmup duration is set at 3,000,000 instructions, which was decided on the basis of results obtained by [8].

4.2 Results

Detailed results for our experiments can be seen in Table 4.2 for Sniper and Table 4.3 for gem5. Results show the max speedup as averaged across all benchmarks to be 31.5x for Sniper and 323.1x for gem5. Average latency error from Sniper is 0.89% while the average energy error from Sniper is 0.10%. Likewise the average latency error from gem5 is 4.16% while the average energy error from gem5 is 2.87%.

Phase Curves For illustration of the model output, we have provided phase curves associated with the benchmarks simulated within the Sniper simulation flow. The phase curves corresponding with latency per ROI can be

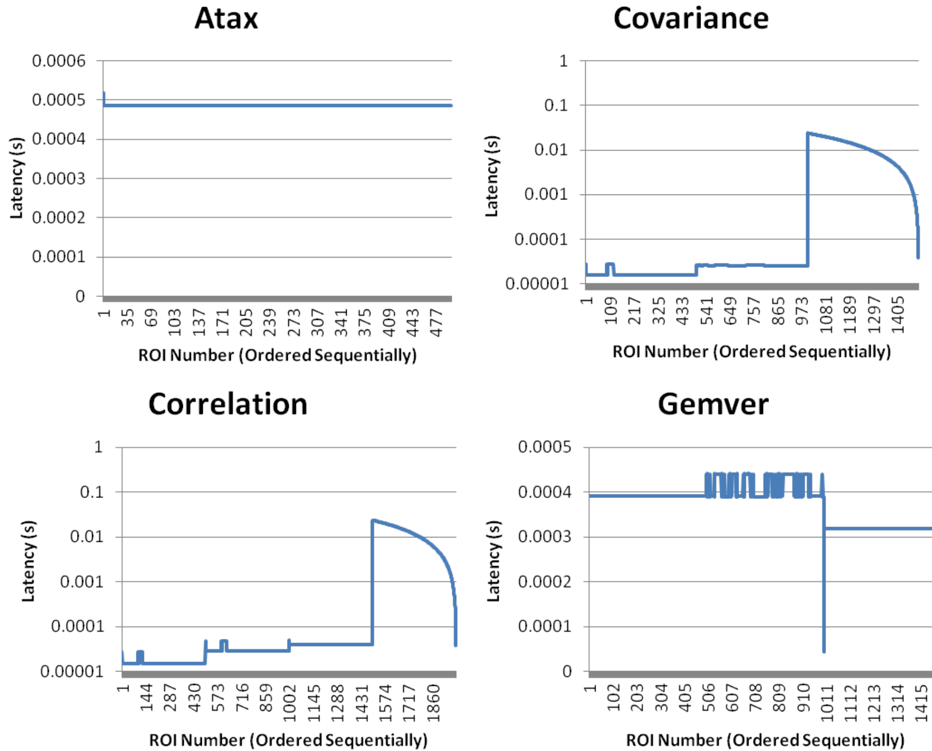

Figure 4.1: Resulting latency phase curves as output from Sniper simulation flow.

seen in Fig 4.1 while the phase curves associated with power per ROI can be seen in Fig 4.2. In general, these phase curves demonstrate the benefits of linear step-wise convergence modeling, even though some benchmarks, namely Correlation and Covariance, contain non-linear behavior. Sharp edges within the phase curve represent either changes in control flow or system events such as cache misses scattered throughout execution. We do however note the limitations of our methodology as captured within the region in Gemver between ROI numbers 500 and 1000. At this ROI granularity, the phase curve is periodic in nature, which is inherently difficult to capture using only linear likewise convergence, and results in the limited speedup at higher granularities as shown by Table 4.2.

Speedup Results demonstrate that, in general, increasing the granularity of ROI partitioning results in a significant runtime speedup of the simulation profile. We note that the magnitude of the speedup obtained can vary across

Table 4.2: Results obtained using Sniper simulation platform in conjunction with our flow. For each benchmark, overall runtime, latency, and energy are reported for multiple ROI granularities. Speedup and error have been computed relative to full detailed simulation with complete cache states.

Num ROI Per Loop	Runtime (s)	Speedup	Latency (s)	Energy (J)	Latency Error	Energy Error
Atax						
base	1468.02	1.00	0.2430	2.531	0.000%	0.000%
1	773.29	1.90	0.2430	2.497	0.015%	1.339%
8	166.47	8.82	0.2428	2.497	0.076%	1.345%
20	213.43	6.88	0.2427	2.497	0.090%	1.346%
40	104.46	14.05	0.2427	2.497	0.090%	1.346%
50	122.64	11.97	0.2427	2.497	0.090%	1.346%
80	78.84	18.62	0.2427	2.497	0.090%	1.346%
100	71.44	20.55	0.2427	2.497	0.091%	1.346%
200	48.89	30.02	0.2427	2.497	0.090%	1.347%
400	35.56	41.28	0.2427	2.497	0.093%	1.348%
500	32.90	44.62	0.2427	2.497	0.091%	1.349%
1000	34.19	42.94	0.2427	2.497	0.085%	1.352%
2000	34.09	43.07	0.2428	2.496	0.071%	1.358%
3000	32.83	44.71	0.2428	2.496	0.069%	1.360%
4000	32.70	44.90	0.2428	2.496	0.049%	1.369%
guided	47.60	30.84	0.2427	2.497	0.091%	1.348%
Correlation						
base	14631.39	1.00	5.8669	48.915	0.000%	0.000%
1	13327.48	1.10	5.8782	49.456	0.194%	1.106%
20	2889.72	5.06	5.8687	49.325	0.032%	0.838%
50	1245.91	11.74	5.8783	49.456	0.196%	1.107%
100	2348.54	6.23	5.8780	49.456	0.189%	1.106%
500	1545.25	9.47	5.8747	49.454	0.134%	1.102%
guided	1048.25	14.17	5.8783	49.456	0.196%	1.106%
Covariance						
base	14856.39	1.00	5.8567	48.802	0.000%	0.000%
1	14102.28	1.05	5.8655	48.809	0.150%	0.014%
20	3258.44	4.56	5.8573	48.703	0.009%	0.202%
50	1653.64	8.98	5.8708	48.813	0.240%	0.023%
100	2713.92	5.47	5.8650	48.809	0.140%	0.013%
500	1315.62	11.29	5.8637	48.808	0.118%	0.012%
guided	814.25	18.25	5.8638	48.799	0.120%	0.006%
Gemver						
base	2410.18	1.00	0.5623	4.585	0.000%	0.000%
1	666.12	3.62	0.5623	4.603	0.001%	0.393%
20	229.04	10.52	0.5623	4.603	0.010%	0.393%
50	49.43	48.76	0.5623	4.603	0.000%	0.392%
100	154.32	15.62	0.5632	4.604	0.173%	0.404%
500	328.01	7.35	0.5624	4.603	0.024%	0.388%
guided	63.73	37.82	0.5625	4.603	0.050%	0.381%

Table 4.3: Results obtained using gem5 simulation platform in conjunction with our flow. For each benchmark, overall runtime, latency, and energy are reported for multiple ROI granularities.

Num ROI Per Loop	Runtime (s)	Speedup	Latency (s)	Energy (J)	Latency Error	Energy Error
Atax						
base	57888.86	1.00	1.164	3.345	0.00%	0.00%
1	27445.70	2.11	1.194	3.293	2.57%	1.56%
50	761.30	76.04	1.194	3.294	2.59%	1.54%
100	425.40	136.08	1.194	3.294	2.63%	1.52%
500	161.53	358.37	1.196	3.298	2.77%	1.41%
guided	122.61	472.14	1.198	3.303	2.97%	1.26%
Correlation						
base	120467.32	1.00	5.536	16.591	0.00%	0.00%
1	120237.83	1.00	5.754	16.145	3.95%	2.69%
12	19689.72	6.12	5.775	16.203	4.33%	2.34%
24	9882.86	12.19	5.716	16.084	3.26%	3.05%
50	4703.62	25.61	5.713	16.081	3.20%	3.07%
100	2401.08	50.17	5.658	15.980	2.22%	3.69%
250	932.96	129.12	5.659	16.004	2.23%	3.54%
500	634.32	189.91	5.682	16.052	2.65%	3.25%
750	703.54	171.23	5.786	16.266	4.53%	1.96%
1000	715.91	168.27	5.593	16.089	1.04%	3.03%
guided	752.08	160.18	5.788	16.229	4.56%	2.18%
Covariance						
base	762759.16	1.00	45.686	135.460	0.00%	0.00%
1	761991.38	1.00	47.067	136.708	3.02%	0.92%
50	45898.96	16.62	47.047	138.484	2.98%	2.23%
100	22696.44	33.61	47.102	138.582	3.10%	2.30%
500	4704.88	162.12	47.085	138.576	3.06%	2.30%
guided	1597.87	477.36	47.035	131.190	2.95%	3.15%
Gemm						
base	376349.75	1.00	48.246	107.892	0.00%	0.00%
1	375753.16	1.00	52.469	114.907	8.75%	6.50%
50	10994.29	34.23	52.468	114.905	8.75%	6.50%
100	5847.38	64.36	52.466	114.902	8.75%	6.50%
500	2157.33	174.45	52.451	114.872	8.72%	6.47%
guided	668.94	562.61	52.434	114.837	8.68%	6.44%
Gemver						
base	47524.91	1.00	3.197	8.304	0.00%	0.00%
1	17429.17	2.73	3.333	8.369	4.24%	0.78%
12	4219.71	11.26	3.330	8.363	4.14%	0.70%
24	2219.43	21.41	3.322	8.348	3.91%	0.52%
50	1177.00	40.38	3.333	8.370	4.24%	0.79%
100	665.07	71.46	3.334	8.373	4.28%	0.82%
250	370.34	128.33	3.334	8.373	4.27%	0.83%
500	275.06	172.78	3.335	8.377	4.31%	0.88%
750	246.68	192.66	3.205	8.118	0.24%	2.25%
1000	229.53	207.05	3.113	7.935	2.63%	4.45%
2000	206.45	230.20	3.190	8.102	0.22%	2.44%
4000	195.70	242.85	3.347	8.445	4.67%	1.69%
guided	278.80	170.46	3.335	8.379	4.31%	0.90%
Jacobi-2d						
base	2809.52	1.00	0.102	0.292	0.00%	0.00%
1	1504.14	1.87	0.100	0.281	1.12%	3.71%
50	104.71	26.83	0.099	0.286	2.68%	1.86%
100	126.81	22.16	0.099	0.286	2.99%	2.07%
500	92.85	30.26	0.101	0.291	0.53%	0.16%
guided	315.94	8.89	0.099	0.279	2.24%	4.49%
Lu						
base	81798.17	1.00	4.107	11.624	0.00%	0.00%
1	81596.98	1.00	4.445	12.306	8.23%	5.87%
50	4102.34	19.94	4.434	12.293	7.96%	5.76%
100	2115.59	38.66	4.455	12.336	8.47%	6.13%
500	479.66	170.53	4.447	12.326	8.29%	6.04%
guided	285.43	286.58	4.412	12.259	7.43%	5.46%

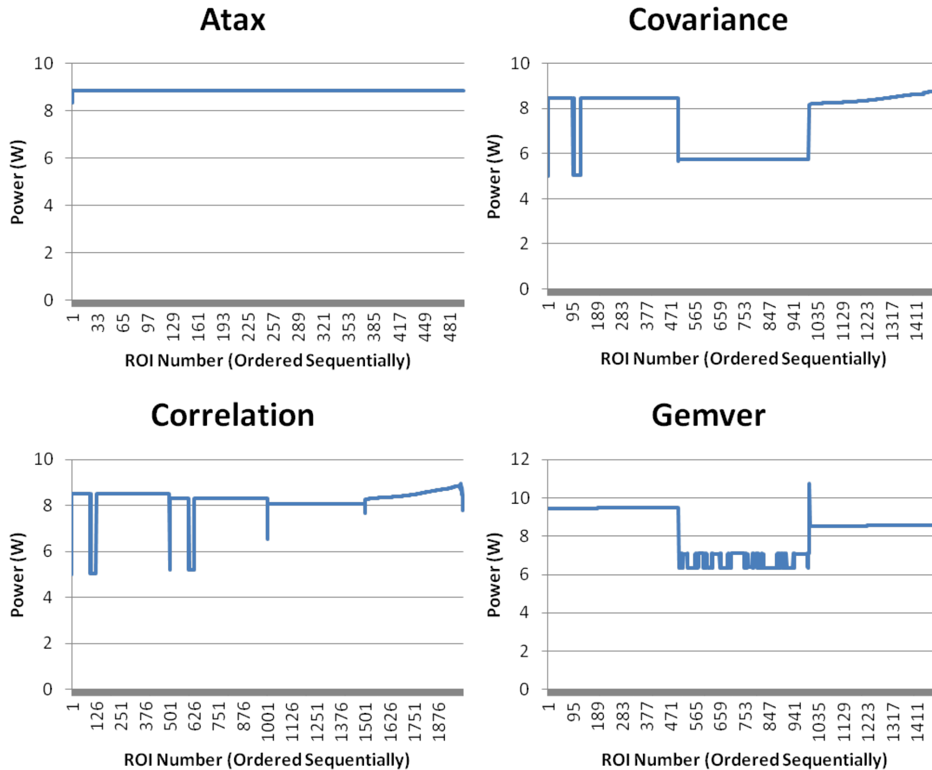

Figure 4.2: Resulting power phase curves as output from Sniper simulation flow.

different benchmarks and simulators. Variation across benchmarks is due in part to the differing phase typologies, which directly affect the rate at which each benchmark converges to our model. Drastic changes in control flow or instruction composition result in sharp edges within the phase curve that require multiple iterations to resolve.

Within Sniper, for the ROI granularities tested, the max speedup obtained was 48.76x achieved using the benchmark Gemver at an ROI granularity of 50. While higher speedups may exist at granularities not tested, time constrains to not permit an exhaustive sweep of all possible granularity levels. More detailed sweeps were considered for benchmarks will relatively short runtimes. Atax received a similar max speedup of 44.90x at a granularity of 4000, while Correlation and Covariance achieved their max speedups of 14.17x and 18.25x respectively at the guided granularities. In considering all benchmarks, it is significant to note that there is no consistent granularity at which optimal speedup is obtained. Nevertheless, our guided ROI

division, by considering maximal granularity with a threshold minimum instruction count, effectively achieves speedup that is either optimal or near optimal across all benchmarks. Overall, across the benchmarks considered with Sniper, the average max speedup achieved is 31.5x.

Similar speedup trends were found within the results of the gem5 experiments. Overall maximum speedup of 562x was obtained from the Gemm benchmark using the guided granularity. We likewise note that within gem5 our guided ROI division was also able to achieve optimal or near optimal speedup across all benchmarks. The smallest maximum speedup 30.26x gained occurred in testing Jacobi-2d, with the overall average maximum speedup equal to 323.1x. For large benchmarks this constitutes the difference between a simulation requiring a few hours vs. a simulation requiring a few weeks!

Speedup differences between simulators can be attributed in part to the magnitude of speedup provided by fast-forward and cache warming modes relative to the speed of detailed profiling. Sniper, although natively faster than gem5, does not offer the same speed advantage provided by gem5's hardware-accelerated fast forwarding. As a result, we see that the benefit of increased ROI granularity can decrease after a certain threshold resolution. In considering why such occurs, we note that increasing the number of ROI partitions also increases the number of simulations required for reaching convergence. This in turn increases the number of times that prefix code (code preceding the ROI) must also be simulated. As the granularity of the ROI becomes finer, the time spent performing detailed simulation for each ROI decreases, and the ratio of simulation time spent executing this prefix code relative to the time spent profiling the ROI increases. Thus, increasing the granularity increases the cumulative overhead associated with partitioned profiling which counteracts and eventually diminishes speedup. Within Sniper, this overhead is particularly pronounced, due to the lack of hardware accelerated fast-forwarding, and can be seen in the slowdown of Gemver at high granularities.

Further Speedup Limitations Our speedup model assumes that detailed profiling of code through cycle-accurate simulation remains the dominant contributor to overall runtime. Techniques employed within our flow specifically work to reduce runtime through limiting the overall detailed profiling

required for code profiling. This is particularly valuable when considering the simulation time required for large programs. However, we note that the following case conditions will invalidate this assumption, and will therefore result in limited speedup:

Case 1: Non-convergence Non-convergence can occur under conditions in which convergence is never realized, either due to an unstable phase function or unachievable convergence tolerances. If such occurs, every ROI must be simulated, and the overall amount of code profiled using detailed simulation will be the entire code space. Alternatively, convergence may eventually occur, but not before the majority of the ROI subspace has been sampled. Such may occur when the resulting phase curve does not mathematically conform well to the target modeling scheme. Within the context of linear step-wise approximation modeling, this condition occurs when the phase curve is highly periodic, alternating rapidly between disjoint values, which results in the consistent misprediction of intermediary points. In both situations, the speedup achieved through convergence modeling is lost. While it may still be possible to achieve an overall speedup by virtue of native parallel ROI simulation, max speedup would be limited by a hard upper bound based on the number of parallel computation resources available on the host system.

Case 2: Prefix Dominance Prefix runtime domination occurs whenever overall simulation time is dominated by fast forwarding and warm-up phases rather than detailed simulation. This can occur when the ROI size is much smaller than prefix code or if the fast forwarding mode provided by the simulator provides an insufficient speed advantage relative to the detailed profiling mode. Under such conditions, increasing the ROI granularity, which has no effect on prefix code run time, will result in minimal speedup.

Case 3: External Dominance External runtime domination occurs typically in conditions in which benchmarks are relatively small. While simulation time is directly proportional to the number of instructions within a ROI, other processes within the flow require near constant runtime regardless of the ROI size. The biggest contributor to external runtime is the power profiling performed by McPAT, specifically when used within our gem5 flow.

We note that for our benchmarks, a standard run of McPAT on the uninitialized CPU is approximately 19.9 seconds. Therefore, for detailed profiling of ROI that does not require significantly more than this time, the non-trivial contribution of constant runtime flow components will significantly degrade the speedup achieved through higher ROI granularities.

Accuracy The accuracy of both latency and energy results obtained from Sniper is shown in Table 4.2. Overall, the latency error ranges between 0.0% and 0.240%, with an average error of 0.10%. The energy error ranges between 0.006% and 1.369% with an average error of 0.89%. From Table 4.3, we see that within gem5, the overall latency error ranges between 0.22% and 8.75%, with an average error of 4.17%. The energy error ranges between 0.16% and 6.50% with an average error of 2.87%.

In general we note that the deviation of cumulative latency and energy relative to the baseline model is greater within the gem5 environment than within the Sniper simulation environment. We note, however, that error reported within gem5 is consistent with the expected error associated with imperfect cache warming associated with a warmup phase duration of 3 million instructions as reported by the pFSA model [8]. This error is directly observable in considering the reported error associated with the ROI granularity of 1. Specifically in considering single kernel benchmarks such as Atax, Gemm, and Lu, in which no parallelism or convergence is exploited, the only functional difference in simulation between the base case and ROI granularity 1 is that the former utilizes a complete cache state, while the latter utilizes an imperfect cache state as determined by fast forwarding and warmup modes. Thus the error reported from ROI granularity of 1 directly corresponds with the error associated with imperfect cache warming and warmup phases of the gem5 simulator. The consistency of results between ROI granularity 1 and higher ROI granularity demonstrates that this error associated with imperfect cache warming remains the primary source of inaccuracy within gem5, even at higher granularities.

Warmup Variation Results obtained from varying the duration of the warmup phase are shown in Table 4.4. Results are provided with respect to the Correlation benchmark running on gem5 at an ROI granularity of 500. For each warmup configuration, we report the run-time associated with

Table 4.4: Warmup Variation Results

Num Warmup Instructions	Runtime (s)	Speedup	Latency (s)	Power (W)	Lat Error	Pow Error
base	120467.32	1.00	5.536	2.997	0.00%	0.00%
0	554.00	217.45	5.928	2.787	7.09%	7.03%
100000	616.98	195.25	5.728	2.819	3.48%	5.95%
1000000	624.71	192.84	5.717	2.820	3.28%	5.90%
3000000	634.32	189.91	5.682	2.825	2.65%	5.75%
10000000	694.14	173.55	5.587	2.839	0.92%	5.28%
20000000	795.24	151.48	5.588	2.839	0.94%	5.29%
50000000	1042.52	115.55	5.587	2.839	0.93%	5.28%
100000000	1252.60	96.17	5.586	2.839	0.91%	5.28%

the complete convergence modeling process, the reported latency, and the reported power. Using the complete detailed simulation as our base case, we further report the speedup, latency error, and power error relative to the base.

In general, increasing the duration of the warmup period decreases the overall speedup. This is to be expected, as cache profiled warmup modes are significantly slower than the corresponding fast forwarding mode. While increasing the warmup period does not decrease the amount of prefix code that must be simulated for each ROI, it modifies the distribution, causing less of the prefix code to be simulated in the fast forward mode and more of the prefix code to be simulated in the slower cache profiling mode. We further note that increasing the warmup duration decreases the error associated with both power and latency. This is also to be expected, as increasing warmup duration also increases the cache history, resulting in a more accurate cache profile at the beginning of detailed simulation. As cache behavior has a direct impact of CPU power and latency, the more accurate cache states result in decreased error within the overall profile.

CHAPTER 5

CONCLUSION AND FUTURE WORK

We have herein developed and implemented a complete flow for application within the domain of HW/SW co-design. Beginning with source code, we identify and subdivide regions of highly iterative code into fine-grained regions of interest. Utilizing phase convergence modeling with fine granularity region-of-interest profiling, we achieve a maximum speedup of 48.76x and 562x for Sniper and gem5 respectively and an average simulation speedup of 31.5x and 323.1x with only minor losses of profile accuracy. We then automatically back-annotate performance data into our original executable code, which we then wrap using the TLM 2.0 framework. The result is a flow that efficiently converts host executed source code into TLM 2.0 compliant modules for direct incorporation into a virtual prototype design.

We recognize that code profiles may not always conform to contours that can be easily converged to using a linear approximation method. Therefore we propose for future work the incorporation of additional curve fitting techniques that can be used to analyze the data. Within the application of convergence modeling, such work would require significant additional programming complexity to track multiple data dependencies within the context of the asynchronous thread scheduling technique. Nevertheless, we recognize that if additional accuracy is desired for irregular contours, it may be worth the additional complexity. We specifically suggest the incorporation of both higher order stepwise polynomial modeling in addition to recognition of periodic functionality within phase curves, noting that the latter would enable additional accuracy and speedup of convergence modeling at higher granularity.

REFERENCES

[1] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, 2003.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti et al., "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[3] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 52.

[4] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[5] W. Heirman, T. Carlson, and L. Eeckhout, "Sniper: Scalable and accurate parallel multi-core simulation," in *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*. High-Performance and Embedded Architecture and Compilation Network of Excellence (HiPEAC), 2012, pp. 91–94.

[6] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 2–12.

[7] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 2003, pp. 84–95.

[8] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer, "Full speed ahead: Detailed architectural simulation at near-native speed," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 183–192.

[9] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 469–480.

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

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "PIN: Building customized program analysis tools with dynamic instrumentation," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[12] R. Strong, "m5-mcpat-parse.py," 2009. [Online]. Available: https://bitbucket.org/rickshin/m5-mcpat-parser

[13] L.-N. Pouchet, "PolyBench/C the polyhedral benchmark suite," 2015. [Online]. Available: http://polybench.sf.net

[14] W. Zuo, W. Kemmerer, J. B. Lim, L.-N. Pouchet, A. Ayupov, T. Kim, K. Han, and D. Chen, "A polyhedral-based SystemC modeling and generation framework for effective low-power design space exploration," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2015, pp. 357–364.