The Dissertation Committee for Hari Daas Angepat
certifies that this is the approved version of the following dissertation:

Logical Partitioning of Parallel System Simulations

Committee:

Derek Chiou, Supervisor

Mattan Erez, Supervisor

Andreas Gerstlauer

Mohit Tiwari

Jim Holt

Logical Partitioning of Parallel System Simulations

by

Hari Daas Angepat

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2019

Logical Partitioning of Parallel System Simulations

Publication No. _____

Hari Daas Angepat, Ph.D.
The University of Texas at Austin, 2019

Supervisor: Derek Chiou

Simulation has been a fundamental tool to prototype, hypothesize, and evaluate new ideas to continue improving system performance. However, increasing levels of processor parallelism and heterogeneity have introduced additional constraints when evaluating new designs. The work embodied in this dissertation explores how to leverage novel ideas in simulator partitioning to improve simulator speed and flexibility for simulating these new types of systems.

The contribution of this work includes the introduction of optimistic partitioned simulation to improve parallelization, and the introduction of warped partitioned simulation for improved flexibility. These ideas are refined and demonstrated through the use of prototypes to demonstrate their benefits compared to state-of-the-art approaches. By leveraging partitioning in a structured manner, it is possible to

design simulators that better address the open challenges of parallel and heterogeneous systems design.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Since the introduction of the first integrated circuit nearly 60 years ago, there has been a steady march towards better, faster, and cheaper digital electronics. This continuous progress has been driven by a variety of factors, chief among them the ability to squeeze more transistors into a given area. This scaling was first observed by Gordon Moore in 1965 (Moore, 1965), and has it become a self-fulfilling prophecy for the semi-conductor industry. More over, by shrinking transistor sizes, it became possible to increase switching frequency as observed by Dennard (Dennard et al., 1974), creating not just more, but faster transistors. The combination of increasing density and frequency has become the basic lever for increasing computing performance over the last five decades.

However, the limits of these scaling laws have become apparent over the last decade. Frequency scaling has been slow to improve due to to a variety of physical limits (e.g. voltage thresholds, power and heat density, leakage currents). As frequency scaling has slowed, CPU performance has slowed with it. While transistor counts have continued to rise, they cannot be made faster at the same rate.

This inflection point has led the industry to try to leverage parallel computing and heterogeneous accelerators to translate increased transistor budgets into practical performance improvements. For the last decade, uniprocessor designs have given way to multi and many-core processor designs with 10s of cores on a single processor.

Moving mainstream computing to adopt parallel processing has been termed a hail Mary pass (Patterson, 2010) for the industry due to the risk and issues that must be solved. Despite many attempts since the 1960s to bring parallel computing to the mainstream, enabling high-performance, high- productivity parallel computing has remained an ongoing problem. Given the hard limits now imposed by physics, solving this issue has become a pressing problem that must be solved to allow computing performance to continue scaling to future systems. This has led to increased innovation in programming models, architectures and tools to make parallel programming easier and more tractable for mainstream usage.

The research embodied in this dissertation focuses on extending the state of the art in the tools needed to design parallel systems.

1.2 Designing and Optimizing for Parallel Systems

The difficulties involved in designing and optimizing parallel systems are numerous. Parallel programming and performance optimization has been likened as a two-way conversation between a programmer and the computer ((McKenney, 2011)). In this light, parallel programming requires a programmer to listen to what the computer has to say about "performance and scalability" in a two-way dynamic conversation.

Reasoning about how parallel tasks may perform in a scalable parallel system can be challenging. A designer must cope with load balancing, synchronization, communication, and resource requirements and interactions with other tasks in the system. Understanding and optimizing such a system is also complicated by bottlenecks that may be hidden until pressure is relieved in another part of the system.

This "whack a mole" approach to optimization makes it challenging to identify where to spend design effort.

While parallel frameworks, languages and architectures attempt to hide some of this complexity from end users, they often do not entirely remove the burden from the designers of such systems. This problem is not restricted to software. The same issues are faced by hardware designers as well, choosing what bottlenecks are valuable to explore for acceleration opportunities.

1.3 Computer Simulation

One of the basic requirements when designing and optimizing a system is to have an environment in which to experiment and evaluate ideas. Whether experimenting with a new parallel synchronization algorithm, or a new hardware cache coherence protocol, a means of evaluating the idea is a core tool. Using a computer to simulate itself provides a means of providing this experimental testbed.

A computer simulation is a program capable of representing the behavior and properties of a given system under some environmental constraints. Using computers to simulate reality was one of the earliest uses of computing (e.g. nuclear, chemical, mechanical simulations).

Using a simulation of a computer system is often necessary due to the constraints reality places on real-world prototypes. Designing, verifying and fabricating a modern parallel processor is a multi-year, multi-million dollar project that is infeasible to act as a testbed for new ideas. While this process can be scaled down for test-chips, the overall effort and costs involved are still prohibitive. Real-world prototypes also can incur limitations in observability, low-level control of operations and can be restrictive in the types of ideas that may be practically implement.

Using a simulator to evaluate an potential optimization frees one from these restrictions while adding new complications. As the system under simulation grows

in complexity, tradeoffs in how to effectively use and organize a simulator to balance speed, development cost and accuracy become an important choice in the design of an experiment.

1.4 Parallel Systems Simulation Issues

Designing parallel systems increases the degree of freedom and design choices in how to partition, communicate and synchronize between processing elements executing in parallel. How a parallel system performs is often a combination of various different operating conditions, which further complicates the number of design points a designer must simulate. Two issues that using parallel simulation to evaluate parallel systems center around speed and flexibility.

One of the difficulties of using a computer to simulate another computer is that modern microprocessors are filled with dedicated circuits to perform a variety of tasks while the instructions available to general purpose software are much more limited. This impedance mismatch makes accurate simulation of digital hardware difficult. For example, detailed simulations used by Intel when designing a processor slow down a million times when simulating a single core (Emer et al., 2002). As we increase the number of simulated cores this problem becomes even worse, as we must again use software emulation to mimic the specialized circuits used for parallel synchronization and communication.

While it is possible to trade simulation detail for speed, this comes with some costs in accuracy. During the early stages of a project such inaccuracy may be tolerated, but as decisions become costlier to implement, such potential inaccuracies are not acceptable. For example, when planning a microprocessor design, extremely detailed simulation is still used to finalize architectural specifications for the design which comes years later.

The second open problem with using simulation to optimize a parallel system is

the effort required to simulate a given design choice. Given the increased dimensions that can potentially be explored, along with the heavy design costs involved in ensuring efficient and correct execution of any parallel optimization, implementing a single choice may be prohibitive. For example, implementing a single software optimization in a parallel framework may require changing multiple algorithms and data structures, which can involve multiple days to weeks of effort (see Chapter 6 for more examples).

The research presented in this work looks at alternative approaches to structuring and organizing parallel computer simulations so they can handle these issues better. If a simulator can be partitioned correctly, it becomes possible to leverage more efficient parallelization better (addressing speed concerns) as well as making it possible to explore optimizations more rapidly (addressing flexibility concerns).

1.5 Thesis Statement

Correct partitioning of a simulator can result in improved (i) performance (time to results) and (ii) flexibility (effort to results).

1.6 Contributions

This work embodies two key contributions towards improving the design of computer systems, specifically in the domain of simulator design. The first contribution is the creation of a new simulator architecture that leverages functional-timing (FT) partitioning to enable safe speculative parallelization. The second contribution is the creation of a new simulator architecture that again leverages FT partitioning to enable high-level sketching and exploration of design choices. These two abstract contributions are further refined with full prototypes that demonstrate some of the design issues in building simulators based on these new architectures.

1.7 Thesis Challenges

While the full design cycle encompasses a wide range of open challenges, we focus on points that have a large impact on decision making and exhibit difficult underlying technical problems. In this regard, this dissertation looks at problems at the ends of the design cycle, architectural refinement and post-production software tuning.

1.7.1 Challenge 1: High-Speed Detailed Simulation

During architectural refinement, simulation results eventually become frozen specifications that can have long-term impacts on a project. As custom chip design is such a costly enterprise, high-accuracy / low-uncertainty simulations are relied on to make final architectural decisions. For large microprocessor companies, this involves extremely detailed architectural simulations that can incur over a million-times slowdown. Given the tightly synchronous nature of this simulation, along with its heavy computational burden and the inability easily to leverage simplification approaches to gain speed, the problem is difficult to tackle with existing approaches. This high-detail, high-speed, and low-uncertainty requirement offers the first challenge addressed in this thesis.

1.7.2 Challenge 2: Highly Flexibility Simulation

The second thesis challenge addresses the basic issue of how to tune parallel systems (either in mixed HW/SW or pure parallel SW). This addresses both the early stages of system design and post-production parallel SW tuning. Implementing a single design optimization in a parallel system can be expensive due to the need to manage the various dimensions of correctness and efficiency. Given the interactive way in which optimizations may shift or create new bottlenecks, the problem of how to explore the space again is not fully addressed by contemporary approaches.

By addressing these two design points through innovations in simulation

design, it may be possible to improve decision making when designing and tuning parallel systems. In doing so, we seek to improve overall innovation in parallel systems.

1.8 Dissertation Organization

This dissertation is organized into chapters that cover simulation background, novel parallelization approaches, and novel high-level sketching approaches.

Chapter 2 provides more details about alternatives in simulator partitioning and the tradeoffs involved in using them.

Chapter 3 provides a high-level description of a novel simulator architecture that uses speculation to enable flexible parallelization while preserving correctness. Chapter 4 goes through some of the concrete design issues encountered when implementing a speculative simulation architecture designed to leverage the fine-grain parallelism offered by FPGA accelerated execution fabrics.

Chapter 5 provides a high-level description of the second novel simulation architecture developed in this thesis, focusing on exposing direct manipulation of simulation execution and mapping these concepts to high-level optimization sketches. Chapter 6 goes through the concrete design issues in implementing a simulator capable of direct manipulation and the handling of correctness and forward progress in the face of user-provided simulation directives.

Finally Chapter 7 concludes with a summary of the new simulation architectures introduced in this thesis and it provides some ideas on how to extend this work.

Chapter 2

Background on Partitioning Simulators

2.0.1 Terminology

In this work, we use the term *simulation* to describe the process of imitating the properties of another computer on a given system. These properties can simply be the behavior of a parallel application to the detailed performance of a hardware arithmetic unit under study. To perform a *simulation*, a concrete program or system is used, a *simulator*.

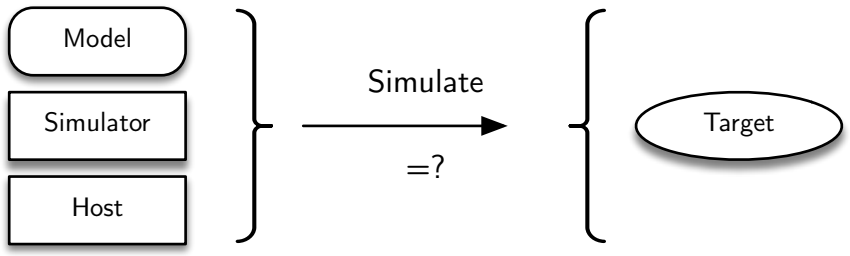

Figure 2.1: Simulation Terminology

Given the desire to simulate the properties of system that exists or could potentially exist, we use the term *target* to refer to the system we would like to study. We can think of the target abstractly as the set of properties or specifications that precisely describe what we would ideally like to simulate. In practice we must realize our desired target in a *model*, written for a simulator to represent our target.

A *host* provides a collection of resources upon which we can run the combined model and simulator. The output of this triplet of host + simulator + model should mimic the properties of a desired target if it can exist in reality. We depict this terminology stack in Figure 2.1. If the model is faithfully implemented and the simulator has no inherent inaccuracies in its implementation, the model will correctly mimic the intended target.

For example, a computer architect may use a simulator (e.g. M5 open source simulator) running on a host (e.g. x86 office workstation) to simulate the performance of a target (e.g. ARM dual-core mobile SoC) using a model (e.g. M5 model written in C++).

2.1 Background

Given the criticality of simulation in the design and tuning of parallel systems, multiple methods have been used to try to mitigate the costs of creating and maintaining such systems. Choices in modeling abstractions and fidelity simplification approaches can reduce the complexity and cost to build a simulator, but they can impact accuracy and still retain large costs to update and adapt the simulation. Simulators can be constructed hierarchically, allowing reuse of pieces of a simulation in a Lego-like fashion. This approach has complexity benefits when much of the system under simulation remains unchanged. However, if the part of the simulation being modified is complex by itself (a processor core for example), the complexity burden still remains.

One approach to increasing reuse is to factor the modeling tasks into reusable components. By capturing the parts of the modeling process that are generic, it becomes possible to reuse engineering effort more widely than composition at the model level. We call such factoring *model partitioning*, a structured way of separating concerns with a given modeling dimension. Each partition may model one or more aspects of the target. For example, a partitioned model may have different partitions for its various properties (e.g. behavior, energy, thermal, timing). By grouping state variables, operations and selected events into a reusable component, it becomes possible to create generic models of properties that are portable across models.

Creating a model from partition components can reduce complexity and simulation costs. By leveraging partition components when constructing a simulator, the multiplicative benefits of engineering $M + N$ components rather than $M * N$ submodels can be beneficial. Further, if partition components are restricted to concentrate on a single property, it can be easier to make changes to each property in isolation.

Identifying points of partitioning has similarity to parallel decomposition activities in simulator design but they differ in purpose. Parallel decomposition may examine state variables and event communication to select logical process boundaries, but does not consider the impact on modeling complexity. For example, Hao et al. (Hao et al., 1996) considers various strategies to choose the size of a logical process to minimize load imbalance, maximize internal shared state variables and minimize event communication. The chief concern of model partitioning is factoring a model for design complexity. However, such a model partition may also serve as a useful boundary during parallel decomposition.

2.1.1 Causality in Parallel Simulation

When simulating a simulator decomposed with either parallel or logical partitions, care must be taken to ensure causality is still retained by the overall simulator.

The key restriction in many simulator parallelization approaches is how to enable parallel execution of tasks while still maintaining causality between events. Given a simulation composed of logical processes, a correct parallel simulation would ensure that no event could be processed such that its effects could be felt by events that preceded it.

Fujimoto et al. (Fujimoto, 1990) describe this requirement as the *local causality constraint (LCC)*. The LCC constraint restricts the execution of events such that each logical process never processes an event out of timestamp order. By enforcing timestamp ordered processing of events, it becomes impossible for events from future timestamps to interact with the processing of events from preceding timestamps.

However, limiting parallel simulation to strict causality and the LCC can be overly restrictive. Some systems may be willing to tolerate small violations of causality if their errors can be shown to be negligible. In return for this relaxation, it is possible to increase parallelization speedup. This approach is often used in higher level simulations, such as SlackSim (Chen et al., 2009), where speed may be more valued than exacting accuracy.

2.2 FT Partitioning

One particularly useful model partitioning approach is along an *FT boundary*. We use the term *functional* to describe an operational view of a model (i.e. *what* does a model do). The term *timing* is then used to describe a performance view of a model (i.e. *when* does a model perform). As the parallel systems we are interested in simulating are engineered with interfaces and abstractions, they often mirror a

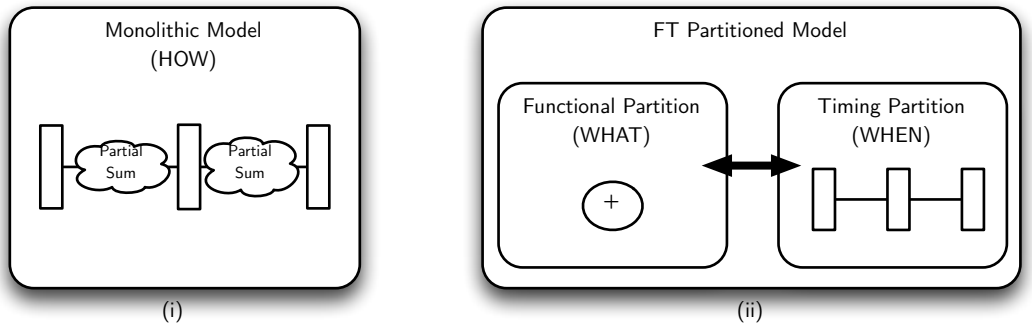

Figure 2.2: Monolithic vs FT-Partitioned Model

similar separation. For engineering and design complexity reasons, systems are often modularized to separate a particular implementation from its abstract interface. This allows the implementation to change over time without impacting the remaining parts of the system. Under an FT-partitioned model, this separation proves useful as it allows the functional partition to remain constant while the timing partition evolves more rapidly.

In contrast to this partitioned approach, a non-partitioned model describes *how* a model operates, and in doing so it captures what functionality it offers and when such functionality should be executed. Figure 2.2 illustrates this difference with a trivial description of an three-stage pipelined adder. In the monolithic approach, the model contains pipeline register state, partial sum operations and the events necessary to propagate the updates to the registers at the next clock cycle. In contrast, one approach for an FT-partitioned model may simply expose the add operation itself while the timing partition only captures the three-cycle delay. By decoupling the two parts of the model, it becomes easier to experiment with changing the latency (for example by simply adding an additional stage to the timing model), without having to refactor the monolithic model and update the intermediate pipeline results.

The goal of the functional partition is to extract the various untimed behaviors from a monolithic model into a common reusable component. If we consider the

abstract view of a model containing state variables, operations, and event sequences, a functional partition of the model may contain these elements save for the actual timestamps attached to the event-sequences. In this manner, we can view the functional partition as a consequence of replacing the simulated clock of the model with an infinitely fast clock such that all timestamps tend towards zero. The event sequences are still preserved to maintain true data dependencies, but the precise time at which state transitions occur is lost in the translation process. Such a model may still retain individual state transitions or it may be optimized to collapse unnecessary transitions. For example, in our three-stage pipelined adder, an alternative functional partition may have retained the untimed partial sum behaviors instead of simplifying to a single untimed add operation.

Given this collection of untimed behaviors exposed in the functional partition, the timing partition is then responsible for providing the necessary timing and sequencing of these behaviors. Computing the timing and sequencing of these behaviors can be quite simple when the timing is not data dependent and it follows simple delays (as observed in the three-stage delay pipeline example). In more complex models, the timing partition may include various schedulers, arbiters and caches as necessary to compute precisely when certain behaviors occur.

While traditional simulation uses logical processes, which pass timed events to communicate, an FT-partitioned model may communicate by transferring *traces* from the functional partition to the timing partition in response to requests. However, unlike their timed event counterparts, traces only contain the value results from the triggered behavior and they do not contain any timestamp information. The trace may contain the entire results of the triggered untimed behaviors, or it may be limited to only those values relevant for computing the timing of the behavior itself. Depending on how the FT partitioning is designed, these trace may contain register and memory values or they may simply act as placeholders to confirm the triggering

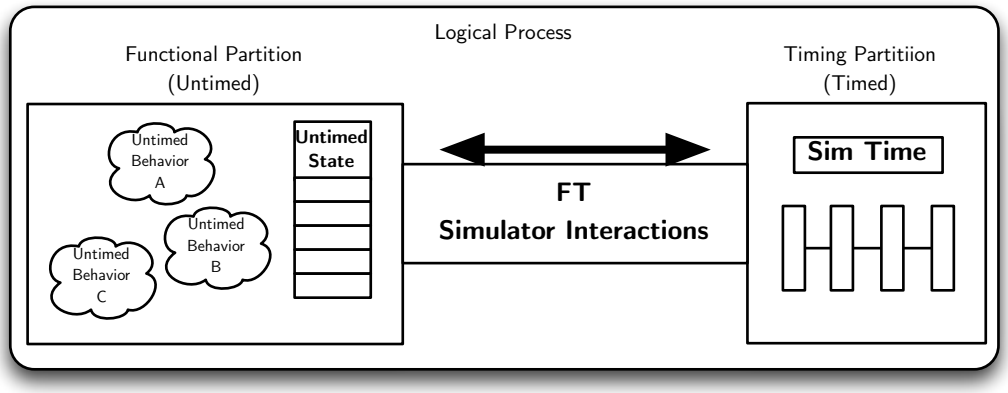

Figure 2.3: FT-Partitioned Model Interaction

of an untimed behavior.

Unlike the generic approach to logical process communication, the particular interaction between functional and timing partitions may be explicitly designed to fit a given model. For example, when designing a partitioned model for a processor, the functional partition may be defined at the level of individual micro-ops. As these micro-ops change relatively slowly as the processor evolves, exposing untimed behaviors that dictate the functionality of each micro-op is a useful boundary. The matching timing partition for this functional partition would then respect this micro-op interface when simulating the flow of instructions and micro-ops through its timing pipeline. By explicitly defining the model in a partition, the micro-op functional partition may be reused across designs of the processor.

2.3 FT Partitioned Design

Designing an FT-partitioned model can be viewed as another modeling activity that adheres to the normal tradeoffs of speed, cost and accuracy. An FT partitioned model must make key decisions regarding (i) which partition initiates interactions (ii) the granularity of partition interactions and (iii) how the state is split across partitions.

Choices made along these three dimensions can have an impact on partition reuse and parallelization.

2.3.1 Initiator

The most basic choice made in FT-partitioned models is selecting which partition initiates the request: functional or timing. In a timing-directed model, the timing partition explicitly sequences the untimed behaviors as required, which allows for triggering behaviors only when they are certain to be needed by the timing partition. Alternatively, a functionally-directed model may compute a sequence of untimed behaviors and provide the results to the timing partition for its later use. By decoupling the execution of the functional partition in this fashion, such an approach allows for greater parallelization and lower synchronization between the partitions. The downside of such an approach is the potential for executing behaviors functionally before they should have done so. We discuss this complication further in Section 2.6.

2.3.2 Granularity

The second design choice captures how many functional state transitions may be captured by a single interaction. While individual untimed behaviors may be exposed individually, it is often helpful to group multiple behaviors under a single FT partition interaction to increase efficiency and to reduce modeling complexity. For example, in the example in Figure 2.2, a single interaction could be used to trigger the single add behavior, or three separate interactions could be used to trigger three separate partial sum behaviors.

2.3.3 State Separation

Finally, the choice of how state variables are spread across the two partitions can have an impact on granularity design choices. At one extreme, a functional partition

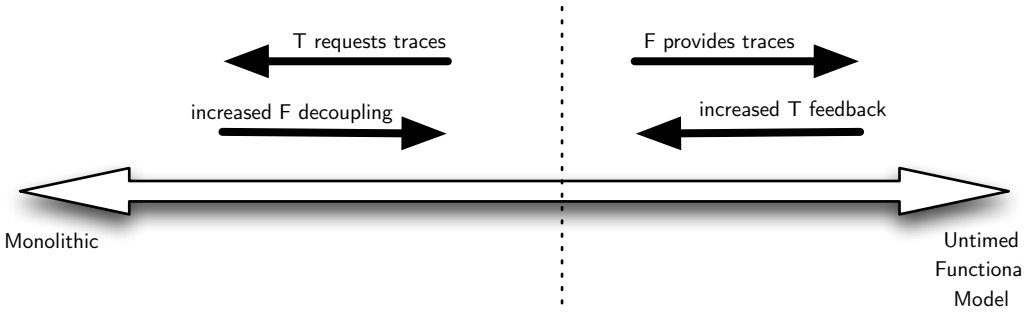

Figure 2.4: Continuum of FT-Partitioned Models

may contain no state, simply offering a set of operators that may be invoked on the state passed via the interaction request. At the other extreme, much of the state may reside in the functional partition while the timing partition state is reduced to a small subset necessary for timing purposes only. Shifting the state into the untimed functional partition can enable more efficient computation (through state sharing, state merging), but it may restrict the choice of granularity and it may have further correctness complications as discussed further in Section 2.6.

2.3.4 Design Continuum

Designing a partition model along these three dimensions creates a design continuum, spanning from tightly coupled timing-directed approaches to loosely coupled functionally-directed approaches. Figure 2.4 illustrates how the choice of initiator sets the first design point, while the choice in granularity and state-splitting extend from there.

2.4 FT Simulator Architectures

Analogous to the specialization of synchronization protocols found in PDES logical process simulation, a similar set of strategies has evolved to meet the particular

constraints of FT partitioned simulation. FT simulator architectures embody choices of how to partition a model and protocols for partition interactions. These architectures do not employ the traditional conservative/optimistic approaches described in A.4.3 as they are not communicating timed events, but they operate at a level below the typical logical process interaction. Indeed, conventional simulator protocols are applicable when treating a pair of FT partitions as a single logical process. Figure 2.5 illustrates some of the basic approaches to creating a FT partitioned model.

Monolithic

At one end of the spectrum is a completely unfactored model, that represents a conventional monolithic logical process. GPGPUSim (Bakhoda et al., 2009) is an example of a GPU simulator that doesn't clearly separate simulator partitions.

Timing-Directed (Stateless)

If we start to carve out common untimed behaviors into a parameterizable library, we can end up with a timing-directed (stateless) simulator architecture. This approach uses the timing partition to initiate requests, using fine-grain interactions with no state held in the functional partition. This level of interaction leads to strong coupling between the partitions but it allows for accurate triggering of behaviors and no potential races between timed and untimed state. Examples of this simulation approach are found in academic simulators such as M5 (Binkert et al., 2006) and Ptlsim (Yourst, 2007), which expose a micro-op library against which timing models may trigger operations by passing micro-op structures as arguments.

Timing-Directed (Stateful)

To reduce the partition coupling and complexity, it is possible to migrate some state into the functional partition. This avoids needlessly having to track state values that may not be useful for timing simulation purposes. For example, in a processor simulator, the particular values stored in architectural registers have no direct impact on timing. Further, by avoiding passing state across the partition,

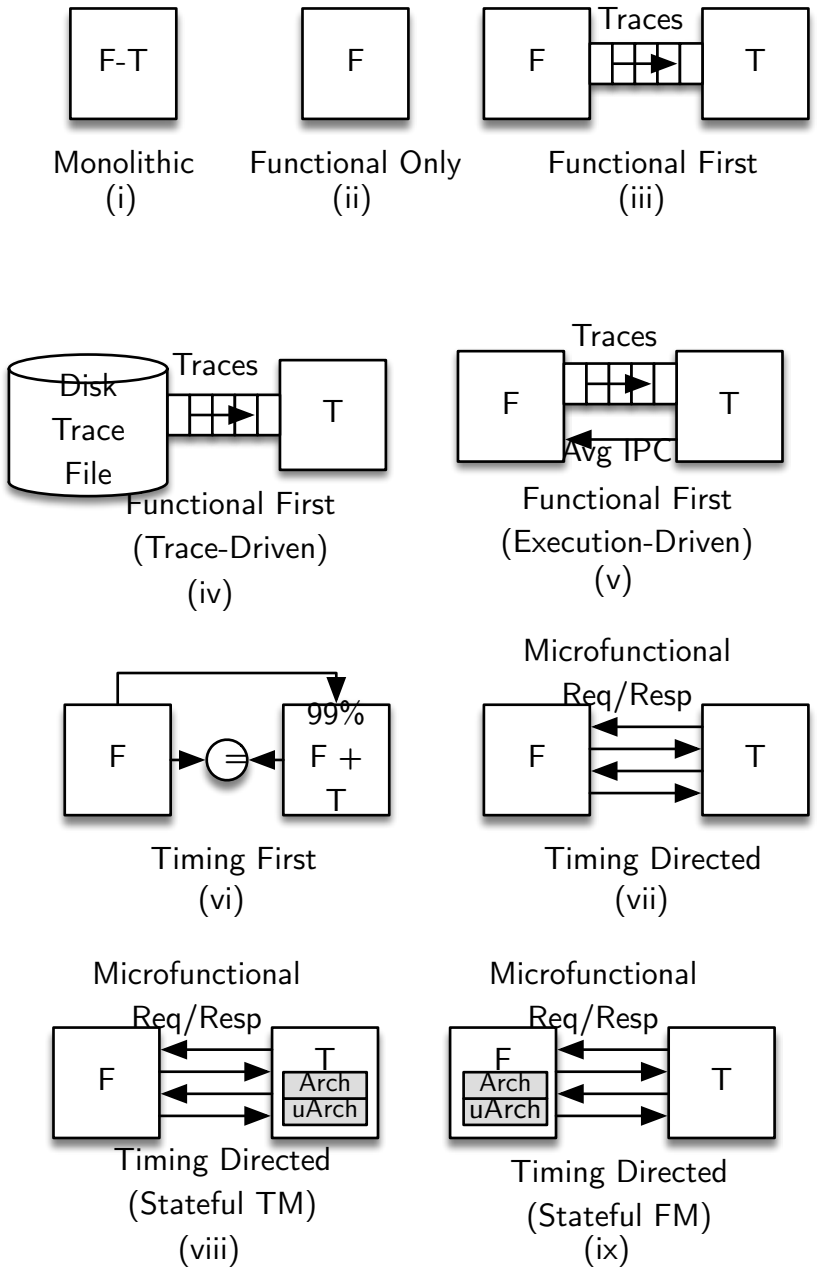

Figure 2.5: FT Partitioned Simulator Architectures

it may be possible to combine interactions that operate on related state in strict sequences. The HASIM simulator (Pellauer et al., 2011) employs this approach using an FPGA to accelerate simulation and a stateful functional partition to reduce communication and synchronization across the partitions.

Functional-Directed (Feedback)

The functionally-directed (or functional-first) approach to simulation drops the requirement for the timing partition to trigger untimed behaviors, and instead it computes functional traces in a decoupled fashion. By removing the strong interactions from the timing partition, simulators in this approach can aggressively optimize for compute efficiency by collapsing state transitions and minimizing interactions to provide only weakly coupled feedback to control functional execution. Many simulators use this approach (Renau et al., 2005; Jaleel et al., 2008) due to its lower design complexity.

Functional-Directed (Offline)

Taking the functional-directed approach to its extreme, we can precompute the entire evaluation of the functional partition and save the trace in an file for later replay. This form of simulation (also known as trace-based simulation), does not rely on feedback from the timing partition to control its execution at all. By pre-generating traces, complex systems may first be instrumented and monitored to record a trace while subsequent simulations may simply consume the results from the tracefile. Simulators such as Dinero (Edler and Hill, 1998) use this approach when simulating cache designs. This approach has fallen out of favor in recent years as compute capacity available for simulation has grown. The strong downside of this approach is the inability to easily detect modeling errors in the simulation easily. As the timing simulation simply consumes the results from the recorded functional partition, any errors that may have actually affected the simulation are hidden.

Untimed

Finally on the other end of the spectrum from the monolithic approach, we can remove timing entirely and only simulate an untimed model. This approach is highly useful for software development, so is often a early simulation artifact despite its limitations. Simulators such as QEMU (Bellard, 2005), Simics(Magnusson et al., 2002) are common examples that use this approach.

Other Variations:

Additional varieties of FT partitioned simulators have been proposed to address different aspects of partitioned simulators. Maurer et al. (Mauer et al., 2002) propose Timing-First which uses a functional-only model alongside a nearly complete monolithic model. Targeting processor simulation, a timing-first simulator attempts to model a target correctly most of the time (e.g. 99% of the time) and to use a functional-only replica running alongside this model to act as a verification mechanism. In the case where the monolithic model encounters an error, it can flush its state and restart the simulation using the state from the functional-only replica. This approach may reduce modeling effort by avoiding fixing corner cases, but it requires significantly increased computation.

2.5 FT Concurrency and Parallelization

While FT partitioning is primarily used for design complexity optimization, it can also be used to express additional levels of concurrency and parallelism in a simulator. A pair of FT partitions executing cooperatively mimic a monolithic logical process with each partition executing as its own concurrent task. As each pair of FT partitions operates as if it were a single logical process, a partitioned model may transparently work with a larger simulation with traditional monolithic logical processes, and bit may benefit from parallelization at this level as well.

One of the benefits of using FT partitions as concurrent tasks is the ability to separate the two categories of computation for specialization and optimization. The

functional partition and its accompanying set of untimed behaviors often express high-level operations applied to the modeled state. As these behaviors need not be synchronized to a simulated clock, it becomes easier to optimize and map them to available host resources efficiently. The timing partition may also be optimized by extracting the untimed behaviors, allowing optimization of simpler remaining timed operations.

We can see this difference between partitions most clearly in processor- centric simulation, where the functional partition maps conveniently to the behaviors implemented by a typical host processor. For example, if the functional partition exposes an 'add' untimed behavior, it may only require a single host instruction to execute. For the timing partition of a processor model, a similar optimization may be made, but it requires the use of non- traditional hosts such as FPGAs. For example, by extracting the functional partition behaviors from the timing partition, it becomes possible to design compact hardware engines to execute the remaining timing partition operations efficiently.

We can further specialize an FT-partitioned simulation by scheduling and mapping the functional and timing partition processes into two groups, each capable of executing either functional or timing partition processes. By separating these collections of tasks, we gain further ability to specialize the implementation and mapping to host resources. We call the collection of functional partitions across multiple logical processes forms a *functional model*. Similarly, the timing partitions from different logical processes may be grouped together to form a *timing model*.

Each pair of partitions still communicates with a structured set of request/traces. However, as functional partitions are not beholden to the communication restrictions of a normal logical process, they allow for state- sharing between functional partitions. For example, when simulating a multi- core processor, each functional partition for each core in the simulated system may be executed under a single functional

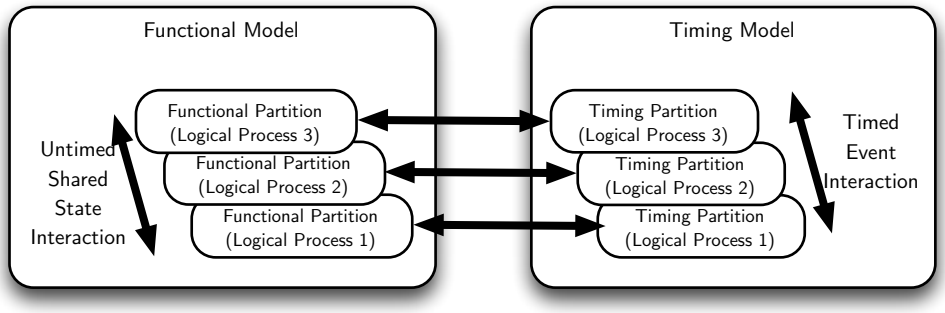

Figure 2.6: Functional and Timing Models

model, which allows sharing of simulated memory between the cores. This sharing offers both benefits and complications as the state sharing occurs without respect to the simulated time. We discuss the complications of these functional-to-functional interactions further in 2.6.

While each partition may present a concurrent simulation task, leveraging this partitioning for parallelization can be challenging. Parallelizing a strongly coupled simulator architecture such as timing-directed is difficult due to the frequent synchronization that occurs across the FT boundary. The HASIM simulator overcomes this difficulty by leveraging an FPGA hardware approach to allow low-cost synchronization and communication, but it still requires both partitions to reside on the same FPGA in close proximity to one another.

With a more loosely coupled simulator architecture, such as functional- directed, it becomes more efficient to map the functional partition and the timing partition to separate host resources and to execute them in parallel. The Cotson simulator (Argollo et al., 2009) allows leveraging of this boundary by running the functional partition in one thread, while the timing model runs in another. In practice, this level of parallelization is not often leveraged in typical general purpose hosts due to large communication overheads. Using non- traditional hardware, this approach becomes more attractive as it is possible to optimize each partition independently

and communication overheads can be managed with hardware specialization. For example, the RAMP Gold(Tan et al., 2010) and Diablo (Tan et al., 2015) FPGA simulators use this functionally directed approach with tight feedback to enable scalable multi and manycore simulation.

2.6 FT Entanglement

Partitioning along an FT boundary has many benefits for design complexity and potential avenues for parallelization. However, as we migrate from tightly coupled timing-directed execution to weakly coupled functionally-directed approaches, our guarantees on equivalence to monolithic execution start to weaken. The basic issue is that by separating *how* into *what* + *when*, there exists a potential for *when* influencing the execution of *what*.

In timing-directed approaches, functional behaviors are only triggered at precisely their required points during simulated time. Thus, any dependencies between the untimed behavior and the state of the simulation are fully captured. If a functional behavior reads from a state variable, it can be assured that the variable has the correct value at the simulated time t.

By contrast, in a functionally-directed approach, the behaviors are executed without explicit synchronization against the simulated clock. In typical cases, this execution is safe as the state variables that are read and written to by the functional partition will eventually be the same as those used in the timing partition when the functional trace is consumed. In fact, one of the primary purposes of adding feedback to a functional-first architecture is to ensure that this assumption is mostly true.

However, this form of execution retains the possibility that the state used to compute a given trace will not match the timing state when it is consumed in the timing partition. We call this dependence of the functional-partition on the timing partition *FT entanglement*. Violations of FT entanglement can be reduced

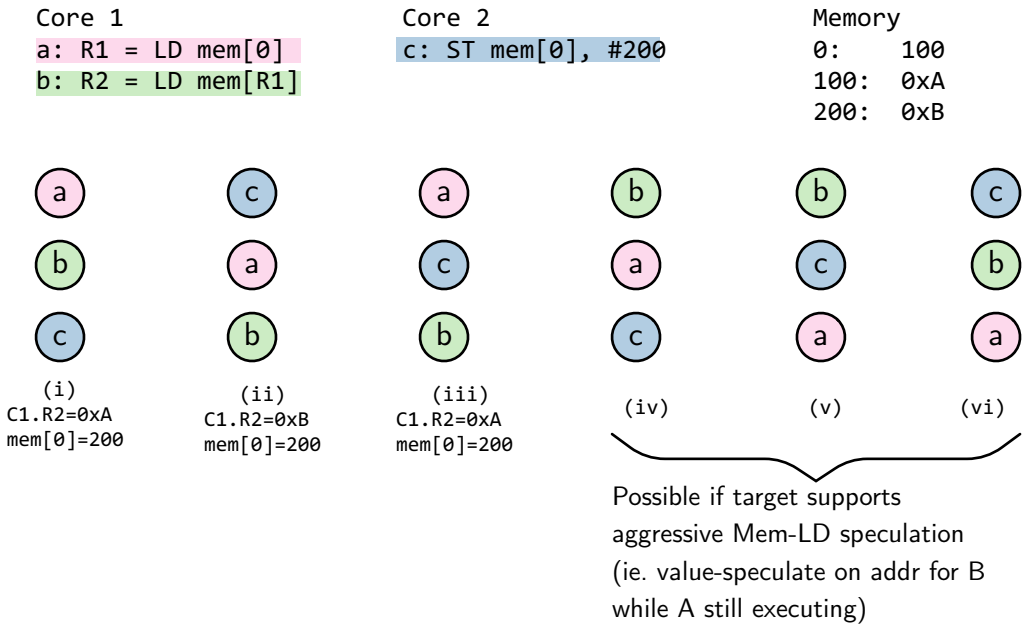

Figure 2.7: Example of FT Entanglement and Violation

by introducing tighter feedback but violations remain a fundamental limitation of functional-first simulation architectures. Even if we devolve into adding feedback on every trace execution, it is still possible to encounter violations due to FT entanglement, as seen in Section 2.6.1

2.6.1 Example: Memory Ordering

The simplest example of FT entanglement may be demonstrated with a pair of functional partitions with some shared memory held between them. In a functional-first execution, each partition may execute without synchronizing against the simulated clock. Thus any load or store against the shared functional memory may occur in any order governed only by the arbitrary choice host ordering of the functional partitions. This simple example is illustrated in Figure 2.7.

Figure 2.7 depicts various instruction execution orderings in a two-core, 3

instruction program snippet. Core1 executes a pair of dependent load instructions, while Core2 executes a store instruction. While the example is intentionally kept simple, it illustrates a problem faced when simulating multiprocessor systems that interact with shared memory locks, barriers, message passing. From the figure, the execution of the three instructions (a, b, c) can result in one of six possible execution orders (i..vi). The execution order followed by a given simulator depends both on the particular details of the target being simulated, as well as the constraints the simulator architecture may impose. In a monolithic simulator, the correct target order is executed and simulated as instructions are not executed separately from their timing-driven simulation.

By contrast, in a Functional-First simulator, the functional partition must blindly choose one of the six orders, regardless of what the timing simulation order may require for correctness. For example, a FF functional partition may execute Core1 for two instructions, then execute Core2 (resulting in order (i)). However, when the trace is simulated in the timing partition, the cache simulated for Core2 may hold address 0, allowing instruction C to simulate first (resulting in order (iii)). This silent error may go undetected as the timing partition in a Functional-First simulator continues simulation, regardless of whether target correctness has been broken.

A Timing-Directed simulator avoids these types of silent errors by only allowing the correct target execution to occur. In the case where the cache in Core2 held address $0x0$, this would result in the timing partition requesting to first execute instruction c, then requesting the execution of b, c, resulting in the correct target order (ii).

Additionally, it can be observed that even if a Functional-First simulator increases its synchronization with the timing partition down to a single instruction, it is still possible to execute a before b. Further, it is possible for alternative execution

orders to exist that may be difficult to execute in a functional partition that only permits architecturally correct execution. Consider order *iii* to *vi*, which can be supported in targets that permit load-value speculation. When simulating such targets, a Functional-First functional partition would require additional execution control not often found in functional partitions that simply execute instruction-at-a-time.

By allowing the functional partitions to execute without knowing timing ordering for any shared memory accesses, we introduce the potential for FT violations. These violations may manifest themselves as timing inaccuracies (e.g. incorrect addresses used when simulating performance of caches) which increases the uncertainty of simulation results. These violations may also result in hidden errors by allowing the functional ordering of the simulation to hide bugs that should have been exposed during simulation. For example, if a new memory consistency protocol is being evaluated, it is important to simulate the new consistency memory ordering faithfully in addition to its timing characteristic to ensure that the protocol is both performant and correct.

2.6.2 Implications of FT Entanglement

We observe that FT entanglement is a natural property of parallel systems that have communicating processes. While in many cases the effects of entanglement may be mitigated by feedback the underlying dependence of functionality on timing remains. This leads us to two observations that influence the design of FT partitioned simulation.

First, to ensure identical equivalence to monolithic process execution and to avoid adding any uncertainty to the simulation results, we require tightly coupled timing-directed simulation. We examine how to relax this requirement by explicitly coping with FT entanglement in Chapter 3.

Second, execution under a functional-first FT-partitioned simulator may result in higher or lower contention for shared resources, leading to different behaviors simply due to timing. We will use this ability to influence functional execution through timing in Chapter 5.

Chapter 3

FT Optimistic Simulation

3.1 Introduction

Given the potential benefits from FT partitioned simulation (particularly functionally driven simulation) if we can solve the issues surrounding FT entanglement, we can unlock a more efficient means of parallelization without compromising on model accuracy.

3.1.1 Addressing Challenge 1: High-Speed Detailed Simulation

If we consider the first open simulation challenge, high-speed detailed simulation (see Section 1.7.1), we have a stringent set of requirements to cope with. Simulation addressing late-stage architectural refinement requires high-accuracy, low-uncertainty modeling that limits many of the approaches used to enable high-speed simulation. Parallelization at this design point is also particularly difficult due to the interactions that occur between logical processes as processor models synchronize with each other at every cycle. This interaction creates low lookahead that limits traditional approaches for parallelization. Further, given the high-levels of detail required, the computation costs for modeling even a single target processor can be significant,

Simulator	Mode	ISA	Speed (MIPS)
SimpleScalar	outorder	Alpha	2.229
SESC	cmp	MIPS	1.000
GEMS	ruby-opal	SparcV9	0.025
M5	atomic	Alpha	3.242
M5	inorder	Alpha	0.052
M5	detailed	Alpha	0.190
Zesto	fast	x86-32	2.570
Zesto	ooo	x86-32	0.139
PTLsim	seq	x86-64	8.330
PTLsim	ooo	x86-64	0.350
GPGPUSim	simt	PTX	0.174

Table 3.1: Simulation Speeds for Contemporary Simulators. Single-core cycle-accurate simulation on commodity x86 single-core @ 2.4Ghz host.

leading to thousand to million times slowdown in simulation rates. Tackling this design point requires simultaneously addressing the computation complexity and parallelization of the simulator without compromising on fidelity.

In Table 3.1, we survey a number of contemporary cycle-accurate and cycle-approximate simulators. We see that most software simulators are limited to a few hundreds of kilo-instructions per second in simulation rates when simulating a single core. Given the difficultly of efficient parallelization, many of the simulators forgo explicit parallelization. Those that do implement parallelization employ both model fidelity simplifications and some form of relaxed synchronization to enable efficient scaling across multiple host resources. The overall result is that cycle-accurate simulation in pure software is currently limited to ten-thousand times slowdowns that only become worse as the number of processor cores increases. For example, if we wish to simulate one second of the target on a 64 processor target with the M5 simulator, we would require nearly eight days of host execution.

To tackle the challenge of high-speed simulation while retaining fidelity, we reexamine the use of FT parallelization. One of the benefits of parallelization at the

FT boundary is the ability to specialize and optimize functional and timing separately (particularly with functionally-directed architectures). Given the high computation costs involved in simulation, one of the key aspects to enabling high-speed simulation is to execute the simulation more efficiently. The complication with this approach is the limitations imposed by FT entanglement that prevent leveraging functionally-directed architectures without sacrificing fidelity and accuracy. If we can solve this issue, we may design simulators that can better tackle the high-speed, high-detail simulation design point more freely without compromise.

3.1.2 FT Parallelization and FT Entanglement

The benefits of leveraging parallelization along an FT boundary in a functionally-directed simulator are numerous. By decoupling the execution of the functional partition from the timing partition, we make parallelization more tractable. This decoupling also enables more aggressive optimization in each partition. In the functional partition, the ability to collapse state- transitions, allow state-sharing between untimed behaviors and efficiently optimize the functional partition for the underlying host resources all lead to reduced compute complexity and higher execution efficiency. Similar optimizations can be made in the timing partition by extracting complex functional behaviors and replacing them with simple trace lookups.

The complication with simply adopting a functionally-directed approach for parallelization is the chance of violating correctness due to potential FT entanglements that may exist between the partitions. As the functional partition accesses shared state that may be timing dependent, it is possible to generate simulation errors due to FT entanglement.

Solving this issue is difficult without giving up on the benefits of the functionally-directed approach. If we pull the timing partition closer, increasing feedback and reducing communication granularity, we limit the chance of violations but we decrease

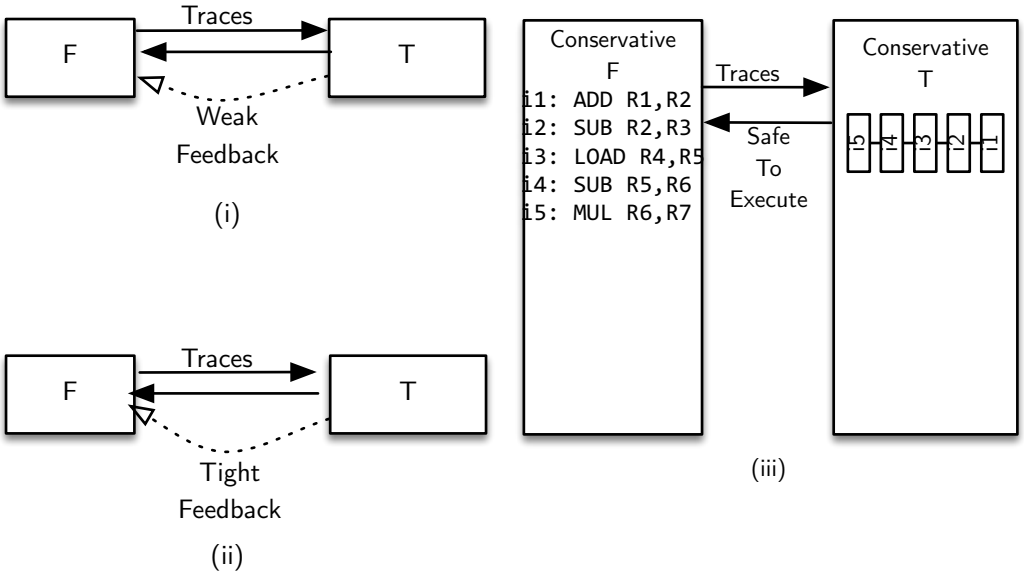

Figure 3.1: FT Entanglement Issues

the ability to parallelize and optimize the partitions due to increased synchronization. We illustrate this approach in Figure 3.1 (ii).

It is also not possible to simply to act conservatively by pausing functional execution when any shared state is accessed. With certain timing partition models, the functional partition may need to generate multiple traces before the timing partition may advance at all. For example, consider the example in Figure 3.1 (iii) with a canonical five-stage processor pipeline that may require five instructions inflight at any time. If the functional partition executes "instruction-at-time", it must necessarily lead the timing partition by five instructions.

As the timing partition cannot make forward progress until a trace is available at the start of the pipeline, it is not possible simply to pause the functional partition waiting for feedback from the middle of the pipeline to prevent a potential FT violation. In the example presented in the figure, we cannot simply wait for clearance

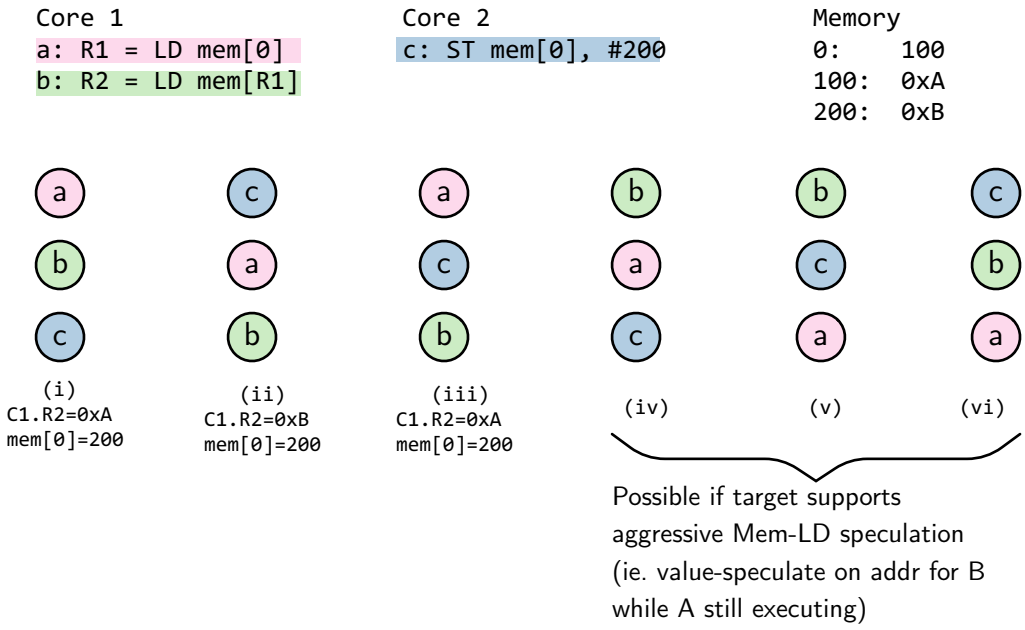

Figure 3.2: FT Entanglement Motivating Example

to execute $i3$, as we require traces from $i4$ and $i5$ simply to allow the timing partition to make forward progress.

3.1.3 Motivating Example

To better illustrate the problem of trying to keep functional and timing partitions decoupled while still maintaining target correctness, the simple example of FT entanglement from Chapter 2 can be revisited. Figure 3.2 depicts the three instruction, two core program snippet again.

As previously described, a Functional-First (FF) simulator executing such a program snippet would choose one of the six possible orders (executing blindly without precise timing knowledge). This may result in the functional partition executing a, b, c (order i). However, when the resulting trace from this execution is passed to the timing partition, the cache in Core2 may hold address $0x0$, resulting in

the timing partition requiring c, a, b (order ii). Such a FT entanglement violation is an inherent possibility with a functional-first simulator, allowing silent errors and simulation inaccuracies to be introduced into simulation results.

Even if the functional partition is forced to synchronize with the timing partition after every instruction, precisely ordering even single instructions (e.g. a relative to b) is difficult without precise timing partition knowledge.

3.1.4 Avoiding FT Violations with Optimistic Speculative Execution

From the previous memory ordering example, potential FT violations are an inherent property when attempting to execute a partitioned simulator without precise timing ordered execution. The question then arises if it is possible to preserve decoupled execution for execution and implementation benefits while simultaneously preserving target simulation correctness.

A common solution to breaking dependencies in parallel systems is to use a optimistic speculative execution approach. Using optimistic simulation has been previously applied to uni-processor simulators to handle execution on mis-predicted branches in functional-first simulators ((Chiou et al., 2007; Schnarr and Larus, 1998; Moudgill et al., 1999)). Such simulators execution instructions speculatively in the functional partition while the timing partition handles detection when this speculation on program-counter addresses is incorrect. In a multi-processor simulator setting, these prior optimistic approaches have limitations (further detailed in Section 3.7), but the basic approach of speculate, detect and correct is a useful approach to avoiding potential FT violations in a parallel simulation.

Using an optimistic simulation approach with the previous example, if the functional partition executes instructions $a, b, c(orderi)$ but also recorded this choice, it becomes possible to later detect if the timing partition required an alternative

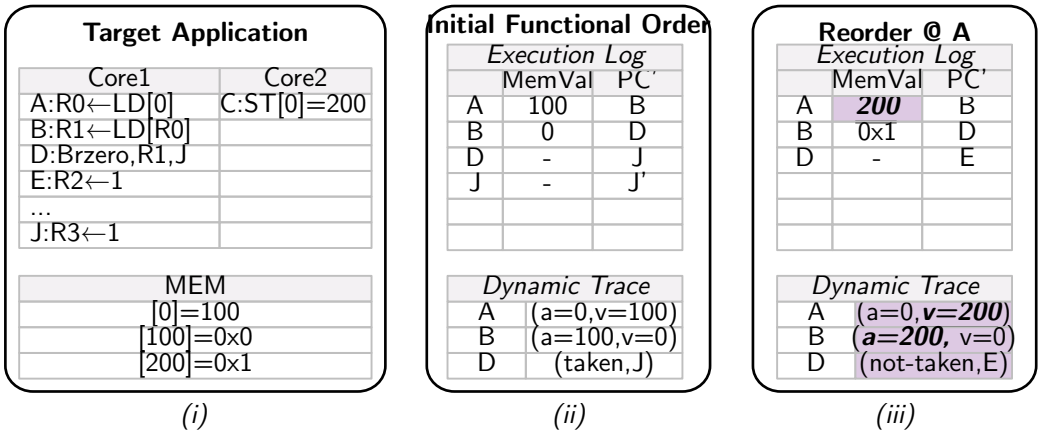

Figure 3.3: Example of solving FT entanglement with speculative execution in a simple non-pipelined target model

order (e.g. $c, a, b(orderii)$). If the functional execution order is incorrect, it can be corrected to prevent the FT violation from creating inaccuracies in the simulation.

Figure 3.3 provides a more detailed example to demonstrate how speculative execution can be used to avoid FT violations. In (ii), an initial functional order is speculatively executed as a result of executing a functional order of a, b, c, d, e. This generates a dynamic trace such that each instruction records both its execution order and additional instruction information such as dynamic address values and data. For example, a records its load from $0x0$ resulting in a value of 100.

If the timing partition ends up simulating such that Core2 has address $0x0$ in its cache with write permissions, instruction c will execute before a, leading to a divergence at b. Assuming the timing partition can detect what the timing correct execution order is, it can then detect the violation before allowing instruction b to simulate in the timing partition. For example, when the timing partition attempts to consume the trace, it verifies it to ensure the simulator speculation is target correct (in this case, verifying that instruction a loaded a value of 0x200 (corresponding to the effect of executing c before a). If this verification fails, the functional partition can be rolled back to instruction a and the correct value of 200 patching up the

functional partition execution (resulting in (iii)).

This simple example illustrates that if the functional partition can record its execution order and the timing partition can validate and provide corrections to this order, it may be possible to retain speed and flexibility benefits of functional-first simulation without compromising of the accuracy benefits of timing-directed simulation. As with any optimistic approach, achieving overall performance benefits requires efficient mechanisms for detecting and correcting these FT violations. For example, a naive optimistic FT implementation could run a duplicate monolithic simulator to act as the timing partition verifier to satisfy optimistic execution requirements while failing to improve performance. Creating mechanisms for efficient and scalable detection and correction optimistic simulation is one of the key contributions in this work. The remainder of the chapter provides details on these mechanisms for implementing this particular simulator architecture which is termed *optimistic-FT* simulation.

3.1.5 FT Entanglement Execution Constraints

One of the difficulties in safely leveraging FT partitioning for parallelized simulation execution is knowing precisely when a simulation may be free from FT entanglement and potential violations. A typical functionally-directed simulation may encounter cases of FT entanglement without any explicit indication that a potential error exists or has occurred during simulation. It is only upon comparison with a timing-directed execution or careful examination of the ordering between access to shared state that such cases can be detected. We cannot simply rely on traditional PDES constraints such as the local causality constraint causality (LCC) as FT-partitioned communication is not structured as timed event communication. While the LCC restriction allows verifying adherence simply by monitoring monotonic event timestamps during processing, FT entanglement and violation prevention requires examining how state

variables are accessed from the functional partition. One of the contributions of this work is the development of a set of useful constraints to guide safe FT parallelized simulators, avoiding this uncertainty.

One approach to constraining FT partitioned simulation is to eliminate the potential for FT violations by treating all untimed functional behaviors as timing dependent. By forcing the functional partition only to access state variables in timed order, we prevent any violations due to executing using untimed state variables. We then have a constraint where *an FT-partitioned simulation is free from FT-entanglement violations if all state variable accesses are ordered with respect to the simulated clock in the timing partition.* This constraint is however heavily restrictive as it effectively forces the use of a tightly coupled timing-directed approach, negating many of the benefits of the functional-first approach.

Instead of treating all untimed functional behaviors as timing dependent, it may be possible to allow only those that are truly timing dependent. Consider a pair of functional partition processes that do not share any state variables. As these private state variables are only read or written to by a single functional partition, we disallow any potential FT entanglement due to ordering between pairs of partitioned logical processes.

While this constraint of only allowing private state variables is useful in preventing violations due to interactions between pairs of functional partitions, it does not fully ensure that all private state variables will be accessed identically between the functional partition and its matching timing partition. Consider the case of a functional partition generating a trace by accessing state variables in the order $s1, s2, s3$ while the corresponding timing partition consumes and simulates the traces so that it accesses state variables in the order $s3, s2, s1$. This scenario could occur if the functional partition generated its trace sequentially while the timing partition consumed the trace with some latency-sensitive ordering of traces. In this case, even

though we have private state variables, the state accesses are still entangled.

To correct this limitation, we can extend the constraint to ensure access order to a given state variable is preserved between pairs of functional/timing partitioned processes. As traces are generated in the functional partition, they read/write state variables and when these same traces are consumed in the timing partition, they implicitly access the state variables used to generate the trace. If we ensure each state variable is accessed in the same order, regardless of how traces are generated or consumed, we can ensure that no FT-entanglement violations are possible.

We then have a constraint: *an FT-partitioned simulation is free from FT-entanglement violations if (i) for each untimed state variable, it is accessed by exactly one functional partitioned process and (ii) for each untimed state variable, the trace generation access order is equal to the trace simulation access order.* While this additional access ordering constraint may seem restrictive, ensuring equivalence is made easier by the structured algorithms for accessing state variables in processors/accelerators. For example, a processor may follow a dataflow dependence ordering of instructions that is guaranteed regardless of whether the timing is a multi-cycle microcoded engine or a deeply pipelined out-of-order processor. In such a case, a functional partition generating a sequential instruction trace can be safely consumed by a timing partition with the understanding that all state variables may only be accessed in dataflow dependence order (ensuring Clause ii of the constraint).

So we are left with two approaches to execute FT partitioned simulation, both with non-trivial limitations. In the first approach, we may only allow timed state variable access, which effectively disallows functional-first approaches (and their accompanying benefits). In the second approach, we can allow a functional-first approach if we can guarantee no functional state sharing and ensure that state variable access ordering is consistent between partitions. If we cannot guarantee both of these restrictions, we must rely on the first approach to ensure a partitioned

simulation does not incur FT entanglement violations.

It is possible in limited cases to use a hybrid between the two constraints. For example, if we have a single state variable that cannot satisfy the state sharing restriction, it is possible to fallback to our first approach of timed state access whenever an access is made to this particular variable. For example, we can interpret the work done by Schnarr et al. (Schnarr and Larus, 1998) as an instance of this approach. As the program counter in a processor simulator may be written by two sources (the correct *nextpc* and the *pc* predicted by a branch predictor), we cannot directly apply the private state variable restriction needed to allow a functional-first approach. Instead of forcing the entire simulator to rely on timing-directed execution, Schnaur limits this fallback only on branch instructions where there is a potential FT entanglement, while retaining functional-first execution for the remaining instructions.

This fallback to a timing-directed approach is limiting when simulating parallel systems that share memory and communication channels. If we require fallback to timing-directed approach for most state variables, the benefits of the functional-first approach are minimized as the optimization opportunities decrease and the synchronization requirements increase. It may be difficult to allow partial fallback on individual state variables due to limitations on how the functional partition executes its remaining untimed behaviors (i.e. moving one state variable to timing partition may disallow a range of common operations). Further, the inability simply to stall execution when entanglement is detected in pipelined processor simulation may result in not being possible to fallback to a timing-directed approach in a partial fashion.

The heavy execution constraints on FT partitioned simulation create complications to using this approach in a practical parallel simulator. If we are forced to rely on timing-directed approaches to gain safety from FT-entanglement violations, we effectively forgo any of the benefits gained from the functional-first approach.

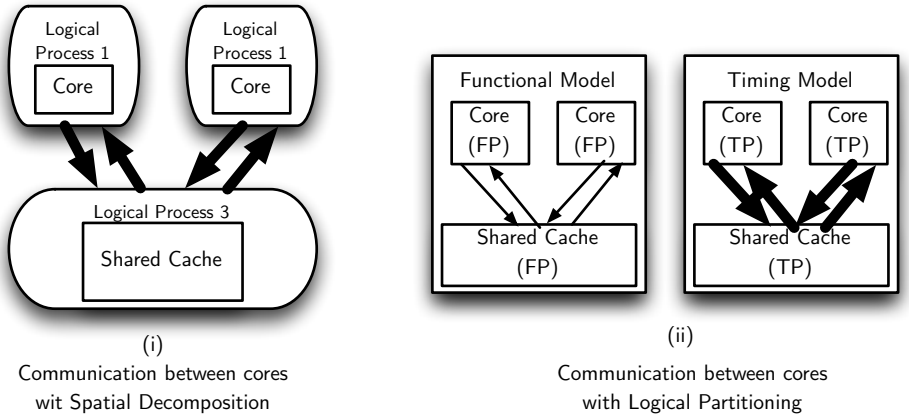

Figure 3.4: Communication Differences between Spatial and Logically Partitioned Simulation

3.1.6 Optimistic Simulation

Given the constraints on safe access to shared state variables, existing optimistic simulation approaches may seem useful as they attempt to solve this tension between safety and performance. Optimistic PDES simulation however must be applied to logical processes, which in a typical multicore simulation may rely on spatial decomposition along processor core boundaries. Figure 3.4 (i) depicts an example simulation of with a pair of processor cores sharing a cache.

For example, we could apply a TimeWarp optimistic simulation protocol on each core of a multicore simulation, allowing each core to speculate on events traded between shared caches. The difficulty here is the high frequency at which events are traded between the cores and caches as these represent bus transactions to various cache addresses. As the cores access different addresses, they may use different parts of the cache interconnect (ports, buffers, banks), and may create contention even if they access non-overlapping addresses. If we attempt to apply an optimistic protocol at this level we may encounter many rollbacks as any cache request could trigger a rollback with the TimeWarp protocol.

Instead of attempting to use existing optimistic approaches on complete

logical processes, we may consider designing an optimistic approach that leverages the partitioned processes instead. Consider Figure 3.4 (ii) which illustrates the difference in communication a partitioned model may bring. By partitioning the model, we extract a different communication pattern that was hidden under the spatial communication required between the cores and caches. In the functional model, the cores trade load and store accesses to the shared cache state. These accesses only truly conflict when they access the same memory address, rather than when they simply occur at the same timestamp. If we design an optimistic approach that can be applied at this FT boundary, we can take advantage of this difference in communication patterns.

From the FT entanglement execution constraints, we can safely simulate a functional-first partitioned process if we can guarantee private state variables and the same access order between partitions. For those state variables where we cannot guarantee this constraint, instead of forcing a timing-directed approach, we may apply an optimistic approach at these points of interaction. Instead of encountering hidden FT entanglement and potential violations, an optimistic approach replaces these state variable accesses with a speculate, detect and correct mechanism.

In contrast to an optimistic PDES approach, instead of speculating on timed events, an optimistic FT simulation speculates on untimed state access. By assuming that each access to a state variable from a functional-partition adheres to the "single owner, same access order" principles, we retain all the benefits of a traditional functional-first simulation. By detecting when such assumptions are broken during simulation runtime, a rollback-repair process similar to the approaches used in PDES TimeWarp simulation can be used. In this way while an optimistic PDES simulator attempts to gain performance without incurring causal errors (rolling back when necessary), a similar approach can be applied to avoid FT-entanglement violations.

Designing an optimistic FT-partitioned simulation scheme relies on three basic

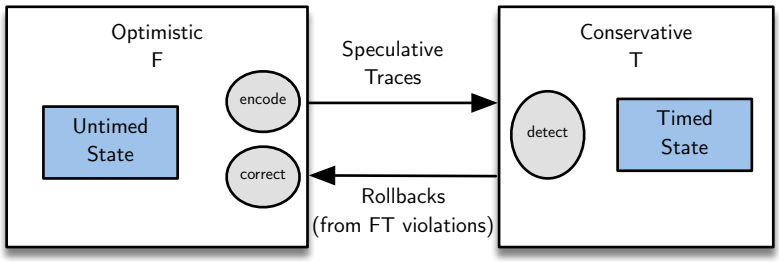

Figure 3.5: Optimistic FT Simulation

actions as illustrated in Figure 3.5: (i) encode speculative decisions, (ii) detect when speculation is wrong, and (iii) correct the mis-speculation. First, when state variables are accessed from the untimed functional partition, each access to a variable that fails to adhere to the FT entanglement execution constraint must be explicitly tracked. Second, when the speculative traces are consumed in the timing partition, a means of detecting when an FT violation has occurred must be available. Finally, given a detected FT violation, the functional partition must be capable of correcting the speculative trace.

3.2 Traces, Prophets and Oracles

3.2.1 Traces

In an FT partitioned simulator, a trace forms the basic unit of communication from the functional partition to the timing partition. As previously described in Section 2.2, a trace contains the functional computation results necessary for the timing partition to simulate the precise timing. Typically, a trace sent from the functional partition to the timing partition only needs to contain timing-relevant portions of the computation. We can typically safely omit register values, intermediate state transitions and other fields only relevant to the untimed functional behaviors. For example, we may encode

opcodes and which registers were used during the computation of an instruction which the timing partition may use to model resource contention and opcode execution latencies.

The key difference in an optimistic FT simulation is that the trace being generated is speculative and may contain invalid computation results due to FT entanglement violations. To cope with this speculation, we can extend the trace to include additional information not strictly needed for timing purposes. The timing partition needs to extract two additional pieces of information from a trace to handle the cases where the trace is invalid due to a FT-entanglement violation.

By extending the trace to include identifying when potentially FT-entangled state is loaded or stored to, we enable the timing partition to identify a precise portion of the functional computation when a violation may have occurred. Any convenient scheme for generating these ids may be used. For example, in a trace generated by a processor, we may simply use a monotonic sequence number to name each trace while loads from shared memory that may be FT entangled are simply marked with an explicit variable id.

By extending the trace to include explicit ids to identify which parts of the functional computation may require repair, we address one part of the optimistic FT correction.

3.2.2 Prophets

From the FT entanglement execution constraints, we see that for any state variable that fails to adhere to the principles of single-owner/same-access-ordering, there is a potential for FT entanglement and violations. In an optimistic FT simulation approach, we first explicitly identify every state variable that may fail to adhere to the execution constraints. When a state variable is treated as potentially FT entangled, we cannot simply access the variable as if it were untimed, as this would

lead to hidden FT violations. However, as the common case is for these variables not to manifest in violations, we must attempt to allow the appearance of untimed access to the state variables. For example, consider the case where a processor has a large physical memory that may be potentially shared, but often it is used as purely private memory. As the execution constraints cannot be guaranteed, we cannot safely allow untimed access and must conservatively treat all accesses as potentially entangled.

A *prophet* is a simulator construct we can insert into a functional partition process to protect access to any state variable that may be FT entangled. In the common case a prophet will attempt to return a value that approximates the actual *timing correct* value. However, as a prophet need not always be truthful, the value returned may sometimes be incorrect and may lead to an FT entanglement violation. As a result, the prophet must also explicitly record its approximation and the state variable that was accessed, inserting both as part of the extended speculative trace. By recording the approximations delivered by all prophets used in the generation of the trace, the timing partition may use this knowledge to detect when a violation has actually occurred.

A prophet may be implemented as a simple mediator around an untimed state variable that simply records the current value of the state variable and adds it as part of the trace. For example, in our shared memory example, a prophet would simply return the value of the shared memory, but would also record the value as part of the trace. By explicitly mediating each access to shared memory with a lightweight prophet, we gain the benefits of untimed functional state sharing necessary for efficiency, while also ensuring safety from hidden FT-entanglement violations.

Prophets may also be implemented with additional behaviors to approximate the timing correct value of a state variable. For example, the *branch predictor predictor* introduced by Chiou et al. (Chiou et al., 2007) in their optimistic simulator can be reinterpreted as a more complex program-counter prophet. A prophet may

also use different methods to record the approximation delivered to the functional partition. Instead of recording the value of the approximation itself, the prophet may use a summary of the approximation, such as version numbers or content hashes, may be used instead. This can be useful when a functional partition may access a large set of state that changes very infrequently (e.g. instruction memory pages that only change on new process startup).

The prophet simulator construct provides an explicit and verifiable means of ensuring the safety of FT partitioned execution that cannot adhere to the execution constraints, while allowing efficient untimed state access. This construct is used to address the first aspect of an optimistic FT approach, encoding speculation.

3.2.3 Oracle

While prophets provide a means of encoding when FT-entangled states are used to generate traces, we still require a method to detect when this usage actually manifests in a violation. In the optimistic FT simulation approach, the timing partition must validate these speculative prophet values by comparing them to the correct timed state variable value. As in our introduction of a prophet, we create a corresponding simulator construct, a *timing oracle* that can provide the source of this information. As an oracle is a simulator construct, it is free to generate this source of information in any way that is convenient for simulation purposes.

The naive approach to implementing an oracle is to implement both the state variable and all is required functionality in the timing partition itself. This approach however reduces the benefits of FT partitioning due to the redundant computation that may be necessary. For example, to implement an oracle for shared memory in a processor simulator, we may need to implement the entire processor functionality in the timing partition itself simply to compute the values required to populate this memory. For state variables that may be small, for example a single 32-bit cycle

counter, this approach may be adequate. As the oracle must only track states to which prophets are attached, it is possible to retain many of the benefits of FT partitioned execution.

An alternate approach to implementing the oracle is to leverage the prophets to capture the update to an FT-entangled state. Typically a prophet only records the reads from its attached state variable, while writes to this variable could be executed without intervention. However, if the prophet records both the read and the write, it is possible to populate an oracle with these writes.

As the speculative trace is simulated in the timing partition, the trace is incrementally validated, and it is only used at the required point in simulated time. By simply allowing the prophet writes to update a timing oracle when the corresponding trace is simulated, the timing oracle will always reflect the correct timed state variable. As the actual functionality required to compute the value of these prophet writes still resides in the functional partition, we only require the storage for the state variable to remain in the timing partition, avoiding duplication of functionality.

For example, to maintain a timing oracle for shared memory, we simulate the timing of the system as we would do even without an optimistic-FT approach (e.g. processors, caches, interconnect, DRAM). When an instruction actually attempts to update the cache or DRAM, in addition to simulating the timing, we propagate the prophet captured store at the same time. By piggybacking on the existing timing simulation and leveraging the values computed in the functional partition, the overhead of maintaining this timing oracle can be kept to a minimum.

The timing oracle (whether implemented by full implementation or via prophet write propagation) solves the final piece of the optimistic FT simulation puzzle, *speculative trace correction*. By identifying a trace by id (as our extended speculative trace allows), and by comparing the FT entangled reads (via the prophet) against

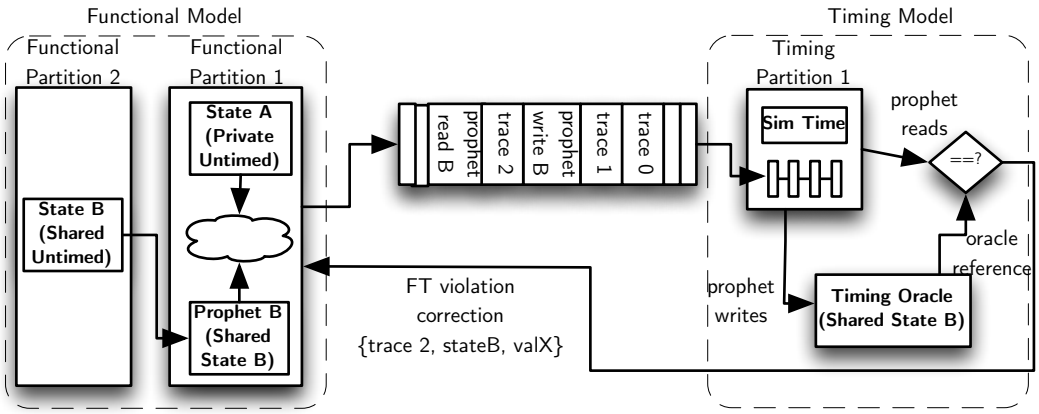

Figure 3.6: High-Level Architecture of Optimistic FT Simulator

a reference value (via the timing oracle), we have enabled an optimistic simulation scheme.

3.3 Optimistic FT Simulation

Figure 3.6 depicts a high-level architecture of a optimistic FT simulator with a pair of functional partitions with shared memory. Under normal execution, the functional partition reads values provided by the shared memory prophet, while the trace is appended with these values. Similarly for updates to shared memory, prophet writes capture the values and append them to the trace. When the timing partition consumes the trace, it uses the normal timing parts of the trace for simulation, passing the prophet reads and writes to a timing oracle for validation. Under normal conditions, this validation passes and there is not entanglement violations.

If the timing partition continues simulation it may encounter a scenario where the timing oracle value is different than the value provided by a prophet write. In that case, the simulation for the timing partition in question is paused, and a correction request is sent to the corresponding functional partition. The correction request includes the trace that encountered the violation, the prophet variable that was not

correct, and the corrected value that should be used.

When the functional partition observes a correction request, it must rollback to the trace named in the request and patch the prophet read with the provided value. The functional partition then enters a replay/re-execute mode until all traces that are currently in use by the timing partition are patched. The functional partition re-executes all traces, but it replays all prior prophet reads (including the newly corrected one). By re-executing traces and replaying prophets, the functional partition can be rolled back and corrected incrementally, correcting each prophet read one at a time. This incremental correction with prophet replay ensures that the prophet load may only cause a violation once, with the oracle corrected value being used after the violation is detected.

3.4 Motivating Example Revisited

We can revisit the example from Section3.2 to describe more precisely how Oracles and *Prophets* allow efficiently implementing an optimistic execution simulator to avoid FT entanglement violations.

Figure 3.7 depicts the LD/LD/ST example again as being simulated in a two-core non-pipelined target. The figure illustrates a functional partition holding two cores (FCore1, FCore2) and a timing partition with TCore1, TCore2. Initially, the functional partition executes order A, B, C, only allowing the *store* instruction to update $0x0$ after the *load* has executed. As each memory access is performed in the functional partition, a memory prophet records the addresses and values in the trace.

In the example, consider the timing partition selecting TCore2 to simulate C first (e.g. cache in TCore2 has $addr = 0x0$ in it). As instruction C simulates, it uses the value recorded by the prophet to update the oracle with $mem[0] \rightarrow 200$ denoted in step (i).

The timing partition then attempts to simulate A and the trace containing

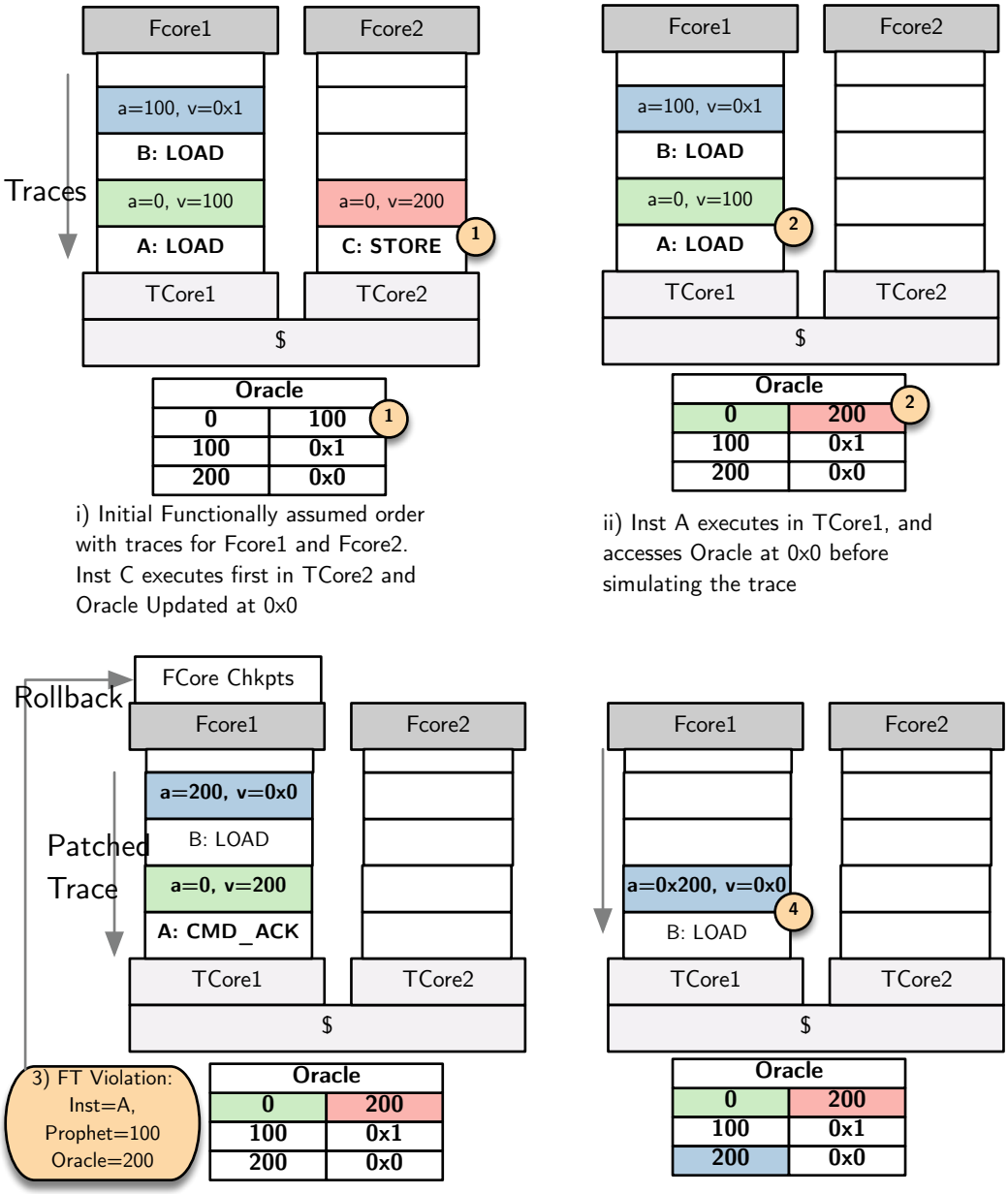

Figure 3.7: Example of an Optimistic-FT simulator running a simple non-pipelined two-core target. Functional partition executed $A(load), B(load), C(store)$ while timing partition attempts to simulate $C(store), A(load), B(load)$. 1) Store $0x0$ updates oracle. 2) Load $0x0$ accesses oracle. 3) FT violation detected, corrected and trace updated. 4) Load $0x200$ simulates. Functional trace has been corrected to the desired order C, A, B

the prophet load is validated. As the prophet recorded ($mem[0] == 100$), while the timing memory oracle indicates it should be 200, a FT violation is detected as denoted in step (ii).

A correction command of *{inst=A, val=200}* is sent back to FCore1, which rolls back to A and updates the prophet read to return 200 instead of 100 as denoted in step (iii). As the target being simulated is non-pipelined, all instructions not yet currently fetched by TCore1 can be re-executed. Inst B re-executes using the updated register value from A, causing it to load from 200. The repaired trace is sent back to TCore1 to resume the simulation as shown in step (iii).

As inst B is dependent on inst A, the simulated processor pipeline has not yet executed the second load. Load B simulates in TCore1 (accessing the timing oracle for $addr = 200$) and is safely validated by the oracle without any further violations as shown in step (iv).

The functional partition has thus been incrementally forced to execute the desired order C, A, B by using value comparisons in the timing partition.

3.5 Correctness

Ensuring the correctness of optimistic FT simulation relies on two basic properties: prophet insertion on all potentially entangled states, and enforcing selective replay and re-execution of prophet approximations.

On the first point, a prophet is inserted on all state variables that may potentially violate the FT execution constraints outlined previously. For each of these prophets, a matching oracle must be maintained in some fashion in the timing partition to verify prophet approximations. By guaranteeing that all states that cannot be proven to hold to the strong FT execution constraints explicitly use a prophet, we remove the potential of hidden FT entanglements leading to incorrect simulation.

The second point that ensures correct execution is the use of selective re-execution and replay. During the rollback/replay/re-execute phase, the functional partition must ensure that (i) previously accepted prophet corrections are retained (ii) execution eventually converges to the correct timing-directed trace. To ensure these two properties, the functional partition uses selective re-execution explicitly to regenerate traces but it also ensures that prophet reads are replayed. As traces are regenerated, if they depend on a corrected prophet, the traces will be updated with the correct values. By replaying all prophet reads, we ensure that once a prophet is validated by an oracle, its value is fixed, regardless of successive corrections that may occur later. By incrementally correcting prophets and regenerating traces with these updated values, the trace converges to the timing-directed trace as each FT violation is removed.

We can summarize the rules necessary to ensure functional correctness under an optimistic FT simulator:

- Functional partition executes all instructions correctly, preserving the required behavior of an operation
- Functional partition accesses all state variables outside its functional partition via prophet-read trace mechanism
- Functional partition updates all state variables outside its functional partition via oracle-update trace mechanism
- Functional partition can restore to any prior point of execution, restoring all functional partition state variables consistently
- Functional partition can selectively re-execute from a prior point of execution, using corrected prophet-reads

- Functional partition preserves prophet-read corrections supporting arbitrary nested re-execution

3.6 Causality

As the optimistic FT simulation preserves equivalent execution to a timing-directed simulator, each pair of FT partitioned processes still preserves the appearance of a single monolithic logical process. As such, the timing partition may be safely connected to other logical processes, communicating with conventional timed events. The details of both partitioned execution and optimistic evaluation of the functional partition are hidden from any logical processes that may interact with a timing partition. This ensures that an FT violation cannot induce a causality violation, so long as traditional PDES simulation approaches are adhered to in the timing partition (for example using conservative null synchronization between timing partitions).

We can summarize the rules necessary to ensure timing causality and precision under an optimistic FT simulator:

- TCore simulates all events in correct target order, preserving the required accuracy of the model
- TCore accesses traces when simulating events at time of use, preserving timing order when accessing trace fields
- TCores within the timing partition may consume any non-prophet fields within a trace arbitrarily (while still preserving 1)
- TCores attempting to consume any prophet fields must validate by value the prophet prediction against the current timing oracle holding the given state-variable

- TCores encountering a FT violation due to attempting to apply rule 4, must stop advancing target time for the given TCore
- TCore may simulate in parallel so long as normal LCC causality rules are followed

3.7 Prior Work in Optimistic Simulation

3.7.1 Comparisons to Optimistic Parallel Simulation

Optimistic FT simulation shares many characteristics with previously proposed optimistic simulation approaches such as TimeWarp (Jefferson, 1985) as both rely on the general properties of speculative algorithms (speculative decisions, detection and correction). However, the key distinction between prior PDES optimistic simulation approaches is their focus on applying optimism at the logical process level, speculating on timed event ordering. In contrast, optimistic FT simulation attempts to speculate on untimed state values accessed in the functional partition where the frequency of mis-speculation is lower.

3.7.2 Comparisons to Optimistic FT Simulation

There have also been some prior attempts to employ optimistic simulation at the FT boundary. However these works have been limited in various ways that prevent their approach from being used as general parallelization or simulation speedup strategies.

One of the earliest prior works that employed a very limited form of optimistic FT simulation is the FastSim simulator by Schnarr (Schnarr and Larus, 1998). The simulator employs a hybrid functional-first simulation approach that allows the functional partition to execute against untimed state except for branch/jump instructions that fallback to accessing the program counter held in the timing partition. We briefly described this simulator in Section 3.1.5 as a demonstration of using fallback

to timed state accesses when FT execution constraints cannot be guaranteed. As the functional partition only falls back to timing-directed execution on branches, the functional partition executes optimistically for at most one basic block ahead of the timing partition. As FastSim is designed for uniprocessor simulators, the result of this optimistic execution only results in additional load and store instruction traces being generated rather than true FT-entanglement violations. In contrast, our approach uses aggressive optimistic execution to enable latency-tolerant parallelization between FT, a non-goal in FastSim as both partitions run in the same host thread. Our approach also attempts to generalize the optimistic approach so it may be applied to any FT-entangled state, rather than creating special cases for different types of state.

Moudgill (Moudgill et al., 1999) presents another restricted form of optimistic FT simulation in the IBM Aria/Turnadot simulator. The simulator allows the functional partition to execute optimistically, including across branches/jumps. As the timing partition consumes these traces, it may detect an FT violation created by the program counter being updated by the branch predictor misprediction rather than the functionally correct program counter the functional partition assumed. However, unlike the optimistic FT simulation presented in our work, the correction strategy uses a lossy approximation to "correct" the trace. The Aria functional partition relies on the fact that the mispredicted instruction traces will eventually be squashed in the timing partition, so providing an approximation will not significantly affect simulation results. As our approach addresses simulation of multicore systems, using lossy approximation may hide modeling errors and increase uncertainty in simulation results.

Yoo (Yoo et al., 2000) present an interesting approach of using optimistic simulation with direct execution to drive a detailed simulator of an SoC. This approach more closely adheres to optimistic simulation between a pair of logical processes (one simulating the processor core and the other simulating the SoC and its environment).

The approach employed allows checkpoint and rollback of the core logical process, speculating across interrupts and exceptions (events that may be generated by the SoC simulator). While the approach uses optimistic execution of the core logical process, it does not employ the FT partitioning we employ in our work.

Comparison to FAST Simulator

The closest prior work to optimistic-FT simulation is found in the FAST simulator (Chiou et al., 2007). This simulator, which is an ancestor of the work presented in this thesis, shares many similarities but also has some distinct differences that stem from the uni-processor vs multi-processor centric focus in this work.

In a FAST simulator, FT partitioning is used across a CPU and FPGA. In addition, FAST uses an optimistic FT style approach to handle the communication between partitions. A FAST functional partition executes optimistically, speculating across branches that may be mispredicted by the timing partition. When the timing partition detects a misprediction, the functional partition is rolled back and the timing partition resumes consuming the trace at the mispredicted branch target trace.

The key difference compared to the optimistic-FT approach is the generalized approach to using value-based prophets and oracles to detect and correct entanglement in a scalable and efficient manner. By contrast, the original FAST simulator has two key restrictions which make applying the approach that worked for uni-core targets difficult when simulating parallel targets.

First, a FAST functional partition could not run on parallel hosts as the functional partition was executed speculatively, but was effectively checkpointed/rolled back as a monolith (so individual cores within a functional partition were not rolled back, but instead the entire functional simulation was rolled back).

Second, the detection mechanism used in a FAST simulator relied on a full

implementation for anything speculated on in the functional partition. For example, in order to speculate on mis-predicted branches, a FAST timing partition requires a full implementation of a target branch predictor to act as an oracle. If this approach is applied to parallel targets, some implementation would be required for speculating across any load or store memory value. This could require needing to implement nearly a full duplicate monolithic simulator in the timing partition simply to implement the detection/correction of any potentially FT-entangled shared state. By contrast, the optimistic-FT approach of using the values computed in the functional partition to maintain the state of the oracle in the timing partition removes the need to maintain a full implementation in the timing partition. It is sufficient to simply hold the state, but not the functionality, for all FT entangled state, making simulation of parallel targets tractable with speculative FT execution.

3.8 Summary

By introducing execution constraints, identifying points of entanglement and using abstractions of prophets/oracles, we attempt to specify clearly how to employ optimistic simulation safely at FT boundary. This clearer understanding of optimistic approaches in the context of FT partitioned simulators, makes it possible to build practical simulators (as will be discussed further in Chapter 4).

Chapter 4

FASTMP: FPGA FT Optimistic Simulator

4.1 Introduction

While optimistic FT simulation offers a promise of better parallelization and optimization, the approach is not a panacea by itself. Challenge 1 identified in this thesis focuses on the open problem of how to build high-detail, high-speed, high-accuracy parallel simulation. The optimistic FT approach opens new opportunities for parallelization and optimization, but we must translate these into practical design choices that allow for both parallel scalability and computation efficiency.

Prior attempts at simple parallelization of cycle-accurate simulation (Chidester and George, 2002; Donald and Martonosi, 2006) observed the synchronization overheads limited effective parallel speedups to small non-scalable improvements. Even if we can achieve perfect scaling, the sheer compute complexity to simulate a single core places a lower bound on simulation performance. This is the principal reason why contemporary simulators either remain purely sequential (Binkert et al., 2006), or combine both relaxed synchronization and modeling simplification to trade accuracy

for speed (Miller et al., 2010; Heirman et al., 2012; Sanchez and Kozyrakis, 2013).

One approach to reducing this lower bound on time required to simulate a single cycle of a single core it to leverage heterogeneous computing platforms (e.g. CPUs + FPGAs/GPUs). When a parallel problem fails to achieve the necessary speedup across multiple cores, introducing a FPGA or GPU or alternative domain specific accelerator is a common tactic to break through parallelization limitations. The benefit of the optimistic FT simulation approach is the ability to separate where and how each of the functional and timing partition is mapped onto host resources. We can take advantage of this fact along with this common tactic of heterogeneous accelerator platforms to address the efficiency requirements of our simulation design point.

As traditional CPU frequency scaling has slowed to crawl, heterogeneous systems have become more commonplace to address the performance and power requirements that homogeneous CPU platforms cannot support. A heterogeneous platform may be composed of a collection of CPUs, GPUs, FPGAs and other domain specific accelerators, each with its own set of tradeoffs.

CPU: Designed for general purpose workloads, typically optimized for latency-sensitive tasks. May use high clock frequency, wide out-of-order execution, and large caches to minimize memory latency. A significant fraction of chip area dedicated to non-compute portions of the architecture.

GPU: Designed for highly or embarrassingly parallel workloads. Optimized for throughput with massively parallel compute units, communication and synchronization restrictions. May run at a third of the frequency of CPUs with large fraction of chip dedicated to integer and floating point compute units.

FPGA: Designed for a bring-your-own workload model of computation. Provides a set of basic building blocks that can be stitched together to address a given domain. Offers highly parallel compute units, large on-chip memory bandwidth and

arbitrary execution models (streaming, latency-sensitive with strong synchronization support, throughput oriented with minimal communication support, etc). May run at one tenth the frequency of CPUs, with user-defined choice in how area may be spent (compute, communication, memory).

To make a choice of where and how to map the functional model for an optimistic FT simulation, we observe functional partition for a multicore simulation can be implemented with an existing instruction-set simulator with some optimistic FT additions. There is a large history of prior work in designing efficient instruction-set simulators (highly tuned virtual-machine interpreters: (Ertl and Gregg, 2003); just-in-time compiled simulators: (Bellard, 2005)) that we can take advantage of if we map the functional model onto a CPU. This also allows us to take advantage of existing high-performance untimed instruction set simulators that may be created to support software-only development. Modifying such artifacts for use with an optimistic FT scheme then becomes an engineering challenge while the foundational processor specific features are obtained for 'free'.

For the timing model, we can leverage the fact that for multicore simulation, we must mimic a large number of low-level hardware operations and use frequent synchronization on every cycle between components. If we map the timing model to an FPGA, we can simulate hardware with hardware. An FPGA allows us to reduce the impedance mismatch between the low-level hardware centric operations we want to simulate and the general purpose operations exposed via contemporary CPU instruction sets. A FPGA also offers significant parallelism, allowing the various timing tasks to run in parallel with low synchronization overheads.

4.1.1 Design Principles

Given our basic choice to map the functional model to CPUs and the timing partition to an FPGA, we can enumerate some design principles to refine our design choices

further:

Trace Design: Try to balance costs for trace manipulation between the two partitions. The compute cost to encode traces, bandwidth encoding efficiency and FPGA storage required for on-chip buffering form a key design point, and they should be co-designed between hardware and software.

Communication/Synchronization: Minimize communication and synchronization requirements between the partitions. Even though optimistic FT allows for relatively decoupled execution, a common CPU+FPGA platform may be connected via a PCIe link. Such a link may offer reasonable bandwidth but it may incur moderate latencies and limit choices for coherent communication. While tighter integration platforms are becoming available (integrated on the same die or via coherent cachelines), targeting a coarsely integrated CPU + FPGA pairing allows for more flexibility in host platform choices.

Parallelization: Enable parallelization in both functional (multiple threads/multiple cores) and timing models (large FPGA with multiple compute units). As the design point being targeted in the simulator is to simulate large multicore and manycore systems, both functional and timing systems must be capable of scaling to handle the increased computation required to simulate larger systems.

We describe the prototype simulator that was built to refine the ideas introduced by our generalized optimistic FT simulation approach. The simulator, FASTMP, is a conceptual successor to the original FAST simulator (Chiou et al., 2007) but it is a complete reimplementation from predecessor uniprocessor-centric simulator. FASTMP differs both in its focus on simulating multicore, manycore systems, and as its use of the generalized optimistic FT simulation approach which was incrementally refined as the concrete prototype was developed. A more detailed comparison of FASTMP vs FASTMP is presented later in the chapter.

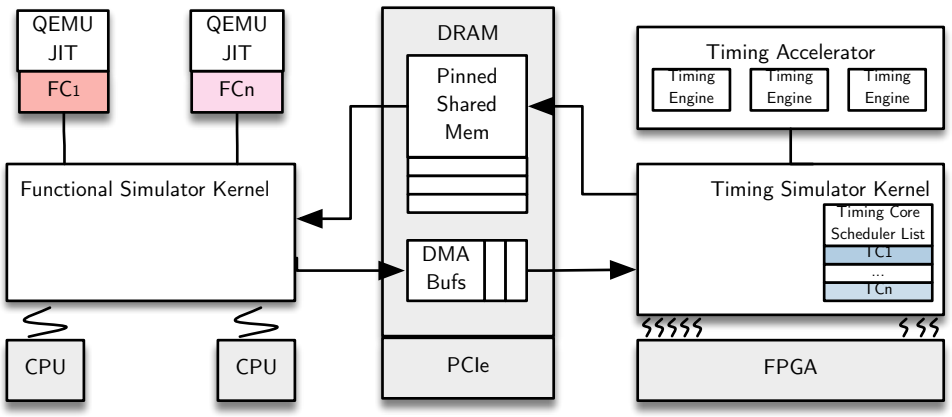

Figure 4.1: FASTMP System Architecture

4.2 System Architecture

Figure 4.1 depicts the system architecture of simulator prototype. The simulator uses a hybrid CPU+FPGA platform that connects the FPGA via an adding PCIe card. The functional model is mapped to software running on the CPUs, orchestrated by a functional simulator kernel. Similarly, the timing model is mapped to custom hardware designed for the FPGA, orchestrated by its own timing simulator kernel.

This pair of kernels implement the various required operations specified by the optimistic-FT simulation approach, isolating the details of supporting HW/SW communication and detection/correction of optimistic simulation from the modeling of the target itself. Model specific functional and timing partitions are then built on top of the kernels to implement the required target. As the system is designed to support multicore and manycore simulations, multiple logical processes may share a single SW thread or HW accelerator, allowing simulation of targets that exceed the number of cores/accelerators supported in the host.

HW/SW communication is implemented with DMA and pinned shared memory to enable efficient communication between the partitions. Traces are sent via DMA transfers, while feedback containing FT violation correction commands and

trace completion notifications is written to pinned shared memory.

In the common case where no violation is encountered, the simulator flow operates as:

- (i) The functional simulation kernel running on the host CPU core selects a functional core to run. The FCore runs by invoking portions of a heavily modified QEMU JIT-based instruction set simulator to provide the base x86-32 instruction set functionality.

- (ii) As the QEMU JIT executes, it invokes various simulator operations exposed by the simulator kernel (generating traces, invoking prophet read/writes) that land in an SW trace buffer.

- (iii) The functional simulation kernel sends a portion of the pending per-core traces via a PCIe DMA buffer. The hardware simulation kernel receives DMA buffers and splits traces for different cores into on-chip trace buffers. The hardware simulation kernel selects a timing core that is ready to run (has available traces, is not blocked by a pending violation command) and dispatches the TCore to run on the timing accelerator.

- (iv) The timing accelerator is implemented as a collection of specialized hardware engines, optimized for executing various parts of the timing simulation. Engines may be specialized to simulate processor-pipelines, caches, on-chip-networks or dram. As engines execute a TCore, they may consume various parts of the trace by accessing the on-chip trace buffers. When a TCore has been completely simulated by the engines, it signals completion back to the timing kernel which advances the simulated cycle and can reschedule the TCore.

- (v) As traces are consumed and cycles are completed in the timing accelerator, the timing simulation kernel monitors which traces have been completely consumed. When a trace can no longer be accessed by the timing accelerator, it is 'committed' by updating the pinned shared memory. The functional simulation kernel may use

these trace commits to reclaim storage used to support rollback.

The case for handling optimistic FT violations is nearly identical to the common case flow with some additional steps to handle the detection and correction operations:

- (i) When a trace containing a prophet read is accessed, the trace buffer will compare the prophet recorded value with the value provided by a matching oracle. We implement two oracles in the system, one for the processor program counter (e.g. handling branch mis-predicts) and another for shared memory.

- (ii) If the prophet recorded a value that differs from the oracle, a command is generated to notify the functional simulation to rollback and correct the trace. The timing core is paused waiting for its original trace lookup to complete while the correction occurs in the background. Unrelated timing cores may continue to simulate while the command processing occurs in the background, provided they have available traces and some amount of simulation lookahead.

- (iii) The functional simulation kernel periodically polls for commands held in shared host memory. If a new command is posted, the kernel will restore the associated functional core to the required offending trace, correct the prophet, and use replay and re-execute to repair any traces inflight and send the repaired traces to the timing simulation.

- (iv) The timing simulation receives the repaired traces, updating the on-chip trace buffer and it allows the timing core to resume simulation transparently.

4.3 Partition Communication Design

4.3.1 Overview

One of the key components of FASTMP is how to generate, transfer and consume traces across the hybrid platform. A variety of custom structures (implemented in

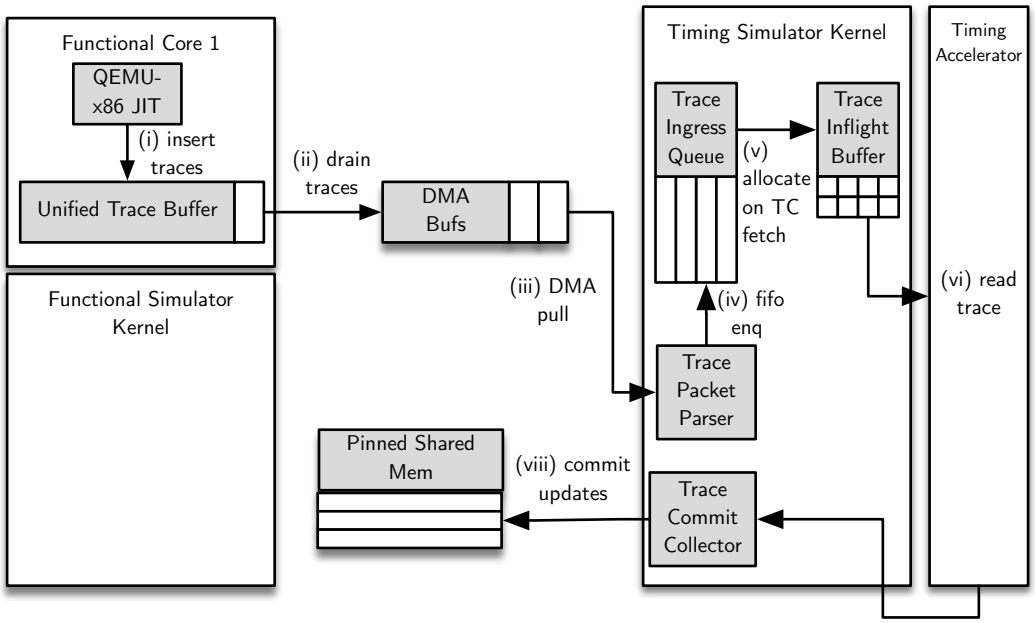

Figure 4.2: Trace Communication High-Level Architecture

both hardware and software) were designed to cope with the unique demands of high-efficiency and speculative nature of the traces passed between software and hardware.

Figure 4.2 illustrates a high-level view of the trace communication architecture (further details on selected components are provided in subsequent sections. The basic flow of traces from generation to consumption may be described as:

- (i) Traces are generated during QEMU JIT execution and placed into per-core trace buffers.

- (ii) The functional simulator drains traces from each functional core and packs them into DMA buffers for transfer into the FPGA via PCIe.

- (iii) The timing simulator pulls DMA buffers via the PCIe link, and it unpacks the packed traces to determine the core for which a given trace is destined.

- (iv) The stream of unpacked traces is pushed into an ingress holding queue,

waiting for the a given timing core to fetch the instruction trace.

- (v) When the timing simulator activates a given timing core and it fetches a new instruction, the timing simulator moves the trace from the ingress queue into the on-chip trace buffer.

- (vi) As the timing cores execute in the various timing engines, different parts of the trace may be accessed from the on-chip trace buffer in parallel.

- (vii) As instructions complete in each timing core, the associated trace entries attached to the instruction are marked as committed. These commits are collected and packed by a collector. The timing simulator periodically updates the functional simulator with notifications of completed traces, allowing it to reclaim storage held for optimistic rollback operations.

4.3.2 Trace Messages

The traces passed between hardware and software must contain a variety of information (e.g. instruction traces, memory addresses, prophet reads, prophet writes). To balance encoding compute complexity and decoding hardware complexity, we encode a trace as a series of 64b words, reserving three high-order bits to select between different encoding formats. Each of the different information types (instructions, memory ops, prophet memory ops, prophet pc ops) are encoded with data-formats specialized for each. The trace generated from a single executed instruction may then contain one or more 64b trace words.

As traces are drained from multiple functional cores and multiplexed over the same physical PCIe DMA link, we pack a bundle of traces prepended with the functional coreid and the number of traces in the bundle. This allows the receiving HW trace packet parser to identify which per-timing-core queue should be filled from the incoming trace stream easily. With a simple 64b word encoding, the parser can steer trace words of different types into specialized storage buffers. For example,

given the difference in how instruction trace words are processed compared to prophet reads and writes, the trace parser can simply decode the upper three bits of the trace and steer the word into the appropriate storage queue.

4.3.3 Platform Trace Channel

To provide the base communication infrastructure on the FPGA, we use the OpenCPI (OpenCPI, 2012) communication infrastructure that allows DMA-based communication between the CPU and FPGA. This communication link is designed to operate in a streaming fashion, where each DMA buffer must be sent in a sequential fashion (i.e. must send Buffer 3 before sending Buffer 4). To isolate from the particular details of our platform, we create a streaming FIFO abstraction from this contiguous set of DMA-able pages. As this virtual FIFO is populated, it tracks when DMA buffers are crossed, automatically triggering a buffer send. To preserve the illusion of a single unbounded FIFO, we *mmap* the DMA-able pages into a virtually contiguous region of twice the size. This oversized virtual mapping allows us to avoid checks for wraparound on each insertion.

4.3.4 FM Trace Buffers

As each functional core executes, it composes and inserts traces into a per-core trace buffer filled with trace words. This trace buffer may then be drained by the functional simulator as space becomes available in both the platform DMA channel and the on-chip trace buffers in the timing simulation. While this buffer requires a 1-copy send to deliver traces to the FPGA, keeping the per-core buffer separate from the platform send buffer allows the decoupling of functional execution, inter partition communication and timing simulation (consuming and freeing on-chip trace buffers).

Using a per-core trace buffer tracked separately from a platform send buffer also simplifies the handling of prophet corrections. As is discussed further in Section

4.4.4, a functional core may use its per-core trace buffer to hold corrected prophet reads, avoiding the need for additional replay logging.

4.3.5 TM Distributed Trace Architecture

Splitting computation across CPU and FPGA requires additional buffering on the FPGA side to allow for high-bandwidth, low-latency, and highly-parallel access to traces. We cannot simply consume the raw DMA stream, as we may need random access to traces, and we may require traces to feed multiple timing engines in parallel. The trace buffering must also cope with the speculative nature of traces. If a trace is repaired due to prophet mis-speculation, any on-chip buffers containing the invalid data must be tracked and repaired as well to prevent incorrect simulation. This set of constraints led to the design of a custom streaming memory architecture. The full details of this memory architecture are provided in Section 4.5.2, while a brief high-level view is provided here to capture the end-to-end simulation flow of traces in the system.

The on-chip trace buffer is split into two parts, one to hold traces before they enter the TCores, and another to hold them while they are being simulated by the timing engines. This split architecture specializes for the differences in access patterns between the two use cases. The *trace ingress queue* maintains storage in a FIFO pattern using dense memory arrays statically allocated across the TCores hosted in the simulator. This provides a fixed latency access to traces when a TCore is activated and scheduled on the timing engines.

When a trace is fetched by a TCore, it is placed into a second *trace in-flight buffer*. This buffer uses a set of linked random-access memories that allow parallel high-bandwidth access to different parts of the instruction trace. This buffer presents a logically centralized view of traces as they are simulating, but it is physically implemented as distributed memory banks to allow high-bandwidth feeding of the

timing engines.

4.3.6 Trace Flow Control

As on-chip resources for trace buffering are limited, we backpressure the sending of traces in software to prevent overflowing the trace ingress queue. As the functional cores maintain their own internal trace buffers, they may execute and generate traces in these local buffers even if the on-chip trace buffer is completely full. When a trace is drained into the DMA buffer, the timing simulator must be capable of placing these traces in on-chip storage to prevent head-of-line blocking.

Backpressure from the ingress queue to the sending SW is implemented by a tracking a read pointer that is updated via a shared pinned memory. As traces are inserted from the trace parser into the ingress queue, a per-TCore write pointer register is incremented. When the traces are dequeued and inserted into the trace in-flight buffer, a matching read pointer is incremented. Periodically, these read pointers are collected and written back to the pinned shared memory region. When an FCore is drained of its pending traces, it eagerly updates its own copy of the ingress queue write pointer, while its view of the read pointer is updated lazily based on updates to the shared memory region. This conservative tracking scheme ensures safety and only requires communicating a single small counter (eight bits in our design) between HW and SW, minimizing synchronization overheads.

4.4 Functional Model Design

4.4.1 Overview

The FASTMP functional simulator is comprised of two parts, a common simulator kernel and a per-core library of simulator actions. The common simulator kernel is responsible for scheduling FCores onto host threads, managing sending of traces, and

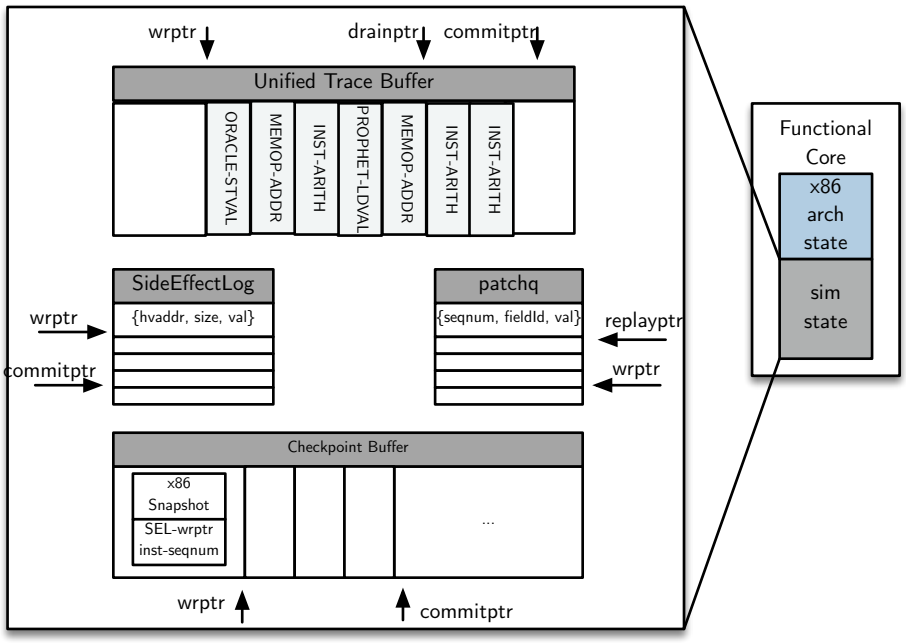

Figure 4.3: Functional Simulator Architecture

supporting the handling of rollback corrections. The per-core library provides a set of customized data-structures and associated operators to transform a conventional untimed functional-only instruction set simulator into an optimistic functional core. Figure 4.3 depicts the architecture of the functional simulator.

We use QEMU (Bellard, 2005), a high-performance full-system instruction set simulator to provide base functionality for modeling the x86-32 instruction set. This functional-only simulator was extensively modified to support host parallelization, trace generation, prophet accessors and controlled execution for rollback and replay purposes.

4.4.2 Generating Traces

To implement support for generating our speculative traces, we design a set of simulator micro-ops that can be invoked by the untimed QEMU execution flow as

it simulates an instruction. As each micro-op is executed, it may insert records in the various simulator maintained per-core data structures, such as pushing a trace into the trace buffer. When this JIT translated block of instructions is executed, the various simulator micro-ops are triggered as well.

For instruction tracing, these micro-ops are added during JIT translation to compose the various properties of the instruction (op type, op width, register operands, etc.). For memory tracing, we use a combination of micro-ops to allow the generation of traces needed for timing simulation, as well as tracking prophet reads and writes. When a memory operand is accessed during the simulation of a single x86 instruction, a memory operand trace is generated with the various dynamic properties (mem op type, memory address, mem operand size).

When attempting to access the memory (after the address is computed), a memory prophet micro-op is used instead of directly accessing the simulated untimed memory. This memory prophet micro-op treats all memory as potentially FT-entangled, as all the simulated FCores in the functional simulation may access the memory. For loads, this prophet micro-op generates a prophet-read trace entry before returning the underlying simulated memory. Similarly, for stores, a prophet update trace entry is generated before allowing the untimed memory to be modified. The memory prophet micro-ops provide both tracing capabilities and controlling access to the untimed shared memory (a capability used to support replay execution described in Section 4.4.4.

4.4.3 Supporting Rollback

As each FCore is allowed to execute optimistically, the traces generated may be incorrect due to a mis-speculation by a prophet. To support the trace repair process, we must be able to restore the FCore to arbitrary instruction points. Implementing rollback uses similar design space alternatives to traditional optimistic PDES

simulation: snapshots, logging or hybrid combination of the two. We use a hybrid scheme, using fine-grain logging for sparse and large state variables, and snapshots for frequently overwritten state variables.

To support rollback to arbitrary instructions, we first restore to a consistent checkpoint (restore the snapshot, rollback the fine-grain logs). We then replay the execution deterministically and stop execution once we have reached the desired instruction. As one of the preconditions for correctness in an optimistic FT simulation relies on selective replay and re-execution for trace correction, full replay-only execution is a minor additional requirement.

One advantage of the optimistic FT simulation approach is its application to a logical process containing a simulated target core. As the optimistic execution is hidden outside this boundary, the rollback and reclaiming of state only occurs between a pair of interacting FCore/TCores. In contrast, to support rollback in an optimistic PDES simulation like TimeWarp, an expensive operation to compute the minimum Global Virtual Time was required periodically to reclaim checkpointed memory. The FCore need only examine which traces have been completed by its single matching TCore to determine when rollback logs may be safely discarded.

To implement the fine-grain logging for rollback a simple circular buffer called a "side effect log" is useful. The buffer maintains records as a 3-tuple of (host virtual addr pointers, pointer size, old memory value). This simple structure allows us to treat arbitrary memory locations in the simulation as logged states, simply by instrumenting the accessors. For example we track state changes to target physical memory, infrequent x86 control registers, x86 interrupt and segment table updates amongst others. Insertion into the log is computationally inexpensive as the accessor already has the host pointer and it simply costs an additional dereference to record the value prior to modification.

When a checkpoint is restored, it records the position of the write pointer

into the side effect log. Restoring the log simply requires rewinding the current write pointer until it reaches the checkpointed write pointer, and then applying the records in reverse order while restoring the old values. As the log is maintained as circular buffer, when checkpoints are committed, the pointer to the last reachable record in the buffer is also updated. Tracking the commit pointer into the buffer is useful for detecting implementation bugs due to overflow and attempts to rollback to records that have already been committed.

A checkpoint buffer is used to hold a series of checkpoint entries. Each entry consists of an architectural snapshot along with the write pointer into side-effect-log and the instruction trace number of the FCore when the checkpoint entry was taken. To restore to a given instruction trace, the checkpoint buffer is searched for an entry with an instruction trace less than the requested trace number. As prophet corrections affect the trace number being requested for rollback, the checkpoint entry must record the state of the FCore prior to the execution of the trace. Once the entry is identified, any successive checkpoints are discarded (as they reflect the now invalid FCore state without the prophet correction) and the side effect log is restored to the recorded position in the checkpoint entry. This restores the FCore to a consistent checkpoint prior to the offending trace. Deterministic replay can then be used to reach the point immediately prior to the corrected trace number if required.

When the simulator starts, an initial checkpoint is created that contains a an architectural snapshot that records the reset-state of the processor. Once this initial checkpoint entry is created, there is a guarantee of retaining at least one valid checkpoint entry at any moment during the course of simulation. Checkpoint entries may only be reclaimed for space if they have a successor checkpoint that may take their place. For example, consider a pair of checkpoints: (chkpt: i, trace_number: N) and (chkpt:$i+1$, trace_number: M). Checkpoint i may be reclaimed once the TCore commits trace M. This ensures that no rollback will ever be generated for

traces less than M, ensuring only entry $i+1$ needs to be kept. As trace numbers are periodically committed by the timing simulation, this comparison is checked to reclaim entries.

4.4.4 Supporting Replay

Replay execution is used during both rollback and trace repair phases of functional simulation. During the rollback phase, replay execution is used to rollback to a specific instruction trace identified by a timing partition command. In this phase, replay execution is completely repeatable and deterministic, which allows the simulator to return to any trace from a starting checkpoint. During the trace repair phase, replay execution is used both to regenerate prior computations deterministically and to propagate and repair traces due to incorrect prophet reads. This phase must use selective replay and re-execution rather than simple deterministic replay as this will regenerate the same incorrect trace that triggered the FT violation in the first place.

Implementing this form of selective replay and re-execution requires a slightly more involved design than straight deterministic replay. Under a purely deterministic replay scheme, we can record all non-deterministic inputs and replay them to ensure completely repeatable re-execution. However, in the case of a prophet correction, we must explicitly change the value of a prophet and re-execute in the presence of this modification. Moreover, while the prophet correction must be allowed to affect dependent traces, prior prophet reads already validated by the timing partition must be retained. To represent this behavior we cannot simply execute instructions to completion, but we must instrument execution both at instruction boundaries and during the execution of a single instruction. We implement this selective-replay behavior in three parts: (i) prophet micro-ops (described briefly in Section 4.4.2), (ii) a unified trace buffer to replay prophet reads, and (iii) an unexpected patch log for inserting infrequent prophet corrections.

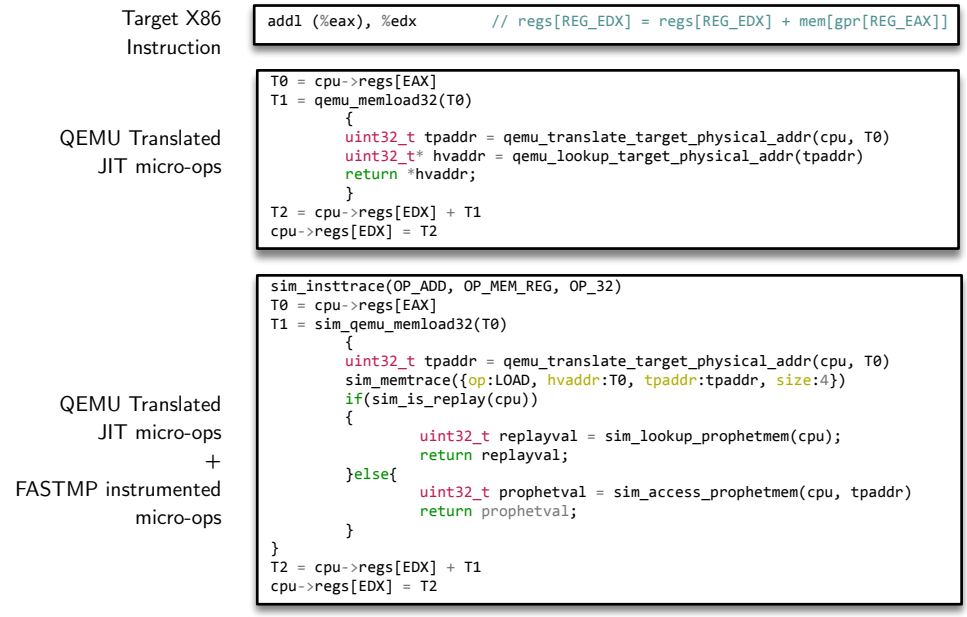

Figure 4.4: Supporting Replay with Prophet Micro-ops

As described previously, we insert simulator micro-ops to integrate tracing, logging and replay functionality directly into the execution of an instruction. Prophet-mem micro-ops are embedded into the simulation of instructions that may access shared untimed memory as memory arguments. As we support the x86 instruction set which allows register-memory based computation, this requires instrumenting a variety of instructions beyond a typical set of load and store instructions. Figure 4.4 illustrates an example translation of an add instruction with an implicit memory load. Simulator micro-ops are inserted for instruction and memory operand tracing (required for timing simulation). Prophet micro-ops are inserted to replace direct access to the untimed shared memory. During replay, these prophet values are replayed without accessing the underlying shared memory.

When a prophet micro-op is executed during replay execution, it must access the value that was used during the first generation of the trace or the corrected

value provided by an oracle if the prophet has already been corrected. Instead of creating a separate log to hold these prophet values, we merge the tracing and replay functionality into a single unified trace buffer. The per-core trace buffer is simply used to hold pending traces before they are sent to the timing partition for simulation. The buffer may contain a mix of instruction and memory address traces necessary for timing simulation, prophet reads for verification by a timing oracle and prophet writes to populate a timing oracle. However, as the prophet reads must be provided in the trace for verification by a timing oracle, the same buffer can serve as a replay buffer for prior prophet reads.

Merging the replay and tracing functionality into a single buffer simplifies and reduces the overhead of replay execution, but it also requires some additional restrictions during trace generation. As the trace buffer is organized as a circular buffer, we must ensure that trace entries are reserved deterministically, even in the presence of corrected prophet values. This requires that each instruction will generate a fixed number of trace records, even if this requires reserving placeholder records to support the worst case execution of a instruction.

For example, a conditional load instruction that evaluates to false during the first functional execution must insert a placeholder prophet-read trace entry. As it is possible the load may evaluate to true under some future correction scenario, we must reserve the trace record unconditionally. Using this pre-reservation scheme allows the trace buffer to be used for replay without incurring the cost of shifting the entire trace buffer simply to insert a conditional prophet load value.

The unified trace buffer also makes it trivial to handle prophet corrections. After a rollback has returned the FCore to the desired trace number, the trace buffer can be directly patched with the corrected oracle value. This in-place patch of the trace buffer ensures that any subsequent correction to other prophet reads will observe this initial correction transparently. This greatly simplifies the handling

of prophet corrections as they effectively only require a single update to the trace buffer. Replayed re-execution may then repair the trace transparently, regenerating dependent traces as required.

The unified trace buffer replay approach works well when we can reserve space for prophet reads in the common case. However certain state variable accesses are highly predictable, and reserving a replay entry for every access can be expensive (e.g. program counter reads on non control-flow instructions, implicit read of interrupt pin on every instruction). These accesses are effectively non-entangled, but they must still support rare scenarios where FT entanglement may occur (e.g. exceptions, interrupts, mispredictions to the middle of an instruction).

To support the replay of this form of prophet, rather than passing prophet reads along with every instruction trace, we record only when these state variables are corrected and hold them in a secondary patch queue. For example, when generating an instruction trace we explicitly record updates to the program counter between basic blocks, while each instruction provides its instruction size. This allows the timing simulation to derive the program counter prophet read for every instruction without explicitly needing to send the value on every instruction trace. However, as the timing simulation validates this prophet PC against the timing oracle PC for every instruction, it is possible to require prophet corrections at arbitrary points in a basic block, for which no explicit replay storage exists in the trace buffer.

To solve this problem, we keep unexpected prophet corrections in a secondary patch queue. Each entry in the patch queue holds a 3-tuple record containing a trace number, a prophet id and a prophet value. This structure is generic enough to address any state variable in the functional core state (e.g. program counter, interrupt pins, and other unexpected inputs into the processor core). Keeping unexpected prophet corrections separate from the common-case allows supporting a wide range of TCores that may require infrequent unexpected updates to nearly non-entangled state. For

example, the TCore may require accurate simulation of register reliability events where a private internal register state variable may potentially change due to timing (e.g. wearout, cosmic rays, etc). Instead of requiring all private general purpose registers to be recorded in the common case, we can used the patch queue to hold prophet corrections instead.

Through the use of prophet micro-ops, the unified trace buffer and the patch queue, supporting prophet corrections is straightforward:

- (1) restore the FCore to a consistent checkpoint by finding the closest checkpoint entry. The architectural snapshot is restored, the side-effect log is restored (restoring old values modified by stores and infrequent architectural states) and the trace buffer write-pointer is restored to its position recorded in the checkpoint entry. We also set the patch queue replay pointer to the value recorded as its write pointer in the checkpoint entry.

- (2) The patch queue is examined to see if the head entry contains a pending patch for the trace numbers that will be repaired through re-execution. If a patch entry exists and the next trace number matches, the entry is applied and the replay pointer is bumped.

- (3) We continue in pure replay mode until the trace number requested for correction is hit. As this mode is purely deterministic replay, the traces generated are idempotent. However to avoid additionally specializing in the JIT-ed code and to detect simulation errors, we regenerate the traces inplace and we optionally verify their idempotence when debugging simulator errors.

- (4) Once the requested trace number in the correction command is reached, the trace buffer is patched with the corrected prophet value.

- (5) The simulator resumes execution in replay re-execution mode, repairing traces and replaying prophets. As instructions are executing they regenerate/re-insert traces into trace buffer. As the prophets only replay from the trace buffer itself,

both the newly added prophet correction and all prior corrections are transparently reflected in the updated trace.

- (6) We continue executing in replay mode until the trace number for the last trace to be consumed by the TCore is reached. We can then return to normal execution only mode

By interleaving patch queue, replay lookups and selective re-execution, we can efficiently construct corrected traces with minimal disruption to the common case execution.

4.4.5 Supporting Target Speculation

Using the common prophet record/correct/replay to cover any FT-entangled state can cover cases of both simulator speculation (where the functional simulation assumes a value for the FT-entangled state incorrectly) and target speculation (where the timing partition incorrectly speculates and must discard instructions and restart the instruction execution). This transparent handling is supported by allowing the prophet corrections to name both the trace number to rollback to and the desired oracle value for the correction.

For example, when a TCore with a branch predictor mispredicts a branch, the first prophet correction occurs with (trace number i, $badPC$). This regenerates a trace of the target mis-speculated instructions for the TCore to use in the rest of the simulation. When the TCore detects its own mis-speculation, it may flush or drain its pipeline and restart its instruction fetch inducing a prophet correction at (trace number i, $goodPC$). As trace numbers are opaquely tracked in the timing simulator, the TCore is unaware of these corrections. It simply provides the timing oracle PC and the timing simulator is responsible for ensuring that the provided trace matches.

4.4.6 Host and Target Parallelism

As the hybrid CPU + FPGA execution scheme allows significantly faster simulation of the timing partition, we needed to ensure the functional partition did not become a new bottleneck. We added parallelized execution to the baseline QEMU instruction-set simulator to allow individual FCores to execute in parallel with minimal communication or coordination.

As the design of tracing, rollback, and replay mechanisms were designed to operate locally in the context of a single FCore/TCore pair, all simulator micro-op actions could be safely executed in parallel. From an optimistic FT perspective, if one host thread simulates N FCores sequentially or if N FCores simulate in parallel, little changes in the optimistic simulation protocol. The complications of adding parallel execution in the functional simulation stemmed from the starting point of the legacy QEMU instruction set simulator. As the baseline simulator was purely serial, identifying where locks, barriers and atomics were required incurred most of the implementation complexity.

One point where the optimistic FT simulation approach intersects with the QEMU parallelization effort is around centralized translation cache used by the QEMU JIT. As QEMU relied on a global translation cache, locking was added around the various manipulation operations (inserting new basic blocks, invalidating code blocks, etc). This required adding locking around both the x86 translation process, and the page-level protection tables QEMU uses to maintain a (read-write XOR read-execute) invariant to track code loading/unloading.

As the translation process of JIT-ing new code required a serialized lock, using the JIT to specialize code for the normal versus replay modes of operation was avoided. Further, as the JIT operates at the level of basic-blocks, if a checkpoint restore required execution to stop at the middle of a basic block, this would require a re-translation and a global lock. To avoid this bottleneck, we introduce a small

per-host thread private code cache that can be used to translate and break basic blocks during precise instruction-level replay. As this code buffer was purely transient and only used for a single execution, it did not require integration with the shared translation cache. This allows the processing of prophet corrections without requiring the use of any QEMU-based locks, ensuring scalable processing of corrections.

4.4.7 Supporting Uncore Functionality

While the optimistic FT simulation approach is not strictly limited to processor simulation, it is the central focus of the FASTMP simulator. However as a multicore system still requires some uncore IO functionality to operate, different strategies were developed to support these IO behaviors. We treat all IO devices in the simulated platform (USB, PCIe, etc) as a single functional partition (i.e. an IO 'core'). This core may be interacted with using mmio or ioport instructions from other processor cores in the system.

From the perspective of an FCore, mmio and ioport access is simply treated as a different type of untimed shared memory access, thus requiring the use of a prophet. We insert prophet micro-ops on ioport x86 instructions and memory pages marked as memory-mapped IO.

4.5 Timing Model Design

4.5.1 Overview

Similar to the organization of the functional simulator, the timing simulator is composed of a common simulator kernel and a per-core component. The simulator kernel handles many similar actions to its software counterpart (scheduling, communication, handling oracle and prophet corrections). The simulator kernel also manages the various on-chip trace buffers to allow the traces sent over DMA transfers to be accessible

more efficiently. For the per-core simulation component, no existing FPGA timing model simulator fitted our requirements, so a domain-specific timing accelerator to simulate x86 multicore timing models.

When designing the timing simulator, the characteristics of how to achieve speedup on an FPGA were constantly kept in mind to drive design choices. An FPGA may only offer 1/20th of the frequency of a modern CPU core, so different strategies must be used to ensure simulation speedup. As one of the basic restrictions for high-speed simulation is the large amount of computation required to simulate a timing model in SW, achieving good speedups on this specialized workload is key to enabling high-speed simulation. We identify a few common methods that were used to achieve speedup and efficiency gains from the various hardware components implemented in the timing simulation:

- **compute operator customization**: Use custom accelerator operators rather than trying to fit behaviors ideally suited for a custom circuit into an existing general purpose ISA. For example, we use custom pipelines for each TCore to allow performing various low-level hardware modeling tasks as dedicated circuits (e.g. cache-tag searches, wake and select scheduling actions, etc)

- **memory customization**: Leverage the high bandwidth available on-chip via dedicated memories. As the FPGA doesn't have to rely on conventional L1/L2/L3 caching schemes, memories can be allocated and use with different policies. For example, heavy use of both shallow 64-entry memories and deeper 1k-entry memories are used extensively through out the design rather than trying to rely on a generic caching subsystem for the entire simulator.

- **communication operator customization**: The adhoc execution model offered by the FPGA allows specialization of the communication primitives to match characteristics of how parallel tasks interact. For example, to allow low-

cost work-conserving scheduling of simulation tasks, a pipelined synchronization barrier network is implemented.

- **compute parallelism**: We take advantage of FPGA capacity that can support a large number of parallel operations (as each compute unit can leverage the three customizations listed above to minimize its resource requirements). This makes it possible to build simple compute units that offer which by themselves might not accelerate the simulator compared to a software alternative. However, in aggregate, as all compute units may run in parallel, the total number of operations completed per second can offer significant speedups compared to commodity CPUs.

The design of the trace memory architecture, the timing accelerator and the timing oracles all involve many design choices used to optimize for some characteristic of split hardware/software execution or to cope with iterations with the optimistic functional-first simulation approach. We describe the details of these components some of their unique architectural implementation features in the following sections.

4.5.2 Trace Memory Architecture

The trace memory architecture forms the basic foundation of supporting CPU + FPGA execution. As described in Section 4.3.6, on-chip trace memory is split into two parts: one for buffering traces on the chip and another for use while instructions are simulated in timing accelerators. The design of this memory system had to cope with multiple competing requirements including:

- decoupling execution of SW and HW
- reducing hardware design complexity by providing deterministic access to traces
- efficiently sharing storage across large numbers of TCores

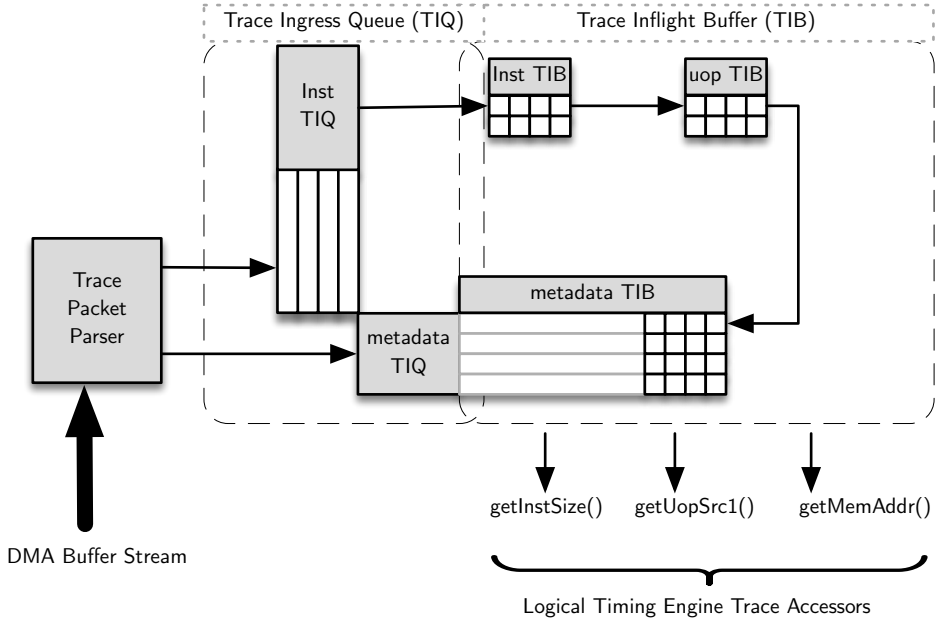

Figure 4.5: Trace Memory Detailed Architecture

- enabling high levels of compute parallelism in the timing accelerator

These requirements would be needed even for a standard functional-first simulator mapped to a CPU+FPGA system. The added complication is that optimistic FT simulation approach requires the ability to update traces that may depend on an incorrect prophet read. To cope with these requirements, different strategies were employed to design the trace memory: (i) splitting ingress queue buffering between static and dynamic portions of the trace, (ii) splitting trace inflight buffer into multiple levels, and (iii) presenting a simple high-level abstraction to the timing accelerator.

Ingress Queue

The first design optimization used in the trace memory was to split how traces were stored in the ingress buffer. The purpose of the ingress buffer was to buffer traces in

FIFO order, decoupling the CPU send from the consumption in the FPGA. As this memory is not directly accessed by the timing accelerator, we can avoid expensive highly multi-ported/replicated memory and use a simple FIFO.

We further specialized the storage of the ingress queue based on the contents of the trace. By splitting the queue into two parts, one to hold instruction traces and another to hold instruction metadata (memory traces, prophet read/writes), we could simplify how the trace memory was organized. As a single instruction trace may contain 0 or more metadata traces, each of which could only be accessed *after* the instruction was fetched, keeping the metadata in separate buffer allowed more efficient buffering of traces on the chip.

In-Flight Trace Buffer

The second design optimization for the trace memory was to use a highly multi-ported, banked buffer to hold the in-flight traces. As these trace buffers support random access, they allow the timing accelerator to simulate different cores and to access traces in any desired order. This trace buffer was designed to mirror the way the trace was split and accessed in a TCore pipeline. Instead of a monolithic trace entry with multiple fields, three separate buffers were used: one for instructions, another for instruction micro-ops, and a third for micro-op metadata.

When an instruction trace is first fetched, it may be allocated in the first instruction trace buffer. As this instruction is decoded into one or more micro-ops, each of these are allocated into the second trace buffer and matched with any accompanying metadata from the third buffer. Each buffer supports multiple readers by sharding different parts of the trace into different physical buffers. This nested memory structure supports the high bandwidth requirements of the timing accelerator while allowing compact on-chip buffering by dynamically allocating space between the buffers.

Supporting Trace Repair

The trace ingress queue and the trace buffer were explicitly designed to support easy correction of traces during a prophet correction. As instruction traces are validated before they are inserted into the in-flight buffer, traces that have yet to enter the TCore may simply be dropped and continue being dropped. When the repaired trace is received with a command acknowledgment, these traces may resume being buffered in the ingress queue.

Supporting trace repair once the instruction trace has been fetched by the TCore is somewhat more involved. We rely on the fact that the prophet corrections may affect the instruction metadata, but it may not change the static instruction trace. As the instruction metadata is kept separately from the instruction and micro-op trace buffers, only a single trace buffer need be refilled with repaired traces. By centralizing the metadata buffer and referring to this metadata via pointers that are only referenced on first use, we may transparently patch the metadata buffer and have the changes reflected immediately. As the other buffers in the timing accelerator may only contain valid trace information *after* an oracle has confirmed the validity of the trace, we do not need to invalidate any further memory structures.

Logical Trace Memory Abstraction

The multiple levels of split buffering offered by the trace memory architecture would add significant design complexity to the timing accelerator if exposed directly. Instead we adopt software-engineering style principles to hide the various trace field accessors under a single logical API. By simply choosing the required accessor and passing an opaque trace pointer, a timing engine in the accelerator may view the trace buffer as a single monolithic buffer. All the details of selecting the required memory storage, computing the required memory address and traversing multiple linked pointers is hidden under this simple abstraction. For example a call to read the source register id

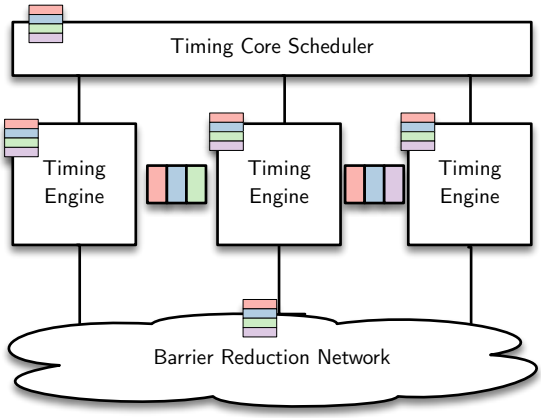

Figure 4.6: Timing Compute Accelerator Architecture

of a trace can be made with a simple invocation of $getSrcRegId1(ctx, instid, uopid)$. This physically accesses the uop trace buffer and accesses the read port that may return register id1.

By combining different access policies (FIFO, random access), different port counts (single, dual, banked multi reader) and overlaying storage where possible, the trace memory architecture can meet the requirements of CPU + FPGA execution along with the unique constraints required for optimistic FT support.

4.5.3 Timing Engines

To take advantage of benefits of FPGA execution for timing simulation, we must leverage fine-grain parallelism and expose efficient computation primitives. The trace memory architecture is designed to support multiple compute engines by exposing multiple banks and memory ports capable of reading multiple parts of the trace simultaneously. We can take advantage of this high-bandwidth trace memory by using parallel decomposition at spatial boundaries in the timing simulation to expose additional levels of parallelization. The TCores, timing caches, interconnect and DRAM timing simulation partitions may all execute in parallel in the timing

accelerator.

While executing each TCore, cache, etc in parallel offers some additional parallelism, many actions to simulate a single timing cycle may still require multiple sequential steps. To cope with these sequential dependencies that occur within a single simulated cycle, we adopt a multi-threading approach. Using hardware multi-threading is an approach used by many FPGA simulators (Protoflex (Chung et al., 2008), RampGold (Tan et al., 2010)) to ensure high utilization of FPGA execution resources in the face of sequential dependencies. As the trace memory architecture was designed to allow random access to different TCores and different parts of the trace simultaneously, allowing multiple TCores to execute on a multi-threaded timing engine offers no additional complications.

To support multiple timing engines (each supporting multiple TCores mapped to different contexts) with variable execution latency, we create a dedicated synchronization network to detect when all engines for a single context have completed their simulated cycle. As TCores are completed from a given timing engine, they issue a cycle completed notification to the barrier network with their corresponding core ids. When a TCore has collected all its engine completions, the synchronization network signals the completion of the TCore, allowing it to advance to the next cycle.

4.5.4 Timing Oracles

The requirement of optimistic FT simulation for the timing simulation is to ensure that when a trace includes prophet reads, the untimed prophet reads matches the current timed oracle value before consuming the trace. The trace memory presents a useful abstraction of a single centralized trace buffer with predictable fixed latency. This makes the design of timing engines simple as they consume different parts of a trace in a fashion not too dissimilar from simply accessing a C-struct.

However, as these different parts of the trace are accessed, some of them may

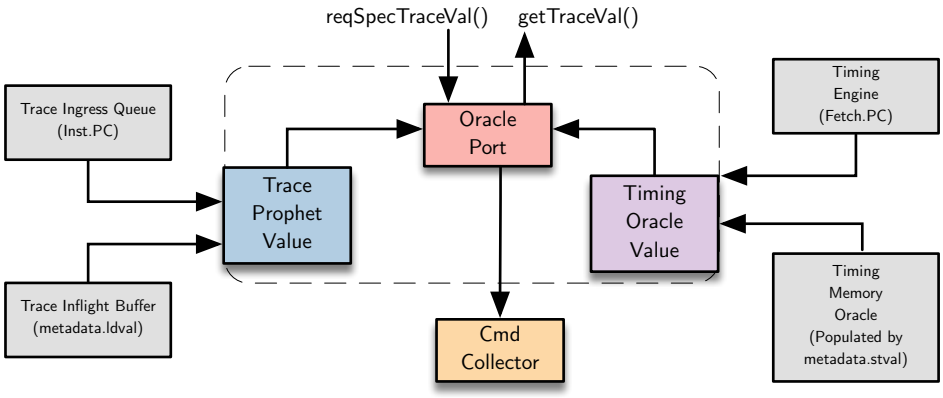

Figure 4.7: Timing Oracles Architecture

require comparison against an oracle. If an oracle detects a FT-violation, the timing simulation must invoke SW rollback and correction before allowing continued access to the trace. To avoid complicating the design of the timing engines, we need to hide the complications of this access and validation in similar manner to the non-optimistic parts of trace. We do this by introducing an extension to the logical trace buffer abstraction we call oracle ports.

If we use a use a two-phase request/response abstraction to access a trace field, we add a minor additional complexity to the timing engine, but we create a sufficient interface that can hide the speculative nature of the trace access. In the first phase of a trace field access, an engine can indicate a desire to read from a potentially incorrect trace field. In the second phase, an engine consumes the trace field with the assumption that the trace field return is correct. All details of optimistic detection and correction are hidden under this abstraction. This simple two-phase access addition to the logical trace buffer interface enables the timing engines to be isolated from the optimistic simulation details.

Figure 4.7 depicts the architecture of this two-phase oracle port. From the perspective of a timing engine, the port appears to be a simple request/response

interface. Inside the port, we may access the prophet read value from the trace and compare it against a reference oracle value. If the oracle comparison detections a violation, a command to rollback is generated and the request is placed in a queue in the port until the command is acknowledged and the trace is repaired.

This basic port architecture can be used with minor adaptations to validate different parts of trace by connecting to different sources. For example, if we connect the prophet value to the $ingressQueueBuffer\_pc$ and the oracle to $tmPipeline\_fetch\_pc$ we implement a program counter oracle port. Similarly, by connecting the prophet value to $traceInflightBuffer\_metadata\_memval$ and the oracle to $timingMemoryOracle\_memval$ we implement a load value oracle port.

The oracle port hides the optimistic characteristics from trace consumers, but we still need to provide reference values to compare with prophet values. The timing memory oracle must provide a value for each memory address in simulated system. We could implement this oracle by keeping all data values in simulated caches, DRAM. In that case, we could simply use the value returned by simulated L1 cache to provide the reference oracle value for the load value oracle port. In the prototype simulator, as we model a fully sequentially consistent memory architecture, there is always exactly one memory value for any given memory address in the system. We choose to keep these values in FPGA memory separate from the simulated caches, DRAM and to only access it once a hit is detected in cache or DRAM.

Updating the timing memory oracle is conveniently simplified by reusing the oracle port for both load and store instructions. Using the oracle port for both allows the same structure both to detect violations for loads, and to update the memory oracle if trace field is a store. The only distinction between two paths is that load validation requires a value to be returned by the memory oracle, and that it can create an FT violation command. A store on the other hand simply updates the memory oracle and signals the response immediately via same two phase oracle port

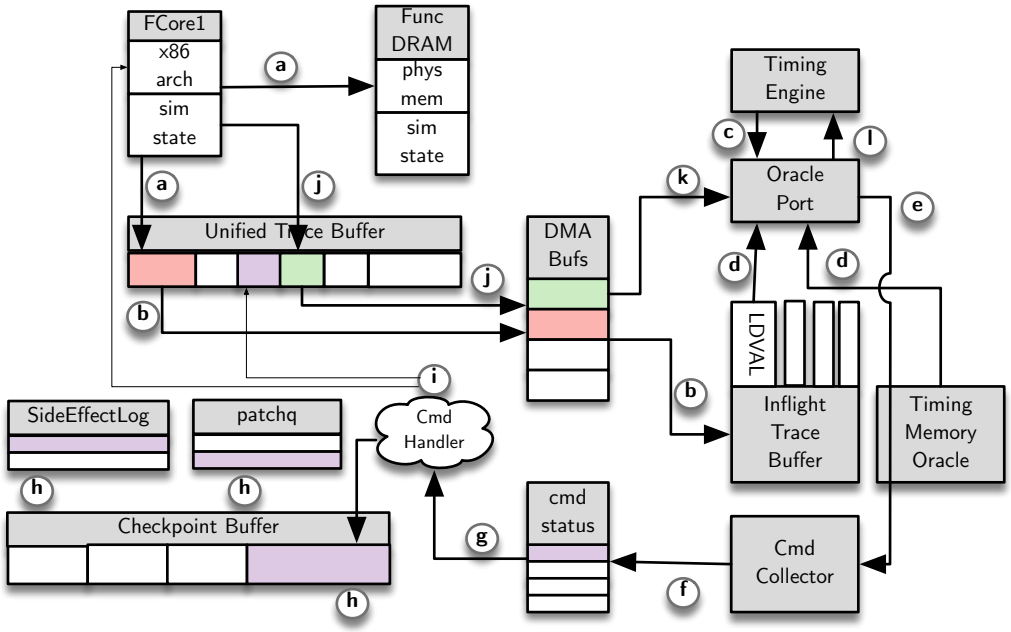

Figure 4.8: Simulation of an FT violation with Detection and Correction

abstraction.

4.6 Optimistic Simulation Flow

1. (a) - FCore fp_1 executes a load instruction that reads from shared memory held in the functional model (e.g. shared target DRAM) and inserts a trace into the unified trace buffer containing the prophet-read value.

2. (b) - The functional simulator drains this prophet-read trace into the DMA buffer, which is parsed, buffered in the ingress queue, and then finally lands in the inflight trace buffer.

3. (c) - The timing simulator simulates TCore tp_1, which when it attempts to access the trace memory address component, invokes the request via an oracle channel.

4. (d) - The oracle channel reads the prophet readval from the inflight trace buffer, along with the current timed value of the same address from the target memory oracle.

5. (e) - The oracle channel detects a mismatch between the prophet and the oracle, and it signals an FT violation.

6. (f) - A rollback command is composed based on the associated sequence number and the prophet-id attached to the trace record. The command is written back to the pinned command memory region, along with updating a bit in the cmdstatus bitvector to indicate the publication of a new command.

7. (g) - The functional simulator polls the command status bitvector, and it detects the new command.

8. (h) - The functional simulator invokes a rollback on FCore fp_1 (which searches for the closest checkpoint record, restores from the side effect log and sets up the patchq), and it then replays until the requested trace identified in the command.

9. (i) - The unified trace buffer is updated with the corrected prophet value.

10. (j) - The functional simulator replays the set of inflight traces identified in the command. It then drains the updated trace into the DMA buffer it and marks the trace as as command acknowledgment.

11. (k) - The timing simulator observes the command acknowledgment and it updates the inflight trace buffers with the new trace.

12. (l) - The timing simulator signals the completion of the FT violation correction and the oracle channel transparently returns the load-value back to TCore tp_1, which continues simulating the remainder of the cycle.

Host Machine Properties	
CPU	Intel Core i7 x980, 3.3Ghz
FPGA	Xilinx ML605 fpga card (Virtex6-240T, 37k slices, 416 BRAMs)
PCIE	PCIe Gen2x4
Target Machine Configuration	
Core	Atom-style, 7-stage in-order processors (1 to 256 mulitcore)
L1-Cache	256 tags, configurable line-size
L2U-Cache	64k tags, configurable line-size

Table 4.1: FASTMP Simulator Configuration

This correction process is optimized for both throughput and latency. From a throughput perspective, the two-phase nature of the oracle channel and the ability of the timing partitions to continue simulation allows some of the rollback penalty to be hidden. Further, as the command handling is treated as a latency-sensitive task in the functional simulator it may bypass normal fair execution interleaving and trace draining policies to resume the paused timing partition as soon as possible.

4.7 Results

4.7.1 Methodology

To evaluate the benefits and limits of the optimistic FT approach, we focus on evaluating FASTMP using a variety of targeted benchmarks designed to exercise the simulator and demonstrate the tradeoffs of speculation and parallelization in both SW and HW domains.

While the FASTMP simulator is written to be generic and could be adapted to work on a variety of FPGA boards, for evaluation purposes we restrict the configuration to a commodity workstation with a ML605 FPGA acceleration card.

Table 4.1 presents the various parameters used in the FASTMP evaluation studies. The timing model configured in the simulator supports simulation of a large many-core processor configuration with 1 to 256 cores with private L1 caches and a

Component	Mode	Action	Host Cycles	Host Memory (bytes)
Tracing	Normal	insert	3	8
Checkpoint	Normal	create	500	966
Checkpoint	Replay	restore	410	966
SideEffectLog	Normal	insert (per store)	15	32
SideEffectLog	Replay	restore (per store)	15	32
ReplayProphetMem	Replay	replay load value	5	8
PatchQ	Replay	replay GPR-regs patch	3	5

Table 4.2: Costs of Rollback Operations

Functional Partition Mode	Per-Thread Simulation Rate (MIPS)
Functional Only	127.9
Functional First	61.3
Optimistic Functional-First	45.4

Table 4.3: Overhead of Different Simulation Approaches

shared distributed L2.

4.7.2 Impact of Optimistic Execution on Functional Simulation

As optimistic FT simulation requires additional support from a pure Functional-First simulator, the costs of supporting optimistic execution are studied. Using microbenchmarks and code analysis the overhead in supporting optimistic execution in the functional partition can be measured. Table 4.2 presents the space and time overheads of supporting various rollback capable operations. The most significant overheads come from capturing checkpoints, which for an x86-compatible core requires a large snapshot state. The other logging/tracing operations have been made efficient through the use of JIT optimizations, precomputing trace properties where possible and trace buffer population.

Table 4.3 shows the effective overhead of supporting an optimistic FT approach by comparing the simulator against both a Functional-First (FF) and a Functional-Only (FO) variations in the the functional partition. In FF mode, the simulator

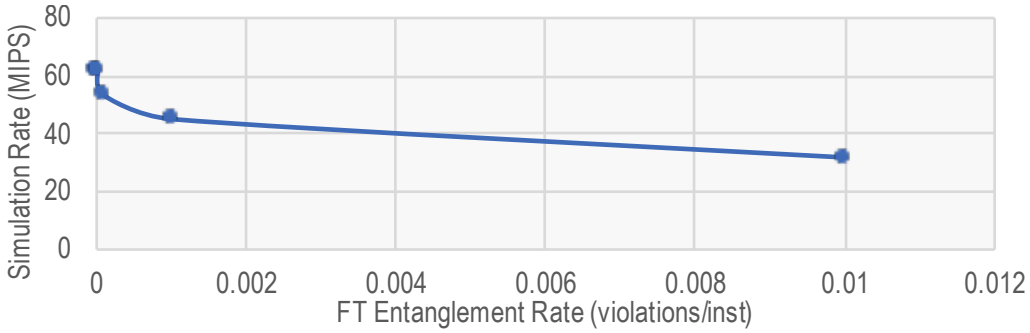

Figure 4.9: Impact of FT Violations on Simulation Throughput

is configured to disable actions to support rollback, while in FO mode, all tracing actions are further disabled. To find the limits of each approach we use a small test program alongside a null timing partition that immediately commits traces after consuming them. Comparing the various modes, most of the overhead is attributable to simply tracing instructions. In comparison, the additional overhead required for the supporting rollback and optimistic execution is relatively inexpensive.

4.7.3 Impact of Simulator Mis-Speculation on Overall Simulator Performance

To understand the impact of how simulator mis-speculation impacts overall simulation throughput, a small benchmark capable of inducing FT violations artificially is created. Using a magic NOP instruction, the functional partition can record a load value that that timing partition will treat as a FT entanglement violation. Using this magic instruction, varying amounts of simulator mis-speculation can be created with some given probability of occurrence.

A parallel application running on a multi-core processor must typically keep communication and inter-core interactions limited to maintain parallel scaling. This results in practical occurrences of load-value inducing FT violations being a rare occurrence for any highly scalable parallel application. Figure 4.9 depicts the per-

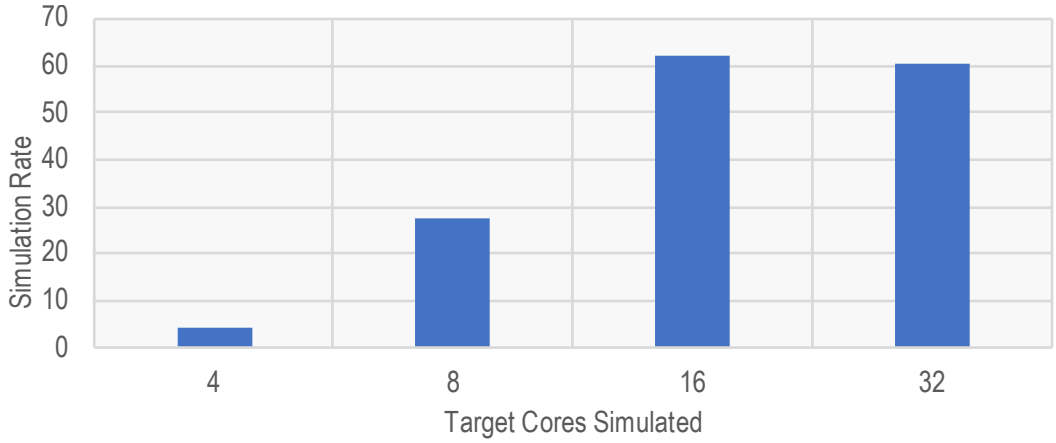

Figure 4.10: Impact of Target Core Scaling on Simulation Throughput

formance impact of from 0% violations to violations 1% of the time. We see that under low expected violation rates, simulation performance is not impacted. As the violation rate increase, simulation throughput drops gradually as well.

4.7.4 Impact of Target Core Scaling on Simulator Performance

The FASTMP simulator architecture was designed to leverage the decoupled synchronization offered by the optimistic-FT approach, there are still overheads incurred as target core counts increase. We configure the simulator with different numbers of simulated cores using a small benchmark to stress simulation throughput.

Figure 4.10 shows the result of this core count scaling study. The FASTMP simulator relies on hardware multi-threading to maintain efficient simulation on the FPGA. Under low core counts, the throughput of the simulator suffers as there is insufficient target parallelism for the simulator to keep its timing partition engines occupied. Once these engines are fully occupied, peak simulation throughput can be achieved. As target core count continues to scale, some gradual degradation of performance occurs due to limitations in scheduling timing engines on-chip and ensuring traces are brought on-chip with large numbers of cores.

4.7.5 Impact of Parallel Communication Patterns on Simulator Performance

To measure the actual impact of FT-entanglement and how common parallel synchronization patterns behave under the FASTMP simulator, we construct a set of common parallel patterns that cover various common synchronization and communication patterns. To cope with some limitations on the timing model and x86 micro-op compatibility, we use a light bare-metal C runtime and thread library as part of the benchmark runtime environment.

The benchmark suite is constructed to allow parametrization on how often communication occurs (configured in terms of how many work items can be processed independently without attempting to communicate), allowing study of how target communication and simulator performance. Four communication patterns are studied:

- *atomic* Parallel pattern using lock-free atomics to allow multiple threads to cooperate on some shared state without having to take a lock. An atomic add is used to model the impact when multiple threads are executing in parallel with read-modify-write occurring within a single instruction.

- *barrier* Barrier synchronization occurs periodically as work items are being processed. Once all threads have reached the barrier they are all released to continue processing work items.

- *critsect* Critical section mutexes protect a shared state variable that is periodically accessed by one or more worker threads.

- *sfifo* Pair-wise communication in producer-consumer relationship. Read and write pointers are synchronized periodically.

At low core count, we notice simulation throughput is limited as the FPGA contexts in the various multi-threaded timing partition engines have not yet been

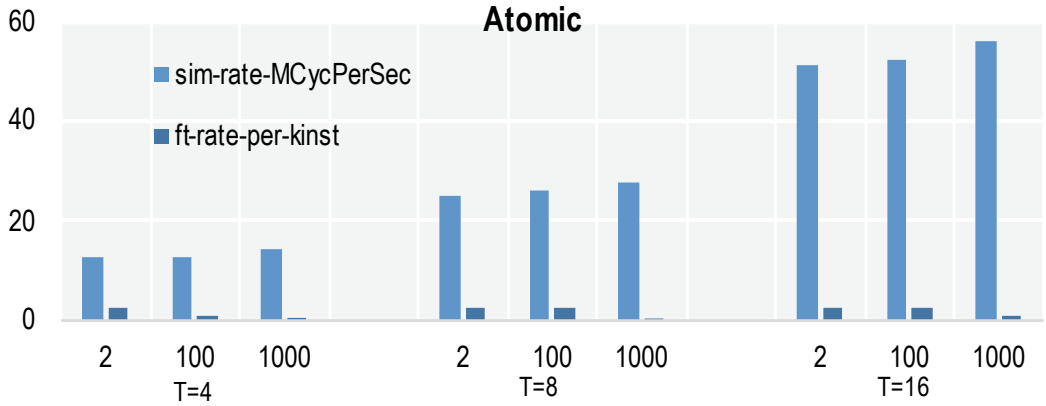

Figure 4.11: Atomic Performance of target and simulator on Atomic microbenchmark pattern

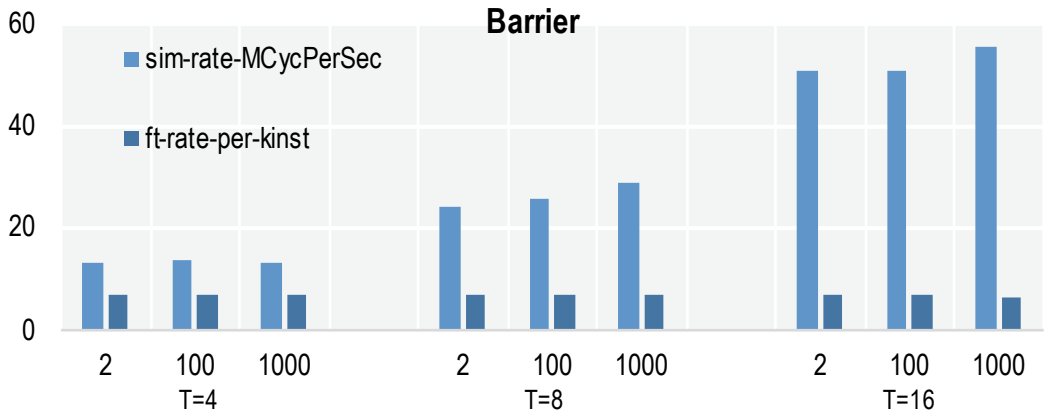

Figure 4.12: Performance of target and simulator on Barrier microbenchmark pattern

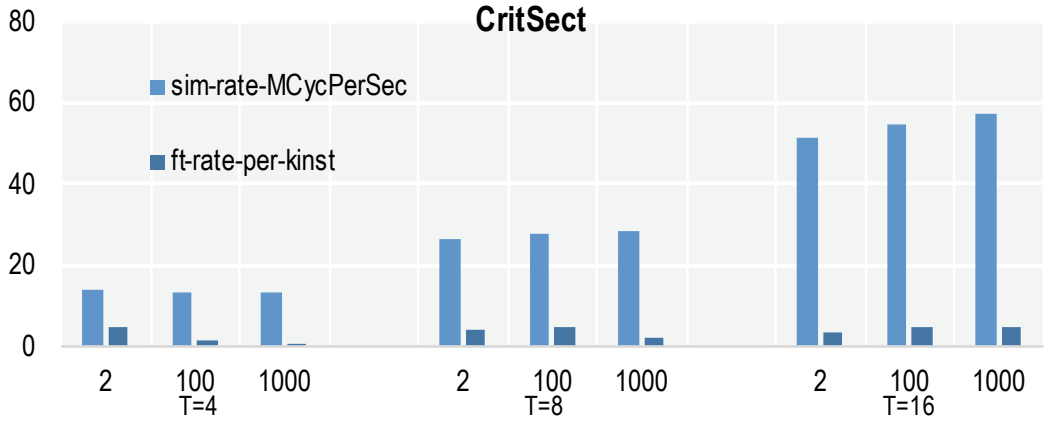

Figure 4.13: Performance of target and simulator on CritSection microbenchmark pattern

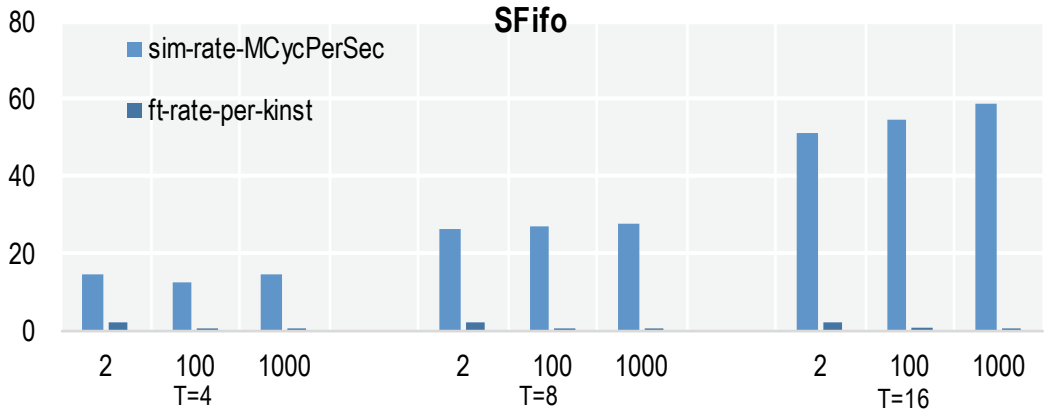

Figure 4.14: [Simulation Perf with Shared Fifo communication pattern]Performance of target and simulator on Shared-FIFO microbenchmark pattern

filled. As the core count increases, effective simulation throughput also rises. Further, at this higher core count, the actual impact of FT violations is minimal. In the parallel patterns, even though communication frequency (work items per communication event) scales from 2 to 100 to 1000, the actual FT violation rate is less than five in a thousand.

4.7.6 Comparative Study of FPGA Simulator Efficiency

Methodology

To compare the FASTMP simulator against other FPGA simulators, a fair method of comparing both speed and resource efficiency is required. A complication with hardware accelerated simulation approaches is the difficulty in creating comparisons on an equal footing due to differences in the hardware resources selected for a given implementation. To compare the efficiency of the FASTMP simulator, we need to create a fair accounting of how FPGA resources are used by different simulators. With a standard method for comparing resources, it is possible to allow fair comparisons against other hardware-accelerated simulation or prototyping methodologies that have appeared in recent publications.

Given the differences and limitations of the various FPGA simulators, it is not possible to normalize for certain differences (e.g.. hardware platforms, modeled target ISAs, and modeled target micro-architectures and target accuracy). We attempt to make as fair a comparison as possible by normalizing host and target differences where possible. We can then compare/contrast the characteristics of the techniques themselves, rather than simply those of the implemented artifact.

The first normalization to address is how to normalize for differences in FPGA resources and FPGA families. A modern FPGA is not simply a homogeneous sea of gates, but instead filled with lookup-tables, memories, flip-flops, and dedicated ALUs. Attempting to compare across different FPGA approaches must take into consideration how a given technique uses this heterogeneous collection of resources.

To normalize for the differences between different types of hardware resources (LUTs, SRAMs, ALUs, flip-flops), we construct a synthetic proxy metric *FPGA Resource Units (FRUs)* that is normalized to the cost of a single 18kbit SRAM. We use a fractional weighted ratio to derive equivalence ratios of each of the primitive FPGA resource blocks using a Xilinx Virtex5 LXT series FPGA as our baseline (Xilinx, 2012) as this FPGA family has appeared in previously published works on FPGA prototyping solutions.

As the FPGAs available from commodity vendors often come in relatively constant ratios inside a product family (only changing the total capacity of the part), we can normalize the differences between these heterogeneous resources and instead compare simulation strategies across a metric that is a useful proxy for FPGA area (which is difficult to obtain in openly published data). This metric also allows the convenience of comparing both single and multi-FPGA solutions as we simply sum the FRUs across the entire design.

To derive a canonical FRU metric for comparison we define it based on the ratio of available resources to Block RAMs in a Xilinx Virtex-5 LXT family

FPGA(Xilinx, 2012). In this family of FPGAs, there are between 60 to 80 slices for each BRAM available. Each slice contains four 6-input LUTs and four flip-flops. As such, we assigned one FRU to be equal to 280 6-input LUTs, or 280 flip-flops, or 1 18-Kbit Block RAM. In the Virtex 2, LUTs were 4-input as opposed to 6-input, so we generously assumed that 4 times as many 4-input LUTs would be required as to an architecture with 6-input LUTs, resulting in 1120 4-input LUTs per FRU.

The purpose of this metric is to normalize the various types of resources available on an FPGA for ease of comparisons. Thus, the number of FRUs a design uses is simply the resources of each type used divided by the definition above, then summed together across all types of resources.

The second complication in comparing approaches is the differences in clock frequency. To normalize for the differences between FPGA frequencies, we look at the activation rate of each simulator, combined with its FPGA operating frequency to obtain its effective simulation rate in target-core-cycles per second.

To obtain our net efficiency metric, we compare the effective simulation rate divided by the number of FRUs used to obtain this rate to yield target core cycles per second / FRU. This metric allows us to evaluate how efficiently we are using the FPGA logic to obtain a given simulation rate.

Resource Comparisons

We examine a set of FPGA simulator and emulators from previously published work, alongside the results from the FASTMP simulator. Using the normalization approach previously described, we can present the different simulators under a common rubric for analysis. Table 4.4 presents the details of this comparison.

Analysis

Table 4.5 summarizes the effective FRU efficiencies using the comparative data from Table 4.4.

	FASTMP	HASim (Pellauer et al., 2011)	RAMP Gold (Tan et al., 2010)	Protoflex (Chung et al., 2008)	Intel Nehalem (Schelle et al., 2010)	Intel Atom-like (Wang et al., 2009; Wong et al., 2008)
Methodology	Speculative functional first	Timing directed	Functional first	Functional only	FPGA prototype	FPGA prototype
			Target characteristics			
System	CMP up to 256 cores	CMP up to 16 cores	CMP up to 64 cores	CMP up to 16 cores	Nehalem core + L1	Atom-like core + L1
Core	7-stage in-order pipe	9-stage in-order pipe	IPC of 1 except cache miss	14-stage in-order pipe	Full RTL behavior	2-issue in-order pipe
Branch handling	Predict, resolve, flush	Predict, resolve, flush	Always correct	Always correct	Full RTL behavior	Full RTL behavior
L1 cache	256 tags per core (D-cache only)	256 tags I and D per core [1]	256 tags I and D per core	None	32KB (I and D)	8KB I and D
L2 cache	64K tags shared	4K tags shared [1]	64K tags shared	None	None	None
NoC	Bulk delay model	Detailed NoC model	None	None	Dummy memory unit	Special FSB unit
			Host characteristics			
Platform	ML605 (PCI-E board)	ACP (FPGA on FSB)	XUPv5 (PCI-E board)	BEE2 (custom board)	MCEMU (custom board)	Custom board on FSB
FPGA	V6 LX240T	V5 LX330T	V5 LX110T	V2-Pro 70	Five V4/V5 FPGAs	V5 LX330
FPGA clock	125MHz (64 or fewer cores)	50MHz	90MHz	90MHz	2.08MHz	50MHz[2]
Peak activation rate	125MHz (16-64 cores)	10MHz	90MHz (not sustained)	90MHz	520KHz	50MHz
Avg FMR at full throughput	1-2	5-11	1.9	1	4	1
1-core activation rate	16-18 cycles	19.7 cycles	not reported	14 cycles	4 cycles	1 cycle
			Resource accounting and efficiency			
FRUs (FM)	149[3]	405	114	359 [4]	N/A	N/A
FRUs (TM)	316	549	233	N/A	N/A	N/A
FRUs (total)	819	1196	422	359[4]	4037	504
Target-core-KHz/FRU	127	8.4	112	251	0.13	99.1

Table 4.4: Detailed Specs for Various Hardware Simulators/Emulators

	Type	F/T	*Resources* FRUs (FM)	FRUs (TM)	FRUs (total)	*Speed x Efficiency* (kHz/FRU)
FASTMP	Sim	SW/HW	149	316	819	127
HASim	Sim	HW/HW	405	549	1196	8.4
RAMP Gold	Sim	HW/HW	114	233	422	112
Protoflex	Sim	HW/-	359	N/A	359	-
Nehalem	Proto	HW	N/A	N/A	4037	0.13
Atom	Proto	HW	N/A	N/A	504	99.1

Table 4.5: Comparison of Hardware Accelerated Simulators/Prototypes

First, from a pure simulation performance perspective, the FASTMP simulator offers comparable performance to the even a pure hardware functional-first RAMP-Gold simulator and nearly 5x improvement over the much more detailed simulation found in the HASim simulator. lightweight simplified functional-first RAMP-Gold simulator. Comparing against other x86 FPGA implementations, the FPGA prototypes offer a marked look at how costly implementing the full functional of an x86 processor on an FPGA can be. Comparing against even a simple Atom implementation, the FASTMP simulator offers nearly 1.5x increase in simulation throughput while being able to use a software x86 implementation.

If we compare resource usage across the various FPGA simulators, we see some notable differences. With the FASTMP approach, the FRUs devoted to the functional model are used to implement the various streaming memory and trace support infrastructure required to support a software functional model. Despite this significant buffering and control overhead, the net resource usage of this component is on the same order as a lightweight functional-model (SparcV8-32b from RAMP Gold) or half the size of an FPGA-optimized soft core with a more complex ISA (e.g. SparcV9-64b from Protoflex).

This gap between the FASTMP FRUs required to implement the trace buffering, and actually implementing the functionality on-chip can grow significantly as the modeled ISA grows more complex. For example, a modern X86 instruction set may include 512b vector registers with SIMD operations for both integer and floating point

operations. Expending FPGA resources to implement this functionality will further push the resource benefits of the FASTMP. Leveraging a software model running on a commodity processor can significantly reduce on-chip hardware resources as the buffering/control logic costs are insensitive to the complexity of the functional model being executed in the software partition.

Finally, if we compare overall efficiency, we show comparable results to both the RAMP Gold and Protoflex approaches, while offering the benefit of leveraging existing software through lightweight instrumentation rather than developing hardware functional models completely from scratch. We also see the huge costs involved in both resources and overall efficiency for FPGA prototyping solutions. As these solutions cannot leverage fine-grain time-multiplexing or reusable FT partitioning, the overall area to support even a single core is significant.

4.7.7 Developing, Building and Testing

Like most FPGA-based simulators, building and using an FPGA for cycle-level simulators is a different experience compared to pure software simulators. FASTMP tries to make the overall simulation experience tractable through its use of abstractions, validation and debug environment, and its wide reliance on full featured software libraries to provide much of its functionality.

With a pure software simulator, users may compile a single simulator binary in a matter of minutes and then configure each experiment run using command-line options. Simulation may be extremely slow, and jobs are often launched in the background on compute clusters, running for days or weeks to complete a single run. In a FASTMP simulation, the user builds both software and hardware, then deploys the simulator onto a machine hosting a FPGA accelerator card with software providing the requisite command-line interface. From this perspective building and using FASTMP is structured similar to a traditional SW simulator, but with two key

	kLoC
External Components	
qemu x86 lib	158.0
Hardware (Bluespec)	
sim-core	7.2
generic data structures	6.5
platform support	1.0
timing model	8.7
total	23.4
Software (C)	
sim-core	10.8
jit instrumentation lib	7.1
data structure lib	3.7
platform support	0.8
total	22.4
Unit Tests (C/Bsv)	
hw unit tests	9.0
sw unit tests	7.0
total	16.0

Table 4.6: FASTMP Code Organization and Lines of Code Complexity

differences: compile vs run times and debuggability.

Source Organization

The software for FASTMP is written in C and is structured into a generic simulator core that handles the optimistic FT portions of simulation (e.g.. checkpointing, trace generation, command handling, logging and replay). The functional partition relies on a full x86-32 ISA implementation with a high-speed JIT implemented with QEMU. While extracting and subsetting QEMU for multi-threaded execution, tracing and arbitrary replay, the ability to build on over 150k lines of functionality is a huge benefit of the optimistic FT approach (where otherwise such functionality would have to be implemented on the FPGA directly). Finally additional support libraries and data structures are provided to allow efficient communication with the FPGA.

The hardware for FASTMP is written in Bluespec SystemVerilog, a mid-level RTL description language that builds ontop of SystemVerilog. Using a higher-level

language to program the hardware portions of the simulator reduces the design effort when creating and modifying the simulator. The code is structured similarly as the functional partition, with a simulator core that handles the processing and buffering of traces, handling commands, and scheduling the execution of the various timing model components.

Leveraging the high-order generic programming offered by Bluespec, generic data structures and helper modules are used where possible to reduce design effort. Finally the timing model itself is heavily parametrized to allow configuring various options without rewriting a module. For example, a cache module specifies its replacement function as an argument that the caller provides at compile-time. These properties make writing timing and simulation modules in hardware tractable as the design effort can be amortized across a range of instantiations, each with different parameters.

While the initial development of a new hybrid HW/SW partitioned simulator framework was costly in time and lines of code, the resulting framework can reduce development effort for subsequent experiments. The code in the simulation core, platform support and variety of parametrized hardware blocks and timing model components can greatly reduce the actual development effort experienced by an end-user.

Build Flow

With FASTMP, the user builds two components, software bits (including binary translators, simulator core and various platform support components) along with hardware bits (including FPGA board platform, simulator core and timing model components). Building the software bits is very much like the experience building a pure software simulator. The FASTMP software simulation core is compiled alongside the QEMU JIT with gcc, resulting in a standalone binary that can accept

command-line options, user provided test binaries.

Compiling the FASTMP hardware bits are built using more compute intensive FPGA compilers. The Bluespec compiler takes minutes to generate industry standard Verilog which is then consumed by the FPGA synthesis tools. Most of the build time is spent in this phase, where the Xilinx ISE synthesis, place and route, timing analysis and finally programming file generation occurs. Once the FPGA image is compiled, it can be deployed onto a test machine using a JTAG programming cable. The build times for the hardware may take 3 hours to complete. However, this long build time is traded for rapid simulation speed. Once the image has been compiled, simulation can be performed rapidly, offering 300x+ speedup compared to traditional software simulators.

Test and Debug Flow

As hardware design is more restrictive in its debug capabilities and visibility compared to traditional CPU SW debugging, multiple approaches were used during development to test the simulator. First unit tests were used to verify the majority of core simulator, generic data structure libs and timing model components. The unit tests contribute nearly 25% of total code in the simulator across both HW and SW.

Second, a rich co-simulation environment was created to allow nearly all the SW code to directly execute against a RTL-level simulation of the FASTMP hardware. A thin shim was introduced on both the hardware and software sides to abstract away the direct mmio and DMA transfers used to communicate between hardware and software. The majority of integration debugging could be performed here, with builds only taking minutes with full visibility into both HW and SW.

Testing directly on the final FPGA was reserved for more complex system or FPGA board issues. For these types of issues alternative debugging tools were developed including a GDB shell to talk to the FPGA ((Angepat et al., 2010)), debug

and perf counters for tracking and diagnosing accelerator deadlocks, and signal-level instrumentation for limited waveform debugging with Xilinx Chipscope.

While building and running an FPGA simulator can be a different environment from a traditional software-only simulator, the FASTMP simulator attempts to drive down the friction in using a FPGA simulator in the organization, partitioning and testing methodology.

4.8 Prior Work in Accelerated Simulation

4.8.1 Parallel Simulation Approaches

Compared with fast parallel simulation in SW, prior approaches have been forced to use simplifications to cope with the limitations of contemporary general purpose CPUs. Graphite (Miller et al., 2010) uses functional-first execution (with its previously discussed limitations) along with peer-to-peer approximate synchronization of simulated clocks to gain parallel scalability. Zsim (Sanchez and Kozyrakis, 2013) similarly uses functional-first execution with adaptive synchronization based on when sharing occurs in simulated cachelines. Unlike these approaches, a FASTMP simulator does not need to bake simplification into the simulator itself to gain scalability as we rely on the customization offered by FPGA execution to overcome the limitations of general purpose machines.

4.8.2 FPGA Simulation Approaches

As the capacities of modern FPGAs have increased, there has been an increased body of work focused on how to leverage FPGAs for simulation of parallel systems. Some systems like RampBlue(Krasnov et al., 2007) or TCC(Wee et al., 2007) simply use the FPGA as a prototyping platform using direct-execution. Other systems, like Protoflex(Chung et al., 2008), use the FPGA for untimed functional execution to

	Software FF-Approx	Software TD-detailed	Hardware FF	Hardware TD	Hardware Monolithic
Speedup	∼	✓		✓	✓
Design Reuse	✓	✓	✓	✓	✓
Simulation Correctness	✓		✓		
Hardware Efficiency				✓	✓

Table 4.7: Contrasting benefits of FASTMP approach to alternative simulation approaches

populate a detailed software simulator. Tan et al. (Tan et al., 2010, 2015) implement a tightly synchronizing functional-first simulator using FPGAs while Pellauer et al. (Pellauer et al., 2011) use a timing-directed approach in their system. In all of these approaches, the entire functionality of the processor must be developed and mapped to the FPGA at significant resource and engineering costs. The FASTMP approach advocates using FPGA resources for timing simulation, retaining the CPU to execute the functionality.

The FAST simulator (Chiou et al., 2007) holds the same philosophy as FASTMP in that both attempt to use CPU-FPGA hybrid simulation and optimistic simulation techniques. The primary difference between the approach is the focus on multiprocessor targets and the generalization of the optimistic-FT simulation approach. The architecture of both hardware and software for FASTMP was designed for both parallelism at the target and simulator levels, which leads to different tradeoffs in the design of the simulator architecture and implementation.

4.9 Contrasting Simulation Approaches

The FASTMP simulator offers significant benefits over prior approaches in multiple dimensions. Table 4.7 compares different simulation approaches (both pure software and hardware accelerated variations) along multiple simulator design dimensions. To address the first challenge addressed in this work (high-speed parallel simulation), the primary comparison metric is simulation speed. However additional dimensions of

design reuse (e.g. ability to reuse existing functional-only instruction-set simulators), as well as simulation accuracy are also relevant. In the case of comparing hardware accelerated simulators, the efficiency of how well a given approach can use the available FPGA resources can be compared as a hardware efficiency metric.

Compared against cycle-accurate, detailed pure-software simulators, FASTMP offers its largest speedup benefits, with two to three orders of magnitude improvements (depending on level of detail included in an accurate software simulation model.) For example, against the M5 software simulator a FASTMP simulator offers over 1100x speedup. This shifts the operating regime from the 10s of kIPS to to the 10s of MIPS. This benefit grows larger as the simulation model grows more detailed and accurate, as is often the case with industrial simulators that may operate in the single digit kIPS performance regime or slower. A FASTMP simulator also offers benefits in design effort, allowing the reuse of off-the-shelf high-performance JIT instruction emulators which detailed timing-directed software implementations cannot directly use. With FASTMP, more than 150k lines of x86 processor emulation code could be reused, avoiding the need to fully implement a functional partition from scratch.

Against timing-directed hardware simulators such as the HASim simulator, the FASTMP simulator offers nearly 10x improvements in FPGA speedup with nearly 2x improvement in hardware efficiency. These benefits grow directly by the ability to reuse existing software artifacts, only using FPGA resources for the timing partition. This improves both hardware efficiency as well as speed as complex functionality implemented in FPGA logic doesn't become a limiting factor in simulation performance. The overall simulator can be much faster as it allows more resources to be spent on the timing partition and/or can run at higher clock frequencies as complex functionality is not necessary to implement on the critical path in hardware.

Contrasting against monolithic hardware FPGA prototypes of full Intel x86 cores running on an FPGA, the FASTMP simulator again offers strong benefits in

both speedup and hardware efficiency. For example, contrasting with a complex Nehalem core implemented in an FPGA, the hardware resources dedicated to the functional partition can be reduced by nearly 30x. Even with a simple Atom-like core implemented on a FPGA, the hardware efficiency benefits still offer a 4x reduction in area spent to functionally execute x86 instructions on an FPGA. As we model increasingly complex processor cores the hardware simulator can continue to deliver speedup without exceeding the capacity of the FPGA. For example, for the same resource cost of implementing a single light-weight x86 Atom-like core, the entire FASTMP functional partition can be implemented which can support any number of processor cores.

Compared to functional-first hardware simulators like Ramp-Gold, FASTMP offers comparable speed but with full simulation correctness. When modeling simple processor cores, the hardware resources to support consuming traces and handling commands in FASTMP are similar to the cost of implementing the functionality directly in hardware but with significantly more complex functionality available from a FASTMP functional partition running on software on host processor cores.

4.10 Summary

The FASTMP simulator prototype was designed to leverage optimistic FT simulation with a hybrid CPU-FPGA platform. The simulator avoids having to implement functional partition directly in FPGA logic without sacrificing equivalence to monolithic/sequential simulation.

4.10.1 Contribution Summary

Optimistic FT simulators and the FASTMP prototype allow accurate functionally-directed simulation for multicore targets. Prior approaches either were limited to uni-processor targets or relied on heuristics to avoid FT-entanglements and violations

(leading to inaccuracies). An optimistic FT simulator has no intrinsic inaccuracies caused by its simulator architecture and can produce identical results to a timing-directed or monolithic simulator.

The second capability made possible by this work is a simulator architecture that can use hybrid SW/FPGA execution resources without compromising on accuracy or speed. Prior works may use SW running on the CPU for infrequent or complex operations that would be inefficient to implement in FPGA logic. Previous approaches again rely on heuristics, trading simulation performance for potential inaccuracies or force tight coupling between the HW and SW. The FASTMP prototype demonstrates an optimistic execution between HW and SW can be both loosely coupled and efficient. This changes the calculus of what the most efficient use of FPGA resources can be for parallel, high-accuracy simulators.

4.10.2 Lessons Learned

The design of the FASTMP prototype evolved as an experimental engineering project that encountered multiple challenges along the way including nearly two full simulator redesigns with experimental FPGA platform switches due to physical reliability issues. Many of the ideas presented in Chapter 3 and Chapter 4 only evolved through trial and error. We summarize some of the key learnings that were gained through this development.

Optimistic Simulation Complexity

While the abstract optimistic FT simulation presented in Chapter 3 presents simple means of detection and correction, translating these ideas into practical engineering artifacts can become challenging if done manually. Indeed, there are parallels between the complexity issues that early TimeWarp systems faced until the appropriate languages and abstractions were introduced (e.g. HLA time management abstractions).

In contrast, the FASTMP simulator evolved not as an generic optimistic framework, but rather as an attempt to transform a large (100k+ lines of code) untimed functional-only simulator into an accurate optimistic FT simulator.

The evolution of the abstractions and data-structures presented in this work only emerged after multiple iterative re-implementations. The rich set of simulator abstractions described in this thesis can form the foundation for a more fully generic optimistic simulation framework that can hide the complexity of using optimistic simulation from the simulator writer (following in the footsteps of how modern TimeWarp-like simulators evolved from initial TimeWarp simulation over subsequent frameworks).

Parallel Systems Complexity

The second lesson from building the prototype simulator is that contemporary approaches to the design, verification and debugging of a co-designed HW/SW partitioned system still have open problems. We adopted a high-level hardware language for FPGA design hoping to mitigate some of the productivity and correctness issues that hardware designs typically face. While we wrote extensive unit-testing testbenches for both hardware and software simulator components, co-simulation and co-debug was still necessary to identify many simulator bugs. As we adopted a large legacy instruction set simulator that used JIT execution, this co-debug meant coping with multiple abstractions, multiple languages, and multiple host execution platforms. As accelerator-based systems become more common, we expect many of these design and debug issues to mature.

4.10.3 Reducing Prototype to Practice

While the FASTMP simulator prototype offers the promise of fast, flexible and accurate simulation, it has a few hurdles to overcome before it can become a mainstream

simulator for research or industrial purposes. We enumerate some of these remaining challenges and some of ideas and trends that indicate such issues are tractable given the benefits of the proposed approach.

First, the availability of custom hardware to run simulations has become easier with the advent of FPGAs-As-A-Service (e.g. Amazon EC2 FPGAs rentable per hour (AWS-F1, 2017)). As the general approach described in Chapter 3 is not strictly restricted to hardware accelerator simulators, using a mixed implementation approach of running the simulator locally in software can make using FPGA acceleration a more practical resource to scale across more users. For example, for small experiments and initial debugging of models, a optimistic FT simulator running in pure software could be used. Then for large long runs, FPGAs hosted in the cloud may be used to run large-scale simulations with short turnaround times.

Second, the design effort of writing a simulator code for an FPGA has seen some progress over recent years as high-level synthesis (e.g. Catapult-C, Vivado HLS, and Intel OpenCL for FPGAs) has gained adoption for programming FPGAs. Such languages maintain the current expectations of simulator users who may write their models in C/C++. As these tools continue to gain traction and resource efficiency, some of the friction with using an FPGA-based simulator can be removed.

Third, while high-speed simulation is necessary to maintain short time-to-results when an designer is waiting for results from an experiment, debugging the model and understanding performance bottlenecks in the model are also critical usability factors. During the development of FASTMP, similar issues were encountered when trying to debug models on the FPGA, leading to the development of a GDB plugin to directly debug the FPGA without any explicit debugging hardware written by the user ((Angepat et al., 2010)). When combined with HLS-languages that allow executing purely in software, FPGA-based simulation acceleration can approach similar tradeoffs developers face when debugging highly optimized release binaries vs

debug binaries.

Taken all together, the available of hardware, efficiency of high-level synthesis languages and flexibility when debugging accelerated simulations, the remaining hurdles in reducing a FASTMP-style simulator to mainstream practice are tractable.

Chapter 5

FT Warped Simulation

5.1 Introduction

5.1.1 Overview

While the optimistic simulation approach described in previous chapters addressed the challenge of simulation speed, it did not offer any significant improvements in simulation flexibility. In fact, due to the focus on high-accuracy, modeling a new HW multi-core optimization or a new SW parallel framework optimization may take significant development effort. During late-stage development, such engineering effort can be completely warranted. However early in a design, users are often more interested in quickly evaluating large numbers of alternatives and only investing further development effort to promising choices. This need for high-flexibility is the second challenge addressed in this thesis.

Improving the scaling of parallel programs and systems, especially ones with complex, irregular data-structures can be difficult. Scaling problems have many causes, including long hold times on locks, false-sharing, unbalanced load, and communication through memory. How and when these problems occur is difficult to determine due to code complexity. Non-trivial programs often make extensive use

of irregular data-structures, including task schedulers, graphs, sets, and trees (with implementations from a variety of sources) often with hidden dependencies on lower level support and runtime libraries written by designers other than those performing the debugging.

Performance issues manifest themselves differently at different levels of load. For example, when there are few threads, long lock hold times may be the bottleneck. At this thread load, reducing lock hold times may provide the biggest win to improve performance. However, at higher thread-counts, memory locality may completely swamp the gains of this new locking scheme. This give and take is common in performance tuning making it a black art.

To further complicate matters, high-performance programs often incorporate adaptive algorithms that behave differently under different loads. A simple example is work stealing. Such adaptive behavior responds to changes in timing of execution paths by changing the behavior of other parts of the system in unpredictable ways. Subtle interactions such as these make it difficult if not impossible to precisely predict the performance after a potential code change.

The simplest solution is to use existing tools to measure the original performance and make a best guess on where the current bottlenecks in the system might be. Then make and debug the necessary changes and finally run the new program to see how it behaves. Such changes may require significant restructuring of the code and/or even changing the fundamental algorithms. Currently there is no other way to accurately predict the effects of making such changes. There is, however, no guarantee of success of following such a path. Many highly experienced programmers have had cases where they spent a significant amount of time going through this exercise to discover that their performance actually got worse.

While simulation allows a user to evaluate an idea without requiring a full implementation, it can still require significant effort to construct the model itself.

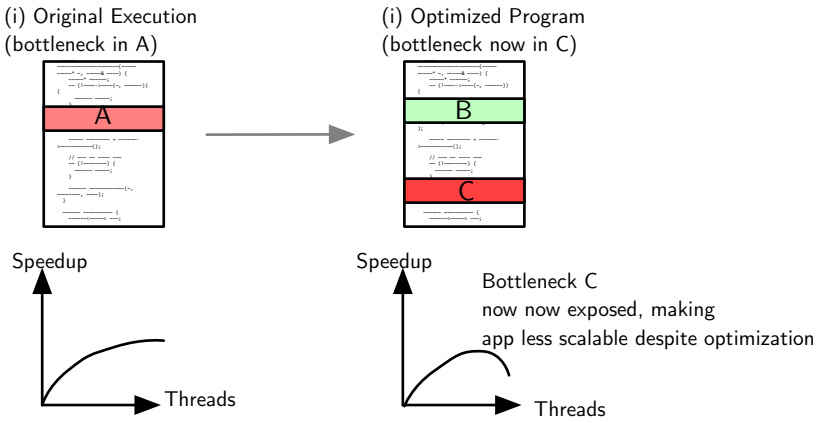

Figure 5.1: Illustration of complications when optimizing parallel systems. In (i), code section A may appear to have high contention leading to the poor parallel scalability. In (ii), a user may rewrite the same code section into A', reducing contention in the region. However, this can expose another bottleneck that was hiding, code section C, which has worse scaling and even higher contention. Overall scaling actually decreases (due to a hidden bottleneck that becomes visible only when the first is removed)

For example, to evaluate a new parallelized SW data structure, the effort to create the model may exceed that of simply implementing the new data structure. For HW development, developing models is sufficiently labor intensive that design points are chosen carefully using parameterization where possible to amortize design effort. These approaches can limit the set of design alternatives that can be explored to those that can be reasonably implemented or modeled.

Figure 5.1 illustrates two motivating scenarios that are difficult to tackle with existing tools. Due to nested interactions, it can be difficult for a user to fully comprehend the effort required to remove a bottleneck or how much benefit a single optimization might have on overall latency, through or scalability. These problems apply equally to both hardware and software. When optimizing SW data-structures for a parallel framework or deciding on potential code to try and offload to a hardware accelerator, it can be difficult to work out which optimization path is most profitable.

In a parallel system, evaluating a single optimization idea often requires

understanding how the rest of the system will react to the change across multiple dimensions including:

- execution correctness (e.g. locking paradigms, serialization, atomicity guarantees)
- resource management (e.g. contention on data structures and cache, memory bandwidth)
- hidden bottlenecks (e.g. optimizations stranded without hidden bottlenecks also being resolved)

Ideas may be hypothesized in isolation but it may be difficult to quantify their benefit without fully implementing and evaluating the idea using some form of system execution. When tuning parallel software, this incremental implement and evaluate loop can lead to an expensive backtracking as each optimization may not be the most valuable to get to desired performance goals.

For example, when tuning an application, each incremental optimization may be retained in the code as long it provided some benefit at some point in the tuning process. As refactoring a data structure to improve locality or contention may be a non-trivial task, such choices become accretive. Even if future optimizations make a previously implemented optimization irrelevant, the effort required to unwind the development may be prohibitive.

5.1.2 Motivating Example

When evaluating larger parallel system design questions (e.g.. parallel algorithms, datastructures, hw offloads), large amounts of functionality may be necessary to correct implement (or modeled accurately in a simulator) before evaluating a single idea.

```
objptr = 0x1000;
struct Foo {
  uint32_t lock;
  uint32_t data[32];
};

// Core1                                    // Core2
R0 = LOAD(objptr)                           STORE(objptr + offset(lock, Foo), 1);
R1 = LOAD(R0 + offsetof(lock, Foo))
if R1==0: goto J
SLEEP(500);
...
J:
  PROCESS(objptr);
  ...
```

Figure 5.2: A simple LD/LD/ST/Branch code fragment in a SW-centric demonstrates the same execution ordering complications discussed in previous chapters. Consider an optimization that reduces the frequency of how often Core2 must take a lock with the hypothesis to reduce the frequency of Core1 pausing its processing. Evaluating the impact of this requires understanding if Core1 reduces its pause periods that the PROCESS() operation continues to scale. However, a hidden bottleneck in PROCESS may only be discovered once the pause periods are removed.

Consider a slightly revised and more SW centric example revised from the prior chapter where an object containing a lock variable is tested by Core1 while it is locked by Core2.

Even in this small snippet, a variety of hypothetical optimizations could be proposed including changing backoff strategies during contention observed Core1, changing the granularity of how locks are distributed across objects, or changing how frequent Core2 may mark an object as locked. Questions of this nature may require significant refactoring of software algorithms and data structures to implement both correct and high-performance parallel implementations.

This toy example can be expanded to a more practical code snippet to better illustrate the challenges involved with optimizing a parallel system. Figure 5.3 illustrates a parallel find over an array. In the example, the index of each array element matching a given value is inserted into a synchronized linked list.

In a typical performance debugging scenario, the designer may use a sampling profiler to track the time spent in their code. Running tests while varying the number

```
void locked_insert(int value, List C) {
  N = new Node();
  N.value = value;
  C.lock();
  N.next = C.head;
  C.head = N;
  C.unlock();
}

parallel foreach (0 .. x) {
  if (A[x] == v) {
    locked_insert(x, C);
  }
}
```

Figure 5.3: Motivating Example: Inserting into a linked list the array indicies of all array elements equal to value v.

of threads enables differential profiling, which would indicate that *locked_insert* was a significant scaling bottleneck in this code.

After finding this bottleneck, the user considers their options and the requirements of the project. Since the code must scale to many threads, the user hypothesizes that a per-thread list will eliminate the bottleneck.

The designer may spend some significant effort both to implement a correct per-thread list that adheres to the program requirements, as well as refactoring any dependencies that may have not assumed the entire list may not be easily accessible at all times. Once finishing this refactoring, the user may rerun the application and use differential profiling again.

With this optimization the scaling may be improved considerably, but still not sufficient for program requirements. The second profiling run now indicates that allocation of list nodes is now a bottleneck in the newly optimized code.

The user decides to store multiple indices per list node to reduce the number of allocations. The user hypothesizes that contention in the allocator is driven primarily by the number of allocations and not the size of allocation and chooses to store eight indicies per list node.

We can make some observations from this simple example about the limits of

```
void locked_insert(int value, ListWithPerThread C) {
  N = ListAllocManager.new()
  N.value = value;
  C.lock();
  N.next = C.head;
  C.head = N;
  C.unlock();
}

parallel foreach (0 .. x)
  if (A[x] == v)
    locked_insert(x, C);
```

Figure 5.4: Motivating Example: Optimized code fragment after implementing both per-thread lists and packing multiple indicies per list node.

differential profiling and how parallel system optimizations may be reasoned about.

Even though differential profiling indicated the list locking was a useful candidate for optimization, it couldn't indicate that the node allocation in the same scope would also be required to be optimized. Coping with this incremental discovery of bottlenecks can lead to cases where removing contention pressure in one part of the application may cause pressure to increase in another part of the application. For example, consider the case where the ListAllocManager allocator fails to scale as load increases. The user then finds the application has worse scaling with the new optimized list locking scheme.

Using a simulator instead of differential profiling on a native machine doesn't improve the designer experience significantly. While a simulator may offer more detailed visibility into the processor pipeline and memory hierarchy, it does little to predict that the allocator would become the bottleneck after the list lock contention was resolved.

We observe a key limitation of simulation and profiling; we can only influence scalability through a direct implementation of an optimization, with only indirect influence over the performance and resource characteristics of the optimization itself. While the primary goal in evaluating an optimization is to observe the system under some modified performance or resource changes, it is only possible to do so by

indirectly by implementing the optimization explicitly.

The other point of interest in the optimized code fragment is how close it resembles the original code fragment. While the data-structures backing the list locking and the node allocation have been completely rewritten, from the application perspective, these data structures and operators are functionally equivalent. The development effort to create these new data structures may have required weeks to create, reason about correctness, and refactor the application to use them. But from the high-level functionality of the calling functions, the overall behavior has remained the same.

This functionality-preserving property of optimization can hold widely across various optimizations. Optimization in a parallel system may follow the the maxim "make it work, then make it fast". Optimizations may replace entire subsystems or data- structures in a parallel system, preserving functionality and simply altering the implementation with different performance/resource characteristics. For example, loops may be restructured to have better locality, or locks may be split to increase parallelism, or critical sections optimized to reduce lock holding times. In all these cases, while there may be large amounts of code changing to implement the optimization, the overall program functionality is preserved while reducing cache pressure or decreasing execution time due to reduced lock contention.

The user hypothesizes about these substitutions and the replacements may even by micro-benchmarked to gain intuition about their performance and resource characteristics. However, the fundamental limitation is that the user must still implement the optimization and evaluate the system as a whole to gain insight into overall system performance.

When taken together, these two observations pose an tantalizing alternative to traditional differential profiling and simulation approaches. If it is possible to directly manipulate performance and resource characteristics from a simulator rather

than indirectly through implementation, the effort required to optimize a parallel system could be dramatically reduced.

5.2 Alternative Workflow

Instead of this incremental optimization pattern, an alternate work-flow based on a simulator that is able to predict the performance effects of coding changes is proposed. Using the simulator, users ask "what if" questions about programs and get back performance predictions they use to guide their decisions about what and how to optimize. What if memory allocation was moved from this point in the program to that point in the program? What if fine-grained locking was used there? What if this critical section was shorter? What that mutex could be avoided most of the time? Such "what if" questions have a simple answer, namely the effect on in scaling, but that answer must be produced a-posteriori due to the many effects listed earlier. Such a tool would enable users to explore the space of potential changes through predicted impact compared to the coding effort. This would enable a user to find the simplest design that meets the scaling goals and avoid wasting time on a complex implementation that, in the end, has marginal performance benefits or even degrades performance.

Such a simulator replaces the iterative steps of profiling, hypothesizing and implementing with a simulation-driven performance projection that short-circuits the intuition building. Such intuition is often limited by the relatively slowness of implementing complex parallel, scalable data-structures and algorithms and the complex interactions that may only appear when an optimization is integrated into an application.

To answer these what-if questions, a user introduces annotations to their existing code to describe the properties of an optimization. The simulator would then execute the original code but use the annotations to change the execution properties

```
void locked_insert(int value, List C) {
  N = new Node();
  N.value = value;

#pragma START no-contention, no-invalidation
  C.lock();
  N.next = C.head;
  C.head = N;
  C.unlock();
#pragma END

}

parallel foreach (0 .. x) {
  if (A[x] == v) {
    locked_insert(x, C);
  }
}
```

Figure 5.5: First Annotation: Simulates no lock contention and no cache-invalidations, a reasonable approximation of a private linked list per thread.

of the simulated code. This allows the simulator to predict the performance and scaling of a proposed code change without implementing it.

Using annotations, the user has only to specify how the change should behave in isolation, for which they may have a relatively good understanding of a proposed optimization. The simulator then handles the harder task of figuring out how the optimization would impact the overall scalability, latency or throughput of the application.

5.2.1 Optimization Example

Using this annotation-based approach to optimization exploration, the original motivating example can be revisited.

The user starts by using the profiling information as a guide, looking for opportunities to improve the list locking behavior. The user annotates function *locked_ insert* as shown in Figure 5.5 to mark the lock-acquisition and linked list insertion as having no contention and no memory-level invalidation. No-contention means that the tool will never interleave executions of the region, so the lock will

```
void locked_insert(int value, List C) {

#pragma START 7:8 no-contention, zero-cost
  N = new Node();
#pragma END

  N.value = value;

#pragma START no-contention, no-invalidation
  C.lock();
  N.next = C.head;
  C.head = N;
  C.unlock();
#pragma END
}

parallel foreach (0 .. x) {
  if (A[x] == v) {
    locked_insert(x, C);
  }
}
```

Figure 5.6: Second Annotation: Simulating larger list nodes storing multiple items by simulating one eighth the contention

always appear uncontended. Further, no-invalidation signifies that the tool will not simulate the cost of cache coherence in the region. These changes to the target behavior simulate the expected behavior of making the list thread-private.

Simulating this annotated program reveals that scaling improved considerably, but still not enough. The simulation run indicates that allocation of list nodes is now a bottleneck in the transformed execution. The user then makes a second hypothesis that multiple indicies can be packed into a single node. As seen in Figure 5.6, the user annotates the node allocation with a zero-cost annotation that sees no contention and instantaneous execution seven-eights of the time.

The simulation of this model indicates that the user's scaling requirements are met. They now have some confidence that their modeled implementation, thread private linked lists with blocks of eight indices in each node, will perform acceptably and implements the changes accordingly.

Throughout this new workflow, the user only makes modifies annotations rather than implementing each optimization. Measuring the impact of a proposed

optimization at the system-level before needing to implement all the details can greatly improve the flexibility and agility when exploring optimizations for parallel systems.

At each step, the user is only required to reason about the performance and resources impacted by the code section being marked with annotations and understanding a concrete implementation of the optimization would be functionally equivalent. They are shielded from the reasoning about how the private allocation scheme would perform in the overall program execution.

5.3 Warped FT Simulation

5.3.1 Introduction

Given the benefits of this new proposed optimization workflow, the question is how to organize a simulator to implement these new features. From the discussion of the motivating example, we observe a few interesting properties about the proposed simulation flow. First, optimizations are often functionally preserving and may only affect the timing or resources consumed when the optimization is applied. Second, system-level interactions are necessary to understand if an optimization will be sufficiently valuable. Finally, that directly influence the performance characteristics and resources are necessary to enforce annotations provided by user.

These properties align well along the logical partitioning between functional and timing simulation. Furthermore the combination of both functional preservation in local code but potentially highly sensitive at the application level matches observations of FT entangled state. Using this knowledge, it is possible to craft a simulator capable of supporting this new workflow using the properties and interactions of functional and timing partitions. By leveraging properties of the how FT partitions interact, it is possible to enable this new workflow to a wide class of simulators, rather

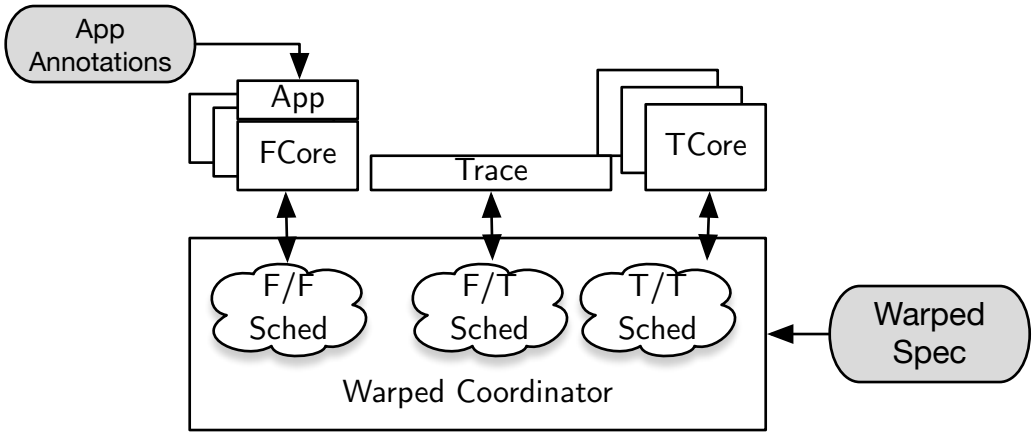

Figure 5.7: Abstract Architecture for a Warped FT Simulator

than as a specific tool.

5.3.2 Simulator Architecture

A *warped FT* simulator is used to describe a simulator where user-specified scheduling constraints are used to influence the execution of the simulator and as a consequence, predict performance under different constraints. In a conventional FT partitioned simulation architecture (e.g. Functional-First or Timing-Directed), each functional and timing partition may be executed and interact to strictly maintain simulated correctness and causality. By contrast a warped FT simulator explicitly controls *how* and *when* each F and T partition may execute according to some user constraints. Controlling partition execution in this way allows the simulator itself to become a building block in exploring optimizations.

Figure 5.7 depicts a warped FT partitioned simulation. The simulator is divided into sets of Functional (F) and Timing (T) partitions, communicating via a trace buffer holding traces generated by F and consumable by T. Each partition and the the trace buffer between partitions is hooked explicitly into a warped FT simulation coordinator. The coordinator handles the scheduling enforcement between and within

partitions, translating user directives into execution constraints. The coordinator is fed a warping specification that describes a particular set of optimizations the user is considering.

The warped FT coordinator is responsible for three key activities:

- functional co-execution (e.g. is it permissible to execute a given Functional thread/core when another Functional thread/core is also executing in concurrently)
- relative advancement of time (e.g. how fast or slow should a given Timing thread/core advance relative to other Timing thread/cores)
- trace interaction (e.g. how should the trace between a given Functional thread/core and its Timing thread/core counterpart be controlled)

By inserting the warped simulator scheduler/coordinator into these activities, its possible to insert additional constraints to enforce a specific artificial schedule for exploring a particular optimization.

5.3.3 Selective Warping of Code Sections

To implement the proposed annotation workflow discussed in the example, we can require two features to describe an annotation: i) identifying code sections to warp and ii) describing how to warp the performance or resource characteristics of a code sections.

A *Region* is used to describe a particular code section that is annotated by the user as being interesting to warp. A region is delimited by a start and end and may cover individual instructions, collections of program statements or entire function call graphs. While a region may be described as starting and ending at two given points in a program, the portion of code contained within the region covers the entire dynamic sequence of instructions that are executed between these two start and end

```
void locked_insert(int value, List C) {
  N = new Node();
  N.value = value;

  WARPED_MARKER(REGION_START, R1);
  C.lock();
  N.next = C.head;
  C.head = N;
  C.unlock();
  WARPED_MARKER(REGION_END, R1);
}
```

Figure 5.8: Region Markers: A user instruments a code section of interest with a pair of markers to describe where the region should start and end and some way to name the region. Region R1 describes a region that inclusively covers the lock and unlock function call execution, along with the update the head and next variables. It does not cover the allocation of the Node as that exists outside the region.

points. Thus, function calls and other control flow instructions are permissible in such regions. Even though instructions and functions are statically delimited between begin and end markers, they are not part of the region if they are not executed, for example, due to conditionals.

To implement region delimiters, code markers specifically recognized by the simulator can be used. A "begin" marker and an "end" marker can be implemented with an empty function call that has no impact on program behavior unless run on a warped simulator. Region markers simplfy inform the simulator when a thread or core is entering or leaving a region, but does not describe specifically how to warp the FT simulation to enforce a given desired performance prediction. Figure 5.8 provides an example of a marker used to describe a region named $R1$ that contains all the dynamic instructions required for the unlock and lock functions along with the update of the head and next variables.

Some care must be taken to ensure that begin and end markers are dynamically paired. For example a user wishing to warp a region that encounters an exception should eventually terminate the region with an end marker. In practice this constraint is easy to meet given the desire to warp interesting portions of parallel code where throwing exceptions is often handled locally in scope if at all.

```
[R1]
name = NodeUpdate
mode1 = no-contention
frequency1 = 1.0
mode2 = no-invalidation
frequency2 = 1.0
```

Figure 5.9: Warping Specification: A user provides the hypothesis of an optimization they would like to evaluate before implementing the changes. In the previous example, the user can explore the benefits of using thread-private list of Nodes.

5.3.4 Describing Performance and Resource Constraints

The second feature required to implement the proposed annotation workflow is the ability to describe how to warp the performance or resource characteristics of a *region*.

A *Warping Specification* is a user-defined file that describes a particular set of hypotheses to evaluate in a warped simulator.

Figure 5.9 shows an example of a warping specification for exploring the impact of turning the Node update operations into a thread-private operation. By describing both no-contention and no-invalidation in the specification, the warped coordinator can enforce these constraints when an FCore or TCore is actively processing region $R1$. A more comprehensive description of the variety of warping constraints supported in a warped simulator is described further in Section 5.3.6

As the simulation progresses, threads may enter and leave regions, each having their own active set of warping specifications being applied. As threads leave a region, the warp scheduling constraints are removed, allowing normal execution to resume. The side-effects of executing the region under warped constraints can then simply be measured to see if the user's proposed optimization is beneficial to overall application performance.

5.3.5 Example of Warped Scheduling

Given these mechanisms, we can revisit the initial motivating example of array traversal and creation of a list of array indicies. It was observed that removing lock

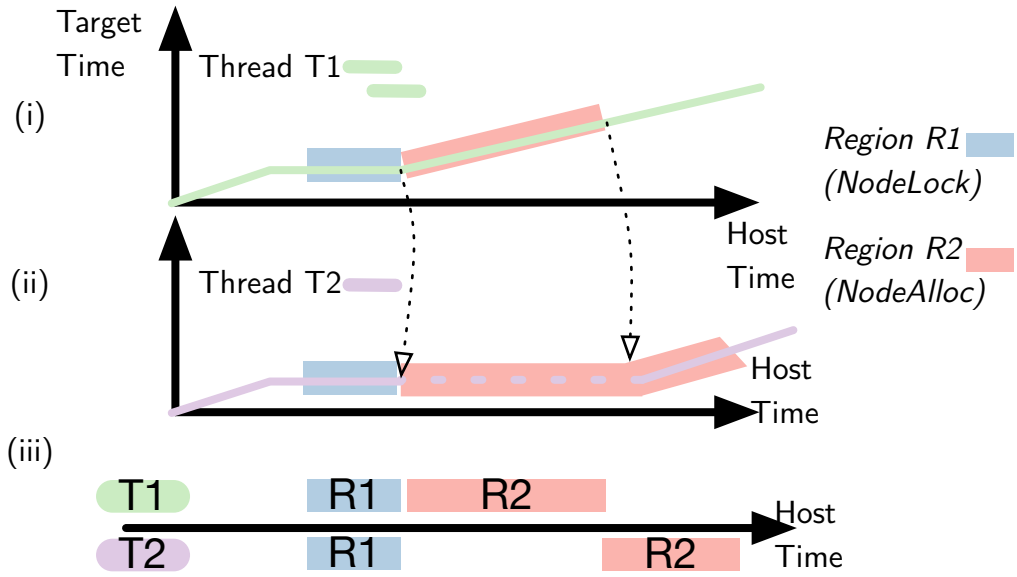

Figure 5.10: Example of exploring optimizations with region warping

contention around updating nodes in the list and time spent allocating nodes was required to remove scaling bottlenecks. Figure 5.10 depicts the impact warping can have on this example.

To understand how the warping interventions work, we use a consistent illustration to capture the model of execution used by the simulator. For all examples we use the same organization: 2 threads, each represented by a different color. The x-axis represents host time running on the machine running the simulator. The y-axis represents the target simulated time associated with a single thread. In a conventional simulator, each thread would make forward progress in both dimensions, with the slope of the line representing the simulation speed of a given thread. Convention simulators also keep target simulated time in lock-step, so the thread-local target time is the same as the global target time.

A warped simulator enables the user to modify how time is accounted for by

the simulator. By doing so, it is possible to simulate many cycles of instructions and account for them differently in the target time visible to the thread. For example, we may simulate 100 cycles of effective execution in the target, but account for this as 10 cycles in thread time. Thus, it is as though the thread was executing on a 10x faster machine for this period than other threads (achieving 100 cycles of execution in the "time" other threads can complete 10).

When the threads enter region R2 (representing the node allocation region of code), threads are temporally warped, reducing the time spent in the code by 7/8ths. When the threads enter R1 (representing the list locking region), threads take turns entering this region and pause if necessary in the functional partition. This prevents the threads from executing down any adaptive code that may be run when observing contention.

5.3.6 Representing Optimizations as Partition Constraints

In a warped FT simulator, a few points of interaction can be controlled with user-directed scheduling. Specifically by controlling the scheduling and interaction of logical partition process (LPP) execution amounts to controlling three types of interactions:

- F/F interleaving (e.g. controlling how much contention/cooperation might be observed when jointly executing a region)
- T/T interleaving (e.g. controlling relative pacing of how each partition advances through simulated time
- F/T interaction (e.g. controlling how traces are generated/consumed between partitions)

Partition interactions (e.g. F/F, F/T, T/T) are exposed as first class simulation primitives that are available for direct manipulation by the warped simulator.

The simulator can then use schedule and interaction constraints to form a basic primitive to allow exploration.

Exposing simulation scheduling in this way allows keeping the warped simulator coordinator to have a small interaction with the simulator. Controlling partitions instead of explicitly attempting to control simulator scheduling, provides a generic mechanism for warping simulation to integrate with existing partition simulator implementations. This allows warped simulation to be applicable to FT-style simulators in general. In particular, it avoids needing to construct detailed parameterized models to mimic some particular effect on T (as might be necessary if attempting to do similar limit studies in a conventional simulator).

A warped simulator is most easily applicable to functional-first style architectures as starting point. A FF-style simulator allows more flexibility in enforcing schedule constraints, as functional correctness still preserved and T tolerant to timing skew. Applying the approach to timing-directed is also possible at the cost of additional metadata tracking within the timing partition to track regions and more aggressive detection of unsatisfiable constraints may need to resolved. For example, section 5.4 describes in more detail how enforcing an artificial schedule may induce deadlock issues during simulation and mitigation techniques.

From these three partition interactions, three useful warping specs can be supported:

- Temporal Warping
- Ordering Warping
- Locality Warping

The *temporal warp* changes the runtime of a region within the simulator. A region can be charged fewer cycles, thus making the region appear to have executed faster.

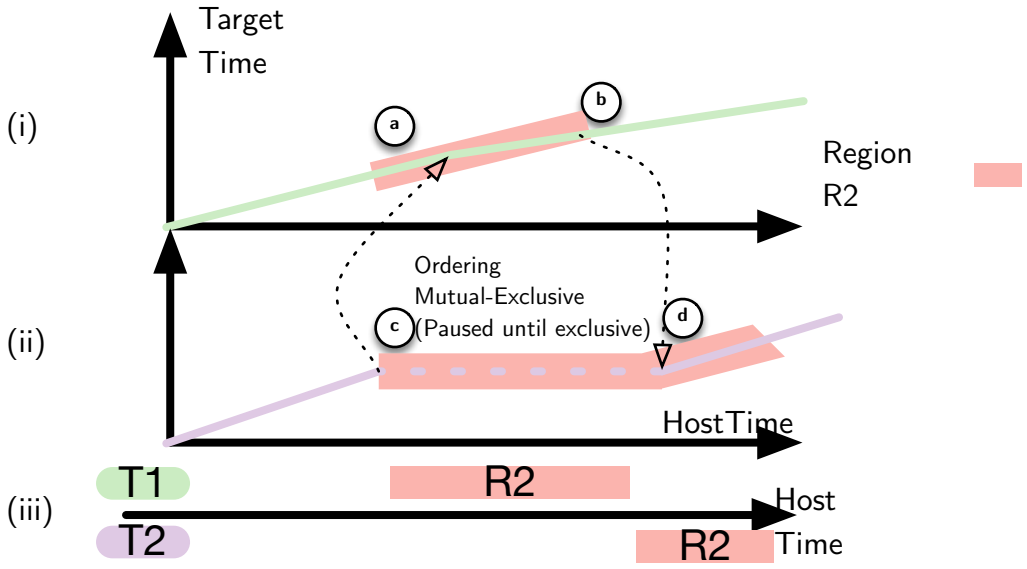

Figure 5.11: Ordering Warping. At point (a), thread T1(green) enters a mutual exclusion region. Thread T2(purple) is also at the boundary of this region, so the purple thread is not scheduled until after thread T1 leaves the region at (b). At (c), the thread T2 enters the region, having never seen the fact that thread T1 was in the region, but observing all the side effects of the green thread. Thread T2 leaves the region at (d).

The *ordering warp* eliminates the waiting time to enter a critical section. A core attempting to enter a held critical section is automatically paused by the simulator, making time appear to not pass for it until it is able to enter the critical section. In this manner, the paused threads never see the fact that the region was already entered, they simply see the side effects of the previous thread. The time consumed is adjusted accordingly as well.

The *locality warp* allows a thread to observe that other threads have entered a region and controls how often and for how long the thread observes that contention.

Ordering

The warping adjustment offered allows controlling the contention observed within a region. Mutual exclusion warping prevents contention on software resources by ensuring that the region executes in isolation. This isolation is enforced in the simulator; the protected region always appears to only be executing on one thread. Mutual exclusion allows the user to ask questions like "what if I used fine-grained locking here?". A probability filter, or a higher limit on the number of allowed threads in the region supported to allow mimicking various optimization strategies to reduce contention.

Mutual exclusion allows threads to skew relative to each other, which is demonstrated in figure 5.11. Here the thread T1 (green) and thread T2 (purple) will contend for a single mutual exclusion region. At (a), thread T1 enters the region. The basic scheduling would have allowed thread T2 (purple) ton continue running, since it was the furthest back in time, but this thread is waiting (at (c)) to enter the region thread T1 is still in. Thus the scheduler allows thread T1 to continue running while thread T2 is halted in target time. At the next scheduling point, thread T2 is still the furthest back and still ineligible to execute. Thus the scheduler chooses the second furthest back thread which is still thread T1. The thread T1 leaves the region at (b), allowing the scheduler to now run the thread T2, which is the furthest back in time and free to enter the region. As thread T1 now leads thread T1 in target time, thread T2 will continue processing until it synchronizes its thread-local target time with the global target time.

Temporal

The base time used for temporal warp retiming is the execution time of the region after any locality changes discussed later. Conceptually, this execution establishes the time-stamp for various events such as instruction completion, cache access, invalidation

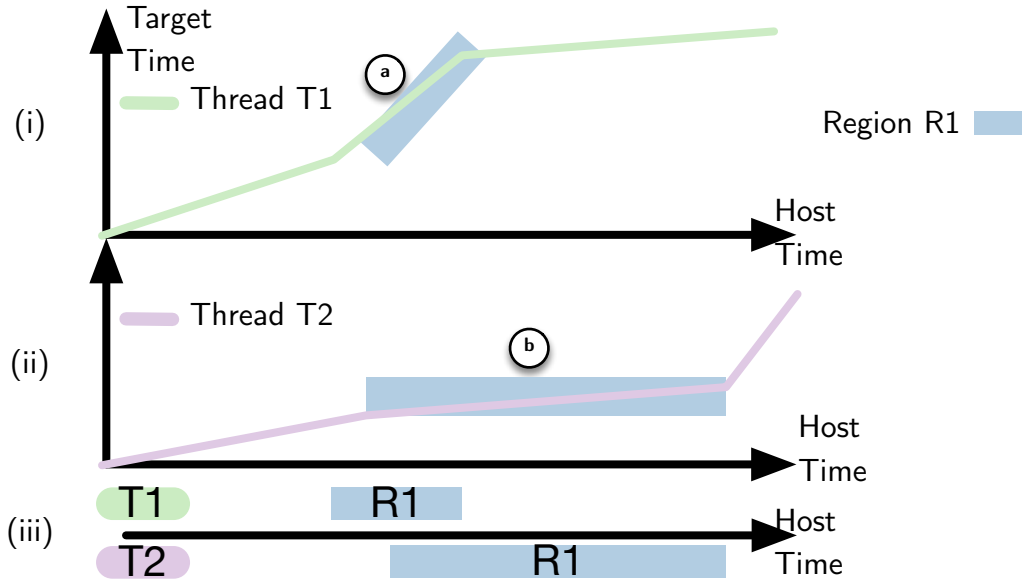

Figure 5.12: Example timeline of a thread executing and hitting region $R1$ with temporal warping constraints. Time compression and dilation. During (a), the thread T1 (green) experiences a time compression ratio of 2:1 in which instructions take twice as long to execute. During (b), the thread T2 (purple) experience a time dilation ration of 1:2 during which instructions take half as long to execute.

messages, and so on. These time-stamps are then scaled by the dilation factor to compute the time on the global target time reference clock when those events appear to other threads.

The temporal warp dilates the time taken by a region relative to the base time and thus relative to other threads. The retimed region thus appears to have executed either faster or slower than it would have normally. Retiming allows the user to ask questions like "What if this region ran faster?" to model, for instance, an hardware accelerator being added to the baseline system. By adding a probabilistic filter, for example, one can simulate adding a statistical cache to memoize that accelerator; 75% of the time, the region would run with a small fixed cost to simulate a cache hit and the rest of the time, the region would run without retiming.

Figure 5.12 shows the thread T1 (green) running in retimed mode until point (a). The thread T2 (purple) executes in zero target cycles (x-axis). On the other hand, the thread T2 also runs retimed until point (b), but then time is warped in the other direction. Then, the purple thread executes more cycles than it would have executed otherwise.

Locality

While the temporal and ordering warps modify coarse software behavior, many parallel optimizations reduce the utilization or increase the efficient use of critical hardware resources. The key among these are the shared memory hierarchy.

Changes to the locality of a region affect how the cache and memory are timed, updated, and used. An access can be timed as though it was satisfied by a specific cache. A cache update can be made to appear to propagate as an instantaneous, or fixed latency broadcast. Invalidations can be prevented while still maintaining coherence.

By changing the behavior of the cache, the latency of loads and stores is

changed. The modified latencies are used to establish be baseline time of a region before other changes are applied.

Changing the locality of a region enables asking questions about a wide variety of parallel optimizations. Annotating a mutual exclusion region as always hitting in no-invalidations simulates the privatization of the data and lock used in that region.

5.3.7 Contrasting to Optimistic FT

A warped FT simulator shares some similarities with optimistic FT simulation by allowing F and T partitions to execute under some simulator scheduling policy but differs in exactly how these partitions are scheduled and executed.

While an optimistic FT simulator decouples execution and scheduling of F partitions without strong interactions from the T partitions. If scheduling interactions occur, they only do so in response to entanglement violations which force rollback and repair operations. When this style of simulator detects a potential violation in the T partition, the simulator corrects the F partition, forcing the previously unconstrained schedule to align with the T partition expectations.

By contrast, a warped FT simulator applies an artificial constrained scheduling policy when executing F and T partitions. Rather than following the T partition for scheduling, a warped FT simulator attempts to selectively enforce a scheduling constraint specified from user specified directives. As the scheduling policy is explicit, rather than allowing unconstrained execution with rollback/repair as in optimistic FT, a warped FT simulator tightly constrains execution to match some particular ordering, interleaving of the F and T partitions.

5.4 Correctness and Causality

In a traditional cycle-accurate simulator, correctness relies on having a valid implementation of the simulator model, and having a legal program execute on the

simulator. In a warped simulator, due to the ability to enforce artificial scheduling constraints, it is possible to introduce new correctness, causality issues that may simply be due to poorly formed schedule.

5.4.1 Correctness

Given the changes the simulator can induce on program execution, one must consider whether the changes result in a correct execution. This question comes in several forms. The fundamental question is whether executing blocks of instructions serially with the techniques we use, including retiming and mutual exclusion, produces a correct program execution.

To answer this question, consider the functional, timing split of the simulation. We argue that the functional simulation itself produces correct executions because a warped simulator does not reorder execution within a single functional partition; all we change is the interleaving of functional partitions with respect to each other. *Therefore, we always guarantee sequential consistency.* From a functional standpoint, this core interleaving is something an OS scheduler could do as well. A warped schedule may enforce that only one core in the functional partition may execute at a time, which is still a valid execution of the target being simulated. The warped simulation technique, functionally, is entirely about controlling scheduling including window sizes and which the functional core to execute next. The simulator produces a possible, but potentially unlikely, execution which could have been produced by an operating system.

There could be a class of programs which depend on detailed timing for their correctness. Such programs use some (non-architected) knowledge of the underlying architecture to, perhaps, avoid synchronization. Such programs do not have sequential semantics and, therefore, we provide no guarantees for such a program. Programs must have sequential semantics to have sequential semantics in our system.

Given correct functional execution, the correctness of the timing is not an issue. What matters for the timing simulation is that the produced timing is useful. A useful timing result is a close approximation to the behavior of the system if the user had implemented the change he proposed and they had accurately estimated the behavior of the change.

At the simulator level, the modeling error between a given simulated machine and a corresponding native machine as well as the previously described error introduced by biased scheduling skew both introduce errors that may be compensated for with better models and more compute time. At the designer level, the accuracy of a projection within the framework is dependent on the ability to predict what benefits an actual code transformation may actually become when the change is implemented. While a warped simulator allows a user to make unrealistic assumptions (for example a user may set the contention on a lock to be exactly zero, when after actual implementation the actual contention may be 5%), the user is equally able to make conservative estimates as well.

Aside from basic functional correctness, enforcing user-scheduling directives can potentially lead to additional concerns for causality issues. In particular, it is possible to have constraints on how partitions execute that may lead to impossible schedules. Such situations are explicitly *not* bugs in the program being optimized but are rather interpreted as simulator schedule restrictions. As scheduling constraints are not intrinsic to program correctness, it is possible to break constraints temporarily if necessary to maintain some legal program execution. We describe two situations under which impossible constraints may arise and solutions for each.

5.4.2 Forward Progress and Deadlock

The focus of the simulator is **not** verification, validation. Existing tools (race-detectors, memory checkers, etc) cover this arena to a wide degree. We assume

correct well-labeled parallel programs and discuss the implications of arbitrary retiming, rescheduling as it relates to the correct execution of an arbitrary application. With deterministic, non real-time applications, each thread must be able to make forward progress regardless of the processing rate of other threads in the system. Although we have yet to observe any real instances of such applications, soft real-time applications that expect forward progress from each thread may encounter difficulties within our modified execution environment. Since regions demarcate the start of a dynamic code region, they are constructed as coarse-grain code segments rather than fine-grained resource locks. As a result, a single region may enclose a large number of locks/atomics. Because the user is completely unrestricted in placing region markers in a cyclic-dependence graph, it is possible to introduce deadlock due to mutually exclusion with two or more markers. Such deadlock scenarios is the cost of supporting highly-usable coarse-grained regions without requiring the user to track down every instance of critical resources inside the region and reason about deadlock. Instead, we provide a safety hatch in the form of deadlock detection and correction. By monitoring the forward progress of the each core, our scheduling algorithm detects conditions where region markers have entered into a deadlock scenario. We scan the set of active and pending regions and automatically boost the priority of the cores holding critical resources. This allows the cores to continue execution beyond our normal bounded time-window restrictions, allowing the region to complete, unblocking waiting cores.

5.4.3 A-temporal information flow

One oddity that exists in the simulation framework is the transmission of information backwards in time. This is a mere timing oddity rather than incorrect functional execution. State changes are seen in the order of functional execution, which is decoupled from timing. Thus in reconstructed timing, a functional core may see a

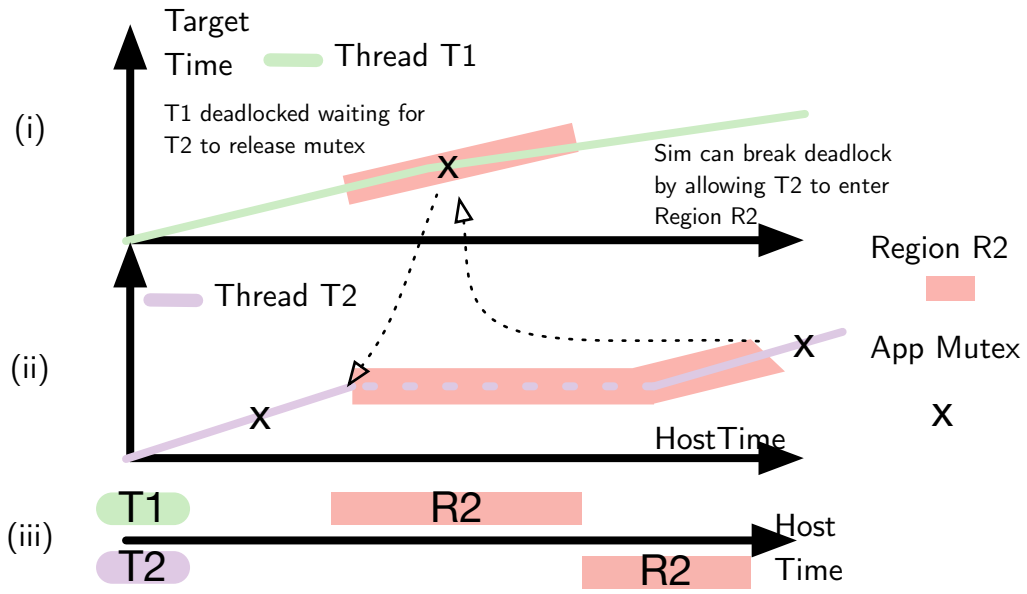

Figure 5.13: Deadlock due to interaction of scheduling and existing synchronization. Thread T1 (green) enters an existing mutex. Thread T2 (purple) enters a scheduling annotation based mutex which the Thread T1 will enter next. Inside the annotated section, thread T2 attempts to take the mutex owned by thread T1. Since the scheduling annotation precludes the thread T1 being scheduled until the thread T2 exists the region, a circular wait is created in an otherwise well synchronized execution.

write from another core (but never from itself) at its time on the reference clock before the other core wrote it, based on the other core's time on its reference clock. Such reorderings are due to two causes.

The first cause, which is avoidable, is the execution of blocks of functional instructions within a core. As one core advances, it may write to memory at reference cycle $n + x$. Another core is then simulated from reference cycle n, but it will see the write at $n + x$. This source is avoidable as the granularity of simulation can be shortened until the effect goes away. Further, by applying concepts for detecting FT entanglement and FT violations from optimistic FT simulation, it is possible to make this granularity dynamic, minimizing information flow when cores may be tightly communicating in this fashion.

The second cause is again due to blocking of functional instructions, but is not avoidable. Under some scheduling constraints such as ensuring mutually exclusive functional execution in a region, the simulator actively prevents one cores clock from updating when it is waiting for another thread to leave a mutual exclusion region. Changes in the mutual exclusion region by the first core are then seen by the second core when it is allowed to execute. This information flow is required to prevent the appearance of contention.

5.5 Prior Work in Virtual Time Simulation

In the domain of simulation, the notion of using virtual-time to account for differences between the progress of wall-clock vs simulated-clock time has been used in various simulators (Jefferson, 1985; Duda and Cheriton, 1999).

In particular, network simulators (Fall, 1999; Barr et al., 2004; Ryckbosch et al., 2012) leverage a form of time-dilation to compensate for the relative differences between the slowdown of simulation and the native speed of network and disk devices. This dilation is similar to our coarse-grained time compression and dilation, but does

not attempt to leverage the flexibility of dilation directly to the simulator-user.

In the area of concurrency bug finding, thread scheduling to manipulate thread interleaving has been an area of active research (Jannesari et al., 2009; Serebryany and Iskhodzhanov, 2009; Musuvathi et al., 2008; Flanagan and Freund, 2010). Such systems either control the operating system or the runtime of an application to artificially construct thread interleavings likely to expose bugs. Burnim et al (Elmas et al., 2013) expose a lightweight specification language to allow a user to declaratively constrain this interleaving correctness exploration, similar to our use of region specifications to explore the performance space of a parallel application. These works have focused on finding bugs rather than exploring optimizations however. Using a warped FT simulator as a common platform to express both testing and optimization may be a useful avenue of exploration in future work.

Warped simulation combines concepts of virtual-time with partitioned FT simulation and introduces ideas on how to map optimizations onto these concepts. This differs from prior approaches in that it controls FT interaction to realize its mechanisms generically, not just IO, process activation, etc. Further by bridging this generic FT scheduling mechanism to potential optimizations, the warped FT simulation approach becomes applicable as a simulator coordinator that can be bolted onto existing simulators. Finally, the warped FT approach proposes solutions to entanglement issues that arise from this warped partitioned execution (e.g. deadlock, livelock, atemporal information flow), given the stronger understanding of how F and T partitions interact when simulating a large parallel system.

The use of simulators for limit studies it not a new concept. Hardware simulators may offer knobs to directly manipulate performance and resources (e.g. ideal caches in a multiprocessor simulation). However, such knobs are restrictive as they are a property of the model itself, rather than as a generic simulation technique applicable across many models. While for any given optimization, we may modify

a simulator to mimic the particular performance characteristics of an optimization, such an effort may be so large to defeat the purpose. Transferring engineering work from implementing the optimization to modeling it.

5.6 Summary

Warped FT simulators are introduced as a new type of simulator capable of allowing direct manipulation of timing properties of a system in a way to allow exploring optimizations. By transforming the problem of evaluating a potential optimization into a set of scheduling and interaction constraints in a FT simulator, a warped FT simulator can be realized from a standard FT partitioned simulator. Using scoped warp regions to selectively apply constrained scheduling allows propagation of local effects of timing manipulation, allowing any potential FT entanglement to occur naturally as the simulation continues. Finally, a set of safety mechanisms are described to preserve correctness and causality int he face of user-level scheduling constraints.

In Chapter 6 a concrete realization of this generic warped FT simulator architecture is implemented and evaluated to further address Thesis Challenge 2 to improve simulator flexibility when optimizing parallel systems.

Chapter 6

MPSIGHT: Flexible FT Warped Simulator

6.1 Introduction

The previous chapter described how an warped simulator provides an approach for exploring optimizations rather than implementing them. A warped simulator can address our second challenge of this thesis, enabling high-flexibility simulation for parallel systems optimizations. Translating the abstract simulator architecture introduced in Chapter 5 into a practical simulation tool is the subject of this chapter.

While warped simulation can be used at any stage of the design life-cycle, many of the flexibility benefits have most impact early in the design phase. Focusing on early stage exploration sets the goals for the amount of effort, accuracy and detail a practical warped simulator must have.

Targeting large optimization wins makes it possible to explore optimizations with limited set of primitives. That is, coarse-grained with inherent accuracy limitations which matches the imperfect understanding of system under design. As a full understanding of system-dynamics and tradeoffs for a potential optimization is

often unavailable during early exploration, it is impractical to require user to provide detailed timing models. Further, we strongly try to preserve existing SW functionality, by focusing on explore optimizations that preserve functionality locally. This doesn't require user to provide detailed functional models / alternative implementations.

6.1.1 Design Principles

Given this focus on large optimization wins, we can constrain the design of a practical warped simulator implementation with an eye towards improving simulation flexibility in these early-stage design explorations where larger optimization wins are possible.

One basic principle we must follow is limiting the set of primitives and to specify warping. The previously described set of three verbs to control ordering, temporal and locality warping are used to explore a variety of optimization and techniques. While richer primitives are possible, restricting to them limits the amount of design effort the user must perform to describe a given optimization.

The second basic principle followed is minimize changes to the application functionality, ideally making no changes to the target code. The basic goal of the approach is to reduce design effort per hypothesis. If a user is forced to deeply modify or model new functionality, the goal of flexibility is not being tackled. For this reason, we try to use simple markers on existing functionality rather than trying to model new functionality.

6.2 System Architecture

6.2.1 Overview

To evaluate the accuracy and flexibility warped simulation, a prototype simulator was created. As the requirements for warped simulation differ from the goals of the FASTMP prototype, an alternative approach was used to implement this form of

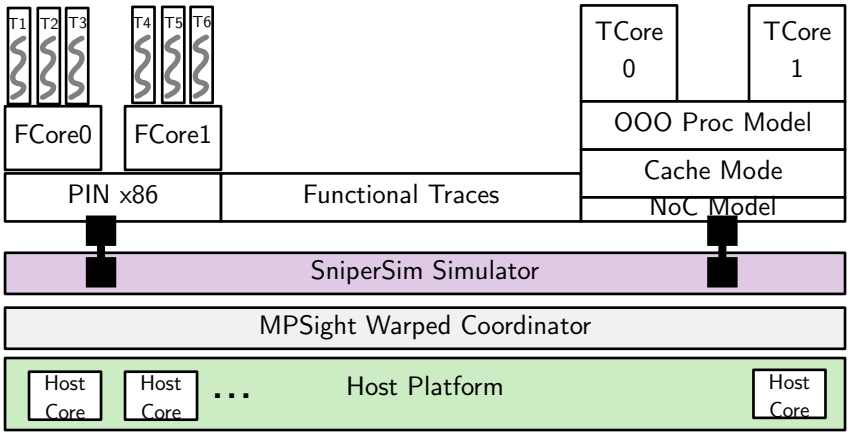

Figure 6.1: MPSight Simulator System Architecture

simulation. Rather than creating an simulator from scratch, the warped simulation approach was modified to work with an existing simulator.

As a warped simulator is principally concerned with exposing simulation primitives to control how FT partitioned processes interact, it is separate the core warped simulation kernel from a underlying simulation library. By creating an external simulation coordinator concentrating solely on these activities its possible to relegating all functional and timing modeling to a existing simulation framework.

Figure6.1 provides a high-level diagram of the prototype warped simulator. The coordinator is responsible for tracking the execution of the various logically partitioned processes running, and managing the sets of regions the processes enter and exit. The coordinator also maintains control over the relative execution of the partitioned processes, allowing the user to directly manipulate the execution of the simulation.

We use an off-the-shelf cycle-level simulator, Snipersim (Heirman et al., 2012), to act as the base simulator. The base simulator is organized as a Functional-First simulator that is parallelized across multiple host-cores and uses binary-translation to implement its functional partition for higher simulation efficiency.

6.2.2 Markers and Warping Specifications

A key aspect of warped simulation is to provide user ability to limit scope of a warp transformation to a subset of application/thread execution. Region warping must be scoped not only to be local to a given FCore, TCore, but must also be scoped to some fraction of the application code itself. To identify regions, the user introduces annotation markers that distinguish where a region starts and stops. For simplicity of the our prototype, we use an explicit pair of START and STOP markers that are inserted directly into the code being optimized. Region markers are mapped to specialized NOP instructions that are interpreted specially by the coordinator.

When the coordinator observes a START marker, it may activate a new warping policy that is used to control partition scheduling decisions until the STOP marker is hit. For each region, the simulator supports a configurable specification that controls how each region should be executed and scheduled by the coordinator.

In addition to region markers, the user also provides a warping specification that describes a particular optimization. Figure 6.2 depicts an example specification describing a potential contention reduction optimization. In the example, region $R1$ specifies an ordering warp to ensure mutual exclusion when it executes the region. With markers placed at the entry and exit to the data-structure being studied, the warping specification for $R1$ will activate.

The warping specification also allows a user to specify a probability for how often the warp scheduling constraint should apply. For example, $R1$ is specified to only apply the constraint 40%, to match the hypothesis of what the user thought was practically achievable by changing the data-structure being studied. Parameterized specifications enable a simple mechanism of controlling these warp dimension to cover cases where an optimization should only apply some of the time.

```
[projection]
num_regions = 1

[projection/regions/r1]
name        = NodeUpdate
region_id   = 1
resource_id = 1
```

```
[projection/regions/r1/ordering]
mode        = mutually-exclusive
arg         = 1.0
prob        = 0.4
[projection/regions/r1/temporal]
mode        = normal
arg         = 1.0
prob        = 1.0
[projection/regions/r1/locality]
mode        = normal
arg         = 1.0
prob        = 1.0
```

Figure 6.2: Example of defining warp specifications to describe parallel software optimizations

6.2.3 Partition Scheduling

To constrain the simulation to a warping spec, the coordinator handles the job of orchestrating how and when each partition should execute relative to each other. A target application may be composed of one or more threads, each executing on a combination of FCores and TCore. The coordinator doesn't need to be aware of the application threads, only needing to track FCore execution. As a functional core may enter a region, and then get descheduled from execution, maintaining a separation between FCores and TCores preserves the ability to specify F/F ordered warping independent of how an OS may schedule application threads to cores.

6.2.4 Region Management

While the coordinator actively selects which FCores and TCores can legally continue execution, a means of tracking what regions each FCore/TCore is occupying is necessary. To implement tracking and management of active warping specifications, a directory is maintained in the coordinator. The coordinator tracks both the warping specifications provided by the user, as well as the active regions currently observed as and when START and STOP markers are observed.

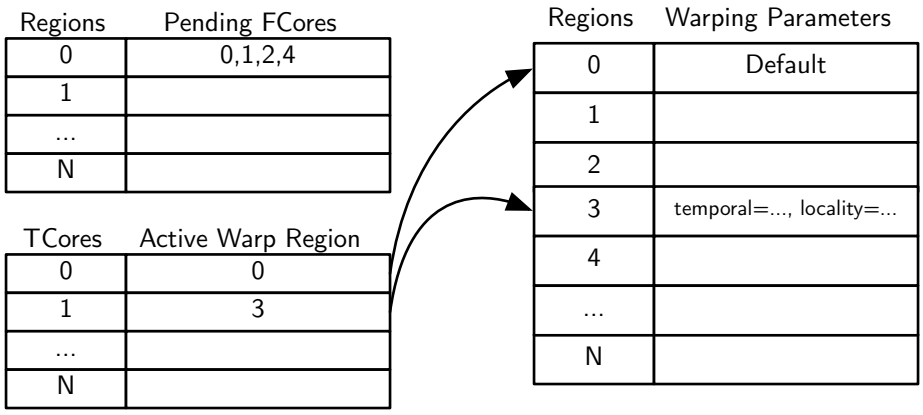

Figure 6.3: Region directory tracks active warping regions and pending functional cores

Each region maintains a pointer to the FCore that is currently executing the region as well as a pending list of FCores waiting to enter the region (e.g.. due to hitting the region with a mutually-exclusive scheduling constraint being active). When a FCore enters a region with a mutually-exclusive constraint and the region is currently being executed by another FCore, the FCore puts itself onto the pending list and yields waiting for the region to be empty. When the previous FCore leaves the region, it wakes up the next pending FCore if there are any.

Similarly, when the START marker is simulated by a TCore, it looks up the region properties that apply to the region. The warping spec for a region may indicate how fast simulated time should progress, or how functional traces should be scheduled and consumed across the partition boundary.

6.2.5 Partition Synchronization

Mitigating Potential Deadlocks

The region directory provides centralized tracking across regions, FCores, TCores and provides a global view of how the application is traversing through warped regions. This conveniently offers a practical approach to leveraging the region directory to

handle possible livelock and deadlock concerns. By monitoring the directory for forward progress or conflicted regions, we can invoke a fallback scheduler to support resolution of livelock/deadlock scenarios. As discussed in previous chapter, warped simulation can suffer from livelock (due to priority inversion) or deadlock (due to region locking) simply by virtue of how the user specifies regions to be warped. In practice, this fallback schedule breaking has not been necessary but provides a measure of generality and forward progress when the user is actively exploring optimizations with complex applications.

The coordinator tracks FCore and TCore progress during simulation. If a given partitioned process stops making forward progress, the particular region or set of active TCores is scanned to see if a scheduling breaking mitigation is required. When a scheduling mitigation is activated for a given set of FCores and TCores, they are free to violate the restricted schedule until they exit the warped region. This simple fallback mechanism provides safety without needing additional state tracking not already offered by the region directory itself.

Maintaining Virtual Time under Warping Constraints

As each TCore maintains its own view of warped target time, keeping the various TCores in sync with each other is handled by bounded synchronization across TCores by the coordinator. As each TCore might be paused waiting for its FCore to gain access to a region marked as mutually-exclusive, TCores must be able to tolerate some slip between their respective target times.

A bounded synchronization scheme is used to enforce limited slack between TCores. This ensures coarse-grained F-T entanglement is preserved by preventing TCores from running too far into future (thereby potentially ignoring global effects induced by local warping in other cores). Figure 6.4 depicts and example of this bounded synchronization scheme.

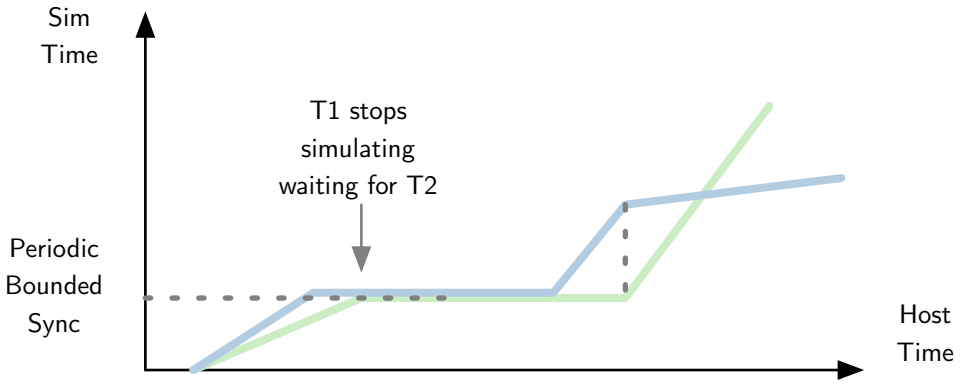

Figure 6.4: Example of bounded sync interactions with temporal warping

6.3 Results

6.3.1 Methodology

To address the second challenge of the thesis, high-flexibility simulation for parallel systems, a warped simulator must be able estimate which optimizations are most beneficial, *before* the optimization is written. We evaluate the MPSight prototype, looking at its behavior in terms of both useful levels of accuracy and design-effort. On the accuracy front, we examine if it is possible to use warp specifications to identify which optimizations may be most beneficial to implement. On the design-effort front, we examine how much effort is required to both instrument the application with warp annotations and write a warp specification.

To study interesting optimizations that require new data-structures and contention aware algorithms, we examine the Galois parallel framework (Galois, 2011). The framework and its accompanying benchmark suite are targeted towards high-performance parallel applications that may have irregular amounts of parallel work and dynamic data dependencies. The framework conceives of a parallel application in terms of operators, schedules and parallel data structures. To reduce programmer effort, a collection of worklists, resource allocators and domain specific parallel data-

structures are provided as part of the core framework. Using the framework as a starting point, it becomes possible to identify both interesting optimizations and applications to evaluate in a MPSight optimization study.

6.3.2 Optimization Details

Developing and refining new optimizations can be a lengthy process often requiring deep domain knowledge and significant trial and error experimentation. We leverage the source repository from the Galois framework as a fertile ground for finding such optimizations that includes a few man-years optimizing and exploring parallel patterns, framework optimizations and scalable data-structures.

The Galois source repository was mined for suitable candidates that could be treated as optimizations that could be evaluated as a 'what-if' style experiment. As warped simulation only permits warping functional ordering and timing properties, optimizations were restricted to be functionally identical. Further, to ensure a common baseline for all optimizations, optimizations were selected so they could be applied to a single baseline software version.

As the Galois system is organized as a common framework with shared code, not all optimizations found in source history are applicable to all applications using the framework equally. However, as discussed in Chapter 5, once an optimization is implemented, it is rarely removed from the repository even if its purpose is superseded by other optimizations.

A set of six candidate optimizations were identified (with feedback from Galois framework developers) covering a range of scalability optimizations to the framework.

- scalable memory allocation (2 optimizations)
- work item distribution (1 optimization)
- handling data conflicts and exceptions (2 optimizations)

- scalable data-structure containers (1 optimization)

Each of the optimizations ranged from 10s of lines to 100s of lines depending on how support infrastructure and refactoring costs are included in the line count and required multiple days of experimentation and implementation (along with multiple failed approaches that were never committed to the repository).

The details of each optimization are described in further detail below:

- *mmap* Replace default glibc malloc with an mmap allocation. Galois uses a variety of pool-based allocators but are backed by some system allocator. Replacing with mmap reduces contention when default malloc incurs additional contention when refilling the pools.

- *periter* Replace a general purpose alloc/free with a small allocator specialized for allocation/free of temporary memory while a work iteration is actively processing. The allocation uses a simple bump pointer to maintain its memory pool as memory is freed once the iteration completes. This reduces contention pressure on the general bulk framework allocator as well as reduces the overall cost of each alloc/free due to simpler management.

- *worklist* Replace a centralized chunk-based work-queue with a distributed chunk-based work-queue with neighbor stealing. This reduces the contention on spinlock protecting the queue and replaces it with both local locking and stealing when work available chunks are exhausted.

- *graphbag* Replace a single linked-list of nodes with one list per CPU. Reduces contention on the central linked-list lock while reducing memory allocation that was maintained in the single bag implementation.

- *abort* Replace the handling of aborted work items from a centralized master to a distributed queue. This reduces contention when moving aborted work items into a master worklist when work items encounter conflicts.

- *exceptions* Replace a low-level lookup in glibc that is triggered during exception handling when new library symbols are checked for updates. The optimization replaces the blocking symbol search with one memoized per-core search, reducing both contention and processing time.

6.3.3 Application/Algorithm Details

Given the optimizations to be studied, a set of applications using the framework was collected from the Galois Lonestar benchmark suite. The benchmarks were modified to allow using both the optimized and deoptimized framework implementations so each application could be evaluated with each optimization. As the Galois framework targets irregular algorithms where dynamic work and data dependencies exist, the applications all use a variety of worklists, graph data processing and dynamic locking and conflict handling.

- *dmr* - Delaunay Mesh Refinement Transforms a mesh described by a set of points in a plane by successively triangulation of the points. Each triangle is constrained to satisfy the Delaunay condition such that no angle in the mesh be less than 30 degrees. The benchmark repeatedly fixes a *bad* triangle by adding new points to the mesh and re-triangulating. The refinement process may create additional bad triangles surrounding it but will converge to a mesh that meets the quality restriction. Refinement can be processed in parallel provided the triangles actively being refined do not overlap.

- *dt* Delaunay Triangulation Triangulates a set of points in a plane such that each triangle in the mesh does not contain any other points in the mesh. The benchmark processes the points finding the nearest triangle currently in the mesh and re-triangulating the cavity created by processing the new point. Processing can occur in parallel with conflicts when cavities overlap.

- *sssp* Single-Source Shortest Path Traverses a directed graph from a provided source node, computing the shortest path to all other nodes in the graph. Edges hold non-negative weights, while nodes are initialized with an estimated distance to the source node. As the benchmark progresses, the estimated distance is recomputed on each node, decreasing from infinite distance to the actual distance. Updates to a nodes distance value can cause other distances to be recomputed. Processing of nodes can occur in parallel but updates to estimated distances to the same node can create conflicts.

- *kmeans* K-Means Clustering Partitions and clusters points from an N-dimensional space into K-clusters. Points are partitioned such that points within a cluster are closely related. Clusters are initialized as empty with their centers placed in the N-dimensional space. Iteratively, points are evaluated to see which cluster they are closest to, then cluster centers are reevaluated. Processing points can occur in parallel but conflicts can occur when cluster centers are updated.

6.3.4 Representing Optimizations as Warping Annotations

In consultation with Galois developers, the selected optimizations were reflected as warping annotations. Markers were added to the deoptimized code pathways and warping specs setup to warp a particular region with ordering, temporal and locality warps. The selected choices reflected the understanding of the developers on how the performance of the optimized code should improve the framework.

- *mmap* To mimic the performance characteristic of the optimized mmap-based page allocator, markers are inserted into the original version of the *pageAlloc* function which is just a wrapper around *malloc*, creating a new warped region. A matching warping specification is used to mark the region with an ordering warp ensuring mutual exclusion when executing the region. As the primary benefit of using the mmap-based page allocator optimization is to reduce the

possibility of hitting a contended futex within the *malloc* call to near zero, warping the functional order to ensure mutual exclusion is used to project the benefits of this optimization.

- *periter* For the *periter* optimization, warping markers are inserted into the Galois pool allocators around operations of *allocate* and *deallocate*. As the performance benefit of applying the periter optimization is to both reduce contention and reduce time spent in managing short-lived allocations within single parallel work iteration, both ordering and temporal warp specification are used. The region is marked as mutually exclusive with temporal warping reducing the execution time down to zero cost.

- *worklist* To estimate the *worklist* optimization, the *pop* chunk locking operation is instrumented in the default WorkList object provided by the Galois framework. As the optimization reduces contention on the centralized worklist when chunks are exhausted, a ordering warp marking the region as mutually exclusive and a locality warp marking the locking with no invalidation costs is used (mimicking the impact of transforming the list to nearly all local).

- *graphbag* For the *graphbag* optimization, the *push* operation has markers inserted to create a function when adding elements to the Bag. The optimization converts to a per-CPU linked list and further reduces compute costs on insertion using simple bump pointers. To mimic the performance of this optimization the region is warped with ordering set to mutually exclusive and temporal set to zero cost.

- *abort* The *abort* optimization is annotated by inserting markers when invoking the *handleAbort* function when inserting conflicting work. A ordering warp is applied to the region with mutual-exclusion and a locality warp to mark the acquire and insertion of work with no invalidations.

Target Machine Configuration	
Core	16-core, Nehalem i7 Cores, 2.2Ghz, dual-socket, 8cores per socket
L1-Cache	D-32K - 8way, I-32K - 4way
L2-Cache	256K - 8way
L3U-Cache	16MB - 24way
Application Properties	
galois	2.2.0 + deoptimization patches
gcc	4.7.2, x64, -O3 optimization
glibc	libstdc++-6.0.17
Test Input Properties	
dmr	250k mesh with 50% bad triangles
dt	500k mesh with 50% bad triangles
sssp	r4-2e20 random graph
kmeans	random 2k points, 16 dimensions, 16 clusters

Table 6.1: Parameters used for Galois optimization warping study

- *exceptions* The *exceptions* optimization is annotated not in the Galois source directly, but instead in the glibc used when running Galois applications. The C++ exception handling function, $unwind_find$ is instrumented with markers around the symbol lookup. Warping specs for an ordering warp indicating mutually exclusion and temporal warp indicating zero cost are associated with the region to capture the benefit of maintaining per-cpu cache when resolving objects in the C++ stack unwinding which is almost entirely static for the duration of the application.

6.3.5 Experimental Setup

MPSight is setup with to model a modern large multi-core processor and is provided with a warping specification as as well as warping specifications that describe the various optimizations selected for the study. (provided to warping coordinator). Table 6.1 describes the configuration parameters for the study.

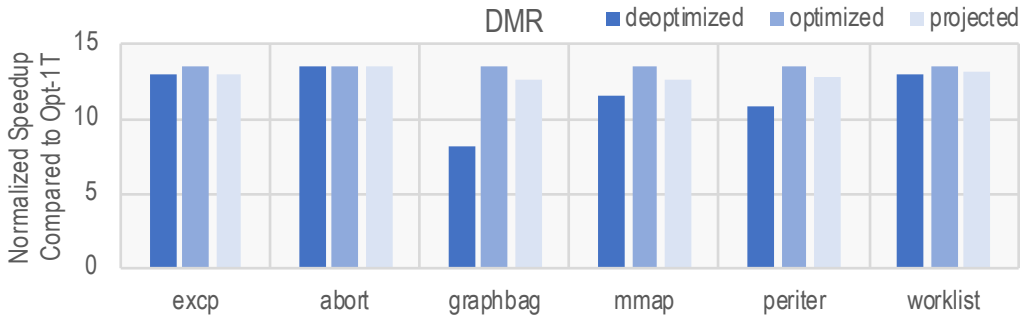

Figure 6.5: Comparison of Speedup for both Projected Simulation and Oracle Implementations for DMR mesh refinement benchmark

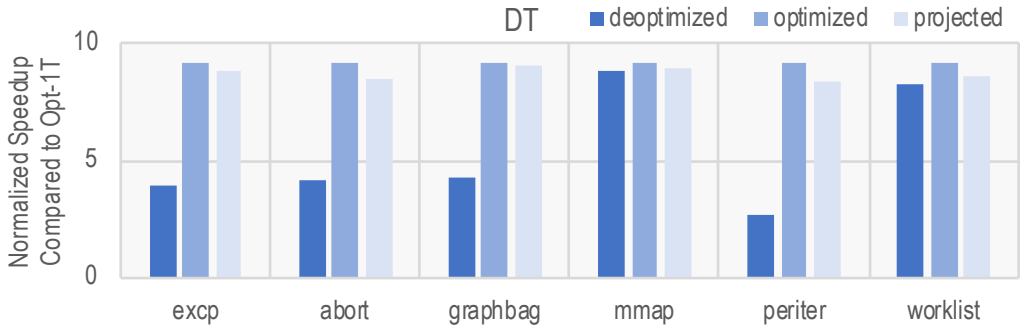

Figure 6.6: Comparison of Speedup for both Projected Simulation and Oracle Implementations for DT mesh refinement benchmark

6.3.6 Study: Galois Optimization Performance Prediction

Using the framework optimizations, application and warping annotations as specified above, we conduct a study to evaluate the warped simulation approach and its ability to improve the flexibility in parallel software tuning. From the fully optimized framework, measure the application parallel scaling compared against a one-core execution. This serves as the *optimized* baseline. A series of deoptimized runs are then evaluated, corresponding to reverting the framework back without a particular optimization, resulting in a set of *deoptimized* runs. Finally, the deoptimized binaries are run with warping annotations inserted and warping specs configured, leading to the *projected* runs.

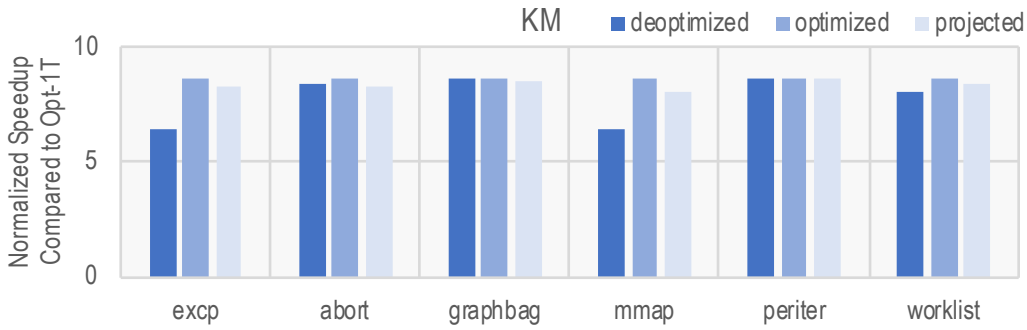

Figure 6.7: Comparison of Speedup for both Projected Simulation and Oracle Implementations of various optimizations for kmeans clustering benchmark

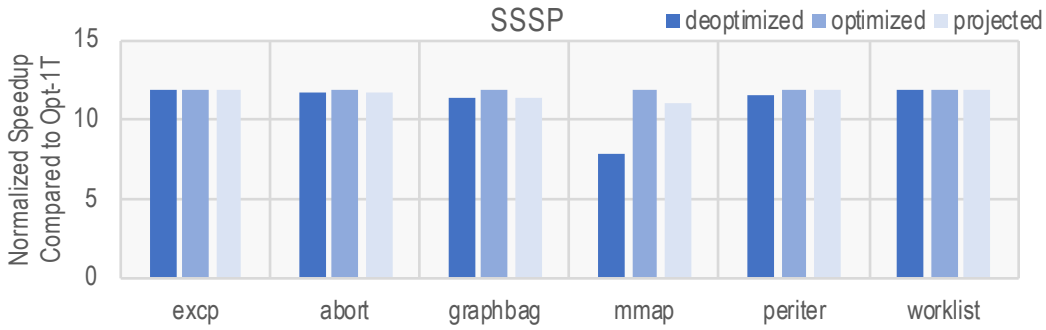

Figure 6.8: Comparison of Speedup for both Projected Simulation and Oracle Implementations of various optimizations for single shortest path graph traversal benchmark

Optimization	Scaling Improvement Factor	Avg. Projection Error (%)	Impacted Apps
mmap	1.15	6.57	sssp, dmr
periter	2.10	5.27	dmr, dt, kmeans
worklist	1.05	4.66	dt
graphbag	1.54	3.33	dmr, dt, kmeans
abort	1.61	6.81	dt, kmeans
exceptions	1.65	7.17	dt, kmeans

Table 6.2: Warped projection of Galois framework optimizations

From the results, we notice that in general warped simulation is capable of capturing the impact of removing the bottlenecks for each optimization when it a hurdle for performance scaling. For example, in Figure 6.5, the graphbag projection indicates that it can greatly improve the scaling of the application. The projection incurs some error (7% absolute error), but the improvement of the optimization is nearly 1.6x so the error is tolerable. By contrast the mmap optimization has a much smaller benefit of only 1.1x with a similar error. Across the various applications and optimizations, we observe a similar pattern.

From a design effort perspective, after the deoptimization study was setup by creating a single baseline and enabling the deoptimized variations of data-structures and algorithms, the effort to insert annotate the six optimizations was modest. For each optimization, two to four markers were inserted into the source code, while another fifteen lines of warping spec configuration files were created for each optimization. In contrast, the accumulated changes necessary to support a single optimization ranged from 10s to 100s of lines of code and spanned days to weeks of experimentation before the final optimization was added to the framework. From this perspective, the MPSight simulator offers a marked improvement in flexibility over existing profiler-centric tools.

6.3.7 Contrasting Benefits of Warped Simulation Exploration

MPSight and its underlying warped simulation approach offers a unique method of exploring parallel optimizations. Table 6.2 summarizes the accuracy results across the range of optimizations and applications studied. For estimating the benefits of optimizations that deliver large scaling improvements, it is possible to use the simulator with good accuracy and be able to rank optimizations that are valuable to implement. For optimizations that may lead to smaller benefits, warped simulation can still be useful but may not be able to precisely rank which optimizations due to the much larger relative error.

The simulator also can find which optimizations have the biggest benefits for a given subset of applications. For example, we observe that three of the four applications saw significant scaling benefits when using the graphbag optimization, while others such as the worklist optimization did not offer as large benefits across the applications. This allows a user to focus their implementation efforts on a smaller targeted set of optimizations without having to implement each one.

From a development effort, instrumenting the applications and writing a warped specification was lightweight enough that it can be used when evaluating optimizations that may require hundreds of lines of code to implement otherwise. The simulator does have limitations in the types of optimizations it can explore as it is limited to optimizations that are functionally equivalent to the existing code with only performance or resource characteristics differing.

6.3.8 Summary

Using warped simulation as prototyped by the MPSight simulator makes it possible to evaluate potential optimizations before implementing and measuring each one. Using warped annotations consisting of markers and warping specs, interesting optimizations in irregular parallel applications can be evaluated.

We show that is approach is capable of finding which optimizations are valuable and show off differences between optimizations and applications. For estimating and ranking optimizations that may lead to large scaling improvements, the approach can rank optimizations within the range of error.

The prototype simulator offers new capabilities not presently supported in current tools With the ability to measure before implementing for irregular parallel applications, users can explore hypotheses that would previously require explicit implementation of new SW or extremely detailed modeling.

6.4 Prior Work in Parallel Exploration

Given the wide variety of tools and approaches available to both port existing serial code as well as optimize parallel code, a few broad categories that target prior work in the parallel exploration space can be created.

Serial conversion estimators such as Kremlin (Garcia et al., 2012), GProphet (Kim et al., 2012) and Parallel-Advisor (IntelParaAdv, 2017) provide tooling to help estimate the benefits of converting serial code to parallel code. These tools operate on the level of tasks (typically dense regular loops). and may contain domain-specific models. For example, Parallel-Advisor provides a model for estimating lock contention using different analytic models for different language frameworks. GProphet extends this basic approach and adds a LLC communication and contention analytic model to measure the saturation of scaling due to memory pressure. In general, the requirement to model behavior analytically reduces generality and makes it susceptible from changes in hardware designs which may invalidate the model itself.

Locality analyzers such as Cachegrind (CacheGrind, 2014), Accumem (Rogue-Wave, 2014) provide instrumentation driven approaches to measure cache locality. Such tools can offer insight into which lines of code or which data-structures are problematic in shared caches, but do little to estimating how much scalability improvement

might be gained by improving locality in some data-structure.

Native machine profilers such as Linux perf, Intel VTune, AMD CodeAnalyst all offer performance monitoring and back-annotation to code. The tools provide a interface to a rich set of counters available on-chip for performance monitoring. By operating at the level of instructions and microarchitectural events (e.g.. pipeline-stalls, cache-misses, etc) along with attribution based on call-stack sampling, the tools can indicate how the code on the current hardware is exactly performing. The downside of these tools are one is often swamped with data and identifying what the most valuable optimization is can be challenging.

Framework-based task analysers and tracers such as MPI Jumpshot, and Cilk CilkView provide domain-specific performance models that target a specific style of application. By instrument framework-specific operations to collect detailed information about potential points of contention, these tools can provide useful information about bottlenecks.

Finally bottleneck decision engines such as HPCtoolkit (Tallent et al., 2008), Paradyn Performance Consultant (Paradyn, 2007), and PerfExpert (PerfExpert, 2014), try to capture best-practices and optimization patterns for recommendation. These tools are closest in goals to our own and falls into a category of bottleneck and resolution engines that attempt to use a mix of dynamic and static analysis to determine where bottlenecks exists. Our work is complementary to these approaches as 'whatif' projections may be simply used to evaluate the benefits of composing proposed optimizations by these engines. A warped simulation approach extracts more information than the normal unmodified dynamic runs used in these approaches as it can answer the 'how much' instead of just the 'where', 'when'.

The various tooling approaches of measurement, profiling, tracing have their own unique advantages and disadvantages. In general, these tools attempt to simplify the presentation of information to the user so they may construct their own mental-

models or hypothesis and present a model-based approach to detecting bottlenecks or parallelization opportunities. The disadvantage of this model-based approach is that it is restricted to well understood framework operations. For example estimating the speedup of parallelizing a sequential for-loop under an specific OpenMP cost model. However, such approaches are hampered by the inability to work with arbitrary parallel code or cope with the system-level interactions that may arise as a consequence of optimizing some unrelated part of the application.

6.5 Summary

6.5.1 Contribution Summary

The warped FT simulation approach and the MPSight tool offer significant benefits over the state of the art. In addressing the second thesis challenge, high-flexibility simulation, using MPSight to explore complex optimizations is both low design-effort and effective.

Prior approaches using profiling or auto-tuning require design-effort to implement an optimization before it can be evaluated. The key result demonstrated by the MPSight prototype is that it is possible to simulate the optimizations that reduce contention and increase application efficiency by warping the simulation to remove the bottlenecks from the application. The design effort required for modeling an optimization only requires the insertion of NOP markers in the application and specifying what amounts of contention, timing or locality reductions would be effected if an optimization was fully implemented.

A warped FT simulator offers a number of capability that are not possible in other simulators or tools. The fundamental new capability is the ability to control simulation scheduling in a way that allows optimization code changes to be measured before they are implemented. While prior parallel testing tools allow exploration

of thread interleaving to find races and architectural simulators have allowed limit study parameters to be used, no tool provides these two features in a single simulator. This results in offering new functionality to model complex irregular optimizations that would be difficult to evaluate without implementing the optimization.

6.5.2 Reducing Prototype to Practice

While the MPSight prototype offers some significant benefits and capabilities over other state-of-the-art parallel performance tuning tools, it does have some limitations that must be addressed to become a mainstream tool. First, matching the speed and overall usability of profiling-based approaches should be addressed. Second, a more integrated method of finding and marking regions should be explored. We briefly discuss approaches to solving both of these issues.

First, as MPSight was developed as a proof-of-concept to evaluate the warped simulation approach of using FT partition warping, it incurs the same tradeoffs made by all simulators (speed, cost, accuracy). The SniperSim base simulator was extended for this purpose, but using a cycle-level simulator has some usability challenges when compared to native profiler-based approaches. Specifically, while SniperSim, a Functional-First simulator that trades some accuracy for its own performance is faster than extremely detailed timing-directed simulators, it is still many orders of magnitude slower than native execution.

To integrate with software development expectations, a faster baseline execution environment is necessary. One approach would be to leverage the learnings of the FASTMP work presented. Using hardware acceleration and extending FASTMP to make it suitable for not just extreme-detail but capable of simulating at near-native speeds with some tradeoffs in accuracy. Another approach would be use native-execution and directly embed the warped coordinator within an application. By treating natively executing threads as part of the functional partition and maintaining

a virtual time per processor, the warped coordinator could enforce the same warping specifications against natively executing applications. This shares some similarities with host-compiled timing simulation (Gerstlauer, 2010) but extended to a parallel execution environment with the user specifying parts of the simulation constraints.

Second, finding and marking regions should be integrated into the existing profiling ecosystem rather than relying on the user to manually instrument their code. While the prototype used a simple approach of forcing the user manually insert NOP instruction markers, this can be exercise in tracing through multiple levels of code, C++ templates and macros before the appropriate points can be discovered. Profiling tools such as gprof are already integrated with compiler-driven instrumentation that could be leveraged for denoting region markers. This would make warped simulation a natural extension to profiling experiments. A user could then profile with instrumentation, then use the same binary to evaluate an potential optimization with the same instrumented binary with a particular set of warping specs.

6.5.3 Conclusion

The MPSight simulator uses the warped FT simulation architecture to answer *what if* optimization questions. These mechanisms are temporal, ordering and locality warping. There are other mechanisms which could be implemented to expand the range of questions the tool can address.

The tool that can be used to estimate the performance of code optimizations, without requiring those optimizations to be implemented. The tool demonstrates less than 10% error across a range of interesting irregular parallel benchmarks and optimizations with low-levels of design effort.

Chapter 7

Summary and Future Work

7.1 Revisiting the Challenges

The previous chapters describe attempts to leverage logical partitioning of simulators for parallel systems to improve their overall design. By leveraging properties of logical partitioning along a functional-timing boundary, we show how improvements to parallelism and simulator flexibility can be achieved.

Chapter 3 and 4 describes a method to parallelize simulation along a logical functional-timing partition boundaries. We show how the parallelization can be made safe and produce the same results as a serial execution, while preserving additional levels of parallelism and opportunities to leverage hardware acceleration in a complexity effective manner. Using an FPGA-accelerated prototype, we demonstrate a simulator competitive with other hardware-only simulations with far smaller resource requirements, large speedups over pure software simulators while not sacrificing fundamental correctness or model generality simplifications.

Chapter 5 and 6 describes a method to improve simulator flexibility by enabling sketching optimizations in parallel systems optimizations using new controls offered by controlling how and when each functional or timing partition executes. Exploring

optimizations using this approach is shown to be sufficiently accurate for high-level exploration and can significantly decrease the cost of experimenting with complex optimization and acceleration design choices.

With the combination of both optimistic and warped FT simulation, we show how leveraging logical simulator partitioning can improve the overall design of simulators for parallel systems in terms of both performance and flexibility.

7.2 Contribution Summary

The results from this work builds on prior simulator approaches in functional-timing partitioned simulation, and makes several contributions that are relevant to both simulator designs and parallel systems architects.

In the high-speed simulation arena, prior simulation tools were forced to trade model simplifications to gain CPU efficiency or limit model sizes or complexity when using hardware-based simulators with fixed resource capacities. FASTMP demonstrates approach of using Optimistic-FT parallelization to safely allow mixing SW and custom HW execution even with complex fine-grain interactions between partitions without compromising on execution correctness. With approach described in this work, simulator designers are now offered ability to create high-speed simulators that can use modest amounts of dedicated hardware acceleration without facing some of the prior issue with resource constraints or requiring modeling inaccuracies when coupling SW execution at high performance.

Secondly, the Warped-FT simulation approach described in this work provides a more direct method of experimenting with optimizations in parallel system that previously required significant effort to evaluate each optimization. Prior tools could offer profile-guidance of bottlenecks, but couldn't easily measure the impact (with all side-effects) of removing the bottleneck. A Warped-FT simulator offers a new set of experimentation capabilities that allow measuring what-if scenarios in complex

parallel applications that is not available in existing tools.

7.3 Future Work

There are many avenues for future work, spanning the abstract as well as concrete engineering domains Simulator utility is often not a competition solely on single axis, often just choose a simulator based on ecosystem, documentation.

7.3.1 Optimistic FT Research Directions

In the optimistic FT arena, extending and automating the transformation of an FT partitioned description into a optimistic FT execution would allow practically using for domain specific processors and more general descriptions of functional/timing models (including non-processor centric accelerators). Leveraging high-bandwidth / medium latency available in rack-scale FPGAs and appliances can provide lightweight RTL emulators if can further work in automating transformation from netlist is refined. Further, leveraging hard cores (processors with trace-buffers, customized processors designed for trace generation and replay) provides further gains in efficiency, leveraging an island of hard cores surrounding tightly coupled sea of reconfigurable logic.

7.3.2 Warped FT Research Directions

In warped FT arena, two primary avenues of research have been open by the concepts presented in this work, both abstractly in terms of abstract simulator architectures, as well as concretely in terms of practical tools to cope with coming changes in parallelism and heterogeneity. In the abstract simulator front, further refinement of partition scheduling and F/T trace interaction could allow extending the scope of optimizations to include allow work item rescheduling. It may be possible to capture alternative work item schedules by executing then trace reordering in a generic

fashion without being tied to specific application. On the engineering front, refining the implementation to work against native execution as a continuous performance guidance mechanism (e.g. background 'intellisense' predictive exploration). Further simplifications of the simulation approach (sampling/hybrid execution) may provide an alternative means of preserving extreme control over partition scheduling without incurring the large slowdowns. Alternatively combining the hardware acceleration approaches discussed in optimistic FT for warped FT simulator might allow with both high speed and high flexibility.

7.4 Conclusion

As Moore's law continues to offer increasing reduced benefits, novel techniques in both software and hardware will be necessary to continue driving performance and efficiency wins. Shrinking time to market pushes hardware design but the increasing complexity in verifying and implementing designs limits what can practically be achieved to try and continue pushing performance forward.

The two approaches introduced in this work on how to design simulation tools offer an avenue to explore and tackle this new landscape.

Appendix A

Simulation Concepts

A.1 Overview

As computer simulation has become an indispensable tool in the design of next generation digital systems, there has been significant work in understanding how to optimize and build such simulations. This chapter provides a background primer on computer simulation, and illustrates some of the tradeoffs involved when selecting a given simulation approach. We finish with a survey of contemporary and state-of-the-art approaches.

A.1.1 Terminology

In this work, we use the term digital system *simulation* to describe the process of imitating the properties of another computer on a given system. These properties can simply be the behavior of a parallel application to the detailed performance of a hardware arithmetic unit under study. To perform a *simulation*, a concrete program or system is used, a *simulator*.

Given the desire to simulate the properties of system that exists or could potentially exist, we use the term *target* to refer to the system we would like to study.

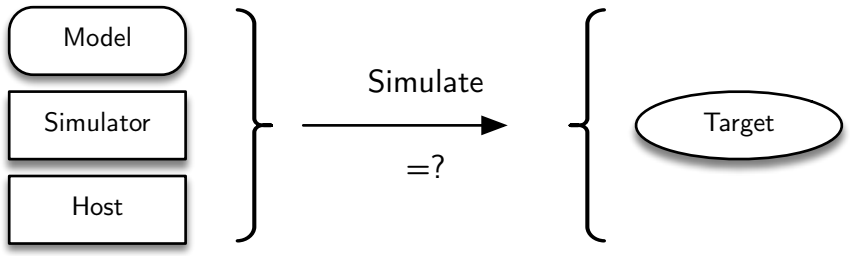

Figure A.1: Simulation Terminology

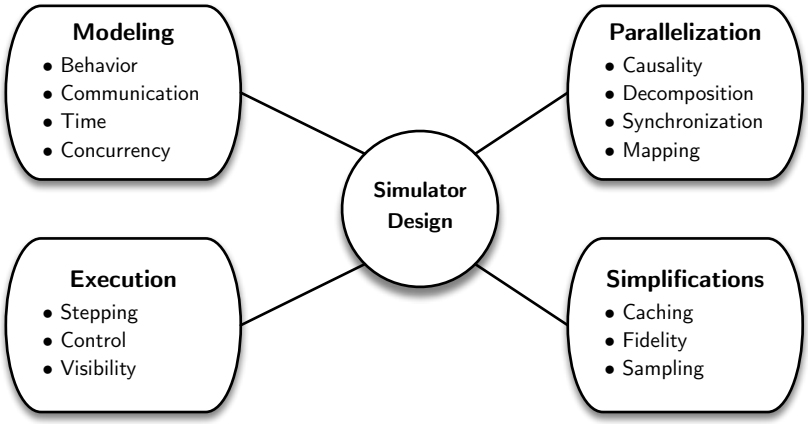

Figure A.2: Simulator Design Space

We can think of the target abstractly as the set of properties or specifications that precisely describe what we would ideally like to simulate. In practice we must realize our desired target in a *model*, written for a simulator to represent our target.

Finally, a *host* provides a collection of resources upon which we can run the combined model + simulator. The output of this triplet of host + simulator + model should mimic the properties of a desired target if it can exist in reality. We depict this terminology stack in Figure A.1.

A.1.2 Design of Simulators

The task of designing a simulator involves a number of choices and tradeoffs to balance competing demands on accuracy, speed, cost and other factors. Figure A.2 illustrates some of these dimensions which are covered in more depth in this chapter.

A.2 Simulator Modeling

The choices made during modeling involve how to bridge the idealized target system with the practical realities offered by a simulator. For example, we could model a circa 2015 smartphone as a collection of atoms from across the periodic table and computer memory simply as electron charges. However, such a simulation would be unnecessarily complex and difficult or impossible to simulate practically. Instead, much of the challenge in computer simulation design relies on choosing the right abstractions that provide sufficient detail while being sufficiently efficient.

While modeling a real world physical system may require modeling in terms of continuous equations, we leverage the discrete nature of digital systems when building computer simulations. A simulated system can be viewed as a system that evolves in discrete steps in response to events that occur at specific moments in time. This form of simulation is known as discrete-event simulation and it has been subject to significant research due to its generality, efficiency and applicability to simulation of computer systems as well as other physical processes that fit into this approach.

Abstractly a discrete-event simulation model can be viewed as having three parts: state variables, a list of pending events, and a representation of the current simulation time. The responsibility of a simulator is to evolve the state variables in response to pending events, updating the simulation time and creating new events as required. Figure A.3 illustrates this high-level view of simulation and an example simulated timeline of events being processed. The choices made of how to represent

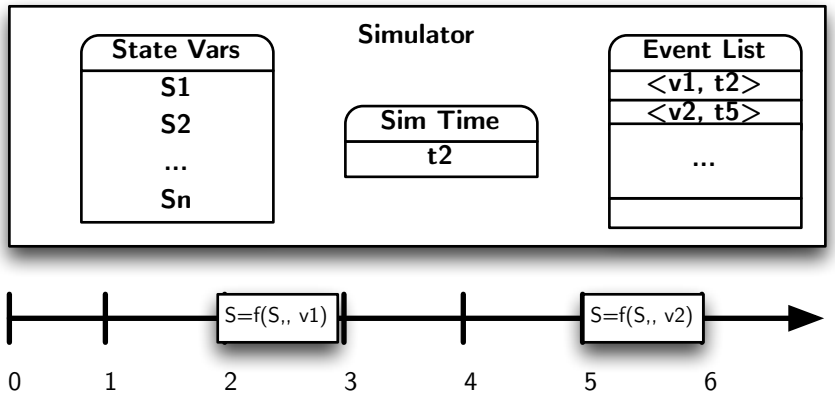

Figure A.3: Simulation Execution

the state variables (behavior) and events (timing), as well as how multiple instances of this simulation may be expressed (concurrency) becomes part of the task of simulator modeling.

Conveniently, digital systems are engineered, and so they have developed a number of abstractions simply for human scalability purposes. From circuits to architectures to programming models, a digital system can be viewed as stack of abstractions. When making choices in simulator modeling, these abstractions are used to select the required level of detail for the given experiment at hand.

The choices in modeling abstractions are illustrated visually in Figure A.4 (adapted from (Cai and Gajski, 2003; Gerstlauer et al., 2012)).

A.2.1 Representing Model Properties

The state variables of a model may be used to represent various properties of the system ranging from digital logic values to power/thermal properties. The most basic property in which we are interested is the models digital logic behavior, or simply *behavior*. Behavior can be represented with a variety of abstractions, each using different ways to model how a digital system may operate. It is often preferable

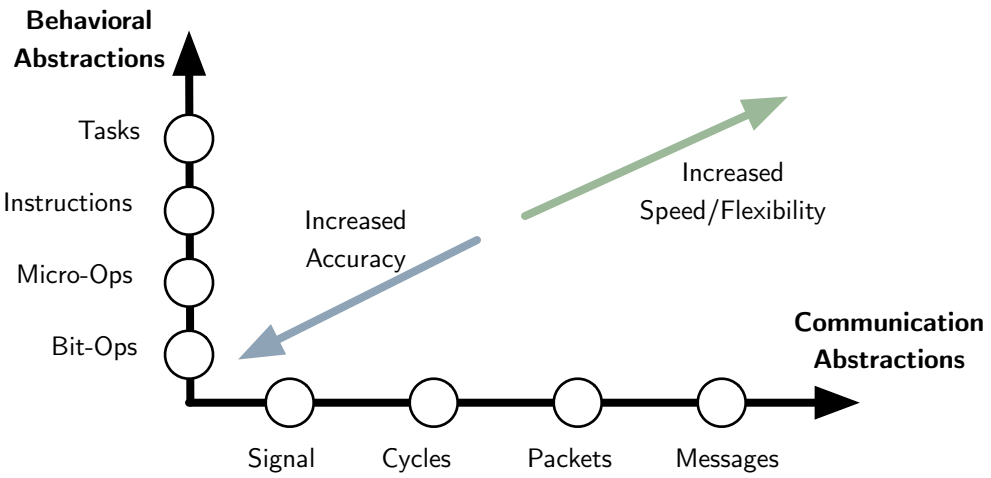

Figure A.4: Behavior and Timing Abstractions

to match the modeling abstraction to the design exploration requirements of the simulator.

We can describe behavior at a variety of levels, each having its own simulation tradeoffs in speed/accuracy. How a model represents its stateful properties can greatly influence the types of state transitions and operations that may be simulated. At the lowest level of abstraction, *bits*, we model a system in terms of digital circuit gates and flip-flops. For example the behavior of a 32-bit adder can be modeled as digital circuit with full-adder gates, carry chains and flip-flops.

Moving up in abstraction, we can think in terms of micro- operations that bundle the individual bit-level operations into *micro-ops*. For example, the same adder could be represented as a set of pipeline registers and control signals with the details of the how the add operation actually works abstracted under a simple add operation operating on these pipeline registers.

We can continue this abstraction and use *instructions* that bundle micro-ops to form more generic operations. For example, an add *instruction* in a processor model can model the add operation applied to two register reads and a register write,

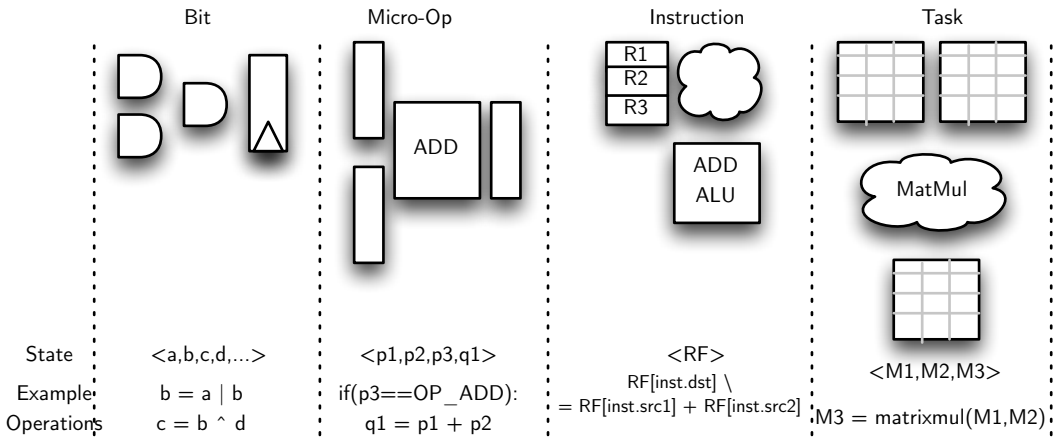

Figure A.5: Behavioral Abstraction Examples

hiding the details of the micro-ops and control signals required for the operation. This level of abstraction is highly useful when modeling programmable systems such as processors where this instruction abstraction is exposed to programmers.

Finally, we can bundle a set of instructions to form a *task* that accomplishes some effective behavior. This level of abstraction is useful in HW/SW co-designed systems, where the set of tasks to be accomplished may be well understood abstractly but the details of how it may transition into a sequence of instructions (for general-purpose processors or dedicated accelerators) is left open as a design activity.

A.2.2 Representing Model Time

To represent time, a simulation must choose a computationally convenient representation for a simulated clock as it models the target clock. For example, a simulated clock may simply be a 64-bit unsigned integer, where each increment of this clock is taken to represent 1 second of target time.

The specific choice of units for this simulated clock may set a bound on how precisely events in the system may be represented due to quantization limits. For digital systems it can be convenient to create derived or unit less clocks for simulated

time. By tracking cycles relative to some configurable simulated frequency, we can express our events in generic terms of elapsed cycles, increasing the configurability of the model. For example, a simulation model using cycles as its unit of simulated time can be configured to use a 100Mhz base frequency (so each tick represents 10ns of target time). Alternatively the same model can be configured to use a 1Ghz base frequency, (where each tick represents 1ns of target time) without redesigning the simulated model itself.

The relationship between the simulated clock and the host clock or 'wall clock' time provides a measure of forward progress as the simulator executes. The ratio of the simulated clock to the host clock provides a measure of the slowdown or speedup of the simulator. For example, a simulator that requires 1000ns of host time to simulate 1ns of target time can be described as having a 1000x slowdown.

A.2.3 Representing Model Communication

How events are modeled also has a significant impact on how time is modeled in the system. Processing events forms the basis for when and how models may communicate. Similar to behavioral abstractions, we can define a set of communication abstractions to simplify modeling of how models may communicate with events and sequence state transitions over time (see Figure A.4). While behavioral abstractions rely on simplifying state representations and their accompanying operations, communication event abstractions rely on relaxing how many state transitions a single event may cover.

At the lowest level of abstraction, individual *signal* events may be used to capture the jitter and glitching that may exist as the state variables evolve and then stabilize. For example, signal-level events can capture the ripple transitions in a carry-chain of digital gates.

Moving up the abstraction levels, we can group a related set of signal transitions

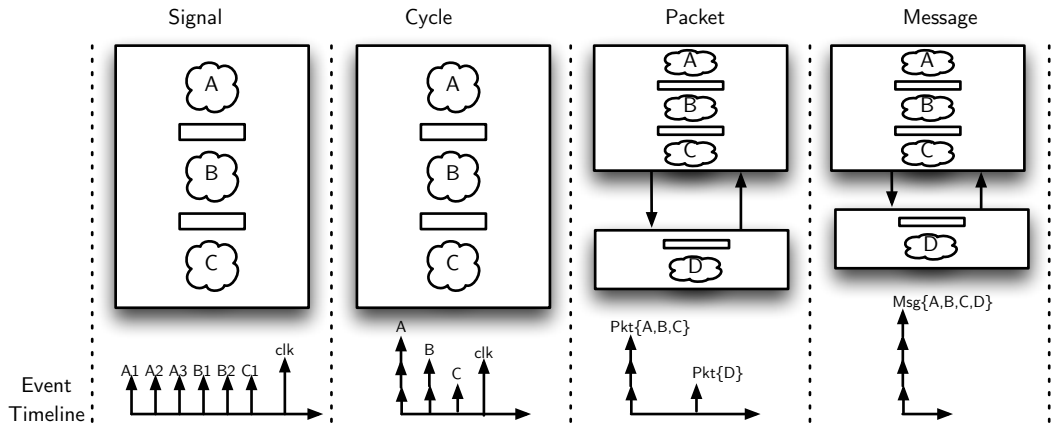

Figure A.6: Communication Abstraction Examples

into a single *cycle* event. This is particularly useful with sequential circuits that may have many transient changes that occur intra-cycle, but that only induce meaningful state changes on the sampling of the signal transition of the clock.

We can further group a set of cycle transitions that semantically share some behavior (e.g. a collection of behaviors/cycles that sequence a single bus request) into a *packet* event abstraction.

Finally we can group multiple such packet transitions so that even multiple phases of round-trip request/response event sequences may be distilled into a single *message* event abstraction (e.g. a single bus message covers both the request and the reply packets). Grouping timing into these abstractions can simplify the computation cost of processing the events but may incur some loss of accuracy.

A.2.4 Representing Model Concurrency

For modeling ease and reuse, we group together a collection of related state- variables and events into a *logical process*. Each process is concurrently executed by a simulator, exchanging events with other such processes in the simulated system. For example, we may group each processor core in a simulated multi-core system into its own

logical process.

Depending on the behavioral abstraction it is possible to model a highly parallel system as a single serial simulation. For example, working at the instruction abstraction, it is possible to view a multi-million gate processor as a single serial state-machine. Similar arguments can be made for communication abstractions that reduce the number of events required to model a given sequence of time. This movement up the abstraction stack reduces observable concurrency but gains modeling and computation efficiency benefits.

A simulator may execute such processes serially or in parallel but it must preserve event ordering to maintain correct execution (see A.4.1 for restrictions on parallel execution).

A.3 Simulator Execution

The execution of a simulator follows a simple high level loop of picking an event to process and advancing simulated time, then processing the event and updating the state variable as required. This simple process has some key choices that can dictate how efficient or useful a simulator may be for a given purpose. Specifically, a simulator must have instructions on how to execute events, schedule event processing and expose the simulated state to the user.

A.3.1 Execution

The choice of how a simulator is implemented can also govern how much control may be available over the simulation process itself. In an *indirect implementation*, the event list and simulated time are represented indirectly using some computation resources. For example, a program running on a host provides an indirect implementation of the simulated system.

By contrast, a *direct implementation* leverages the underlying host to provide

a one-to-one mapping between simulation and host events/time. If we can suitably constrain the host to provide the required behavior of a logical process, it is possible to use the real world itself as the processing engine of events and the advancement of time. An example of this is found in simulators for digital circuits that use reconfigurable logic to map the behavior directly onto the reconfigurable logic. The underlying host clock then drives the simulated clock one to one, while events simply propagate across the wires in the host rather than being posted to some pending event list. This allows higher efficiencies but it comes with a loss of control as the host effectively controls the simulation autonomously.

In the degenerate case, we can view a physical implementation of a target as a form of this direct implementation (albeit with no real ability to change the configuration of the simulation). In practice, some hybrid of this approach for parts of a system can be useful when a model may not exist and/or efficiency concerns are pressing. For example, some simplification approaches may use a direct implementation for some portion of the simulation, and then switch to an indirect implementation for increased control and configuration (see A.5.3 for more examples).

A.3.2 Stepping

We have two basic choices when processing events and updating states: event-driven and time stepped. In a time stepped approach, the processing of a logical process occurs at fixed intervals, regardless of whether there are events to process. In contrast an event-driven approach dynamically invokes the event processing of an logical process at the required timestamp of the event.

In digital systems we may use event-driven approaches in cases where we need extremely fine timing abstractions (e.g. using signal communication abstractions with near-zero advancement of time). In these cases, always invoking the event loop for such small time increments can be inefficient. Similarly, when simulating systems with

extremely predictable events (such as a cycle-level timing abstraction of a processor), using dynamic scheduling can be inefficient. The choice of execution stepping is often implementation and abstraction dependent, although general optimization approaches have been previously studied.

A.3.3 Visibility

Finally, to satisfy the core purpose of a simulation we must provide some ability to peek into the system and to observe the progress of time. The degree of visibility can range from simply the ability to observe final outputs of simulation (for example, just the amount of time it took) to having complete visibility of the simulated system at every simulated timestamp. This spectrum between measurement and introspection can have a significant impact on the performance of a simulation, as it may limit the sets of optimizations that may be possible. For example, certain processor simulators may assume that the state of the processor can only be viewed at instruction branch boundaries. In this case, it is possible to batch updates to state variables so that redundant computation can be avoided, but this comes with the visibility limitation.

A.4 Simulator Parallelization

Given a discrete event simulator we may choose to execute events in parallel either to speedup the simulation or to spread a single simulation across more host resources. Given the importance of certain classes of high-speed and capacity simulation, the topic of how to parallelize and distribute discrete event simulation (PDES) is a well-studied area.

Parallelizing a simulator has many design similarities to parallelizing a typical algorithm, but the structure provided by the simulation task itself and the need for certain ordering constraints introduce additional challenges. Issues such as maintaining event causality, decomposing into parallel tasks, enforcing event and

state synchronization across tasks and mapping tasks onto host resources are amount the key design areas that must be addressed. We provide a high-level summary of some the strategies that have been previously proposed, with particular emphasis on their application to digital systems simulation.

A.4.1 Causality

The key restriction in many simulator parallelization approaches is how to enable parallel execution of tasks while still maintaining causality between events. Given a simulation composed of logical processes, a correct parallel simulation would ensure that no event could be processed such that its effects could be felt by events that preceded it.

Fujimoto et al. (Fujimoto, 1990) describe this requirement as the *local causality constraint (LCC)*. The LCC constraint restricts the execution of events such that each logical process never processes an event out of timestamp order. By enforcing timestamp ordered processing of events, it becomes impossible for events from future timestamps to interact with the processing of events from preceding timestamps.

However, limiting parallel simulation to strict causality and the LCC can be overly restrictive. Some systems may be willing to tolerate small violations of causality if their errors can be shown to be negligible. In return for this relaxation, it is possible to increase parallelization speedup. This approach is often used in higher level simulations, such as SlackSim (Chen et al., 2009), where speed may be more valued than exacting accuracy. We further describe some of the design choices when tolerating simulation inaccuracy in A.6.

A.4.2 Decomposition

One of the biggest choices made in simulator parallelization is how to decompose the system into parallel tasks. While a simulation may be expressly written with multiple

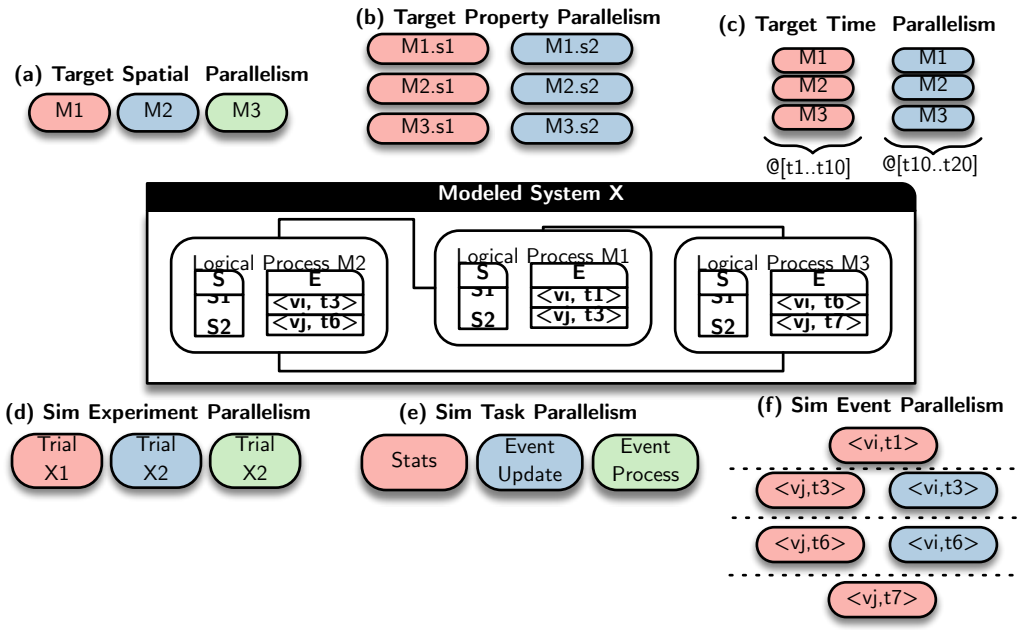

Figure A.7: Approaches for Parallel Decomposition of Simulation

logical processes, this provides one of many options. Figure A.7 illustrates some of the alternative options available to harness parallelism to speedup a simulation.

We can group this approach into two categories, model specific and simulator specific. From a model specific perspective, we can decompose a model into tasks in three basic ways. First, if we simply take each explicitly defined logical process and treat it as a parallel task, we can leverage model level parallelism. As each logical process is effectively its own sequential simulator, as long as we can ensure synchronization between local clocks in each logical process, and enforce some notion of consistency, we can run each logical process in parallel.

Second, if we partition a logical process into parts based on what the state variables that are updated for a collection of events, we can model property level parallelism. This approach may be useful when multiple independent activities are grouped into a single logical process for convenience, but they are not strictly related.

Third, if we partition the simulation across ranges in time such that each parallel task covers some disjoint portion of the entire simulation, we expose model time-level parallelism. This approach either requires some loss in accuracy (as each simulation must make assumptions on the final simulated state), or it requires recomputation until the initial state assumptions converge to some threshold.

If instead of looking at the characteristics of the model, we focus on the work performed by the simulator itself, we can expose a different level of parallelism. First, as a simulation is often used to explore alternatives, it is possible simply to run multiple simulations in parallel. This does not directly improve the speed of a single simulation, but it can reduce the time required to sweep across a variety of potential options. For example, if we are attempting to determine the best size of cache for a system, each cache size simulation could be its own parallel task. Such an approach is also commonly used when a number of inputs (e.g. programs/datasets) must be simulated across a single design point.

Second, we can identify the different tasks used to execute the simulator and treat them as parallel tasks. For example, computing and writing statistics to a file, displaying visualizations and event processing could all be treated as parallel tasks. Finally, we can try to parallelize the event handling within a single logical process. For example, with time-stepped simulations where there are a large number of events that have the same timestamp, we could map each of these events to parallel tasks.

A.4.3 Synchronization

Once a decomposition strategy (or strategies in the case of hybrid decomposition) is chosen, a technique for synchronizing tasks must also be specified. In a typical parallel application this may use atomics, locks, barriers, and other synchronization primitives. The peculiarities of simulation do not come into effect here, but the standard issues with parallel programming still still make designing a high-efficiency

synchronization approach challenging.

However, for model-level decomposition, the structure and the need for ensuring event causality has led to specialized approaches for synchronization. There has been a range of work focused on how to allow multiple logical processes to execute with minimal synchronization overhead. These efforts can be grouped into two categories, conservative and optimistic.

Conservative Approaches

Conservative protocols adhere strongly to the LCC and explicitly prevent causality errors by ensuring that only 'safe' events are processed. These protocols rely on two basic phases: determining what events are safe to process and processing safe events. If we keep all logical process clocks in lockstep, this synchronization protocol is called a synchronous conservative approach This approach may work well with time- stepped simulations or similar models where a large number of events with the same timestamp require processing and each event incurs a heavy computation burden. In such scenarios, the overhead of maintaining strict lockstep logical clocks in all logical processes does not impose too much of a burden.

Alternatively, if we allow each logical process to proceed independently (and allow each logical clock to run freely), this approach is called an asynchronous conservative approach. While each logical process may execute freely, they must all still adhere to the LCC. However, simply applying LCC can lead to deadlock as each logical process must receive at least one event from all communicating logical processes to ensure timestamp-ordered event execution is upheld. Various schemes have been proposed to resolve this issue including deadlock avoidance schemes using null messages (Chandy and Misra, 1981), or deadlock recovery using detection and breaking of logical processes (Chandy et al., 1983).

Using conservative approaches can lead to overly pessimistic parallel execution

as a logical process must halt processing when it cannot determine whether an event with a lower timestamp could potentially exist. To mitigate this, conservative approaches may use domain-specific knowledge about minimum event processing latencies or the minimum latency when sending events between logical processes. This minimum latency bound, known as *lookahead*, can be leveraged by a conservative synchronization protocol to allow forward progress for upto L timesteps into the future given a lookahead of L. However, leveraging lookahead can require static topologies and deep model-specific knowledge that is difficult to embed generically into simulator frameworks.

Optimistic Approaches

In contrast to conservative approaches, optimistic approaches relax the constraint only to allow 'safe' events to execute, allowing causality errors to occur temporarily. The first example of this approach is found in Jefferson's TimeWarp synchronization protocol (Jefferson, 1985). In a TimeWarp synchronized simulator, each logical process allows events to process assuming no event with a lower timestamp will ever be received. If such a *straggler event* does in fact arrive, the process rolls back the computation and sends *anti messages* to cancel any events that may have been generated erroneously. A minimum global virtual timestamp (GVT) is periodically generated so that logical processes can reclaim resources dedicated for rollback during a process known as *fossil collection*. A number of variations have been proposed to the original TimeWarp protocol to reduce the overhead of supporting rollback scenarios. Lazy cancellation (Ghosh et al., 1993) avoids sending anti-messages if the straggler event does not affect the generation of output events while lazy rollback (West, 1988) avoids full restoration of the state variables if the straggler event does not influence some portion of the speculatively executed events.

When the rollback rate is sufficiently small, such an approach can allow signif-

icant speedup as the potential parallelism of each logical process is not constrained. However the memory requirements to support rollback and the complexity involved in executing events speculatively can be challenging. For this reason, while optimistic approaches for PDES simulation may be implemented in generic frameworks such as ROSS (Carothers et al., 2002) or HLA (Fujimoto, 1998), it has found limited adoption in digital systems simulators.

Other Model-Level Approaches

For model-level time decomposition approaches, each task is simply a sequence that is given some initial state. As the simulation progresses, it passes on its final state to the next simulation covering the subsequent range of time. A simulation repeats until the set of processes converges or a sufficiently small difference exists between the initial state and the state provided by the preceding simulator. Typically such an approach may use normal parallel programming synchronization constructs to pass state and events between each simulator slice.

Finally, for model-level property decomposition approaches, partitions are typically restricted such that tasks do not interact with each other. For example, Jones et al. parallelize a network simulation at the MAC access layer (Jones and Das, 2001). If a model is decomposed such that the tasks require interaction, each decomposed task can be treated as its own logical process and the standard logical process synchronization can be applied.

A.4.4 Mapping

The last task required for parallelization of a simulator is mapping parallel tasks on host resources. With the proliferation of multi and manycore systems in recent years, this process has become more complex. A host may offer a mix of general purpose processors (CPUs), high-throughput but more specialized graphics processor

cores (GPUs), reconfigurable hardware engines (FPGAs) or even dedicated custom application-specific hardware accelerators (ASICs).

Most commercially available simulators still restrict mapping to CPUs only for generality. However, some research efforts have demonstrated how to map to GPUs (Wang et al., 2010) or FPGAs. Efficiently leveraging these specialized processors requires careful partitioning, load- balancing and optimization of the simulation execution loop to match the constraints offered by these systems.

Other research efforts have attempted to use design ASICs for specific simulator-level tasks such as computing global timestamps (Lynch and Riley, 2009) or state saving for optimistic protocols (Fujimoto et al., 1992).

A.5 Simulator Simplification

When simulating a digital system, one may be able to make some simplifying decisions to trade accuracy for speed. We can group these approaches into three basic categories: caching, fidelity and sampling simplifications.

A.5.1 Caching

The first simplification approach is to leverage the classic space-time tradeoff and to cache or memorize the results of a simulation. It is often the case that certain parts of a simulator may not change between runs/experiments. In these scenarios, it may be useful to precompute parts of the simulation and save the results rather than recomputing them during runtime. This allows a form of offline parallelization or place-shifting of the computing.

This form of simplification can take different forms, including those that may cache computation in a lossless fashion, and those that may incur some loss when saving/restoring the cached state. For example, the PinPoints (Patil et al., 2004) uses a lossy approach by initializing the state of the simulation using a checkpoint

that approximates the behavior as if the simulation had been run to that point in time. In contrast, Schnarr (Schnarr et al., 2001) design a language and simulator to cache microprocessor actions in a completely lossless fashion.

A.5.2 Fidelity

The second simplification approach is to leverage approximations in the model. That is, given some target T, we assume some restrictions in fidelity to create Model B that should approximate the same results as if we simulate Model A. This simplification is often used to concentrate detail on selected parts of a model (such as modeling a processor pipeline in great detail, in which the memory is modeled with a more simplified approach). Such fidelity simplifications are different than selecting a level of abstraction during modeling as we are still attempting to simulate some target T, but we are explicitly not modeling some effects under the assumption that they will have a negligible effect on the final simulation results.

A.5.3 Sampling

The third simplification approach is to use sampling to change the level of detail in the simulation dynamically. For example, when simulating a system we may switch periodically between a behavioral instruction-level modeling abstraction and a micro-instruction abstraction. Frameworks such as SimPoints (Sherwood et al., 2002) or SMARTS (Wunderlich et al., 2003) leverage this dynamic sampling process along with strategies to select appropriate points to switch and warmup strategies to minimize discontinuities when switching model abstractions. Sampling approaches can also be combined with caching approaches to allow the creation of checkpoints at a high-level of abstraction for generality (Patil et al., 2004).

A.6 Accuracy, Error and Uncertainty

Given the wide range of simulator design choices, we finally touch on the issues of accuracy and error. As previously described, some simulator design choices may impact accuracy and introduce errors to trade for speed, cost, etc. We briefly examine the types of errors that can manifest and how such tradeoffs may be justified.

Given a target and a model, we can use the term *accuracy* to denote how close the results of simulating the model are to the target. We can then use the term *error* to denote observable differences in accuracy. Using the terminology from Oberkampf et al. (Oberkampf et al., 2002), errors can either be *acknowledged* or *unacknowledged*. Acknowledged errors may be introduced due to explicit choices in modeling abstractions (e.g. selection of behavior or communication abstractions) or choices in fidelity (e.g. level of detail in a particular area of a model). These types of errors may also be introduced by simulator execution choices (e.g. relaxed causality synchronization). Unacknowledged errors are typically modeling or simulator bugs (e.g. misuse of a modeling abstraction, misconfiguration of a simulator).

In contrast to errors, we may use the term *uncertainty* to denote how much variation may potentially exist in the results. Again from Oberkampf, we can use the terminology of *aleatory* and *epistemic* uncertainty. Aleatory uncertainty includes the variation due to intrinsic random processes in the target (e.g. the stochastic workload variability (Alameldeen and Wood, 2003) in multiprocessor programs may introduce such uncertainty). Epistemic uncertainty may include potential differences in accuracy introduced by lack of knowledge or assumptions. These potential differences may not actually manifest, but if they do, it can be due to acknowledged or unacknowledged errors.

Using this terminology, it is possible to describe tradeoffs between accuracy and uncertainty in return for speed and cost. When a system is early in the development cycle, we may choose a simulator that may use modeling abstraction and fidelity

simplifications that are tuned to have relatively low average error but that incur moderate uncertainty. As the design progresses, these simplifications and abstractions may be reduced to drive down error and uncertainty further, but this will incur increased cost and reduced speeds. We survey these tradeoffs in the context of the design cycle in subsequent sections.

Bibliography

Alameldeen, A. R. and Wood, D. A. (2003). Addressing workload variability in architectural simulations. *IEEE Micro*, 23(6).

Angepat, H., Eads, G., Craik, C., and Chiou, D. (2010). Nifd: Non-intrusive fpga debugger–debugging fpga'threads' for rapid hw/sw systems prototyping. In *2010 International Conference on Field Programmable Logic and Applications*, pages 356–359. IEEE.

Argollo, E., Falcón, A., Faraboschi, P., Monchiero, M., and Ortega, D. (2009). COTSon: infrastructure for full system simulation. *ACM SIGOPS Operating System Review*, 43(1).

AWS-F1 (2017). Amazon EC2 F1 Instances webpage. https://aws.amazon.com/ec2/instance-types/f1.

Bakhoda, A., Yuan, G. L., Fung, W. W. L., Wong, H., and Aamodt, T. M. (2009). Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

Barr, R., Haas, Z. J., and Renesse, R. V. (2004). Jist: Embedding simulation time into a virtual machine. In *Proceedings of the EuroSim Congress on Modelling and Simulation (EUROSIM)*.

Bellard, F. (2005). QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*.

Binkert, N. L., Dreslinski, R. G., Hsu, L. R., Lim, K. T., Saidi, A. G., and Reinhardt, S. K. (2006). The M5 simulator: Modeling networked systems. *IEEE Micro*.

CacheGrind (2014). Valgrind CacheGrind webpage. http://valgrind.org/docs/manual/cg-manual.html.

Cai, L. and Gajski, D. (2003). Transaction level modeling: an overview. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/software codesign and system synthesis*. ACM.

Carothers, C. D., Bauer, D., and Pearce, S. (2002). ROSS: A high-performance, low-memory, modular time warp system. *Journal of Parallel and Distributed Computing*, 62(11).

Chandy, K. M. and Misra, J. (1981). Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–206.

Chandy, K. M., Misra, J., and Haas, L. M. (1983). Distributed deadlock detection. *ACM Transactions on Computer Systems (TOCS)*, 1(2):144–156.

Chen, J., Annavaram, M., and Dubois, M. (2009). SlackSim: a platform for parallel simulations of CMPs on CMPs. *ACM SIGARCH Computer Architecture News*, 37(2).

Chidester, M. and George, A. (2002). Parallel simulation of chip-multiprocessor architectures. *ACM Transactions on Modelling and Computer Simulation (TMCS)*, 12(3).

Chiou, D., Sunwoo, D., Kim, J., Patil, N. A., Reinhart, W. H., Johnson, D. E., Keefe, J., and Angepat, H. (2007). FPGA-Accelerated Simulation Technologies (FAST):

Fast, Full-System, Cycle-Accurate Simulators. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

Chung, E. S., Nurvitadhi, E., Hoe, J. C., Falsafi, B., and Mai, K. (2008). A Complexity-Effective Architecture for Accelerating Full-System Multiprocessor Simulations Using FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*.

Dennard, R. H., Rideout, V., Bassous, E., and LeBlanc, A. (1974). Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5).

Donald, J. and Martonosi, M. (2006). An Efficient, Practical Parallelization Methodology for Multicore Architecture Simulation. *ACM SIGARCH Computer Architecture Letters*.

Duda, K. J. and Cheriton, D. R. (1999). Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *ACM SIGOPS Operating System Review*, volume 33. ACM.

Edler, J. and Hill, M. D. (1998). Dinero IV trace-driven uniprocessor cache simulator.

Elmas, T., Burnim, J., Necula, G., and Sen, K. (2013). CONCURRIT: a domain specific language for reproducing concurrency bugs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM.

Emer, J., Ahuja, P., Borch, E., Klauser, A., Luk, C. K., Manne, S., Mukherjee, S. S., Patil, H., Wallace, S., Binkert, N., Espasa, R., and Juan, T. (2002). Asim: A performance model framework. *IEEE Computer*, 35(2).

Ertl, M. A. and Gregg, D. (2003). The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5.

Fall, K. (1999). Network emulation in the vint/NS simulator. In *Proceedings of the IEEE International Symposium on Computers and Communications (ISCC)*. IEEE.

Flanagan, C. and Freund, S. N. (2010). Adversarial memory for detecting destructive races. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Fujimoto, R. M. (1990). Parallel discrete event simulation. *Communications of the ACM (CACM)*, 33(10).

Fujimoto, R. M. (1998). Time management in the high level architecture. *Simulation*, 71(6).

Fujimoto, R. M., Tsai, J.-J., and Gopalakrishnan, G. C. (1992). Design and evaluation of the rollback chip: Special purpose hardware for time warp. *IEEE Transactions on Computers*, 41(1).

Galois (2011). Galois lonestar benchmark suite. http://iss.ices.utexas.edu/?p=projects/galois.

Garcia, S., Jeon, D., Louie, C., and Taylor, M. B. (2012). The kremlin oracle for sequential code parallelization. *IEEE Micro*, 32(4):42–53.

Gerstlauer, A. (2010). Host-compiled simulation of multi-core platforms. In *Proceedings of the IEEE/IFIP International Symposium on Rapid System Prototyping (RSP)*. IEEE.

Gerstlauer, A., Chakravarty, S., Kathuria, M., and Razaghi, P. (2012). Abstract system-level models for early performance and power exploration. In *Proceedings of the Conference on Asia South Pacific Design Automation (ASP-DAC)*. IEEE.

Ghosh, K., Fujimoto, R. M., and Schwan, K. (1993). Time warp simulation in time constrained systems. *ACM SIGSIM Simulation Digest*, 23(1):163–166.

Hao, F., Wilson, K., Fujimoto, R., and Zegura, E. (1996). Logical process size in parallel simulations. In *Proceedings of the Winter Simulation Conference (WSC)*. IEEE Computer Society.

Heirman, W., Carlson, T., and Eeckhout, L. (2012). Sniper: scalable and accurate parallel multi-core simulation. *Proceedings of the Conference on High-Performance and Embedded Architecture and Compilation Network of Excellence (HiPEAC)*.

IntelParaAdv (2017). Intel Parallel Advisor webpage. https://software.intel.com/en-us/advisor.

Jaleel, A., Cohn, R. S., Luk, C. K., and Jacob, B. (2008). CMP$im: A pin-based on-the-fly multi-core cache simulator. In *Workshop on Modeling, Benchmarking and Simulation (MoBS)*.

Jannesari, A., Bao, K., Pankratius, V., and Tichy, W. F. (2009). Helgrind+: An efficient dynamic race detector. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE.

Jefferson, D. (1985). Virtual time. *ACM Transactions on Programming Languages and Systems (TPLS)*, 7(3).

Jones, K. G. and Das, S. R. (2001). Time-parallel algorithms for simulation of multiple access protocols. In *Proceedings of the IEE Interational Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE.

Kim, M., Kumar, P., Kim, H., and Brett, B. (2012). Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance

model. In *International Parallel and Distributed Processing Symposium*, pages 1318–1329.

Krasnov, A., Schultz, A., Wawrzynek, J., Gibeling, G., and Droz, P.-Y. (2007). RAMP blue: A message-passing manycore system in FPGAs. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.

Lynch, E. W. and Riley, G. F. (2009). Hardware supported time synchronization in multi-core architectures. In *ACM Workshop on Principles of Advanced and Distributed Simulation*, pages 88–94. IEEE Computer Society.

Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., and Werner, B. (2002). Simics: A full system simulation platform. *Computer*, 35(2).

Mauer, C. J., Hill, M. D., and Wood, D. A. (2002). Full-system timing-first simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

McKenney, P. E. (2011). Is parallel programming hard, and, if so, what can you do about it? *Linux Technology Center, IBM Beaverton*.

Miller, J. E., Kasture, H., Kurian, G., III, C. G., Beckmann, N., Celio, C., Eastep, J., and Agarwal, A. (2010). Graphite: A Distributed Parallel Simulator for Multicores. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

Moudgill, M., Wellman, J.-D., and Moreno, J. H. (1999). Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, 19(3).

Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., and Neamtiu, I. (2008). Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association.

Oberkampf, W. L., DeLand, S. M., Rutherford, B. M., Diegert, K. V., and Alvin, K. F. (2002). Error and uncertainty in modeling and simulation. *Reliability Engineering and System Safety*, 75(3).

OpenCPI (2012). OpenCPI webpage. http://opencpi.org.

Paradyn (2007). Paradyn Performance Consultant webpage. http://www.paradyn.org/html/overview.html.

Patil, H., Cohn, R., Charney, M., Kapoor, R., Sun, A., and Karunanidhi, A. (2004). Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society.

Patterson, D. (2010). The trouble with multicore. *IEEE Spectrum*.

Pellauer, M., Adler, M., Kinsy, M., Parashar, A., and Emer, J. (2011). HAsim: FPGA-Based High-Detail Multicore Simulation Using Time-Division Multiplexing. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

PerfExpert (2014). PerfExpert webpage. https://www.tacc.utexas.edu/research-development/tacc-projects/perfexpert.

Renau, J., Fraguela, B., Tuck, J., Liu, W., Prvulovic, M., Ceze, L., Sarangi, S., Sack, P., Strauss, K., and Montesinos, P. (2005). SESC simulator. http://sesc.sourceforge.net.

RogueWave (2014). Acumem RogueWave webpage. https://www.roguewave.com.

Ryckbosch, F., Polfliet, S., and Eeckhout, L. (2012). VSim: Simulating multi-server setups at near native hardware speed. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4).

Sanchez, D. and Kozyrakis, C. (2013). ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the IEEE International Symposium on Computer Architecture (ISCA)*. ACM.

Schelle, G., Collins, J., Schuchman, E., Wang, P., Zou, X., Chinya, G., Plate, R., Mattner, T., Olbrich, F., Hammarlund, P., et al. (2010). Intel nehalem processor core made FPGA synthesizable. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*.

Schnarr, E. and Larus, J. R. (1998). Fast out-of-order processor simulation using memoization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Schnarr, E. C., Hill, M. D., and Larus, J. R. (2001). Facile: a language and compiler for high-performance processor simulators. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Serebryany, K. and Iskhodzhanov, T. (2009). ThreadSanitizer: data race detection in practice. In *Workshop on Binary Instrumentation and Applications (WBIA)*. ACM.

Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. (2002). Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press.

Tallent, N., Mellor-Crummey, J., Adhianto, L., Fagan, M., and Krentel, M. (2008). HPCToolkit: performance tools for scientific computing. *Journal of Physics: Conference Series*, 125:012088.

Tan, Z., Qian, Z., Chen, X., Asanovic, K., and Patterson, D. (2015). DIABLO: A warehouse-scale computer network simulator using FPGAs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.

Tan, Z., Waterman, A., Cook, H., Bird, S., Asanović, K., and Patterson, D. (2010). A Case for FAME: FPGA Architecture Model Execution. In *Proceedings of the IEEE International Symposium on Computer Architecture (ISCA)*.

Wang, B., Zhu, Y., and Deng, Y. (2010). Distributed time, conservative parallel logic simulation on GPUs. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. ACM.

Wang, P., Collins, J., Weaver, C., Kuttanna, B., Salamian, S., Chinya, G., Schuchman, E., Schilling, O., Doil, T., Steibl, S., and Wang, H. (2009). Intel atom processor core made FPGA-synthesizable. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*.

Wee, S., Casper, J., Njoroge, N., Tesylar, Y., Ge, D., Kozyrakis, C., and Olukotun, K. (2007). A practical FPGA-based framework for novel CMP research. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM.

West, D. (1988). *Optimising time warp: lazy rollback and lazy reevaluation*. University of Calgary, Department of Computer Science.

Wong, H., Bracy, A., Schuchman, E., Aamodt, T., Collins, J., Wang, P., Chinya, G., Groen, A., Jiang, H., and Wang, H. (2008). Pangaea: a tightly-coupled IA32

heterogeneous chip multiprocessor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

Wunderlich, R. E., Wenisch, T. F., Falsafi, B., and Hoe, J. C. (2003). Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the IEEE International Symposium on Computer Architecture (ISCA)*.

Xilinx (2012). Virtex5. www.xilinx.com/support/documentation/datasheets/ds100.pdf.

Yoo, S., Lee, J.-E., Jung, J., Rha, K., Cho, Y., and Choi, K. (2000). Fast hardware-software coverification by optimistic execution of real processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. ACM.

Yourst, M. T. (2007). PTLSim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.