Graphite: A Parallel Distributed Simulator for Multicores

by

Harshad Kasture

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2010

Author .....
Department of Electrical Engineering and Computer Science
January 29, 2010

Certified by .....
Anant Agarwal
Professor
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chair, Department Committee on Graduate Students

Graphite: A Parallel Distributed Simulator for Multicores

by

Harshad Kasture

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis describes Graphite, a parallel, distributed simulator for simulating large-scale multicore architectures, and focuses particularly on the functional aspects of simulating a single, unmodified multi-threaded application across multiple machines. Graphite allows fast simulation of multicore architectures by leveraging computational resources from multiple machines and making efficient use of the parallelism available in the host platforms. This thesis describes in detail the design and implementation of the functional aspects of Graphite. Experiment results using benchmarks from the SPLASH benchmark suite that demonstrate the speed and scalability of Graphite are presented. Results from the simulation of an architecture containing 1024 cores are also included.

Thesis Supervisor: Anant Agarwal
Title: Professor

Acknowledgments

First and foremost, I would like to thank Anant for being a terrific advisor. His support and guidance have been instrumental to this thesis, and I have thoroughly enjoyed working with him. Thanks also to Charles Gruenwald, Geroge Kurian, Jason Miller and Nathan Beckmann, my team mates on the Graphite project whose hard work has made a project of the magnitude of Graphite possible. Much of this thesis was joint work with George Kurian, and a special note of thanks to him for being an exceptionally reliable and hard-working team mate.Thanks also to all other members of the Carbon group, whose support and friendship have helped me immensely during the last two years. And last, but not the least, I would like to thank Aai, Baba, Hema and Ashwini who have been a constant source of support and encouragement in everything I have undertaken.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Simulation has always been a key research tool for hardware architects and software developers alike. This has become especially true as computer systems have increased in complexity, so that purely analytical modeling can rarely be relied on to predict all aspects of system behavior. However, poor simulator performance often hinders the usefulness of simulation by leading to excruciatingly long design turn around times. Typical simulation overheads for today's cycle accurate simulations are in the range of 1000x to 10000x slowdown over native execution (0.01 to 1 million simulated instructions per second) [7] for single core simulations.

This has often limited researchers to using small application kernels or scaled-back benchmark suites for simulation [16, 4]. However, researchers often want to simulate entire applications to be able to get an accurate understanding of system performance. This typically requires approximately 10 MIPS per simulated core in order to achieve acceptable interactivity [9]; today's cycle accurate simulators are many orders of magnitude slower than this.

The situation is aggravated further by the move to multicore architectures. Current trends in industry and academia clearly point towards manycore architectures, with possibly 100s, or even 1000s of cores on a single chip. While this is a rich area of research, it requires fast simulation frameworks. Multiplexing the computational resources of 100s or 1000s of cores on the relatively much smaller number of cores available on today's machines would slow down simulation further.

Many of today's simulators are in fact sequential [2, 13, 25, 19, 3], and thus have to effectively multiplex the simulation of 1000s of cores on a single core. Further, typical benchmarks used for studying multiprocessor architectures often tend to be longer, due to the need to factor out the effects of nondeterministic thread scheduling, the perturbation effects of I/O and the operating system etcetera. The need to simulate large applications on relatively much slower hardware presents an urgent need to develop fast simulation strategies for multicore architectures.

A similar requirement is imposed by the need to develop software for the future generation of manycore processors. Software development typically lags hardware development, often by a generation or more. This is because in the absence of sufficiently fast simulation platforms, software developers cannot start working on software until the hardware itself is available. With the rapid shift to manycore architectures, it is clear that substantial development effort needs to be devoted to developing new programming paradigms and software solutions (programming languages, operating systems, runtime systems etc.) that can leverage the computational power offered by these machines. It is also clear that such development effort cannot wait till the hardware is available, adding further to the need for fast simulation.

Graphite is a parallel, distributed simulator for multicore architectures. Graphite achieves considerable speedup over today's simulators via a variety of novel techniques, including direct execution, distribution and parallel execution with lax synchronization.

Graphite has the ability to fully utilize the computational capacity afforded by today's multicores to simulate manycore architectures of the future by allowing the simulation to execute in parallel. Graphite in fact goes further, by distributing the simulation not only among the multiple cores of a single multicore machine, but across multiple machines. This distribution is done seamlessly, and Graphite provides the functional machinery to maintain the illusion of a single process across multiple machines, including a single, shared address space and a consistent OS interface. The applications that drive a Graphite simulation are vanilla multithreaded application; the application programmer need not be aware of the distribution and the application does not need to be recompiled for different host configurations.

Graphite achieves significant speedup by relying on direct execution for the functional simulation of a large proportion of application instructions. A dynamic binary translator is used to modify the application during execution to provide extra functionality (*e.g.* core-to-core message passing) not present on the host machine, as well as to handle instructions and events that need special handling (memory accesses, system calls, thread spawn requests etc.) [22].

Graphite's high performance is also due in large part to lax synchronization model. Threads in the application are executed in parallel under the control of the host operating system, with each thread maintaining its own local clock. These clocks are only synchronized at special synchronization points (locks, receipt of messages etc.), thus allowing the threads to fall out of synch with each other. Timestamps on communication messages are used for synchronization where required [31]. Graphite does not impose a strict ordering on events in the target architecture; messages are often processed regardless of their timestamps (thus it is possible to process an event from the future before an event from the past in certain cases). Latencies calculated by Graphite often depend on the ordering of events in real time, rather than their true order on the target architecture (Section 3.4. The lax synchronization model presents many interesting modeling challenges in designing Graphite's performance models.

Graphite has a modular design, with each module having well defined interfaces to other modules. This makes it very easy to swap in a different implementation of a module to suit one's needs. For example, one may replace the core model used with a more detailed one without having to change any of the other modules. Further, Graphite maintains a separation between functional and modeling aspects of the simulation; *e.g.* the fact that the host has out-of-order cores does not imply that in-order cores cannot be modeled using Graphite.

Graphite was developed jointly by many members of the Carbon research group at CSAIL. My specific contributions to the project include:

- Developing the mechanism to implement a single, coherent address space across multiple host processes by redirecting application memory references using Pin
- Implementing the machinery to maintain a consistent OS interface across threads running in

multiple host processes

- Early exploratory work on the functionality and modeling of the network layer, including the message passing API
- High level design of Graphite, done jointly with other team members
- Helping with the design of the lax synchronization scheme used in Graphite

The first two items are joint work with George Kurian (gkurian@csail.mit.edu).

To our knowledge, Graphite is the first simulator to enable distributed execution on an unmodified application with a single, coherent address space and a consistent view of the system. Graphite also introduces the lax synchronization scheme which allows loose synchronization of application threads so that they can run in parallel with small synchronization overheads. Experimental results indicate that Graphite has low simulation overhead and good scalability across a wide range of applications, including simulations involving 1024 simulated cores.

This thesis presents a detailed discussion of the design and implementation of Graphite, focussing particularly on the various functional features implemented by Graphite in order to facilitate distributed simulation. A thorough evaluation of Graphite's scalability and speed of simulation is presented, including results showing performance gains as more cores are added to the simulation, both within a machine and across a cluster. The results indicate that Graphite scales well, with performance improving steadily till the number of physical cores equals the number of cores being simulated. The mean slow down across several applications from the SPLASH benchmark suite is 3250x, with the slowdown being as low as 77x for some applications. Further, the results indicate that performance improves irrespective of whether the extra cores are on the same or different machines. The mean speed-up for simulations of SPLASH applications with 32 target cores is 2.15 as the simulation is distributed over 4 machines instead of 1 (where each machine has 8 physical cores).

The thesis is structured as follows. Chapter 2 presents an overview of Graphite's architecture. Chapter 3 describes how Graphite models various components of a multicore architecture. Chap-

ter 4 describes in detail how Graphite maintains program correctness in distributed simulations. Chapter 5 evaluates Graphite's speed, scalability and accuracy. Chapter 6 discusses related work and Chapter 7 present directions for future work. Finally, Chapter 8 summarizes our findings.

Chapter 2

Graphite: A high level overview

Graphite is an execution driven, application-level simulator for multicore architectures. Throughout this thesis, the term *target* is used to refer to the architecture being simulated, while *host* refers to the physical machine(s) on which the simulation executes. A simulation consists of executing a multi-threaded application and modeling its behavior on a target multicore architecture defined by the simulator's models and runtime configuration parameters. Graphite maps each thread in the application to the processing core on a *tile* in the target architecture (Section 2.1). Each target core can only execute a single application thread at a time, and the number of threads in the application at any time cannot exceed the total number of cores in the architecture as specified in the runtime configuration parameters. The simulation spans multiple host processes running on one or more host machines (each target tile is assigned to one of the host processes), each potentially a multicore machine itself. The host processes communicate using TCP/IP sockets. Figure 2-1 illustrates how the target architecture is mapped to the host machine. Graphite is fully parallel: each thread in the application maps to a thread on one of the host platforms. Application threads are part of a host process and are scheduled and executed under the control of the host OS. Dynamic binary translation is used to insert traps into the simulator at events of interest to maintain functional correctness as well as to model the execution of the application on the target architecture.

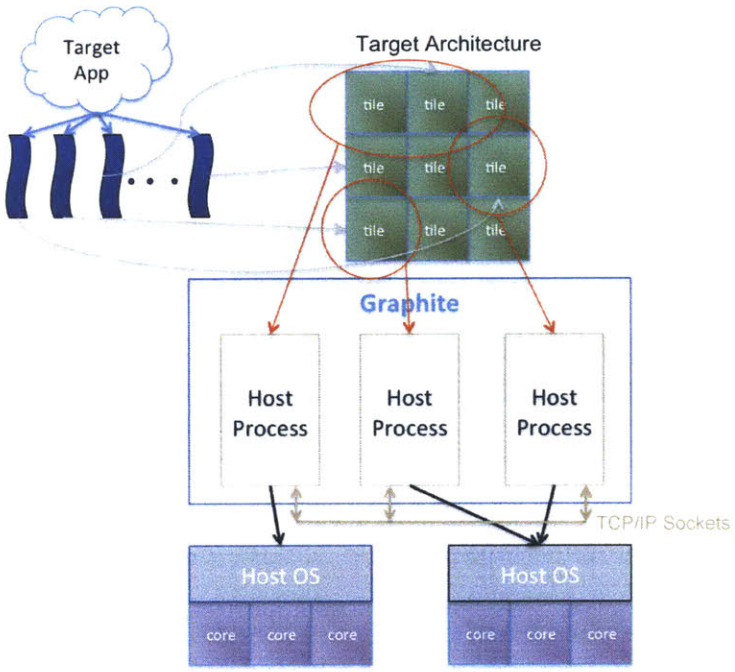


Figure 2-1: High-level Architecture

2.1 Architecture Overview

Figure 2-2 shows the components of a target architecture in a Graphite simulation in more detail. As shown in Figure 2-2, each target architecture contains multiple tiles where each tile may contain a computing core, a network switch and a memory subsystem (cache hierarchy and DRAM) [27]. Each application thread is mapped to a target core and executes on one of the host machines. Tiles are connected together using the on-chip interconnection network(s), and may communicate with each other using either message passing or shared memory. Both types of communication eventually make use of the on-chip interconnection network(s), which in turn relies on Graphite's physical transport API (built on top of TCP/IP sockets) for communicating data.

Figure 2-3 illustrates how the simulation maps to the host machines. The simulation of each target tile maps to a host process. The simulation of each tile involves many interacting components that together ensure correct functionality and modeling. The simulation is driven by the execution of the user application, which may make use of the user-level message-passing API for explicit message passing between cores. This comprises the simulation's *User Layer*. Events of interest

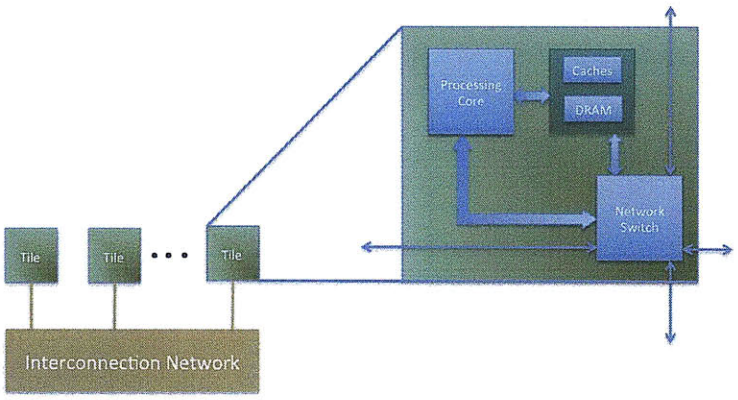


Figure 2-2: Target Architecture in a Graphite Simulation

in the simulation generate a trap into the Graphite core model, and may further make use of the memory management unit (MMU) for correct functionality and modeling (the *Core Modeling Layer*). All communications, including cache coherence traffic and explicit message passing, make use of the communications API provided by the interconnection network models (the *Network Layer*), which in turn use the *Physical Transport Layer* for the actual transfer of data. The physical transport layer is build on top of TCP/IP sockets.

Additionally, each simulation has some additional threads that provide various functional features required for simulation. In particular, the Master Control Program (MCP) is homed on a host process and is responsible for creating a consistent view of the system among the multiple host processes. It participates in thread spawning (Section 4.4) and in handling certain classes of system calls (Section 4.3). Each host process also has a Local Control Program (LCP) which is responsible for communication with the MCP to update and retrieve system state. The LCP also plays a role in thread spawning.

2.1.1 Design Overview

Graphite has a modular design where each component is implemented as a swappable module that has a well defined interface to other modules in the system. Each module can be configured through run-time parameters. Alternatively, one may replace a particular implementation of a module with a different implementation in order to study a different set of features or to study the same set of

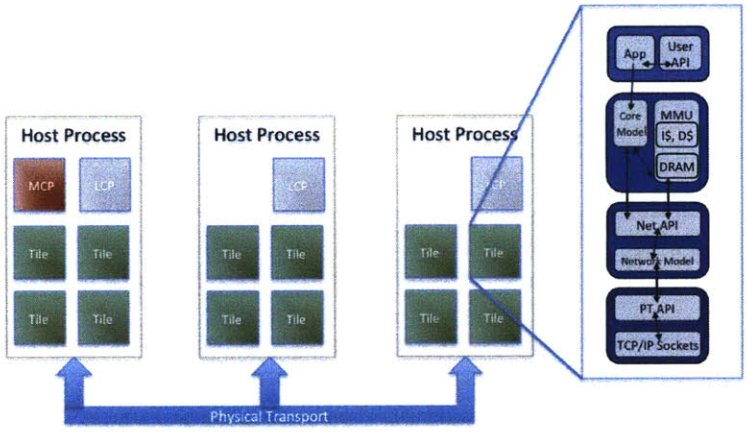


Figure 2-3: System Architecture

features in greater or smaller detail; all that needs to be ensured is that interfaces to other modules are correctly implemented.

Graphite uses a dynamic binary translator front-end to modify the application to insert simulator callbacks. In particular, Graphite rewrites parts of the application and inserts code to trap into the simulator on events of interest, such as memory references and system calls, as well as to generate a stream of instructions used for modeling. Many classes of instructions in the application, such as arithmetic and logical operations do not need to be emulated and run natively on the host machines, providing significant speedup. Currently, Graphite uses Pin [18] as the front end, although Graphite's modular design means that another dynamic translation tool such as QEMU [3] or DynamoRio [5] could easily be used instead.

Features implemented by Graphite's simulation back-end can broadly be divided into two categories: functional and modeling. Modeling features model various aspects of the target architecture, while functional features ensure correct execution of the program.

Modeling Features

As shown in Figure 2-3, the Graphite back-end comprises many components that model various parts of the target architecture. In particular, the core model is responsible for modeling the processor pipeline. The memory hierarchy is modeled by the memory model (Section 3.2), . The memory

model itself consists of models for various levels of caches as well as DRAM. The network model (Section 3.3) handles the routing of network packets over the on-chip network and accounts for various delays encountered due to contention, routing overheads, et cetera.

Note that these models interact with each other to determine the cost of each event in the application. For instance, the memory model uses the round trip delay times from the network model to compute the latency of memory operations, while the core model relies on latencies from the memory model to determine the time taken to execute arithmetic and logical operations.

The chief modeling challenge is presented by Graphite's lax synchronization model (Section 3.4), characterized by unsynchronized local clocks for each core as opposed to a single global clock. Each tile maintains and updates its own clock according to operations it performs, and these local clocks are allowed to go out of synch with the clocks of other cores. This leads to many challenges in modeling certain aspects of system behavior, such as network contention and DRAM access latencies. Section 3.4 talks in greater detail about these design challenges and how we address them.

Functional Features

Graphite's ability to execute an unmodified multi-threaded application across multiple host machines is central to its scalability and ease of use. In order to achieve this, Graphite has to address a number of functional challenges to ensure that the application runs correctly:

1. **Single Address Space:** Since threads from the application execute on different hosts and hence in different address spaces, allowing application memory references to access the host address space won't be functionally correct. Graphite provides the infrastructure to modify these memory references and present a uniform view of the application address space to all threads and maintain data coherence between them. Section 4.2 describes this in greater detail.

2. **Consistent OS Interface:** Since application threads execute on different host processes,

Graphite implements a system interface layer that intercepts and handles all application system calls in order to maintain the illusion of a single process. This is described in Section 4.3.

3. **Threading Interface:** Graphite implements a threading interface that intercepts thread creation requests from the application and seamlessly distributes these threads across multiple hosts. The threading interface also implements certain thread management and synchronization functions, while others *e.g.*, mutexes, are handled automatically by virtue of the single, coherent address space.

4. **Message Passing:** Graphite provides a message passing API for user applications and implements the functionality for these function calls to trap into the simulator and transport data between threads in the same as well as different processes.

2.2 The Anatomy of a Graphite Simulation

This section presents a high level overview of how a Graphite simulation proceeds. At the start of a simulation, Graphite spawns all host processes that the simulation would be distributed across; the host machine on which each process is spawned is specified in a configuration file. Auxiliary threads for the simulation, such as the MCP and the LCP are spawned and all configuration parameters are read from configuration files. All processes begin executing the same statically linked binary executable. After process initialization is done in all host processes (Section 4.2, Section 4.3), only one process is allowed to execute main() while the rest await thread spawning requests. Periodic traps into the simulator are inserted into the application code by Graphite's dynamic binary translator front end that are used to simulate the behavior of the application on the target architecture. Additionally, events such as memory accesses and system calls are intercepted to ensure correct execution as described in Section 4.2 and Section 4.3, respectively. Additionally, thread spawn requests in the application are intercepted and forwarded to the host processes where the thread is supposed to execute, and the application thread is spawned in this process. Graphite's shared memory system ensures that this new thread gets a view of application memory that is con-

sistent with other threads on all other host processes. Synchronization events, including file i/o, barriers and thread join requests are handled centrally by Graphite to ensure functional correctness. Threads run in parallel and synchronize periodically on synchronization events (waiting for locks, receipt of network messages *etc.*). When the application finishes execution, Graphite shuts down all the application threads as well as all the auxiliary threads spawned for the simulation.

2.3 The case for distribution

Distributing a simulation across multiple machines allows Graphite to utilize the resources available on multiple machines instead of being limited to a single machine, and opens up the possibility of using as many computational resources as are available. The most important potential downside to distributing across machines is the latency of communication: typical communication latencies between machines (over TCP/IP) are much higher than communication costs between threads in a single process (using shared memory), and could potentially slow the simulation down. However, a number of factors offset this increased cost. Since there are more physical cores available on which threads can be scheduled, a lot of the computation that would previously have been serialized can now proceed in parallel. This also reduces the context switching costs, which can be substantial. The cost of a context switch can also add to communication latencies in the event of communication between threads - if one thread is waiting for a message from another, and they are being multiplexed on a single core, the first thread has to wait till the second is scheduled (which could involve multiple context switches, the least number of context switches required would be two) before it can make progress. Distributing across multiple machines also allows the simulation to use the aggregate of all resources available on all machines - the simulation thus has a much bigger effective cache as well as much higher aggregate bandwidth to main memory. All of these factors combine to offset the increased communication latencies. In fact, the results presented in Chapter 5 indicate that the performance gain achieved by adding more cores to the simulation is the same irrespective of whether the cores are on the same machines or on different machines.

Chapter 3

Performance Modeling

This chapter describes how Graphite models the behavior of an application on a target architecture. As mentioned in Chapter 2, Graphite's modeling back-end consists of multiple modules, each modeling a part of the system under study. The static as well as dynamic information required by these models is provided by the Graphite front end through dynamic binary instrumentation. These models often interact with each other to model certain aspects of system behavior. In order to model the execution time of an application on a target architecture, Graphite uses a novel modeling technique we call lax synchronization. The following sections discuss Graphite's performance models for each part of the system. Finally, Section 3.4 discusses Graphite's lax synchronization model in detail.

3.1 Core Performance Model

The core performance model is a purely modeled component of the system in that it does not influence the functionality of program execution. The core performance model updates the local simulated core clock in response to events in the target architecture. The core model follows a producer-consumer design: as instructions are retired by the functional component of Graphite, information about these instructions is fed to the core model which is then used to model their execution on the target architecture. Most of these instructions are produced by Graphite's binary

translator front end. Additionally, "special" instructions are produced by other parts of the system to model unusual events. For instance, the network model produces a "message-receive special instruction" on the receipt of a message as a result of a network messaging API call (Section 3.3) from the application, and a "spawn special instruction" is produced when a thread is spawned on the core. This allows special events to be handled appropriately and allow the addition of instructions to the target ISA that are not present in the host ISA (network send/receive, for instance).

Correctly modeling an instruction requires, in addition to static information, some information that is only available at execution time, *e.g.* data access latencies for memory operations and paths for branches etc. This instruction is produced either by other components of the simulator back-end (*e.g.* for memory operations) or by the dynamic binary translator (*e.g.* for branches) and is consumed by the core performance model via a separate interface.

Since the core performance model has no impact on functionality, it can fall out of synch with the functional part of the simulator. This isolation between functional and modeling aspects of Graphite's execution allows a lot of flexibility in implementing the modeling components to closely match the target architecture even if it differs considerably from the simulator's functionality. For instance, although the simulator is functionally in-order with a sequentially consistent memory model, it is entirely valid to have a core model that models an out-of-order core with a more relaxed memory model. Further, since all other parts of the system ultimately use the core clock, the effect of the core model will be reflected in all other parts of the system, *e.g.* network utilization numbers will reflect an out-of-order architecture since network time stamps use core clocks.

Graphite currently supports an in-order core model with an out-of-order memory system. Components of the core and the memory system such as store buffers and branch predictors are configurable through run time parameters.

3.2 Memory System

The memory system is composed of several modules such as instruction- and data-caches, and DRAM controllers, each associated to one of the simulated tiles and connected using the network layer. The memory system is responsible for simulating the cache hierarchies, memory controllers and cache coherence engines of the target architecture under study. For this purpose, the various modules of the memory system interact with each other using additional messages that simulate various aspects of the target memory subsystem such as the cache coherence scheme.

The memory system in Graphite also has a functional role, namely to maintain a single address space between application threads, many of which may be executing on different host machines and hence in different host address spaces. Graphite redirects memory references in all application threads to access data resident in the target address space rather than in their respective host address spaces. The memory reference redirection is achieved using dynamic binary translation, either by rewriting the memory references in place or, in a small number of special cases, emulating them in software. It is the responsibility of the memory system to service these redirected memory accesses and efficiently manage the application's data. It accomplishes this by statically partitioning the application's address space among the different machines participating in simulation; the data corresponding to that portion of the address space is "homed" on that machine. However, such a naive strategy of managing data could prove detrimental to the performance of simulation, as the different memory modules would have to frequently send messages over the network to service the memory requests of their local threads. To overcome this limitation, data frequently accessed by an application thread is cached at its local memory module and all such cached data is kept consistent using a cache coherency protocol.

If the modeling and functional aspects of the memory system behavior were kept completely independent, it could lead to inefficiencies since each application memory request may result in two sets of network messages, one for ensuring the functional correctness of simulation (actually retrieving the data) and the other for modeling the performance of the target memory architecture. Graphite addresses this problem by modifying the software data structures used for ensuring func-

tional correctness to operate similar to the memory architecture of the target machine. In addition to improving the performance of simulation, this strategy automatically helps verify the correctness of complex hierarchies and protocols used to implement the target machine's memory architecture, as their correct operation is essential for the completion of simulation. Performance modeling is done by appending simulated time-stamps to messages sent between the different memory modules and is explained in great detail in Section 3.4.

Currently, the memory system is Graphite simulates a target memory architecture with L1D, L1I and L2 caches. Cache coherence is maintained using a directory-based MSI protocol in which the directory is uniformly distributed across all the tiles.

3.3 Network

The network component provides high-level messaging services between cores built on top of the lower-level transport layer (subsection 3.3.1).

The network component contributes both to functionality and modeling. Functionally, it provides a message-passing API directly to the application, as well as serving other components of the simulator back end, such as the memory system (Section 3.2) and system call handler (Section 4.3). All network messages are eventually transported over the transport layer.

The network component consists of one or more network models that are responsible for modeling events over the network. The network provides common functionality, such as bundling of packets, multiplexing of messages, a high-level interface to the rest of the system, as well as a common interface to the transport layer. The various network models perform tasks such as routing packets, modeling contention and updating the time-stamps on messages to account for network delays. This separation between functionality and modeling is not absolute, however, since the route of the packet computed by the network model affects traffic through the transport layer. Functionally, the packets are transported to their respective destinations regardless of their timestamps -thus packets may arrive at the destination "earlier" in simulated time than they are meant

to. Also, the network only preserves order among packets in real time, not in simulated time. This leads to many interesting modeling challenges, discussed in greater detail in Section 3.4.

Graphite currently implements several distinct network models. The network model to be used for a message is determined by the message type. For instance, all system messages unrelated to application behavior (*e.g.* updating utilization statistics for the network) use a separate network model from the application messages and thus not interfere in modeling the behavior of the application. Further, Graphite by default uses separate models for application and memory traffic, as is common in modern multicore chips [27, 29].

Each network model shares a common interface. Network model implementations are thus swappable, and each network model is configured independently. This allows exploration of various network topologies and parameters for particular subcomponents of the system. Graphite currently implements a "magic" (zero delay) network model for special system messages, a mesh model that determines network latency simply by counting the number of hops (no contention modeling), and a more complicated mesh model that accounts for contention using an analytical queuing model.

3.3.1 Transport Layer

The transport layer provides an abstraction for generic communication between cores. All inter-core communication as well as inter-process communication required for distributed support goes through this communication channel. The current transport layer uses TCP/IP sockets for data transport, however this could be replaced with another messaging back end such as MPI.

3.4 Lax Synchronization Model

In order to achieve good performance and scalability, and make effective use of the parallelism available in the application, Graphite allows target cores to run independently with minimal synchronization. This is necessary for performance reasons, since synchronizing across multiple host

machines after every simulated clock cycle would impose too great an overhead on the simulation. However, this relaxed approach to synchronizing cores means that the cores' local clocks do not always agree, and events may be seen and processed out-of-order in simulated time, leading to many modeling challenges as described in the following paragraphs. We term this relaxed approach to synchronization the "lax synchronization model".

In Graphite's modeling framework, each core's local clock is updated by the core performance model. A majority of the events that lead to the updating of the clock are local in nature, in that the modeled time for the event is independent of the rest of the system (Section 3.1). However, the modeling of some operations, such as memory accesses (Section 3.2), message send/receive via the application message-passing API, thread spawn/join etc. depends on interaction with the rest of the system. This interaction happens exclusively through network messages, each of which carries a time-stamp that is initially set to the clock of the sender. In the case of the memory operations, the latency of the operation depends on the round trip delay of the relevant network message and therefore leads to synchronization with the rest of the system (as explained later). Other operations, such as message receive (via the application message-passing API), thread spawn/join and explicit synchronization operations in the application (locks, barriers etc.) lead to explicit synchronization. In these cases, the clock of the core is advanced to the time when the event completes. If the core has the largest cycle count of all participating cores in such an event, its core clock is not updated.

Graphite's strategy to handle out-of-order events is to process them as they are received irrespective of the simulated time when they occurred. An alternative would be to preserve the order of events in simulated time, either by buffering and re-ordering events, or by rolling back if ever a violation of the simulated time order of events is discovered (a strategy implemented in BigSim [31], among others). The former strategy is difficult to implement since Graphite does not have a notion of a global clock, can have very large overheads and is difficult to implement in a deadlock-free manner. The latter strategy also has many problems, most significantly to maintain large amounts of history and the need for frequent rollback in the case of frequent communication, as happens in the case of memory operations, thus leading to large overheads. Empirical results indicate that

Graphite's strategy, while not completely accurate, yields correct performance trends.

This complicates modeling, however, particularly of system behavior that depends on interaction between events, since it is not straightforward to figure out which events have an overlap in simulation time. One example is modeling contention at network switches and queuing delays in memory controllers. If the simulated-time order of events were maintained, a queue could be easily implemented by buffering incoming packets, and dequeuing the packet at the head of the queue every time a new packet can be processed. Since packets in a Graphite simulation arrive out of order, computing queuing delays is not straightforward.

Queuing delays are instead modeled by maintaining a separate "queue clock", that measures the time until which the queue is busy *i.e.*, the time at which the tail of the queue will be processed. For an incoming packet, the delay experienced by it is the difference between the queue clock and the "global clock". Each incoming packet also advances the queue clock by the amount of time required to process the packet.

The "global clock" used in this computation is itself estimated from the various packet time stamps. Packet time stamps may differ significantly for a variety of reasons. One example is packets originating at cores that are not running an active thread, which implies that their local clocks are not advanced. Such cores still participate in the simulation via the associated network switches and memory controllers. The "global clock" is therefore estimated by averaging the time stamps of a window of most recently seen packets. To minimize the effect of outliers, these windows need to be reasonably large, typically a few hundred packets. Since messages are generated quite frequently (*e.g.* on each cache miss), even a large window gives a fairly up-to-date representation of global progress.

Combining these techniques yields a queueing model that works within the framework of lax synchronization. Error is introduced because packets are modeled out-of-order in simulated time, but the aggregate queueing delay is correct. Other models in the system face similar challenges and solutions. Results indicate that local core clocks do not get too far out-of-synch (Section 5.3), and thus the error introduced by processing events out-of-order is not expected to be too great.

Chapter 4

Distributed Simulation

Distributed execution of a single multi-threaded binary is central to Graphite's ability to deliver good simulation performance for large parallel simulations and its ease of use. In order to correctly executed a single program across a cluster of workstations, Graphite needs to address a number of challenges. The functional features implemented by the Graphite back-end were briefly outlined in Section 2.1.1. This chapter discusses the implementation of these functional features in greater detail.

The choice of solutions to these problems, as well as many of the implementation details depend on the specific functionality provided by Pin [18], the dynamic binary translation front end used by Graphite. The chapter therefore begins with an introduction to Pin and the features it provides. This is followed by sections discussing each of the functional features.

4.1 Pin: A Dynamic Binary Instrumentation Tool

Pin is a free tool for dynamic instrumentation of program binaries provided by Intel. Code running under Pin can be dynamically instrumented to insert arbitrary C/C++ code in arbitrary places. The instrumented code is cached and reused, so that one only has to pay the cost of instrumenting the code once. Pin defines many logical entities within a binary:

- **Image**: E.g. the main program image, various libraries loaded and used by the main program etc.
- **Trace**: A trace is a sequence of instruction with a single entry point that ends with an unconditional branch. There may be multiple potential points (*e.g.* conditional branches) where control may exit the trace between the start and the end
- **Basic Block**: A basic block is a sequence of instructions characterized by a single entry point and a single exit point (conditional or unconditional branch)
- **Routine**: A routine within the binary
- **Instruction**: A single instruction

Pin allows the code to be instrumented at various granularity levels, e.g. one may insert instrumentation code for each instruction, or for each basic block. Additionally, one may specify whether the inserted code executes before or after the corresponding application code. Multiple blocks of code may be inserted corresponding to the same chunk of application code; Pin provides limited facility to specify ordering among these dynamically inserted blocks of code.

The code to be inserted into the instrumented application, as well as the places where it should be inserted, is specified in a *Pintool*. The Pintool registers *instrumentation routines* with Pin that are called whenever Pin generates new code. The instrumentation routines inspect the new code and decide where to inject calls to *analysis routines*, also defined in the Pintool, that are called at run time. Various static and dynamic information, such as the thread id of the thread executing the instruction, the values of various registers, various properties of the code block being instrumented (*e.g.* the addresses of memory references for a memory access instruction) etc. may be passed to the analysis routine.

Pin as well as the Pintool reside in the application's address space.

All of Graphite's back end, including the various models as well as the functional features, reside in a Pintool. Graphite uses Pin for two purposes:

1. **To collect static and dynamic information about program execution** Graphite uses Pin analysis routines to feed dynamic information about the program execution to its various models. In particular, analysis routines inserted at the granularity of basic blocks are used to generate an instruction trace that drives the core models.

2. **Change program behavior** Graphite uses Pin to control and change the execution of the program. Program behavior may be changed using Pin by modifying program context, *e.g.* contents of registers and memory, by inserting jumps at arbitrary points in the program, and by deleting program instructions. Graphite instruments each memory reference in the application code to redirect all memory accesses to the shared address space. Additionally, all message passing functions within the application are also instrumented to provide both functionality and modeling for user level messaging. Additionally, callback functions may be registered with Pin that are called at specific events of interest, *e.g.* start of program execution, the start of a new thread, a system call etc. Graphite uses some of these callback functions to provide a consistent address space, as discussed in detail in this section as well as in Section 4.3

The following sections provide an in-depth discussion of the implementation of the various functional features in the Graphite back end.

4.2 Address Space Management

As mentioned earlier, Graphite has to maintain a single "simulated" address space across all the threads participating in a simulation, which may be executing in different "host" address spaces *i.e.* they are part of different host processes. This presents two problems that Graphite needs to solve:

4.2.1 Issues

There are four main issues in maintaining a single address space across multiple host processes:

1. **Data coherence:** All application threads running across all host processes should have a coherent view of data

2. **Ambiguity:** A single address in the simulated address space should not refer to two or more separate data items in the program

3. **Validity:** All memory accesses that would have been valid on the target architecture should be valid in the simulation, *i.e.* they should not cause exceptions

4. **Sandboxing:** Data accesses from the application and the simulator should not interfere with each other

The data coherence requirement implies that memory writes performed by one thread should be visible to all others, and all memory reads from an address by all thread between subsequent writes to that address should yield the same value.

The problem of ambiguity does not arise for static data items in a program. This is because all host processes participating in the simulation run exactly the same statically linked binary and thus have exactly the same address corresponding to a given static data item. However, requests for dynamic memory may present a problem - since each host process has its own address space, requests for dynamic memory from threads in different host processes (which eventually result in a `brk`,`mmap` or `mmap2` system call on the host system) may return the memory blocks with the same or overlapping address ranges. Since these two memory requests may correspond to completely different data items in the program, this will lead to ambiguity regarding the data value associated in the simulated address space with a given memory address

Similarly, the problems of validity and sandboxing are relevant in the context of dynamically allocated data: a range of addresses, dynamically allocated in one host process may not be part of the valid set of addresses in another host process, or may correspond to a set of addresses being used by the simulator (since Pin and the Pintool execute in the same address space as the application). If a thread in a process other than the one in which the data was allocated attempts to

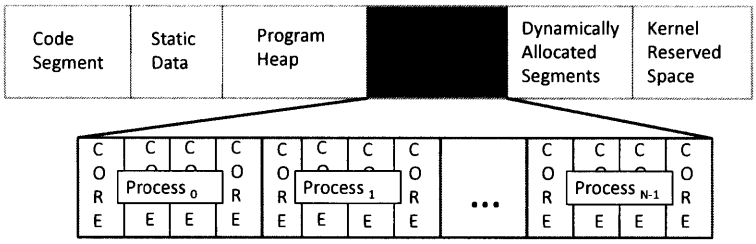


Figure 4-1: Segments within the application address space

access that memory address, it may lead to an exception, or may corrupt simulator data. Similarly, memory accesses within the simulator may corrupt application data.

Graphite solves the problem of ambiguity by implementing its own memory manager that centrally and unambiguously handles requests for allocation and deallocation of dynamic memory. The problems of validity, sandboxing and coherence are solved via Graphite's shared memory system, that maintains coherent copies of each piece of data (Section 3.2). Graphite intercepts every memory reference in the application and modifies it to access the correct data stored in the shared memory system.

4.2.2 Graphite Memory Manager

The Graphite memory manager divides up the application address space as depicted in Figure 4-1. It is responsible for servicing all application requests for dynamic memory. It also reserves a part of the address space for thread stack, as explained below.

The first two segments - the code segment and the static data segment are determined entirely by the program binary that Graphite is executing.

The program heap and the dynamic data segment are used to allocate dynamic data for the application. Dynamic memory requests are intercepted by trapping on the `brk`, `mmap` and `mmap2` system calls as explained in Section 4.3. These requests are then forwarded to the MCP, which serves as the simulations dynamic memory server. Here, Graphite services these requests via its own memory manager.The program heap is a range of the address space that is used by the memory manager to service the `brk` system calls by extending the end of the data segment into

this space (the data segment only extends as far as the end of the static data segment at the start of the program). The dynamic data segment is used to allocate data chunks in response to `mmap` and `mmap2` system calls. Since all dynamic memory requests are eventually serviced by the MCP, all application threads get a consistent view of the application address space.

Thread stacks present a special case of the problems of validity and sandboxing of addresses discussed above. Parts of the Pin functionality used by Graphite, *e.g.* routine replacement, discussed in subsection 4.4.1, access the addresses on the thread stack in the host address space. Unlike the application memory accesses (subsection 4.2.3), Graphite cannot redirect these memory accesses to the simulated address space. This can lead to two problems - it can cause an exception if an address is invalid, or it can cause useful data to be corrupted. For this reason, Graphite needs to treat stacks specially. In particular, addresses corresponding to thread stacks need to be valid in all host address spaces, and should not be used for any other purpose. Graphite thus reserves a part of the address space for thread stacks (Figure 4-1). The size of the stack for each thread may be specified as a configuration parameter. This part of the address space is also reserved in each process' host address space via an mmap call before execution begins. This ensures that the entire range of stack addresses is valid in all host address spaces and does not contain any other useful information that may be overwritten. Of course, since the host address space does not contain the correct data values, the information read by Pin from the host address space is incorrect and will lead to correctness issues. Subsection 4.4.1 discusses how Graphite addresses this problem.

4.2.3 Redirecting memory references

Graphite redirects each memory access in the application inn order to make sure that memory accesses retrieve data from the simulated address space, stored in Graphite's coherent shared memory system, and not the host address space. For rewriting most memory accesses, Graphite uses the following Pin API function:

```
INS_RewriteMemoryAddressingToBaseRegisterOnly(INS ins, MEMORY_TYPE
mtype, REG newBase)
```

This function rewrites the specified memory reference to use only a base register instead of the fully array of components used in specifying the address, namely displacement, base, index and scale factor.

Additionally, Graphite inserts an analysis routine that is called during execution before the memory reference executes. This routine is passed the address being accessed and the number of bytes of data being read/written. If the access is a read access, Graphite reads the corresponding data from the shared memory system and places it in a scratch memory area. The address to the start of this data is then placed in the base register for the memory access. Write accesses are also redirected to a scratch memory area in a similar manner, and another analysis routine is inserted in the code to execute after the write access has completed that writes the data written in the scratch area back to the shared memory system. An ia32 instruction may contain up to three separate memory references: two reads and a write. Each of these memory references needs to be rewritten in this manner.

Many instructions that have memory accesses may also have a LOCK prefix: this means that the instruction should be executed atomically. However, the process described above for redirecting memory accesses is not atomic by itself. For such instructions, Graphite ensures atomicity of operations by 'locking' the private L1 cache of the core executing the LOCKed instruction[1]. A locked cache defers processing cache coherence messages until after the instruction has finished executing.

This approach, however, cannot be used for two classes of memory references: memory references with implicit memory operands and some types of string instructions. These need to be handled differently as described below. memory operands and some classes of string instructions.

[1]This strategy assumes that L1 caches are private, and that the data being operated upon is always present in L1 caches; L1 caches have to be write-allocate, for example. If either of these is not true, this strategy won't work. One can still ensure correctness by using a coarser grained lock, *e.g.* locking the entire memory system for the duration of execution of the instruction, but this strategy would obviously be slower.

Memory accesses requiring special treatment

Instructions such as the stack operations contain implicit memory references, *e.g.* a POP instruction reads reads data from the top of the stack (pointed to by the ESP register), and places the value in a register/memory. Pin cannot rewrite these instructions in the manner described above, *e.g.* the POP instruction can not be rewritten to access data from a location other than the one pointed to be ESP. Graphite handles these by deleting the original instruction, and inserting an analysis routine in its place that emulates the instruction in software. Instructions that need to be handled in this way include PUSH, POP, CALL, RET, LEAVE, PUSHF, POPF, PUSHA and POPA.

Some string instructions such as CMPSD and SCASD present a different problem. When these instructions are preceded by a REP prefix, the number of times the instruction is repeated depends on values in memory. For example, the instruction SCASD compares the value in EAX with the value stored in the memory location specified by ES:EDI, updates EFLAGS accordingly and either increments or decrements EDI according to the setting of the DF flag in EFLAGS. When preceded by a REP prefix, this operation is performed repeatedly until either the ZF flag is set, or the operation has been repeated N number of times, where N is the value specified in the ECX register. The number of times the instruction is repeated thus depends on the values present in memory during execution. When these instructions are rewritten using the Pin API function described above, Pin determines the number of times the instruction should be repeated by accessing memory. Since Graphite cannot intercept Pin memory accesses, these accesses go to the host address space directly and thus give wrong results. Graphite avoids this problem by deleting these instructions and emulating them in software. String instructions that require special handling include CMPS[B/W/D/Q] and SCAS[B/W/D/Q].

Figure 4-2 summarizes how Graphite redirects memory references in the application.

System Initialization

At process startup, before control is passed to the user application,the host system writes data in the process address space that is essential for the correct execution of the program. This data

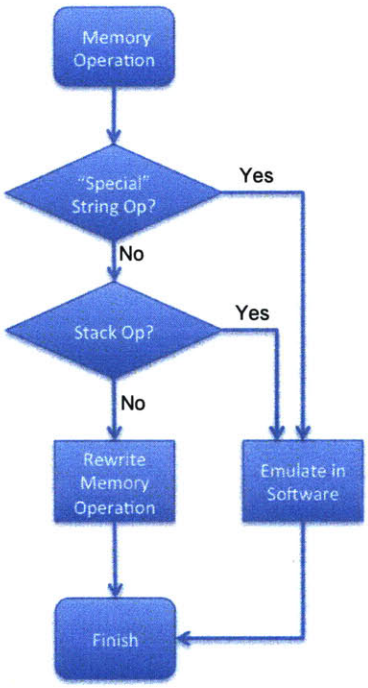


Figure 4-2: Redirecting application memory references

includes command line arguments, environment variables etc, as well as pointers to these values, all of which is placed above the program stack. All this data is in the host address space. Graphite registers a callback function with Pin that is invoked before control passes to the application. This function copies all these values to the simulated address space before transferring control to the application.

4.3 Consistent OS Interface

System calls in an application running under Graphite need special handling for many reasons. The first is the need to maintain the illusion of a single simulated process executing an application across multiple host processes. This becomes an issue with system calls that may be used to communicate or synchronize among different threads in the application; since the threads execute in different host processes which are not aware of each other, simply allowing the system calls to execute on the hosts would yield incorrect results.

Another set of system calls that needs special handling are the ones that pass pointers to data

in memory to the kernel. Since Pin can not modify memory accesses in kernel space, the kernel would try to access the data in the host address space, and will thus either get incorrect data or may even cause an exception.

Lastly, system calls that deal with allocation and deallocation of dynamic memory need to be intercepted and forwarded to the Graphite Memory Manger, as discussed in Section 4.2.

To get around these problems, Graphite implements a system interface that intercepts and handles system calls in the target application. The system interface handles each system call from the application in one of three ways: centrally, locally or fall-through.

Examples of system calls that may be used for inter-thread communication include system calls used for file I/O, such as `open`, `close`, `read` and `write`. For example, in a multi-threaded application, threads might communicate via files, with one thread opening a file (using the `open` system call), writing to it (using the `write` system call) and passing the file descriptor to another thread which then reads the data using the `read` system call. In a Graphite simulation, these threads might be in different host processes, and thus a file descriptor in one process may either point to a different file or be invalid in another process. Thus, simply executing the system calls on the respective host systems will not yield correct results. Instead, these system calls are handled centrally by Graphite's system call server running on the MCP. Graphite intercepts all instances of these system calls in the application, forwards the arguments to the system call server, which executes the system calls on its local host. The results are then sent back to the application. Since all system calls execute on the single host process (the one that hosts the MCP), all threads in the simulation get a consistent view of the system. For example, the file descriptor returned from an `open` system call may be used by any of the threads to read from or write to the file; since the resulting `read` or `write` system call will eventually be executed on the same host process that executed the `open` system call, the program behaves as expected. Other such system calls, such as `fstat64` and `access` are handled in a similar manner.

Similarly, system calls such as `futex` are used to achieve synchronization between threads. For example, `futex` may be used to implement a simple mutex lock, so that the threads execute

the FUTEX_WAIT call to atomically check the value of an integer and enter the critical section if no other thread is accessing it, or are put on a wait queue by the kernel to be woken up when the thread that currently holds the lock executes a FUTEX_WAKE. Clearly, this system call cannot simply be executed locally in a Graphite simulation. Instead, Graphite intercepts all `futex` system calls in the application and forwards the arguments to the system call server, which implements the functionality for the `futex` system call normally provided by the kernel.

To intercept the system calls and update the results, Graphite registers callback functions with Pin that are called immediately before and after a system call. The system call entry callback function reads the system call arguments, bundles them up and send them over the network layer to the MCP, where the correct system call is executed by the system call server and the results are shipped back. It also changes the local execution context to turn the system call into one that won't affect the functionality of the program, such as `getpid`. The system call exit callback function receives the results returned by the system call server and updates the current execution context. This includes updating the system call return value (located in the EAX register) and also updating memory buffers where appropriate (*e.g.*, for a `read` system call).

The system calls related to allocation and deallocation of dynamic memory, such as `brk`, `mmap`, `mmap2` and `munmap` are handled in a similar manner.

Many system calls such as `uname` and `clone` pass as arguments to the kernel pointers to chunks of data in the memory. These arguments may be read or written. Since Pin cannot modify kernel memory references, the kernel would try to access data in the host address space, which would be functionally incorrect. To avoid this problem, Graphite intercepts all system calls that pass arguments in memory and modifies their arguments before allowing them to execute locally *i.e.* on the host machine. Thus, if an argument in memory is passed as input to the kernel (the kernel reads this value), Graphite retrieves the data from the simulated address space, places it in a scratch memory area, and modifies the corresponding system call argument to point to this data. Output arguments that update memory are similarly modified to point to scratch memory areas. After the system call has finished executing, any updated values are written back to the simulated

address space. This modification of system call arguments and updating of memory values after system call execution is accomplished using callback functions that are called before and after every system call.

Finally, some system calls such as `exit` do not need any special handling and are allowed to fall through, *i.e.* execute on the host system without modification.

Since the total number of system calls in Linux is very large, Graphite does not support all of them. Instead, system calls are supported on an as needed basis. Currently, Graphite supports all system calls required to run the entire SPLASH benchmark suite compiled using gcc version 4.3.2 with libc version 2.3.6 running on a Debian system (kernel version 2.6.26). Table **??** lists all the system calls currently supported by Graphite, as well as how they are handled (centrally, locally or allowed to fall through).

Figure 4-3 summarizes how Graphite handles system calls in the application.

4.3.1 Process Initialization and Address Space Management

As mentioned in Section 4.2, Graphite copies over data written to the host address space by the system into the simulated address space before transferred is controlled to the application. However, other aspects of process initialization, such as the setting up of thread local storage (TLS), require special handling as well. Each participating host process in a Graphite simulation needs to be properly initialized, even though only one of them eventually executes main(). In order to achieve this, Graphite let's process initialization routines (routines executed before control is transferred to main()) execute in each host process. This is done sequentially *i.e.*, the first process runs the routines while others wait, then the next and so on. This is necessary since all the initialization routines are modifying data in the same address space and since the routines are not normally executed in parallel and thus may not be thread-safe, allowing all the threads to execute them in parallel may be incorrect. Once all processes have finished initialization, one process executes main() while the others wait for thread spawn requests from other processes (either the process running main() or any of the other processes that have already spawned threads).

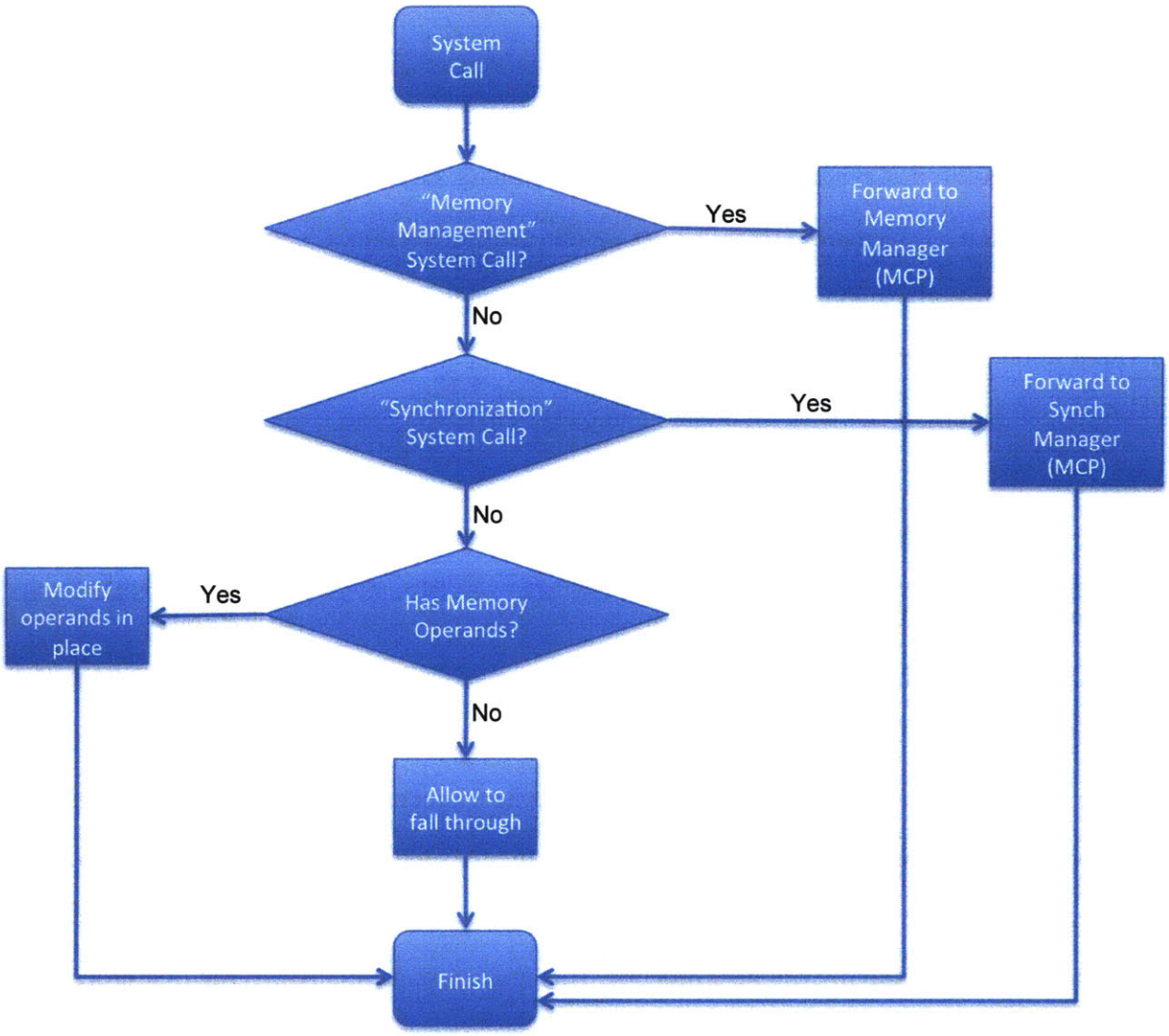


Figure 4-3: Handling application system calls

System Call Name	System Call Number	Handling
open	5	central
read	3	central
write	4	central
close	6	central
access	33	central
getpid	20	central
readahead	225	central
pipe	42	central
brk	45	central
mmap	90	central
mmap2	192	central
munmap	91	central
futex	240	central
mprotect	125	central
set_tid_address	258	local
rt_sigprocmask	126	local
rt_sigsuspend	179	local
rt_sigaction	67	local
nanosleep	162	local
uname	122	local
ugetrlimit	191	local
set_thread_area	243	local
clone	120	local
time	13	local
gettimeofday	78	local
fstat64	197	local
set_robust_list	311	local
exit	1	fall through
exit_group	252	fall through
sigreturn	119	fall through
geteuid32	201	fall through
getuid32	199	fall through
getegid32	202	fall through
getgid32	200	fall through
gettid	224	fall through
kill	37	fall through

Table 4.1: System Calls supported by Graphite

4.4 Threading Infrastructure

Unlike other parallel programming models like MPI, Graphite does not require the application programmer to be aware of the distribution of computation across multiple hosts and manage it explicitly. Instead, Graphite seamlessly distributes the execution of an unmodified multi-threaded application across multiple hosts. From the application programmer's perspective,the application is a simple multi-threaded application, executing in a single process with a single address space; parallelism is expressed via the POSIX threading API (pthreads). The only limitation is that the number of active threads in the application should not exceed the total number of cores for the simulation specified at run time.

To accomplish this, all thread spawning requests in the application are replaced by traps into the Graphite back-end. The arguments to the function and are then forwarded to the MCP. Here, the Graphite distribution engine maps the to-be-spawned thread to an available core and forwards the request in turn to the LCP of the machine to which the core is mapped. The mapping of the threads to cores may be designed so as to achieve good load balancing, Graphite's current strategy is to distribute thread spawn request among host processes in a round robin manner. Similarly, all thread join requests (calls to the API function pthread_join()) in the application are replaced and the requests forwarded to the MCP, where the synchronization is implemented by the distribution engine. Many other API functions, such as pthread_mutex_lock() and pthread_mutex_unlock(), do not need special handling; they can be allowed to execute unmodified in the application and yield the correct results by virtue of the single, coherent simulated address space provided by Graphite.

4.4.1 Routine replacement

The threading infrastructure, as well as some other functional aspects a Graphite simulation such as message passing depend on the ability to replace calls to specific functions within the application with traps into the simulator. The Pin API function

```
RTN_ReplaceSignature (RTN replacedRtn, AFUNPTR replacementFunc, ...)
```

could normally be used for this, but the fact that an application running under Graphite runs in a simulated address space presents a problem. When this function is invoked, Pin replaces calls to the original application function with calls to a replacement function, to which arguments meant for the original function can be passed. These arguments are, however, read from the stack in the host address space, which would lead to functional problems. Instead, Graphite uses a different strategy to "trampoline over" the function to be replaced. This is done by inserting an analysis function to be called before the routine is called. This analysis function reads the function argument from the stack in the simulated address space and passes them to the replacement function. After the replacement function has finished executing, the execution context is modified to update the return value from the function, update the simulated memory to reflect any values changed by the replacement routine, and start executing at the instruction immediately following the replaced routine.

Chapter 5

Results

This chapter presents empirical results from experiments that test Graphite's speed, scalability and accuracy. The results indicate that simulations of large target architectures can be sped up by running the simulation on more physical cores, and that the speed up is achieved irrespective of whether the simulation is done on a given number of cores on a single machine or across multiple machines. The results also demonstrate that distributing the simulation across multiple machines does not impact simulation results. Finally, the chapter presents empirical evidence to show why lax synchronization makes sense. Section 5.1 describes the experimental methodology and configurations used in subsequent sections. Section 5.2 presents scaling results on a single machine as well as across a cluster of machines, including results for a 1024-core simulation. Section 5.3 shows that distributing the simulation across machines has minimal impact on simulation results. Finally, section 5.3 discusses the impact of the lax synchronization model (Section 3.4) on simulation results and demonstrates their consistency.

5.1 Experimental Setup

All experimental results provided in this section, with the exception of the single machine scaling results, were obtained on a homogenous cluster of machines. Each machine within the cluster is a dual quad core Intel(r) Xeon(r) CPU with each core running at 3.16 GHz. Each machine has 8

GB of DRAM and is running Linux with kernel version 2.6.18. Applications were compiled with `gcc` version 4.1.2 using `glibc` version 2.3.6. The machines within the cluster are connected via a Gigabit ethernet controller. This hardware is representative of current commodity server hardware.

The single machine scaling results were obtained on quad quad core Intel(r) Xeon(r) CPU machine. Each of the 16 cores on the machine runs at 3.0 GHz. Each machine has 16 GB of DRAM and runs Debian Linux with kernel version 2.6.25.20. Applications were compiled with `gcc` version 4.1.2 using `glibc` version 2.3.6.

In Table 5.1 summarizes the configuration for the target architecture used for each of the experiments discussed in this section unless otherwise noted. These parameters were chosen to match modern commodity machines.

Architectural Feature	Value
Clock frequency	1 GHz
L1 Caches	4 Kb (per core), 64 bytes per line, 64 sets, 8-way associativity, LRU replacement
L2 Cache	3 Mb (per core), 64 bytes per line, 48 sets, 24-way associativity, LRU replacement
Cache Coherence	Full map directory based
DRAM Bandwidth	5.3 GB/s
On Chip Interconnect	Mesh Network

Table 5.1: Selected Target Architecture Parameters

5.2 Simulator Scaling

Graphite is designed to simulate large, multicore architectures and therefore simulation speed and scalability of performance are first order design objectives. As the results below demonstrate, Graphite gives good simulation speed for a large and diverse set of applications, and scales well as more hardware resources are devoted to the simulation.

5.2.1 Scaling across cores on a single machine

Since Graphite is designed to allow multiple application threads in a simulation to run in parallel, it naturally parallelizes well on multicore host machines. Figure 5-1 demonstrates the speed ups

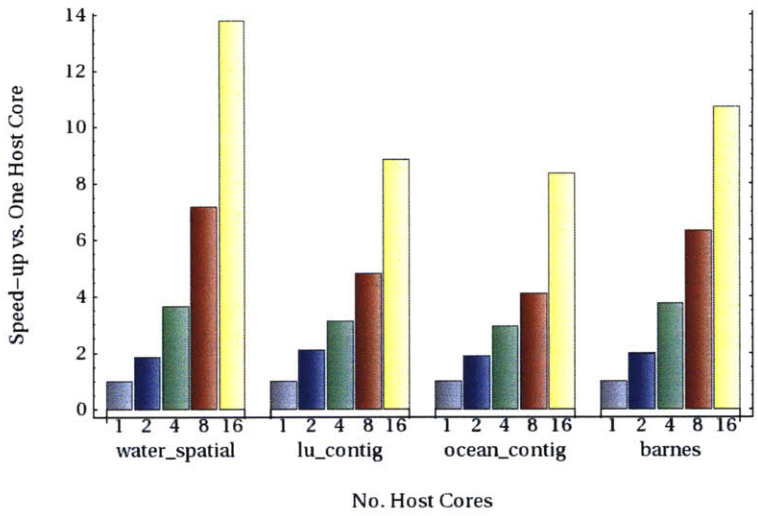


Figure 5-1: Single Machine Scaling

achieved by Graphite as more cores on a multicore machines are devoted to the simulation. These results were obtained on the 16-core machine described in Section 5.1. Figure 5-1 plots the scaling numbers for four applications from the SPLASH benchmark suite. In each case, the application has 16 target cores. All application parameters are identical for the various runs, the only thing that changes from one run to the next is the number of host cores devoted to the simulation. For comparison purposes, the run-times of each application are normalized to a single-core.

As can be seen from the results, all four applications exhibit significant simulation speed-ups as more cores are added to the simulation, with water-spatial exhibiting near-linear speedup (1.85x for 2 cores, 3.65x for 4 cores, 7.16x for 8 cores and 13.82x for 16 cores). Even in the worst case (lu-contiguous), the simulation speed when using 16 host cores is 8.81x compared to the 1 core case. These numbers demonstrates that a Graphite simulation is able to efficiently use the parallelism available in the host platform for many different applications.

5.2.2 Scaling across machines

Figure 5-2 shows similar scaling results for simulations distributed across a cluster of machines. Each machine in the cluster is the 8-core machine as described in Section 5.1, with each core running at 3.16 GHz and the total DRAM being 8 GB. The same set of applications as in sub-

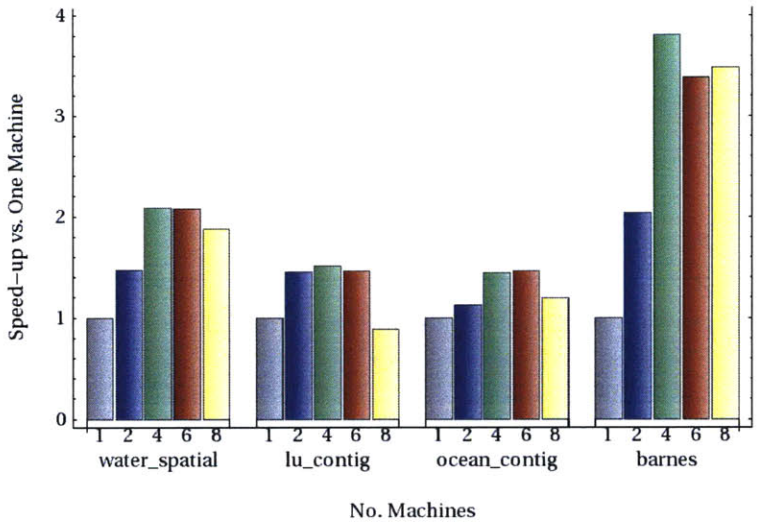


Figure 5-2: Scaling Across Multiple Machines

section 5.2.1 are used, except that each simulation has 32 threads in the application. Performance numbers are given for 1, 2, 4, 6 and 8 machines, which corresponds to 8, 16, 32, 48 and 64 physical cores, respectively.

As can be seen from Figure 5-2, adding more machines to the simulation results in significant performance gains for all applications. For example, performance scaling for barnes-hut is nearly linear (2.03x for 2 machines, 3.80x for 4 machines). Event in the worst case (ocean-contiguous), simulation speeds up by 1.44x when using 4 machines instead of 1.

The results demonstrate that performance improves as more cores are added till the total number of cores becomes 32, beyond which performance levels off or even degrades slightly. Since the application has 32 threads, this is expected: as more hardware resources are employed, performance improves until all the parallelism present in the application is utilized. Beyond that, adding more cores does not provide any additional benefit, while the added communication costs (since the cores and data are distributed across more machines) may cause the performance to degrade.

These results suggest that since the maximum speedup is achieved when the ratio of target to host cores become 1, performance scaling would continue to a larger number of cores with a larger simulation, as demonstrated in subsection 5.2.3.

The most significant hurdle in scaling across machines is the increased communication costs.

Number of machines	1 physical core	2 physical cores	4 physical cores	8 physical cores
1	3904.13	2121.81	1186.67	914.32
2	-	2209.53	1193.43	921.89
4	-	-	1221.44	962.62
8	-	-	-	980.01

Table 5.2: ocean-contiguous: simulation runtime(seconds) for different host configurations

Number of machines	1 physical core	2 physical cores	4 physical cores	8 physical cores
1	31528.99	15608.41	9525.37	6869.06
2	-	15621.43	9578.36	6902.32
4	-	-	9607.81	6963.12
8	-	-	-	6998.18

Table 5.3: lu-contiguous: simulation runtime(seconds) for different host configurations

Number of machines	1 physical core	2 physical cores	4 physical cores	8 physical cores
1	3852.24	2060.02	1005.81	536.52
2	-	2089.83	1043.17	574.28
4	-	-	1024.28	593.38
8	-	-	-	604.39

Table 5.4: water-spatial: simulation runtime(seconds) for different host configurations

Number of machines	1 physical core	2 physical cores	4 physical cores	8 physical cores
1	16019.01	8090.41	4055.45	2554.87
2	-	8112.57	4095.03	2578.74
3	-	-	4121.09	2563.46
4	-	-	-	2591.66

Table 5.5: barnes-hut: simulation runtime(seconds) for different host configurations

Threads in a Graphite simulation can communicate very often (memory references, user level messages etc.), and the increased communication costs between machines can become a serious bottleneck for the simulation. To evaluate the costs and benefits associated with distributing the simulation across multiple machines, the following experiment was conducted: simulation runtime for four benchmarks from the SPLASH benchmark suite were measured on different numbers of physical cores, where the cores could be on the same machine or could be distributed across multiple machines. For example, a simulation could be run on 2 physical cores located on the same machine or on two different machines. Each of the applications had 32 threads. Table 5.2, Table 5.3, Table 5.4 and Table 5.5 summarize the results.

These results demonstrate that the simulation run times are minimally affected by the distribution of the cores, the total number of cores is all that matters. This is a little surprising, since one would expect simulation to be slower when the cores are distributed across multiple machines due to higher communication costs. It is likely that the communication latencies were largely hidden by the fact that other threads running on a core could make progress while one was waiting for a message. Also, running a simulation across multiple machines might offer other advantages such as larger aggregate cache sizes and greater aggregate bandwidth to memory.

To summarize, Graphite displays good simulation speed and scalability. Experimental results indicate that simulations of many-core architectures with large numbers of cores will benefit greatly from distributed execution, with simulation speeds scaling significantly up to a large number of physical cores.

5.2.3 Scaling with Large Target Architectures

This section presents performance results for a large target architecture containing 1024 cores and explores the scaling of such simulations. Figure 5-3 shows the run-time in seconds of a 1024-core `matrix-multiply` kernel across different numbers of machines (each machine is a dual quad-core Xeon, as described in Section 5.1). The run-time in each case is split into two components: initialization and application. The former is a one-time simulation overhead due to the

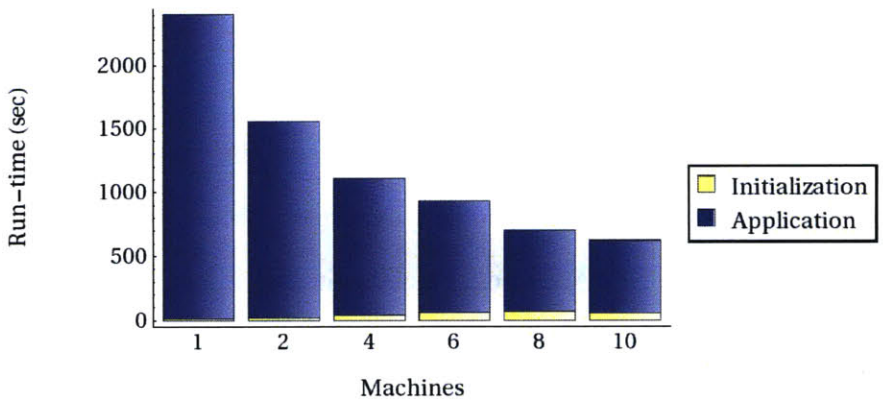


Figure 5-3: Run-times of `matrix-multiply` kernel with 1024 threads mapped onto 1024 target cores across different no. of machines.

initialization of the single global address space, thread stacks, et cetera (Section 4.2). Shutdown cost is negligible and not shown in the figure. Application time accounts for the full run-time of the the application — both sequential and parallel regions. The `matrix-multiply` kernel was run with large matrices (102,400 elements) so that most of the time was spent in the parallel region, even with 1024 worker threads. `matrix-multiply` was chosen because it scales well to large numbers of threads, while still having frequent synchronization via messages with neighbors.

This graph shows steady performance improvement up to ten machines. Application performance improves by a factor of 4.23 with ten machines compared to a single machine. Speed-up is consistent as machines are added, closely matching a linear curve. Adding machines introduces initialization overhead, however, as initialization must be done sequentially for each process. So although application time scales well, it is countered by increasing simulation overhead. However, for a large, compute-intensive application, the initialization overhead would be negligible.

We expect scaling to continue as more host cores are added despite increased overhead, since SPLASH showed optimal scaling when the number of target cores matched the number of host cores (subsection 5.2.2). It is unlikely that the optimal performance for `matrix-multiply` would lie at 1024 host cores, however, since the number of machines required would introduce high initialization cost. But since the maximum number of host cores in this study is 80 and there are 1024 target cores, we expect performance to improve well beyond 10 machines.

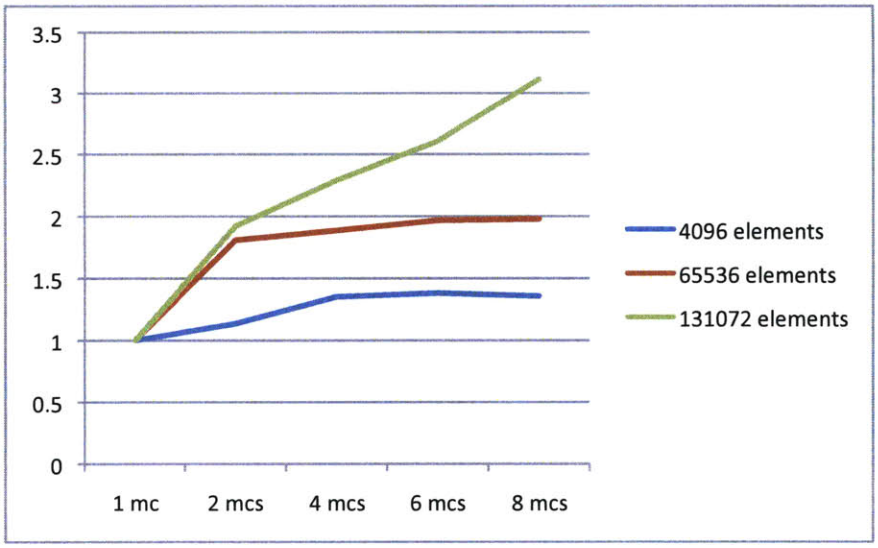


Figure 5-4: Performance scaling across machines of `matrix-multiply` for various problem sizes

Since communication latencies are likely to be a major cost when distributing the simulation across multiple machines, applications with a large computation to communication ratio are likely to scale better. This is confirmed by the results presented in Figure 5-4, which plots simulation speeds for the `matrix-multiply` application for various problem sizes (matrices with 4096, 65536 and 131072 elements, respectively). The computation to communication ratio is higher for larger problem sizes, as each step in the algorithm involves the threads working on a larger submatrix. Run times for each configuration are normalized to the run time on a single machine for purposes of comparison. The results demonstrate that simulation speeds up for each problem size as more machines are added to the simulation. As expected, scaling trends get better for larger problem sizes (which correspond to better computation to communication ratios). For the smallest problem size, the rate of performance improvement is small and flattens out beyond 4 machines (with a maximum speed up of 1.38x over the 1 machine case). For the largest problem size (131072 elements), performance jumps significantly as more machines are added to the simulation, and does not show any flattening. Thus for applications for very little communications between threads, performance is likely to keep increasing as more host cores are devoted to the simulation. Performance gains for applications with significant inter-thread communication, on the other hand,

are likely to be bound by communication costs.

5.2.4 Simulator Overhead

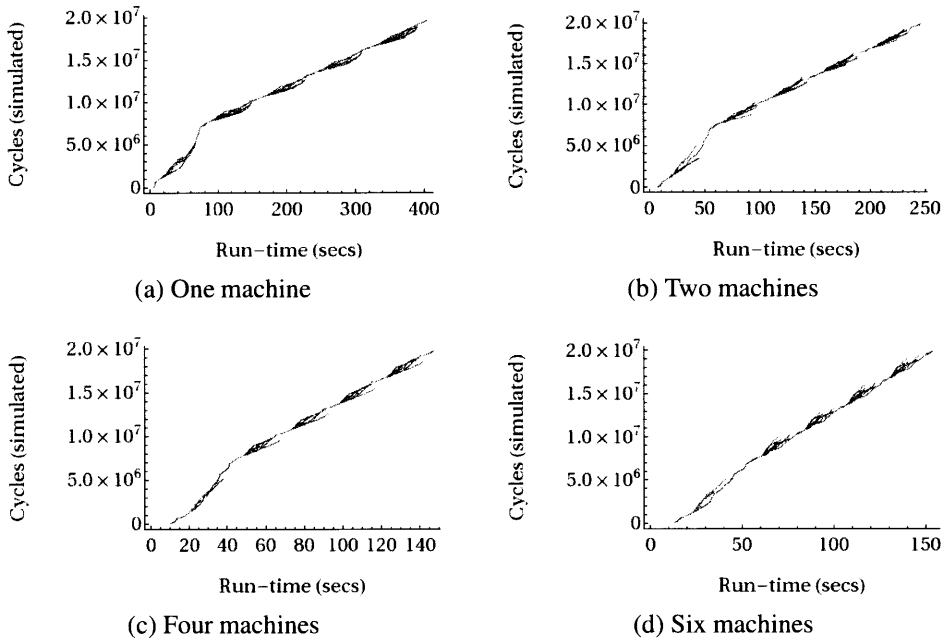
Application	Native (s)	1 Machine (s)	4 Machines (s)	Slowdown (1 machine)	Slowdown (4 machine)	Speed-up
barnes	1.38	8866	2331	6424	1689	3.80
cholesky	13.96	26834	2172	1922	155	12.4
fft	0.1	88	55	880	550	1.60
fmm	1.213	415	94	342	77.5	4.41
lu_contiguous	0.151	4139	1961	27410	12986	2.11
lu_non_contiguous	0.159	1139	610	7163	3836	1.87
ocean_contig	0.32	2163	1498	6759	4681	1.44
ocean_non_contiguous	0.378	2004	781	5301	2066	2.57
water_spatial	0.121	992	476	8198	3933	2.08
radix	0.1	568	253	5680	2530	2.25
Mean	-	-	-	7008	3250	2.15

Table 5.6: Multi-Machine Scaling Results

Table 5.6 shows simulator performance for several benchmarks from the SPLASH-2 suite. The target architecture is as described in Table 5.1. The simulations employ default problem sizes for each application as described in [30]. The host machines configuration is as described in Section 5.1. The number of target cores and worker threads is set to 32 for each experiment.

The first column, labeled "Native", represents the native execution time of each application on a single host machine. The next two columns, labeled "1 Machine" and "4 Machines", represent the overall simulation runtime in seconds for each application distributed across 1 and 4 machines, respectively. The next two columns represent the slowdown factor from running the application in Graphite vs. native (simulated runtime / native runtime). The last column shows the speed-up going from 1 to 4 machines.

The two columns labeled "1 Machine (s)" and "4 Machines (s)" show the effect of distributing a simulation across a cluster. From this data as well as Figure **??**, it is clear that there is substantial gain from parallelizing the simulation across a cluster. The speed-up is heavily application dependent, particularly on the algorithmic scalability of the application and how much it saturates shared resources like off-chip memory bandwidth. Some applications, such as `cholesky`, show scaling much greater than linear; others, such as `fft`, `lu_contig` and `ocean_contig`, show



(a) One machine (b) Two machines

(c) Four machines (d) Six machines

Figure 5-5: Progress of threads during a single simulation of `fmm` across different machines, shown as simulated cycles vs. real time (seconds).

poor scaling. The mean speed-up for four machines is 2.15, after which benefits of scaling these problem sizes diminishes.

In addition to scalability, it is important for Graphite to achieve high enough performance that larger applications complete in a reasonable amount of time. For the applications studied, the slowdown lies between a few 100 and a few 1000x. For most applications the slowdown is less than 4000x, but it can be as low as 77x or as high as 13,000x. Therefore for most applications, Graphite allows realistic problem sizes to be run in a useful amount of time.

5.3 Lax synchronization

Although Graphite is not cycle-accurate, Figure 5-5 shows that application threads remain reasonably synchronized throughout the simulation. Different colors represent different application threads. Simulated cycle time of each thread is plotted on the y-axis, and real-time is plotted on the x-axis. The application is `fmm`, a member of the SPLASH benchmark suite. The plot is discontinuous as threads synchronize with each other and their clocks are forwarded, as discussed earlier

No. machines	One	Two	Four	Six	Mean
Mean	$1.96 \cdot 10^7$	$1.98 \cdot 10^7$	$1.98 \cdot 10^7$	$2.02 \cdot 10^7$	$1.99 \cdot 10^7$
Standard deviation	$7.72 \cdot 10^4 (0.4\%)$	$2.29 \cdot 10^5 (1.1\%)$	$1.50 \cdot 10^5 (0.8\%)$	$4.43 \cdot 10^5 (2.2\%)$	$3.27 \cdot 10^7 (1.7\%)$

Table 5.7: Simulated run-times for multiple runs of `fmm` on different numbers of machines.

Number of machines	ocean	lu	water
1	$6.43 \cdot 10^9$	$5.41 \cdot 10^8$	$6.42 \cdot 10^7$
2	$6.78 \cdot 10^9$	$5.74 \cdot 10^8$	$6.42 \cdot 10^7$
4	$6.92 \cdot 10^9$	$5.43 \cdot 10^8$	$6.42 \cdot 10^7$
8	$6.87 \cdot 10^9$	$4.77 \cdot 10^8$	$6.43 \cdot 10^7$

Table 5.8: Simulated run times for different host configurations

(Section 3.4).

First, note that the shapes of the graphs are qualitatively similar. There is a start-up phase and four subsequent worker phases. During these worker phases, application threads show some deviation in their local clocks before synchronization at the beginning of the next phase. As the number of machines increases, the deviation amongst clocks increases as well. This has negligible impact on simulation results, as each run finishes at nearly identical simulated clock values.

`fmm` was run 5 times for each number of machines, and the simulation results were compared for consistency, as shown in Table 5.7. The means are consistent, showing difference of 3% going from 1 to 6 machines. Standard deviation tends to increase with more machines, as expected. The standard deviation for each set of runs is small — on average, less than 2% of total run-time. In fact, because `fmm` is a multithreaded application with non-deterministic run-time, this standard deviation is somewhat less than what is observed for `fmm` running natively. In that case, on average it takes 7.33 seconds with a standard deviation of 0.31 seconds (4.2%).

Table 5.8 summarizes the simulated run times for 3 applications from the SPLASH benchmark suite running on different numbers of machines. As can be seen, the results are very consistent and distributing the simulation across multiple host machines does not introduce any systematic error to the simulation results.

Chapter 6

Related Work

Because simulation is such an important tool for computer architects, a wide variety of different simulators and emulators exists. Conventional sequential simulators/emulators include SimpleScalar [2], RSIM [13], SimOS [25], Simics [19], and QEMU [3]. Some of these are capable of simulating parallel target architectures but all of them execute sequentially on the host machine. FaCSim[17] solves the opposite problem, simulating a sequential target on a parallel host. However, the parallelism is very limited and consists of breaking the target processor's pipeline into two pieces.

The projects most closely related to Graphite are parallel simulators of parallel target architectures including: SimFlex [28], GEMS [20], BigSim [31], FastMP [15], Wisconsin Wind Tunnel (WWT) [24], Wisconsin Wind Tunnel II (WWT II) [22], and those described by Chidester and George [6], and Penry et al. [23].

SimFlex and GEMS both use an off-the-shelf sequential emulator (Simics) for functional modeling plus their own models for memory systems and core interactions. Because Simics is a closed-source commercial product it is difficult to experiment with different core architectures. GEMS uses their timing model to drive Simics one instruction at a time which results in much lower performance than Graphite. SimFlex avoids this problem by using statistical sampling of the application but therefore does not observe its entire behavior. Chidester and George take a similar

approach by joining together several copies of SimpleScalar using MPI. They do not report absolute performance numbers but SimpleScalar is typically slower than the direct execution used by Graphite.

BigSim and FastMP assume distributed memory in their target architectures and do not provide coherent shared memory between the parallel portions of their simulators. Graphite permits study of the much wider and more interesting class of applications that assume a shared memory model.

WWT is one of the earliest parallel simulators but requires applications to use an explicit interface for shared memory and only runs on CM-5 machines, making it impractical for modern usage. Graphite has several similarities with WWT II. Both use direct execution, and provide shared memory across a cluster of machines. However, WWT II does not model anything other than the target memory system and requires applications to be modified to explicitly allocate shared memory blocks. Graphite also models compute cores and communication networks and implements a transparent shared memory system. In addition, WWT II uses a very different quantum-based synchronization scheme rather than using loosely synchronized local clocks.

Penry et al. provide a much more detailed, low-level simulation and are targeting hardware designers. Their simulator, while fast for a cycle-accurate hardware model, does not provide the performance necessary for rapid exploration of different ideas or software development.

The problem of accelerating slow simulations has been addressed in a number of different ways other than large-scale parallelization. ProtoFlex [9], FAST [7], and HASim [11] all use FPGAs to implement timing models for cycle-accurate simulations. ProtoFlex and FAST implement their functional models in software while HASim implements functional models in the FPGA as well. These approaches require the user to buy expensive special-purpose hardware while Graphite runs on commodity Linux machines. In addition, it is far more difficult to implement a new model in an FPGA than in software, making it harder to quickly experiment with different designs.

Other simulators improve performance by modeling only a portion of the total execution. FastMP [15] estimates performance for parallel workloads with no memory sharing (such as SPECrate) by carefully simulating only some of the independent processes and using those re-

sults to model the others. Finally, simulators such as SimFlex [28] use statistical sampling by carefully modeling short segments of the overall program run and assuming that the rest of the run is similar. Although Graphite does make some approximations, it differs from these projects in that it observes and models the behavior of the entire application execution.

The idea of maintaining independent local clocks and using timestamps on messages to synchronize them during interactions was pioneered by the Time Warp system [14] and used in the Georgia Tech Time Warp [10] and BigSim [31]. However, all of these systems assume that perfect ordering must be maintained and rollback when the timestamps indicate out-of-order events. To our knowledge, our lax synchronization technique has not been previously used.

Separating functional models from timing models is a well-established technique used in many simulators including: FastSim [26], TimingFirst [21], GEMS [20], tsim [8], Asim [12], HASim [11], FAST [7], and ProtoFlex [9].

TreadMarks [1] implements a generic distributed shared memory system across a cluster of machines. However, it requires the programmer to explicitly allocate blocks of memory that will be kept consistent across the machines. This requires applications that assume a single shared address space (*e.g.,* pthread applications) to be rewritten to use the TreadMarks interface. Graphite operates transparently, providing a single shared address space to off-the-shelf applications.

Chapter 7

Future Work

As described in the preceding chapters, Graphite presents a novel way to improve simulation speeds for simulation of many-core architectures with a large number of cores. It also opens up opportunities for future work on the simulator itself, many of which would add significantly to Graphite's capabilities. Many other directions of future research described here are more open-ended, and are closely related to research questions in future many-core systems.

As mentioned in Chapter 3, Graphite's lax synchronization model presents many interesting challenges in modeling the cost of events in target architectures. Most of the challenges stem from the fact that order of events as seen in real time may not be the same as the order of events as seen in simulated time. This presents problems particularly in modeling contention, such as in the network or in the memory controller. For example, network packets that would contend at a switch in simulated time may not be seen at the same wall-clock time. Graphite implements simple queuing models to model contention in such cases, where history over a moving window in simulated time is used to approximate the state of the system at any given time. However, these contention models can prove very fragile. Designing novel schemes for modeling contention presents an interesting direction of future research.

A related issue is that of trading performance for accuracy. At the moment, application threads only synchronize at a small number of specific events in the application. One could design a

scheme where threads additionally synchronize after a certain number of cycles, thus ensuring more accuracy at the cost of simulation speed. The major challenge in implementing such a scheme is preventing deadlocks, *e.g.* when a thread is not making any forward progress because it is waiting for communication from another thread that is waiting to synchronize with the first thread.

Another extension that would provide more opportunities to trade off accuracy for performance is support for hot-swappable modules. At the moment, performance models to be used in a simulation are specified once, at the start of the simulation. Adding support for hot-swappable module would allow a user to use a simpler, less accurate performance models during phases of program execution that are less interesting, thus speeding up the simulation.

Graphite currently only supports 32-bit applications. This can be a serious limitation for simulating target architectures with a very large number of cores, since one is limited to a 32-bit address space. Implementing support for 64-bit applications in Graphite would allow Graphite to simulate much larger target architectures than is currently possible. While this would involve minimal changes in Graphite's performance models, it would require changes to many of Graphite's functional aspects, *e.g.* handling of system calls and function calls, and redirection of memory references.

Finally, Graphite also presents some open-ended research questions that are of broader significance, *e.g.* the placement and live migration of application thread for load balancing, as well as the optimal distribution of addresses among the various DRAM directories.

Chapter 8

Conclusion

In conclusion, Graphite enables fast simulation of multicore architectures with a large number of cores by leveraging the parallelism offered by today's multicore machines and distributing the simulation over multiple hosts. Experimental results indicate that Graphite scales well, both across cores in a single machine as well as across multiple machines. Furthermore, results indicate that for large target applications, performance scaling continues to a very large number of physical cores. Graphite presents a very low simulation overhead (mean slowdown is 3250x across applications in the SPLASH benchmark suite). Furthermore, simulation results are unaffected by the host configuration. It's speed makes it a useful tool for rapid high-level architectural exploration as well as software development for future multicore systems.

Bibliography

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb 1996.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[3] F. Bellard. QEMU, a fast and portable dynamic translator. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2005.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 2008.

[5] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization*, San Francisco, Mar 2003.

[6] M. Chidester and A. George. Parallel simulation of chip-multiprocessor architectures. *ACM Trans. Model. Comput. Simul.*, 12(3):176–200, 2002.

[7] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 249–261, 2007.

[8] S. Cho, S. Demetriades, S. Evans, L. Jin, H. Lee, K. Lee, and M. Moeng. TPTS: A Novel Framework for Very Fast Manycore Processor Architecture Simulation. In *ICPP'08: The 37th International Conference on Parallel Processing*, pages 446–453, Sept 2008.

[9] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi. ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 2(2):1–32, 2009.

[10] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A Time Warp System for Shared Memory Multiprocessors. In *WSC '94: Proceedings of the 26th conference on Winter simulation*, pages 1332–1339, 1994.

[11] N. Dave, M. Pellauer, and J. Emer. Implementing a functional/timing partitioned microprocessor simulator with an FPGA. In *2nd Workshop on Architecture Research using FPGA Platforms (WARFP 2006)*, Feb 2006.

[12] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.

[13] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: Simulating shared-memory multiprocessors with ilp processors. *Computer*, 35(2):40–49, 2002.

[14] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[15] S. Kanaujia, I. E. Papazian, J. Chamberlain, and J. Baxter. FastMP: A multi-core simulation methodology. In *MOBS 2006: Workshop on Modeling, Benchmarking and Simulation*, June 2006.

[16] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[17] J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S. Han. FaCSim: A fast and cycle-accurate architecture simulator for embedded systems. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*, pages 89–100, 2008.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, June 2005.

[19] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[20] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[21] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 108–116, 2002.

[22] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus. Wisconsin Wind Tunnel II: A fast, portable parallel architecture simulator. *IEEE Concurrency*, 8(4):12–20, Oct–Dec 2000.

[23] D. A. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. I. August, and D. Connors. Exploiting parallelism and structure to accelerate the simulation of chip multi-processors. In *HPCA'06: The Twelfth International Symposium on High-Performance Computer Architecture*, pages 29–40, Feb 2006.

[24] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The wisconsin wind tunnel: virtual prototyping of parallel computers. In *SIGMETRICS '93: Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 48–60, 1993.

[25] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(4):34–43, Winter 1995.

[26] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 283–294, 1998.

[27] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffman, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proceedings of the International Symposium on Computer Architecture*, pages 2–13, June 2004.

[28] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, July-Aug 2006.

[29] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown, and A. Agarwal. On-chip interconnection architecture of the Tile processor. *IEEE Micro*, 27(5):15–31, Sept-Oct 2007.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, June 1995.

[31] G. Zheng, G. Kakulapati, and L. V. Kalé. BigSim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, page 78, Apr 2004.