Graphite: A Distributed Parallel Simulator for Multicores

Jason E. Miller, Harshad Kasture, George Kurian,
Nathan Beckmann, Charles Gruenwald III,
Christopher Celio, Jonathan Eastep, and Anant Agarwal

Graphite: A Distributed Parallel Simulator for Multicores

Jason E. Miller, Harshad Kasture, George Kurian, Nathan Beckmann, Charles Gruenwald III,
Christopher Celio, Jonathan Eastep, and Anant Agarwal

Massachusetts Institute of Technology, Cambridge, MA

Abstract

This paper introduces the open-source Graphite distributed parallel multicore simulator infrastructure. Graphite is designed from the ground up for exploration of future multicore processors containing dozens, hundreds, or even thousands of cores. It provides high performance for fast design space exploration and software development for future processors. Several techniques are used to achieve this performance including: direct execution, multi-machine distribution, analytical modeling, and lax synchronization. Graphite is capable of accelerating simulations by leveraging several machines. It can distribute simulation of an off-the-shelf threaded application across a cluster of commodity Linux machines with no modification to the source code. It does this by providing a single, shared address space and consistent single-process image across machines. Graphite is designed to be a simulation framework, allowing different component models to be easily replaced to either model different architectures or tradeoff accuracy for performance.

We evaluate Graphite from a number of perspectives and demonstrate that it can simulate target architectures containing over 1000 cores on ten 8-core servers. Performance scales well as more machines are added with near linear speedup in many cases. Simulation slowdown is as low as $41\times$ versus native execution for some applications.

The Graphite infrastructure and existing models will be released as open-source software to allow the community to simulate their own architectures and extend and improve the framework.

1 Introduction

Simulation is a key technique both for the early exploration of new processor architectures and for advanced software development for upcoming machines. However, poor simulator performance often limits the scope and depth of the work that can be performed. This is especially true for simulations of future multicore processors where the enormous computational resources of dozens, hundreds, or even thousands of cores must be multiplexed onto the much smaller number of cores available in current machines. In fact, the majority of simulators available today are not parallel at all [3, 17, 30, 23, 4], potentially forcing a single core to perform all the work of hundreds of cores.

Although cycle-accurate simulators provide extremely accurate results, the overhead required for such detailed modeling leads to very slow execution (typically between 1 KIPS and 1 MIPS [11] or about $1000\times$ to $100,000\times$ slowdown). In the past, this has limited architectural evaluations to application kernels or scaled-back benchmarks suites [20, 5]. To perform more realistic evaluations, researchers are increasingly interested in running larger, more interactive applications. These types of studies require slowdowns of about $100\times$ to achieve reasonable interactivity [13]. This level of performance is not achievable with today's sequential, cycle-accurate simulators.

Another compelling use of simulation is advanced software research. Typically software lags several years behind hardware, *i.e.,* it takes years before software designers are able to take full advantage of new hardware architectures. In many cases, the hardware has actually moved on to the next generation before the software catches up. High performance simulators can help break this pattern by allowing innovative software research and development (*e.g.,* operating systems, languages, runtime systems, applications) for future architectures. With current industry trends, it is now clear that processors with hundreds or thousands of cores will eventually be available. It is also clear that the computing community is not able to fully utilize these architectures. Research on this front cannot afford to wait until the hardware is available. Existing simulators are not up to this

task because of the difficulty of simulating such large chips on existing machines.

Graphite is a new parallel, distributed simulator infrastructure designed to enable rapid high-level architectural evaluation and basic software development for future multicore architectures. It provides both functional and performance modeling for cores, on-chip networks, and memory subsystems including cache heirarchies with full cache coherence. The design of Graphite is modular, allowing the different models to be easily replaced to simulate different architectures or tradeoff performance for accuracy. Graphite runs on commodity Linux machines and can execute unmodified pthread applications. Graphite will be released to the community as open-source software to foster research and software development for future multicore processors.

A variety of techniques are used to deliver the performance and scalability needed to perform useful evaluations of large multicores including: direct execution, multi-machine distribution, analytical modeling and lax synchronization.

For increased performance, functional modeling of the computational cores is provided primarily through direct native execution on the host machine. A dynamic binary translator is used to add new functionality (*e.g.,* new instructions or a direct core-to-core messaging interface) and intercept operations that require action from the simulator (*e.g.,* memory operations that feed into the cache model) [27].

Graphite is a "multicore-on-multicore" simulator, designed from the ground up to leverage the power and parallelism of current multicore machines. However, it also goes one step further, allowing an individual simulation to be distributed across a cluster of servers to accelerate simulation and enable the study of large-scale multicore chips. This ability is completely transparent to the application and programmer. Threads in the application are automatically distributed to cores of the target architecture spread across multiple host machines. The simulator maintains the illusion that all of the threads are running in a single process with a single shared address space. This allows the simulator to run off-the-shelf parallel applications on any number of machines without having to recompile the apps for different configurations.

Graphite is not intended to be completely cycle-accurate but instead uses a collection of models and techniques to provide a good estimate of performance and various machine statistics. Instructions and events from the core, network, and memory subsystem functional models are passed to analytical timing models that update individual local clocks in each core. The local clocks are synchronized using message timestamps when cores interact (*e.g.,* through synchronization or messages) [36]. However, to reduce the time wasted on synchronization, Graphite does not strictly
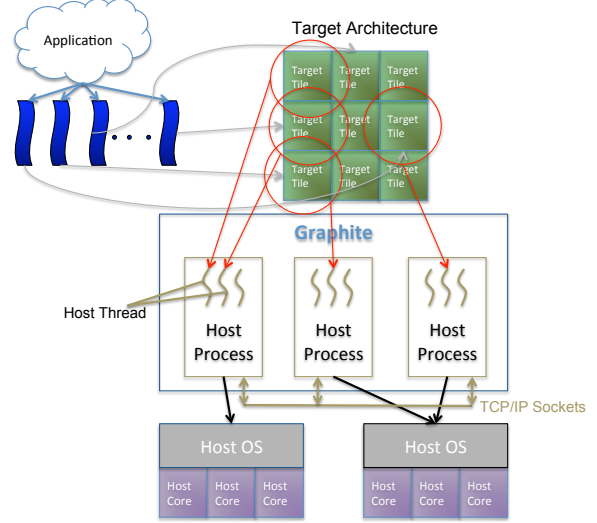

Figure 1: High-level architecture. Graphite consists of one or more host processes distributed across machines and working together over sockets. Each process runs a subset of the simulated tiles, one host thread per simulated tile.

enforce the ordering of all events in the system [9]. In certain cases, timestamps are ignored and operation latencies are based on the ordering of events during native execution rather than the precise ordering they would have in the simulated system (see Section 3.6 on Lax Synchronization).

Graphite has been evaluated both in terms of the validity of the simulation results as well as the scalability of simulator performance across multiple cores and machines. The results from these evaluations show that Graphite scales well, has reasonable performance and provides results consistent with expectations. For the scaling study, we perform a fully cache-coherent simulation of 1024 cores across up to 10 target machines and run applications from the SPLASH benchmark suite. The slowdown versus native execution is as low as $41\times$ when using 8 machines, indicating that Graphite can be used for realistic application studies.

The remainder of this paper is structured as follows. Section 2 describes the architecture of Graphite. Section 3 discusses the implementation of Graphite in more detail. Section 4 evaluates the accuracy, performance, and scaling of the simulator. Section 5 discusses related work and, finally, Section 6 summarizes our findings.

2 System Architecture

Graphite is an application-level simulator for tiled multicore architectures. A simulation consists of executing a multi-threaded application on a target multicore architecture defined by the simulator's models and runtime configuration parameters. The simulation runs on one or more host

machines, each of which may be a multicore machine itself. Figure 1 illustrates how a multi-threaded application running on a target architecture with multiple tiles is simulated on a cluster of host machines. Graphite maps each thread in the application to a tile of the target architecture and distributes these threads among multiple host processes which are running on multiple host machines. The host operating system is then responsible for the scheduling and execution of these threads.

Figure 2a illustrates the types of target architectures Graphite is designed to simulate. The target architecture contains a set of tiles interconnected by an on-chip network. Each tile is composed of a compute core, a network switch and a part of the memory subsystem (cache hierarchy and DRAM controller) [32]. Tiles may be homogeneous or heterogeneous; however, we only examine homogeneous architectures in this paper. Any network topology can be modeled as long as each tile contains an endpoint.

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 certain aspects of the system; all that needs to be ensured is that interfaces to other modules are correctly implemented.

Figure 2b illustrates Graphite's modular design. In Graphite's front-end, a dynamic binary translator (DBT) is used to modify the application and generate events at points of interest. These events cause traps into the back-end including the core model, the memory model and the network model. The core model itself may use the network model for the modeling of certain events. The network model is built on top of the physical transport module, which provides a portable communication interface that abstracts the host architecture dependent details of intra- and inter-process communication.

Points of interest intercepted by the dynamic binary translator include: memory references, system calls, synchronization routines and user-level messages. The DBT is also used to generate a stream of instructions for core 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 [22] as the front end, although Graphite's modular design means that another dynamic translation tool such as QEMU [4] or DynamoRio [7] could be used instead.

Graphite's simulation back-end can be broadly divided into two sets of features: functional and modeling. Modeling features model various aspects of the target architecture while functional features ensure correct program behavior. The following two sections describe the back-end's modeling and functional features.

2.1 Modeling Features

As shown in Figure 2b, the Graphite backend is comprised of many modules that model various components of the target architecture. In particular, the core model is responsible for modeling the processor pipeline. The memory model is responsible for the memory subsystem (Section 3.2), which is composed of different levels of caches and 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 and routing overheads.

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 load and store operations.

The challenge in achieving good simulator performance is mainly solved by Graphite's lax synchronization model (Section 3.6.1). In this model, each target tile maintains its own local clock which runs independently of the clocks of other tiles. Synchronization between the local clocks of different tiles happens only on application synchronization events, user-level messages, and thread creation and termination events. Due to this, modeling of certain aspects of system behavior, such as network contention and DRAM queueing delays, become complicated. Section 3.6.1 talks in detail about how Graphite addresses this challenge.

2.2 Functional Features

Graphite's ability to execute an unmodified pthreaded 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 3.2 explains this in greater detail.

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

(a) Target Architecture
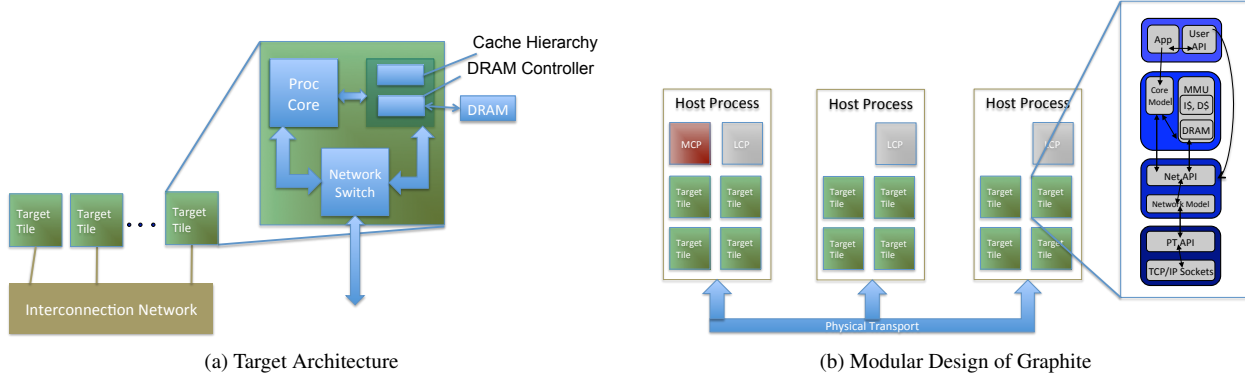
(b) Modular Design of Graphite

Figure 2: System architecture. a) Overview of the target architecture. Tiles contain a compute core, a network switch, and a node of the memory system. b) The anatomy of a Graphite simulation. Tiles are distributed among multiple processes. The app is instrumented to trap into one of three models at key points: a core model, network model, or memory system model. These models interact to model the target system. The physical transport layer abstracts away the host-specific details of inter-tile communication.

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

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 are handled automatically by virtue of the single, coherent address space.

To help address these challenges, Graphite spawns additional threads called the Master Control Program (MCP) and the Local Control Program (LCP). There is one LCP per process but only one MCP for the entire simulation. The MCP and LCP ensure the functional correctness of the simulation by providing services for synchronization, system call execution and thread management.

In addition, Graphite's network component together with the physical transport layer provide the functionality to transport data between threads on the same as well as across different processes.

3 Implementation

This section describes the design and interaction of Graphite's various models and simulation layers. It discusses the challenges of high performance parallel distributed simulation and how Graphite's design addresses them.

3.1 Core Performance Model

The core performance model is a purely modeled component of the system that manages the simulated clock local to each tile. It follows a producer-consumer design: it consumes instructions and other dynamic information produced by the rest of the system. The majority of instructions are produced by the dynamic binary translator as the application thread executes them. Other parts of the system also produce pseudo-instructions to update the local clock on unusual events. For example, the network produces a "message receive pseudo-instruction" when the application uses the network messaging API (Section 3.3), and a "spawn pseudo-instruction" is produced when a thread is spawned on the core.

Other information beyond instructions is required to perform modeling. Latencies of memory operations, paths of branches, etc. are all dynamic properties of the system not included in the instruction trace. This information is produced by the simulator back-end (*e.g.*, memory operations) or dynamic binary translator (*e.g.*, branch paths) and consumed by the core performance model via a separate interface. This allows the functional and modeling portions of the simulator to execute asynchronously without introducing any errors.

Because the core performance model is isolated from the functional portion of the simulator, there is great flexibility in implementing it to match the target architecture. Currently, Graphite supports an in-order core model with an out-of-order memory system. Store buffers, load units, branch prediction, and instruction costs are all modeled and configurable. This model is one example of many different architectural models than can be implemented in Graphite.

It is also possible to implement core models that differ

4

drastically from the operation of the functional models — *i.e.*, although the simulator is functionally in-order with sequentially consistent memory, the core performance model can be out-of-order core with a relaxed memory model. Models throughout the remainder of the system will reflect the new core type, as they are ultimately based on clocks updated by the core model. For example, memory and network utilization will reflect an out-of-order architecture because message times-tamps are generated from core clocks.

3.2   Memory System

The memory system is composed of several modules such as instruction and data caches, and DRAM controllers, each associated with 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. 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. This allows Graphite to overcome the potential bottleneck of always accessing the data at the home location which could potentially be on a different host machine.

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 struc-

tures used for ensuring functional 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 (see Section 3.6).

Graphite currently simulates a target memory architecture with L1 data and instruction caches and local unified L2 caches. Cache coherence is maintained using a directory-based MSI protocol in which the directory is uniformly distributed across all the tiles. The cache hierarchy is easily configurable and more layers may be added with ease.

3.2.1   Process Initialization and Address Space Management

Each process that participates in the simulation executes the same statically linked binary and thus has the same view of static data as every other process. However, maintaining a single address space across multiple processes presents additional challenges in process initialization and dynamic memory management. Graphite replicates the state of system memory at process start-up by copying all the data written on to the process stack by the operating system into the simulated address space before transferring control to the application. Initialization of process state, such as the setting up of thread local storage (TLS), is performed in each process participating in the simulation. Only a single process in the simulation eventually executes `main()`, while the other processes execute subsequently spawned threads as explained later in this section.

Graphite also implements certain memory management functions normally provided by the operating system. Graphite allocates a part of the address space for thread stacks, and makes sure that addresses in this range are a part of the valid host address space on each process. Additionally, Graphite implements a dynamic memory manager that services requests for dynamic memory from the application by intercepting the `brk`, `mmap` and `munmap` system calls and allocating (or deallocating) memory from designated parts of the address space. Figure 3 depicts how Graphite partitions the application address space.

3.3   Network

The network component provides high-level messaging services between tiles built on top of the lower-level transport layer (Section 3.3.1). It provides a message-passing API directly to the application, as well as serving other
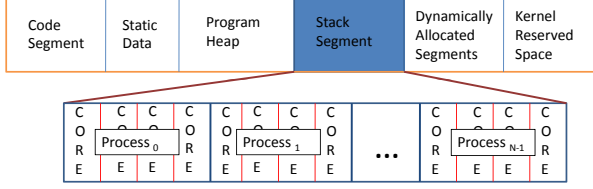
Figure 3: Segments within the application address space

components of the simulator back end, such as the memory system (Section 3.2) and system call handler (Section 3.4).

The network component maintains several distinct network models. The network model used by a particular message is determined by the message type. For instance, system messages unrelated to application behavior use a separate network model than application messages, and therefore have no impact on simulation results. The default simulator configuration also uses separate models for application and memory traffic, as is commonly done in multicore chips [32, 34]. Each network model is configured independently, allowing for exploration of new network topologies focused on particular subcomponents of the system.

The network separates functionality and modeling via the different network models. The network provides common functionality, such as the bundling of packets, multiplexing of messages, high-level interface to the rest of the system, and internal interface to the transport layer. The network models are responsible for routing packets and updating time-stamps to account for network delay. Regardless of the time-stamp of a packet, the network forwards messages immediately and delivers them in the order they are received. This means that messages will arrive out-of-order or "earlier" (in simulated cycles) than the receiving tile. This is discussed in greater detail in Section 3.6.1. However, note that functionality and modeling are not perfectly encapsulated, as the network model determines the route of a packet and consequently impacts the traffic through the transport layer.

Each network model shares a common interface. Therefore, network model implementations are swappable, and it is simple to develop new network models. Currently, Graphite supports a basic model that forwards packets with no delay (used for system messages), a mesh model that uses the number of network hops to determine latency, and another mesh model that tracks global network utilization to determine latency using an analytical contention model.

3.3.1 Transport Layer

The transport layer provides an abstraction for generic communication between tiles. 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 Consistent OS Interface

Graphite implements a system interface layer that intercepts and handles system calls in the target application. System calls require special handling for two reasons: the need to access data in the simulated address space as opposed to the host address space, and the need to maintain the illusion of a single process across multiple processes executing the target application. System calls that do not require special handling are allowed to execute directly on the host machine.

Many system calls, such as `clone` and `rt_sigaction` pass pointers to chunks of memory as input or output arguments to the kernel. Graphite intercepts such system calls, fetches the data passed in as input and modifies system call arguments to point to this data before executing it on the host machine. Any output data is copied to the simulated address space after the system call returns.

Some system calls, such as the ones that deal with file I/O, need to be handled specially to maintain a consistent process state for the target application. For example, in a multi-threaded application, threads might communicate via files, with one thread writing to a file using a `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 need not point to the same file as in the other. Simply executing the system calls on the respective host systems will not yield correct results. Instead, Graphite handles these system calls by intercepting and forwarding them along with their arguments to the MCP, where they are executed. The results are sent back to the thread that made the original system call, achieving the desired result. Other system calls, *e.g.* `open`, `fstat` etc., are handled in a similar manner.

As already described in Section 3.2, memory management system calls are also intercepted and handled by the system interface layer. System calls that are used to implement synchronization between threads, such as `futex`, are also intercepted and forwarded to the MCP, where Graphite emulates the behavior of the `futex` system call.

3.5 Threading Infrastructure

One challenging aspect of our design was seamlessly dealing with thread spawn calls across a distributed simulation. Other programming models, such as MPI, force the application programmer to be aware of distribution by allo-

cating work among processes at start-up. This design is limiting and often requires the source code of the application to be changed to account for the new programming model. Instead, Graphite presents a single-process programming model to the user while distributing the threads across different machines. This allows the user to customize the distribution of the simulation as necessary for the desired scalability, performance, and available resources.

The above parameters can be changed between simulation runs through run-time configuration options without any changes to the application code. The actual application interface is simply the pthread spawn/join interface. The only limitation to the programming interface is that the maximum number of threads at any time may not exceed the total number of cores in the chip. Currently the threads are long living, that is, they run to completion without being swapped out.

To accomplish this, the spawn calls are first intercepted at the callee. Next, they are forwarded to the MCP to ensure a consistent view of the thread-to-tile mapping. The MCP chooses an available core and forwards the spawn request to the LCP on the machine that holds the chosen tile. The mapping between tiles and processes is currently implemented by simply striping the tiles across the processes. Thread joining is implemented in a similar manner by synchronizing through the MCP.

3.6 Synchronization Models

In order to meet performance and scalability goals, Graphite allows cores to run independently with little synchronization. This is necessary to achieve desired throughput for a simulation distributed across several machines, but it also means Graphite is not cycle-accurate. Furthermore, it impacts models throughout the system, as events occur out-of-order and cycle counts may significantly differ. Graphite offers a number of synchronization models with different accuracy and performance trade-offs.

3.6.1 Lax Synchronization

The first synchronization model is *lax synchronization*. Lax synchronization is the most permissive in letting clocks differ and offers the best performance and scalability. To keep the simulated clocks in reasonable agreement, Graphite uses application events to synchronize them, but otherwise lets threads run freely. Lax synchronization is the baseline model for Graphite, and other models add additional mechanisms to it to limit the clock skew.

Lax synchronization is best viewed from the perspective of a single tile. All interaction with the rest of the simulation takes place via network messages, each of which carries a time-stamp that is initially set to the clock of the sender. These time-stamps are used to update clocks during synchronization events. A tile's clock is updated primarily when instructions executed on that tile's core are retired (Section 3.1). With the exception of memory operations (Section 3.2), these events are independent of the rest of the simulation. However, memory operations use message round-trip time to determine latency, so they do not force synchronization with other tiles. True synchronization only occurs in the following events: application synchronization such as locks, barriers, etc., receiving a message via the message-passing API (Section 3.3), and spawning or joining a thread. In all cases, the clock of the tile is forwarded to the time that the event occurred. If the event occurred earlier in simulated time, then no updates take place.

The general strategy to handle out-of-order events is to ignore simulated time and process events in the order they are received [9]. An alternative is to re-order events so they are handled in simulated-time order, but this has some fundamental problems. Buffering and re-ordering events leads to deadlock in the memory system, and is difficult to implement anyway because there is no global cycle count. Alternatively, one could optimistically process events in the order they are received and roll them back when an "earlier" event arrives, as done in BigSim [36]. However, this requires state to be maintained throughout the simulation and destroys performance. Our results show that lax synchronization still yields reasonably correct performance trends even in the presence of this error (as shown in Section 4.3).

This complicates models, however, as events are processed out-of-order. Queue modeling, e.g. at memory controllers and network switches, illustrates many of the difficulties. In a cycle-accurate simulation, a packet arriving at a queue is buffered. At each cycle, the buffer head is dequeued and processed. This matches the actual operation of the queue and is the natural way to implement such a model. In Graphite, however, the packet is processed immediately and potentially carries a time-stamp in the past or far future, so this strategy does not work.

Instead, queueing latency is modeled by keeping an independent clock for the queue. This clock represents the time in the future when the processing of all messages in the queue will be complete. When a packet arrives, its delay is the difference between the queue clock and the "global clock". Additionally, the queue clock is incremented by the processing time of the packet to model buffering.

However, because cores in the system are loosely synchronized, there is no easy way to measure progress or a "global clock". This is particularly problematic for cores with no active thread — these cores will have no local updates to their clock, and therefore no obvious reference for global progress of the simulation. However, these cores still participate in the simulation as memory controllers and network switches.

This problem is addressed by using packet time-stamps to build an approximation of global progress. A window of the most recently-seen time-stamps is kept, on the order of the number of tiles in the simulation. The average of these time stamps gives an approximation of global progress. The large window is necessary to eliminate outliers from overly influencing the result. Because messages are generated frequently (e.g., on every cache miss), this window gives an up-to-date representation of global progress even with a large window size.

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.

3.6.2 Lax with Barrier Synchronization

Graphite also supports quanta-based *barrier synchronization* (LaxBarrier), where all active threads wait on a barrier after a configurable number of cycles. This is used for validation of lax synchronization, as very frequent barriers closely approximate cycle-accurate simulation. As expected, LaxBarrier also hurts performance and scalability (Section 4.3).

3.6.3 Lax with Point-to-point Synchronization

Graphite supports a novel synchronization scheme called *point-to-point synchronization* (LaxP2P). LaxP2P aims to achieve the quanta-based accuracy of LaxBarrier without sacrificing the scalability and performance of lax synchronization. In this scheme, each tile periodically chooses another tile at random and synchronizes with it. If the clocks of the two tiles differ by more than a configurable number of cycles (called the slack of simulation), then the tile that is ahead goes to sleep for a short period of time.

LaxP2P is inspired by the observation that in lax synchronization, there are usually a few outlier threads that are far ahead or behind and responsible for simulation error. LaxP2P prevents outliers, as any thread that runs ahead will put itself to sleep and stay tightly synchronized. Similarly, any thread that falls behind will put other threads to sleep, which quickly propagates through the simulation.

The amount of time that a thread must sleep is calculated based on the real-time rate of simulation progress. Essentially, the thread sleeps for enough real-time such that its synchronizing partner will have caught up when it wakes. Specifically, let c be the difference in clocks between the tiles, and suppose that the thread "in front" is progressing at a rate of r simulated cycles per second. We approximate the thread's progress with a linear curve and put the thread

Feature	Value
Clock frequency	1 GHz
L1 caches	Private, 32 KB (per tile), 64 byte line size, 8-way associativity, LRU replacement
L2 cache	Private, 3 MB (per tile), 64 bytes line size, 24-way associativity, LRU replacement
Cache coherence	Full-map directory based
DRAM bandwidth	5.13 GB/s
Interconnect	Mesh network

Table 1: Selected Target Architecture Parameters. All experiments use these target parameters (varying the number of target tiles) unless otherwise noted.

to sleep for s seconds, where $s = \frac{c}{r}$. r is currently approximated by total progress, meaning the total number of simulated cycles over the total wall-clock simulation time.

Finally, note that LaxP2P is completely distributed and uses no global structures. It introduces less overhead than LaxBarrier and has superior scalability. Results in Section 4.3 show that LaxP2P with a carefully chosen slack encounters a slowdown of only 8% when compared to Lax synchronization and the simulated run-times of a target architecture simulated using LaxP2P are within 1.28% of that simulated using LaxBarrier.

4 Results

This section presents experimental results that demonstrate Graphite's scalability, as well as results from two architectural studies. Section 4.1 describes the methodology and configurations used in subsequent sections. Section 4.2 presents scaling results across a single-machine, across a cluster of machines, and for a 1024-tile simulation. Section 4.3 discusses the impact of synchronization models (Section 3.6) on simulation results, showing empirically that lax synchronization gives consistent results with little impact on accuracy for the studied applications. Results conclude with Section 4.4, which presents two validation studies of application characteristics in the memory subsystem.

4.1 Experimental Setup

The experimental results provided in this section were all obtained on a homogenous cluster of machines. Each machine within the cluster has dual quad-core Intel(r) X5460 CPUs running at 3.16 GHz and 8 GB of DRAM. They are running Debian Linux with kernel version 2.6.26. Applications were compiled with gcc version 4.3.2. The machines within the cluster are connected to a Gigabit ethernet switch
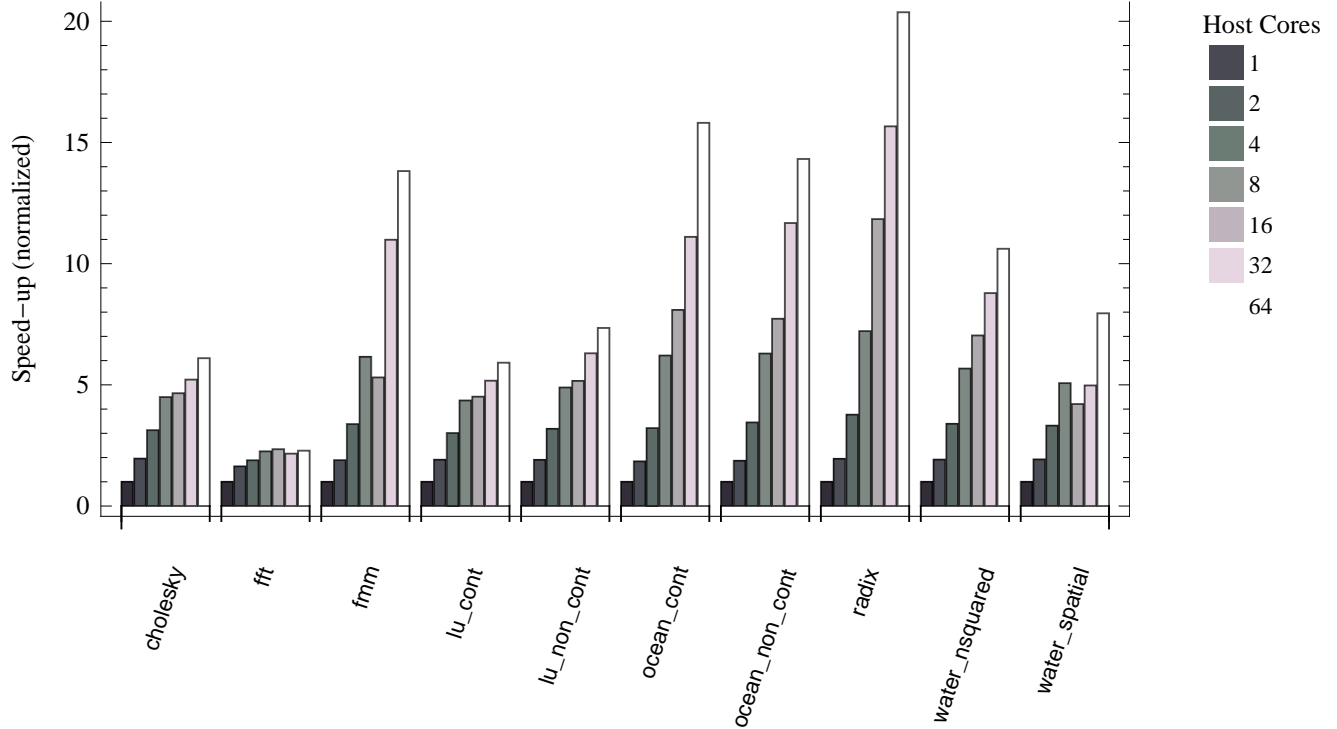
Figure 4: Scaling of SPLASH benchmarks across different numbers of cores. Speed-up is normalized to a single core. From 1 to 8 cores, simulation runs on a single machine. Above 8 cores, simulation is distributed across multiple machines.

with two trunked Gigabit ports per machine. This hardware is typical of current commodity servers.

Each of the experiments in this section uses the target architecture parameters summarized in Table 1 unless otherwise noted. These parameters were chosen to match the host architecture as closely as possible.

4.2 Simulator Performance

Single- and Multi-Machine Scaling. Graphite is designed to scale well to both large numbers of target cores and large numbers of host cores. By leveraging multiple machines, simulation of large target architectures can be accelerated to provide fast turn-around times. Figure 4 demonstrates the speedup achieved by Graphite as additional host cores are devoted to the simulation of a 32-tile target architecture. Results are presented for several SPLASH2 [35] benchmarks and are normalized to the runtime on a single host core. The results from one to eight cores are collected by allowing the simulation to use additional cores within a single host machine. The results for 16, 32, and 64 cores correspond to using all the cores within 2, 4, and 8 machines, respectively.

As shown in Figure 4, all applications except `fft` exhibit significant simulation speedups as more cores are added. The best speedups are achieved with 64 host cores

(across 8 machines) and range from about $2\times$ (`fft`) to $20\times$ (`radix`). It is notable that several apps show a significant benefit when going from 32 to 64 cores, even though the application has only 32 threads. This is due to the extra threads spawned by the simulator as well as the increased total cache and memory bandwidth available when using more host machines. Many apps will show an even greater benefit to additional machines when simulating larger target architectures.

Scaling is generally better within a single host machine than across machines due to the lower overhead of communication. Because each host machine has 8 cores, the transition to multi-machine simulations occurs at 16 host cores. Several apps (`fmm`, `ocean`, and `radix`) show nearly ideal speedup curves from one to eight cores. Some apps show a drop in performance when going from 8 to 16 cores because the additional overhead of inter-machine communication outweighs the benefits of the additional compute resources. This effect clearly depends on specific application characteristics such as algorithm, computation/communication ratio, and degree of memory sharing. If the application itself does not scale well to large numbers of cores, then there is nothing the simulator can do to improve it and performance will suffer.

These results demonstrate that Graphite is able to take advantage of large quantities of parallelism in the host plat-

Application	Native Time	Simulation			
		1 machine		8 machines	
		Time	Slowdown	Time	Slowdown
cholesky	1.99	689	346×	508	255×
fft	0.02	80	3978×	78	3930×
fmm	7.11	670	94×	298	41×
lu_cont	0.072	288	4007×	212	2952×
lu_non_cont	0.08	244	3061×	163	2038×
ocean_cont	0.33	168	515×	66	202×
ocean_non_cont	0.41	177	433×	78	190×
radix	0.11	178	1648×	63	584×
water_nsquared	0.30	742	1317×	396	1317×
water_spatial	0.13	129	966×	82	616×
Mean	-	-	1751×	-	1213×
Median	-	-	1307×	-	600×

Table 2: Multi-Machine Scaling Results. Wall-clock execution time of SPLASH-2 simulations versus native across 1 and 8 host machines. Times are given in seconds. Slowdowns (relative to native) calculated from the wall-clock times.

form to accelerate simulations. For rapid design iteration and software development, the time to complete a single simulation is more important than efficient utilization of host resources. For these tasks, an architect or programmer has to stop and wait for the results of their simulation before they can continue their work. Therefore it makes sense to apply additional machines to a simulation even when the speedup achieved is less than ideal. For bulk processing of a large number of simulations, total simulation time can be reduced by using the most efficient configuration for each application.

Simulator Overhead. Table 2 shows simulator performance for several benchmarks from the SPLASH-2 suite. The target architecture is as described in Table 1. For each application, the simulations employ problem sizes as described in [35]. The host machine's configuration is as described in Section 4.1. The number of target tiles and worker threads is set to 32 for each experiment.

The table lists the native execution time for each application on a single 8-core machine, as well as overall simulation runtimes on one and eight host machines. The slowdowns experienced over native execution for each of these cases are also presented.

The data in Table 2 demonstrates that Graphite achieves very good performance for all the benchmarks studied. The total run time for all the benchmarks is on the order of a few minutes, with a median slowdown of 600× over native execution. This high performance makes Graphite a very useful tool for rapid architecture exploration and software development for future architectures.

As can be seen from the table, the speed of the simulation relative to native execution time is highly application dependent, with the simulation slowdown being as low as 41× for
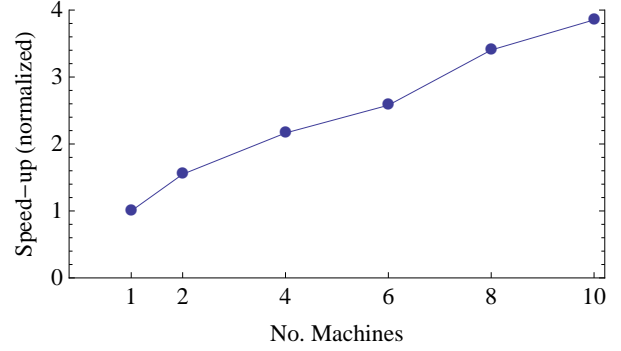

Figure 5: (Run-times of matrix-multiply kernel with 1024 threads mapped onto 1024 target tiles across different no. of host machines.

fmm and as high as 3930× for fft. This depends, among other things, on the computation-to-communication ratio for the application: applications with a high compution-to-communication ratio are able to more effectively parallelize and hence show higher simulation speeds. While the simulation of most benchmarks is slower than native execution by a factor of only a few hundreds, the few outliers with higher slowdowns pull up the mean slowdown factor to 1213×.

Scaling with Large Target Architectures. This section presents performance results for a large target architecture containing 1024 tiles and explores the scaling of such simulations. Figure 5 shows the run-time in seconds of a 1024-thread matrix-multiply kernel running across different numbers of host machines. 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. Performance improves by a factor of 3.85 with ten machines compared to a single machine. Speed-up is consistent as machines are added, closely matching a linear curve. We expect scaling to continue as more machines are added, as the number of host cores is not close to saturating the parallelism available in the application. However, this is countered by initialization overhead, as initialization must be done sequentially for each process. Therefore, scaling will not increase continuously up to 1024 host cores.

4.3 Lax synchronization

As explained previously in Section 3.6, Graphite supports several synchronization models, namely *lax synchro-*

	Lax		LaxP2P		LaxBarrier	
	1mc	4mc	1mc	4mc	1mc	4mc
Run-time	1.0	0.55	1.10	0.59	1.82	1.09
Scaling	1.80		1.84		1.69	
Error (%)	7.56		1.28		1.31	
CoV (%)	0.58		0.31		0.09	

Table 3: Mean performance and accuracy statistics for data presented in Figure 6. Data is averaged over ten runs of three SPLASH2 benchmarks. Scaling is the performance improvement going from 1 to 4 host machines.

nization and its *barrier* and *point-to-point* variants, to mitigate the clock skew between different target cores and increase the accuracy of the observed results. This section provides simulator performance and accuracy results for the three models, and shows the trade-offs offered by each.

Simulator performance. We now present performance results for a variety of benchmarks. Figure 6a and Table 3 illustrate the simulator performance (wall-clock simulation time) of the three synchronization models using three SPLASH2 [35] benchmarks. Each simulation is run on one and four host machines. The barrier interval was chosen as 1,000 cycles to give very accurate results. The slack value for *LaxP2P* was chosen to give a good trade-off between performance and accuracy, which was determined to be 100,000 cycles. Results are normalized to the performance of *Lax* on one host machine.

We observe that *Lax* outperforms both *LaxP2P* and *LaxBarrier* due to its lower synchronization overhead. Performance of *Lax* also increases considerably when going from one machine to four machines (1.8×).

LaxP2P performs slightly worse than *Lax*. It shows an average slowdown of 1.09× and 1.07× when compared to *Lax* on one and four host machines respectively. *LaxP2P* shows good scalabilty with an average performance improvement of 1.84× when the number of host machines is increased from one to four. This is mainly due to the distributed nature of synchronization in *LaxP2P*. Unlike *LaxBarrier*, *LaxP2P* does not force all target tiles to synchronize at regular intervals but instead performs synchronization only between a randomly chosen pair of target tiles. This enables *LaxP2P* to take advantage of the increased number of host cores when going to four machines.

LaxBarrier performs poorly as expected. It encounters an average slowdown of 1.82× and 1.94× when compared to *Lax* on one and four host machines respectively. Although the performance improvement of *LaxBarrier* when going from one to four host machines is comparable to the other schemes, we expect the rate of performance improvement to decrease rapidly as the number of target tiles is in-

creased due to the inherent non-scalable nature of barrier synchronization.

Simulation error. This study examines simulation error and variability for various synchronization models. Results are generated from ten runs of each benchmark using the same parameters as the previous study. We compare results for single- and multi-machine simulations, as distribution across machines involves high-latency network communication that potentially introduces new sources of error and variability.

Figure 6b, Figure 6c and Table 3 show the error and coefficient of variation of the synchronization models. The error data is presented as the percentage deviation of the mean simulated application run-time (in cycles) from some baseline. The baseline we choose is *LaxBarrier*, as it closely approximates a cycle-accurate simulation. The coefficient of variation (CoV) is a measure of how consistent results are from run to run. It is defined as the ratio of standard deviation (of simulated run-time over 10 runs) to mean (simulated run-time), as a percentage. Error and CoV values close to 0.0% are best.

As seen in the table, *LaxBarrier* shows the best CoV (0.08%). This is expected, as the barrier forces target cores to run in lock-step, so there is little opportunity for deviation. We also observe that *LaxBarrier* shows very accurate results across four host machines. This is also expected, as the barrier eliminates clock skew that occurs due to variable communication latencies.

LaxP2P shows both good error (1.28%) and CoV (0.32%). This is because LaxP2P prevents any thread from running far ahead or falling far behind of other threads by putting the faster thread to sleep. Thus, by preventing the occurrence of outliers and forcing target tile clocks to stay within a configurable number of cycles of each other, *LaxP2P* maintains low CoV and error. In fact, *LaxP2P* shows error nearly identical to *LaxBarrier*. The main difference between the schemes is that *LaxP2P* has modestly higher CoV.

Lax shows the worst error (7.56%). This is expected, because only application events synchronize target tiles. As shown earlier, *Lax* allows thread clocks to vary significantly, giving more opportunity for the final simulated run-time to vary. For the same reason, *Lax* has the worst CoV (0.58%).

Clock skew. Figure 7 shows the approximate clock skew of each synchronization model during one run of the SPLASH2 benchmark `fmm`. Simulated clocks for each tile are collected at many points during program execution. This data is used to generate an approximate average "global cycle count" for the simulation at any given moment. The difference between individual clocks and the "global clock"

(a) Performance

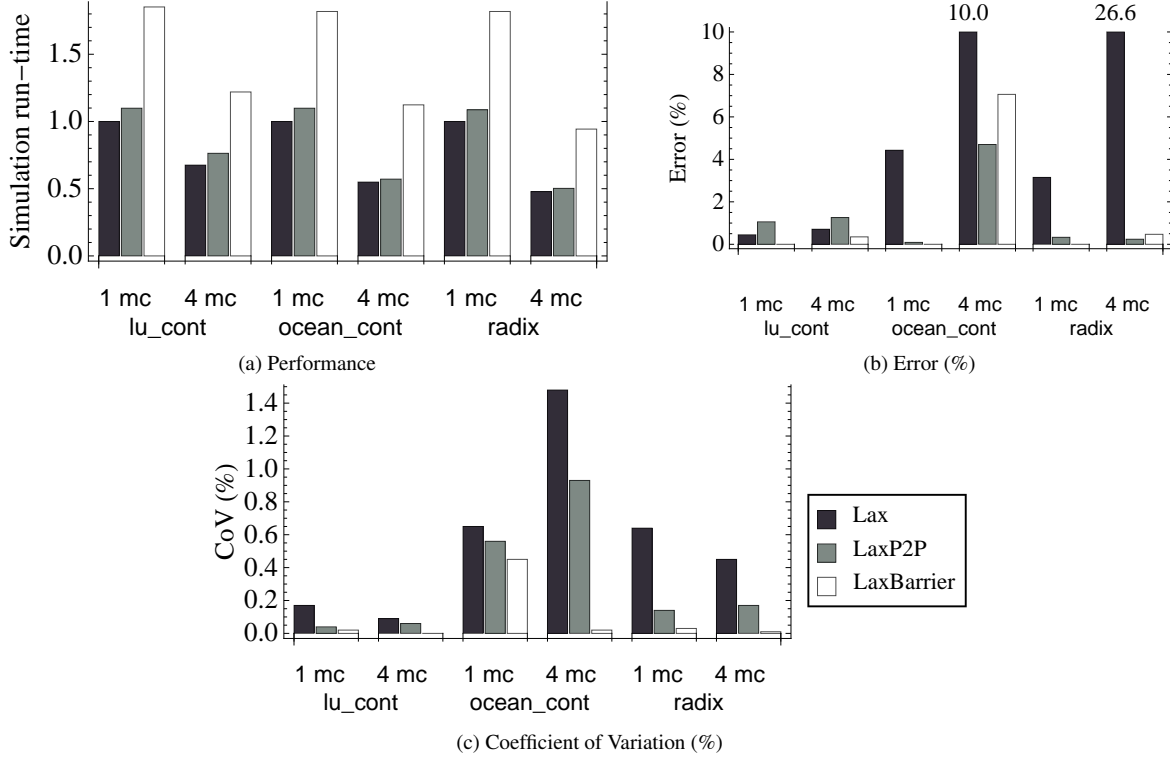

(b) Error (%)

(c) Coefficient of Variation (%)

Figure 6: Performance and accuracy data comparison for different synchronization schemes. Data is collected from SPLASH2 benchmarks on one and four host machines, using ten runs of each simulation. (a) Simulation run-time in seconds, normalized to *Lax* on one host machine.. (b) Simulation error, given as percentage deviation from *LaxBarrier* on one host machine. (c) Simulation variability, given as the coefficient of variation for each type of simulation.
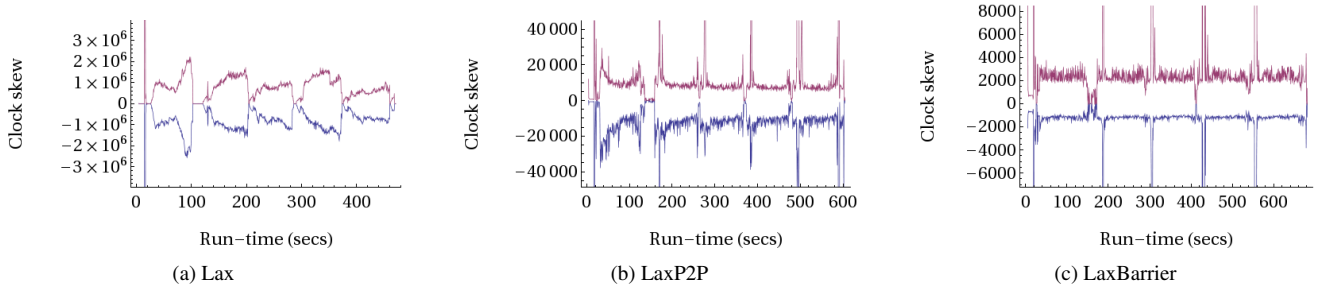

(a) Lax

(b) LaxP2P

(c) LaxBarrier

Figure 7: Clock skew in simulated cycles during the course of simulation for various synchronization models. Data collected running the `fmm` SPLASH2 benchmark.

is then computed. The full simulation time is split into sub-intervals, and Figure 7 shows the maximum and minimum difference for each interval. This method involves some approximation, so spikes are introduced during rapid changes in application behavior, most visible in Figure 7c.

These clock skew results match what one expects from the various synchronization models. *Lax* shows by far the greatest skew, and application synchronization events are clearly visible. The skew of *LaxP2P* is several orders of magnitude less than *Lax*, but application synchronization

events are still visible and skew is on the order of $\pm 10,000$ cycles. *LaxBarrier* has the least skew, as one would expect. Application synchronization events are largely undetectable — skew appears constant throughout execution.

Summary. Graphite offers three synchronization models that give a tradeoff between simulation speed and accuracy. *Lax* gives optimal performance while achieving reasonable accuracy, but it also lets threads deviate considerably during simulation. This means that fine-grained interactions can be

12

missed or misrepresented. On the other extreme, *LaxBarrier* forces tight synchronization and accurate results, at the cost of performance and scaling. *LaxP2P* lies somewhere in between, keeping threads from deviating too far and giving very accurate results, while only reducing performance by 10%.

Finally, we observed while running these results that the parameters to synchronization models can be tuned to match application behavior. For example, some applications can tolerate large barrier intervals with no measurable degradation in accuracy. This allows *LaxBarrier* to achieve performance near that of *LaxP2P* for some applications.

4.4 Application Studies

This section presents two application studies of memory system behavior. The first study characterizes cache misses as line size varies and validates against published results for SPLASH. The second compares performance of different cache coherence schemes for `blackscholes`, a member of the PARSEC benchmark suite.

Cache Miss-rate Characterization. The memory system and network components are validated by characterizing cache miss rates as a function of cache line size for 6 of the SPLASH-2 benchmarks, and comparing the results to those reported by Woo et.al. [35]. Since the memory architecture that Woo et.al. [35] evaluates consists of only a single cache level, the L1I and L1D cache models supported by the Graphite system are disabled and all memory accesses are redirected to the L2 cache for this evaluation. The L2 cache modeled is a 1MB 4-way set associative cache.

Based on Woo et.al's [35] characterization of the 6 SPLASH-2 benchmarks we evaluate, the following application behaviors are expected: In applications such as `lu_contig` and `fft`, miss rates should drop linearly as the cache line size increases because of perfect spatial locality. Perfect spatial locality results from a contiguous allocation of the data structures used by these benchmarks. Hence, using a greater cache line size enables a higher quantity of useful data to be cached as a result of a single cache miss. In `radix`, miss rates should decrease steadily with increasing cache line sizes till 128 bytes. However, at 256 bytes, the false sharing miss rate should become significantly high since at this point, the granularity of interleaving between the writes of multiple processors to the same global array becomes less than that of a cache line. In `water-spatial` and `barnes`, different threads are allocated their own independent set of records. These records may either be molecules (in `water-spatial`) or particles (in `barnes`). Each thread can write any record it owns but can only read from certain fields of other records. In these applications, true sharing miss rates should decrease
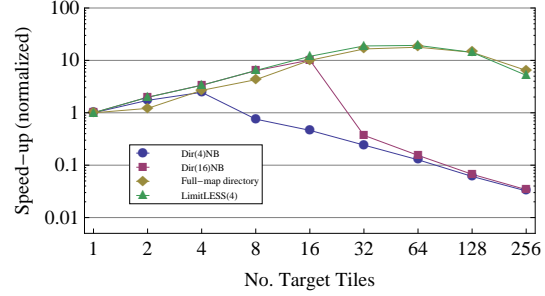

Figure 9: Different cache coherency schemes are compared using speedup relative to simulated single-tile execution in `blackscholes` by scaling target tile count.

and false sharing misses increase with incresing cache line sizes owing to the above data sharing pattern. All the above mentioned application behaviors are captured in Figure 8. A minor difference in the results presented by our evaluation versus those presented by Woo et.al was that our evaluation reported lower overall miss rates. This is because in the evaluations presented in the SPLASH-2 paper, the benchmarks were compiled for SGI machines while for the evaluations presented in this paper, the benchmarks were compiled for the IA-32 architecture. The number of memory accesses will be far higher on IA-32 machines due to register spills and many of these accesses will hit in the cache. Hence, the overall miss rate will be much lower than that for SGI machines. These results demonstrate that although Graphite is not cycle-accurate, it still displays correct application trends in real-world parallel applications.

Cache Coherence Study. As processors scale to ever-increasing core counts, the viability of cache coherence in future manycores remains unsettled. This study explores three cache coherence schemes as a demonstration of Graphite's ability to explore this relevant architectural question, as well as its ability to run large simulations. Graphite supports a few cache coherence protocols. A limited directory MSI protocol with i sharers, denoted Dir_iNB [1], is the base-line cache coherence protocol. Graphite also supports full-map directories and the LimitLESS protocol[1].

Figure 9 shows the comparison of the different cache coherency schemes in the application `blackscholes`, a member of the PARSEC benchmark suite [5]. `blackscholes` is nearly perfectly parallel as little information is shared between cores. However, by tracking all requests through the memory system, we observed some global addresses in the system libraries are heavily shared as read-only data. All tests were run using the `simsmall`

---

[1]In the LimitLESS protocol, a limited number of hardware pointers exist for the first i sharers and additional requests to shared data are handled by a software trap, preventing the need to evict existing sharers.[8]
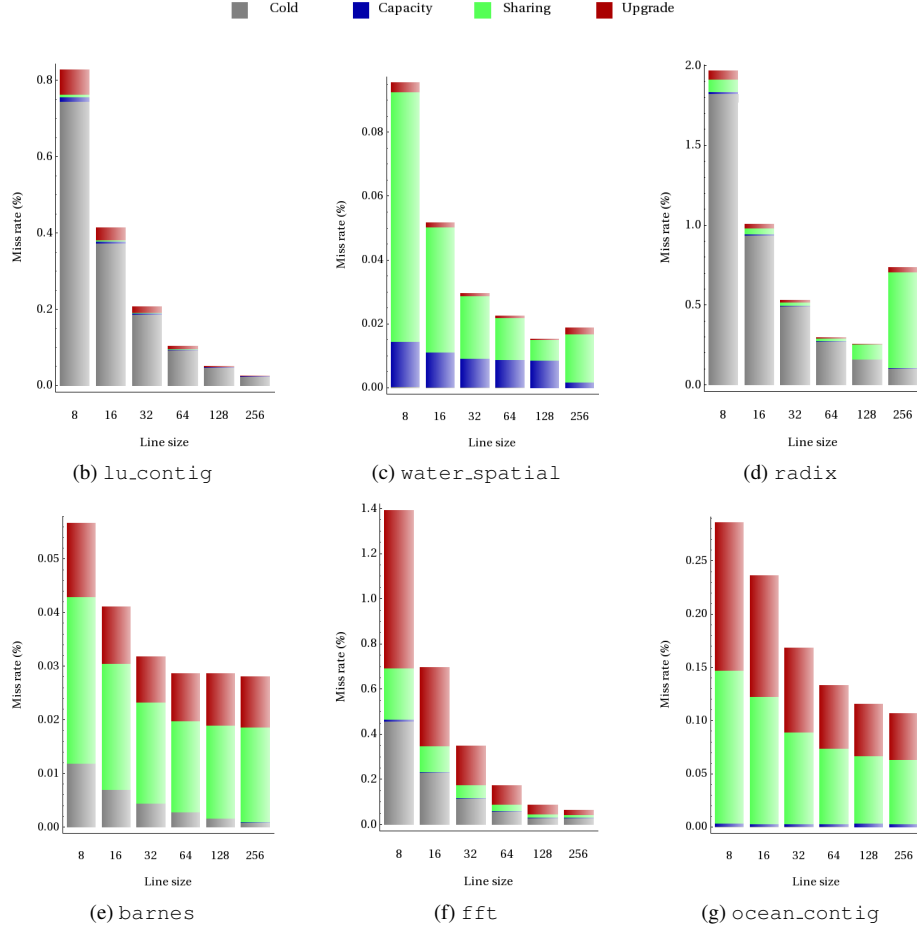
Figure 8: Breakdown of cache misses by type as line size changes for three SPLASH benchmarks.

input. The `blackscholes` source code was unmodified.

As seen in Figure 9, `blackscholes` achieves near-perfect scaling with the full-map directory and LimitLESS directory protocols up to 32 target tiles. Beyond 32 target tiles, parallelization overhead begins to outstrip performance gains. From simulator results, we observe that larger target tile counts give increased average memory access latency. This occurs in at least two ways: (i) increased network distance to memory controllers, and (ii) additional latency at memory controllers. Latency at the memory controller increases because the default target architecture places a memory controller at every tile, evenly splitting total off-chip bandwidth. This means that as the number of target tiles increases, the bandwidth at each controller decreases proportionally, and the service time for a memory request increases. Queueing delay also increases by statically partitioning the bandwidth into separate queues, but results show that this effect is less significant.

The LimitLESS and full-map protocols exhibit little differentiation from one another. This is expected, as the heav-ily shared data is read-only. Therefore, once the data has been cached, the LimitLESS protocol will exhibit the same characteristics as the full-map protocol. The limited map directory protocols do not scale. Dir_4NB does not exhibit scaling beyond four target tiles. Because only four sharers can cache any given memory line at a time, heavily shared read data is being constantly evicted at higher target tile counts. This serializes memory references and damages performance. Likewise, the $Dir_{16}NB$ protocol does not exhibit scaling beyond sixteen target cores.

5 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 [3], RSIM [17], SimOS [30], Simics [23], and QEMU [4]. Some of these are capable of simulating parallel target architectures but

all of them execute sequentially on the host machine. Like Graphite, Proteus [6] is designed to simulate highly parallel architectures and uses direct execution and configurable, swappable models. However, it too runs only on sequential hosts. FaCSim[21] 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 [33], GEMS [24], COTSon [26], BigSim [36], FastMP [19], SlackSim [9], Wisconsin Wind Tunnel (WWT) [29], Wisconsin Wind Tunnel II (WWT II) [27], and those described by Chidester and George [10], and Penry et al. [28].

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.

COTSon uses AMD's SimNow! for functional modeling and therefore suffers from some of the same problems as SimFlex and GEMS. The sequential instruction stream coming out of SimNow! is demultiplexed into separate threads before timing simulation. This limits parallelism and restricts COTSon to a single host machine for shared-memory simulations. COTSon can perform multi-machine simulations but only if the applications are written for distributed memory and use a messaging library like MPI.

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 broader and more interesting class of architectures that use shared memory.

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 lax synchronization.

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 [13], FAST [11], and HASim [15] 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 [19] estimates performance for parallel workloads with no memory sharing (such as SPECrate) by carefully simulating only some of the independent processes and using those results to model the others. Finally, simulators such as SimFlex [33] 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 [18] and used in the Georgia Tech Time Warp [14], BigSim [36], and SlackSim [9]. The first three systems assume that perfect ordering must be maintained and rollback when the timestamps indicate out-of-order events.

SlackSim (developed concurrently with Graphite) is the only other system that allows events to occur out of order. It allows all threads to run freely as long as their local clocks remain within a specified window. Their "unbounded slack" mode is essentially the same as plain lax synchronization. However, their approach to limiting slack relies on a central manager which monitors all threads using shared memory. This (along with other factors) restricts them to running on a single host machine and ultimately limits their scalability. Graphite's point-to-point mitigation is completely distributed and enables scaling to larger numbers of target cores and host machines.

Separating functional models from timing models is a well-established technique used in many simulators including: FastSim [31], TimingFirst [25], GEMS [24], tsim [12],

Asim [16], HASim [15], FAST [11], and ProtoFlex [13].

TreadMarks [2] 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.

6 Conclusions

Graphite is a novel parallel distributed simulator targeting large multicore processors. It uses a variety of techniques to deliver the high performance and scalability needed to perform useful evaluations and software development including: direct execution, multi-machine distribution, analytical modeling and lax synchronization. Our results indicate that Graphite scales well to designs of more than 1000 cores and several host machines. It successfully predicts application characteristics for a number of benchmarks and has low simulation overhead. It enables rapid prototyping and evaluation of new architectures and provides an extensible and modular framework.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 280–298, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[2] 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.

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

[4] 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.

[5] 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.

[6] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: a high-performance parallel-architecture simulator. In *SIGMETRICS '92/PERFORMANCE '92: Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 247–248, New York, NY, USA, 1992. ACM.

[7] 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.

[8] D. Chaiken, J. Kubiatowicz, and A. Agarwal. Limitless directories: A scalable cache coherence scheme. In *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV*, pages 224–234. ACM, 1991.

[9] J. Chen, M. Annavaram, and M. Dubois. SlackSim: A Platform for Parallel Simulations of CMPs on CMPs. *SIGARCH Comput. Archit. News*, 37(2):20–29, 2009.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi. How to simulate 1000 cores. *SIGARCH Comput. Archit. News*, 37(2):10–19, 2009.

[27] 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.

[28] 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 multiprocessors. In *HPCA'06: The Twelfth International Symposium on High-Performance Computer Architecture*, pages 29–40, Feb 2006.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

17