To appear in HPCA-16: Proc. of the 16th International Symposium on High-Performance Computer Architecture, January 2010.

Graphite: A Distributed Parallel Simulator for Multicores

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

Massachusetts Institute of Technology, Cambridge, MA

***Abstract*—This paper introduces the Graphite open-source 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. Several techniques are used to achieve this including: direct execution, seamless multicore and multi-machine distribution, and lax synchronization. Graphite is capable of accelerating simulations by distributing them across multiple commodity Linux machines. When using multiple machines, it provides the illusion of a single process with a single, shared address space, allowing it to run off-the-shelf pthread applications with no source code modification.**

Our results demonstrate that Graphite 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.

I. 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 [1], [2], [3], [4], [5], 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 [6] 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 [7], [8]. 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 [9]. 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. 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. 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. 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 software development for future multicore architectures. It provides both functional and performance modeling for cores, on-chip networks, and memory subsystems including cache hierarchies 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. Through dynamic binary translation, Graphite adds new functionality (*e.g.,* new instructions or a direct core-to-core messaging interface) and intercepts operations that require action from the simulator (*e.g.,* memory operations that feed into the cache model) [10].

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 accurate estimates 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) [11]. However, to reduce the time wasted on synchronization, Graphite does not strictly enforce the ordering of all events in the system. 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 III-F). This is similar to the "unbounded slack" mode in SlackSim [12]; however, Graphite also supports a new scalable mechanism called *LaxP2P* for managing slack and improving accuracy.

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-2 benchmark suite. The slowdown versus native execution is as low as $41\times$ when using eight 8-core host machines, indicating that Graphite can be used for realistic application studies.

The remainder of this paper is structured as follows. Section II describes the architecture of Graphite. Section III discusses the implementation of Graphite in more detail. Section IV evaluates the accuracy, performance, and scaling of the simulator. Section V discusses related work and, Section VI summarizes our findings.

II. 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) [13]. 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.

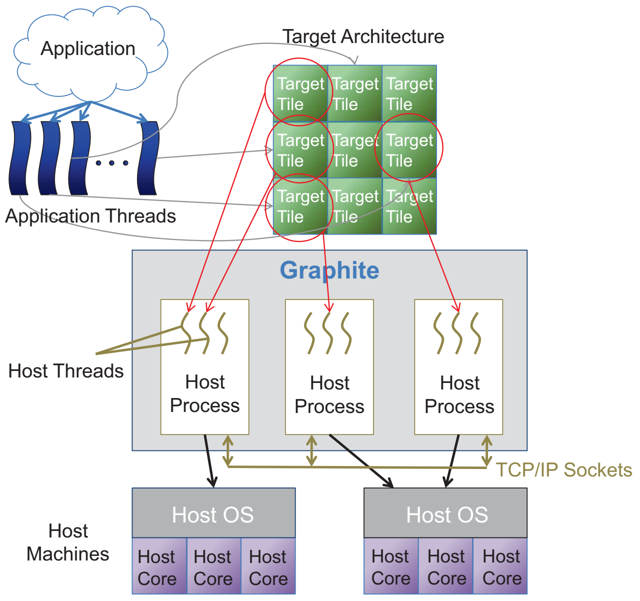
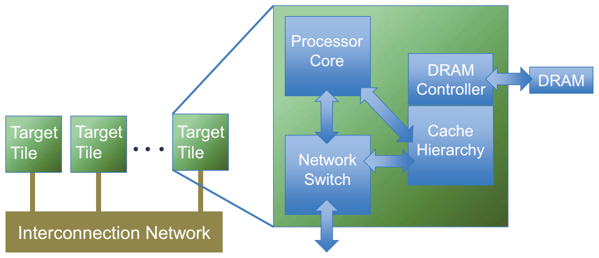


Fig. 1: High-level architecture. Graphite consists of one or more host processes distributed across machines and working together over sockets. Each process runs a subset of the simulated tiles, one host thread per simulated tile.

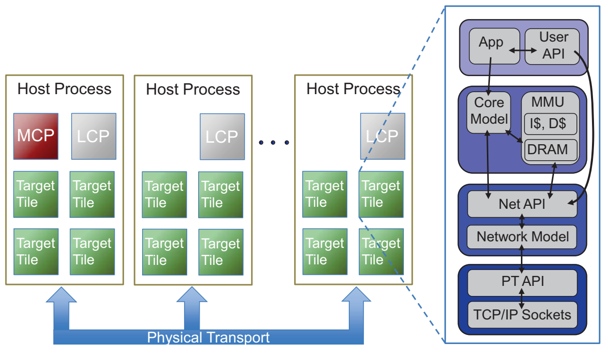
Graphite has a modular design based on swappable modules. Each of the components of a tile is modeled by a separate module with well-defined interfaces. Module can be configured through run-time parameters or completely replaced to study alternate architectures. Modules may also be replaced to alter the level of detail in the models and tradeoff between performance and accuracy.

Figure 2b illustrates the key components of a Graphite simulation.Application threads are executed under a dynamic binary translator (currently Pin [14]) which rewrites instructions to generate events at key points. These events cause traps into Graphite's backend which contains the compute core, memory, and network modeling modules.Points of interest intercepted by the dynamic binary translator (DBT) include: memory references, system calls, synchronization routines and user-level messages. The DBT is also used to generate a stream of executed instructions used in the compute core models.

Graphite's simulation backend 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.



(a) Target Architecture



(b) Modular Design of Graphite

Fig. 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.

A. *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 computational pipeline; the memory model is responsible for the memory subsystem, which is composed of different levels of caches and DRAM; and the network model handles the routing of network packets over the on-chip network and accounts for various delays encountered due to contention and routing overheads.

Graphite's 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.

One of the key techniques Graphite uses to achieve good simulator performance is lax synchronization. With lax synchronization, 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 III-F will talk about how Graphite addresses this challenge.

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

All of the actual communication between tiles is handled by the physical transport (PT) layer. For example, the network model calls into this layer to perform the functional task of moving data from one tile to another. The PT layer abstracts away the host-architecture dependent details of intra- and inter-process communication, making it easier to port Graphite to new hosts.

III. 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.

A. 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 III-C), 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 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 timestamps are generated from core clocks.

B. Memory System

The memory system of Graphite has both a functional and a modeling role. The functional role is to provide an abstraction of a shared address space to the application threads which execute in different address spaces. The modeling role is to simulate the caches hierarchies and memory controllers of the target architecture. The functional and modeling parts of the memory system are tightly coupled, i.e, the messages sent over the network to load/store data and ensure functional correctness, are also used for performance modeling.

The memory system of Graphite is built using generic modules such as caches, directories, and simple cache coherence protocols. Currently, Graphite supports a sequentially consistent memory system with full-map and limited directory-based cache coherence protocols, private L1 and L2 caches, and memory controllers on every tile of the target architecture. However, due to its modular design, a different implementation of the memory system could easily be developed and swapped in instead. The application's address space is divided up among the target tiles which possess memory controllers. Performance modeling is done by appending simulated timestamps to the messages sent between different memory system modules (see Section III-F). The average memory access latency of any request is computed using these timestamps.

The functional role of the memory system is to service all memory operations made by application threads. The dynamic binary translator in Graphite rewrites all memory references in the application so they get redirected to the memory system.

An alternate design option for the memory system is to completely decouple its functional and modeling parts. This was not done for performance reasons. Since Graphite is a distributed system, both the functional and modeling parts of the memory system have to be distributed. Hence, decoupling them would lead to doubling the number of messages in the system (one set for ensuring functional correctness and another for modeling). An additional advantage of tightly coupling the functional and the modeling parts is that it automatically helps verify the correctness of complex cache hierarchies and coherence protocols of the target architecture, as their correct operation is essential for the completion of simulation.

C. Network

The network component provides high-level messaging services between tiles built on top of the lower-level transport layer, which uses shared memory and TCP/IP to communicate between target tiles. 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 and system call handler.

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 [13], [15]. Each network model is configured independently, allowing for exploration of new network topologies focused on particular subcomponents of the system. The network models are responsible for routing packets and updating timestamps to account for network delay.

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), and several mesh models with different tradeoffs in performance and accuracy.

D. 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 target address space rather than the host address space, and the need to maintain the illusion of a single process across multiple processes executing the target application.

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 and modifies their arguments to point to the correct data before executing them 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 (each with its own file descriptor table), and might be running on different host machines each with their own file system. 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. Similarly, system calls that are used to implement synchronization between threads, such as `futex`, are intercepted and forwarded to the MCP, where they are emulated. System calls that do not require special handling are allowed to execute directly on the host machine.

1) Process Initialization and Address Space Management: At the start of the simulation, Graphite's system interface layer needs to make sure that each host process is correctly initialized. In particular, it must ensure that all process segments are properly set up, the command line arguments and environment variables are updated in the target address space and the thread local storage (TLS) is correctly initialized in each process. Eventually, only a single process in the simulation executes `main()`, while all the other processes execute threads subsequently spawned by Graphite's threading mechanism.

Graphite also explicitly manages the target address space, setting aside portions for thread stacks, code, and static and dynamic data. In particular, Graphite's dynamic memory manager services requests for dynamic memory from the application by intercepting the `brk`, `mmap` and `munmap` system calls and allocating (or deallocating) memory from the target address space as required.

E. Threading Infrastructure

One challenging aspect of the Graphite 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 allocating 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 tiles in the target architecture. 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.

F. Synchronization Models

For high performance and scalability across multiple machines, Graphite decouples tile simulations by relaxing the timing synchronization between them. By design, Graphite is not cycle-accurate. It supports several synchronization strategies that represent different timing accuracy and simulator performance tradeoffs: lax synchronization, lax with barrier synchronization, and lax with point-to-point synchronization. Lax synchronization is Graphite's baseline model. Lax with barrier synchronization and lax with point-to-point synchronization layer mechanisms on top of lax synchronization to improve its accuracy.

1) 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 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 timestamp that is initially set to the clock of the sender. These timestamps 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. With the exception of memory operations, 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, 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 [12]. 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 [11]. However, this requires state to be maintained throughout the simulation and hurts performance. Our results in Section IV-C show that lax synchronization, despite out-of-order processing, still predicts performance trends well.

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 timestamp 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 problem is addressed by using packet timestamps to build an approximation of global progress. A window of the most recently-seen timestamps is kept, on the order of the number of target tiles. The average of these timestamps gives an approximation of global progress. 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 while mitigating the effect of outliers.

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.

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 (see Section IV-C).

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 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. Because of this, it introduces less overhead than LaxBarrier and has superior scalability (see Section IV-C).

IV. Results

This section presents experimental results using Graphite. We demonstrate Graphite's ability to scale to large target architectures and distribute across a host cluster. We show that lax synchronization provides good performance and accuracy, and validate results with two architectural studies.

A. Experimental Setup

The experimental results provided in this section were all obtained on a homogeneous 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 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 I unless otherwise noted. These parameters were chosen to match the host architecture as closely as possible.

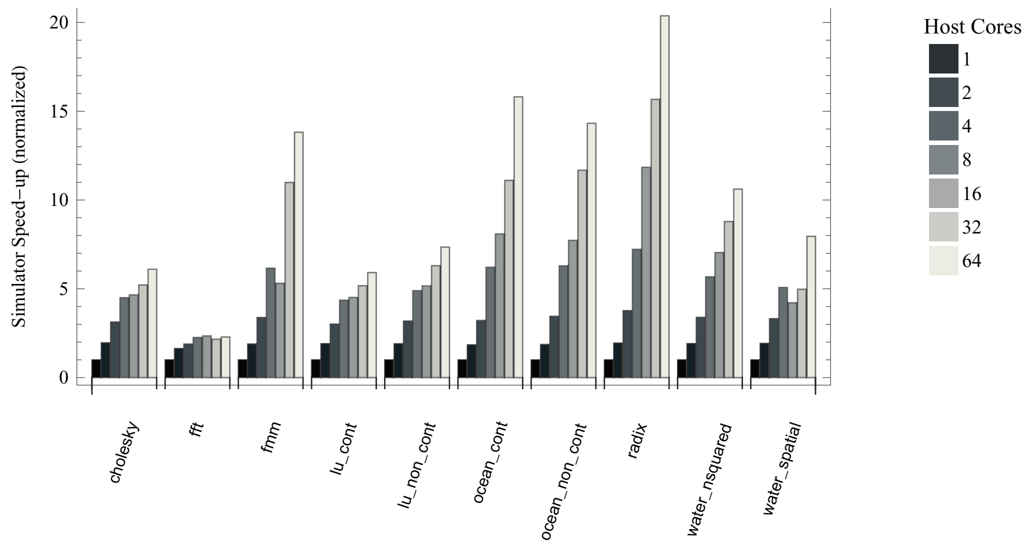


Fig. 3: Simulator performance scaling for SPLASH-2 benchmarks across different numbers of host cores. The target architecture has 32 tiles in all cases. Speed-up is normalized to simulator runtime on 1 host core. Host machines each contain 8 cores. Results from 1 to 8 cores use a single machine. Above 8 cores, simulation is distributed across multiple machines.

Feature	Value
Clock frequency	3.16 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	5.3 GB/s
Interconnect	Mesh network

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

B. Simulator Performance

1) Single- and Multi-Machine Scaling: Graphite is designed to scale well to both large numbers of target tiles 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 3 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 SPLASH-2 [16] benchmarks and are normalized to the runtime on a single host core. The results from one to eight host cores are collected by allowing the simulation to use additional cores within a single host machine. The results for 16, 32, and 64 host cores correspond to using all the cores within 2, 4, and 8 machines, respectively.

As shown in Figure 3, all applications except `fft` exhibit significant simulation speedups as more host cores are added. The best speedups are achieved with 64 host cores (across 8 machines) and range from about 2× (`fft`) to 20× (`radix`).

Scaling is generally better within a single host machine than across machines due to the lower overhead of communication. Several apps (`fmm`, `ocean`, and `radix`) show nearly ideal speedup curves from 1 to 8 host cores (within a single machine). Some apps show a drop in performance when going from 8 to 16 host cores (from 1 to 2 machines) 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 Graphite 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 platform 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 must 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.

2) Simulator Overhead: Table II shows simulator performance for several benchmarks from the SPLASH-2 suite. 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 II demonstrates that Graphite achieves very good performance for all the benchmarks studied. The

Application	Native	Simulation			
		1 machine		8 machines	
	Time	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	2465×	396	1317×
`water_spatial`	0.13	129	966×	82	616×
Mean	-	-	1751×	-	1213×
Median	-	-	1307×	-	616×

TABLE II: Multi-Machine Scaling Results. Wall-clock execution time of SPLASH-2 simulations running on 1 and 8 host machines (8 and 64 host cores). Times are in seconds. Slowdowns are calculated relative to native execution.

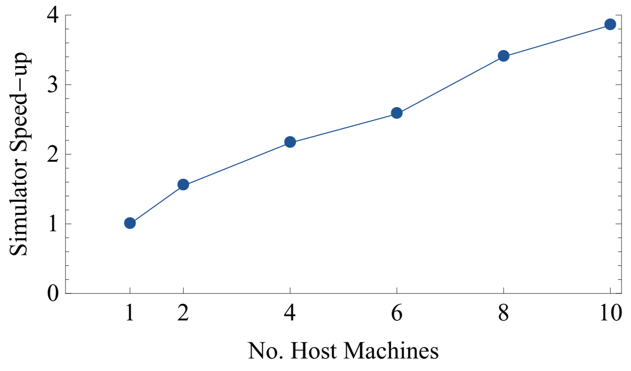


Fig. 4: Simulation speed-up as the number of host machines is increased from 1 to 10 for a `matrix-multiply` kernel with 1024 application threads running on a 1024-tile target architecture.

total run time for all the benchmarks is on the order of a few minutes, with a median slowdown of 616× 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 `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 computation-to-communication ratio are able to more effectively parallelize and hence show higher simulation speeds.

3) 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 4 shows the normalized speed-up of a 1024-thread `matrix-multiply` kernel running across different numbers of host machines. `matrix-multiply` was chosen because it scales well to large numbers of threads, while still having frequent synchronization via messages with neighbors.

	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 III: Mean performance and accuracy statistics for data presented in Figure 5. Scaling is the performance improvement going from 1 to 4 host machines.

This graph shows steady performance improvement for 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.

C. Lax synchronization

Graphite supports several synchronization models, namely *lax synchronization* 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.

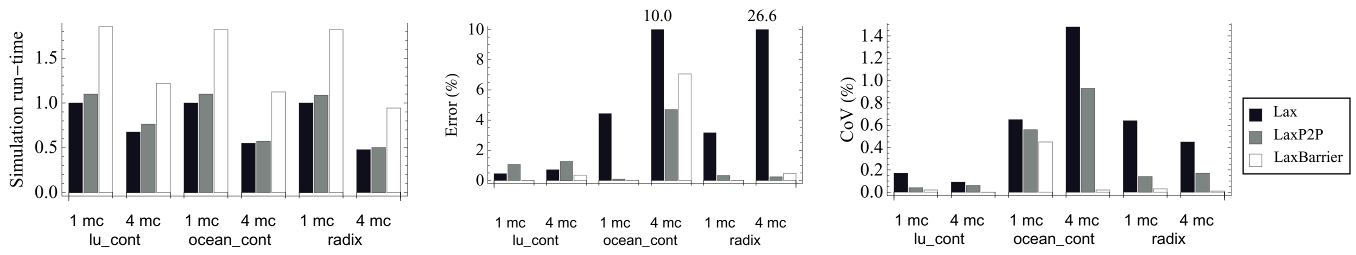
1) Simulator performance: Figure 5a and Table III illustrate the simulator performance (wall-clock simulation time) of the three synchronization models using three SPLASH-2 benchmarks. Each simulation is run on one and four host machines. The barrier interval was chosen as 1,000 cycles to give very accurate 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 only slightly worse than *Lax*. It shows an average slowdown of 1.10× and 1.07× when compared to *Lax* on one and four host machines respectively. *LaxP2P* shows good scalability with a performance improvement of 1.84× going from one to four host machines. This is mainly due to the distributed nature of synchronization in *LaxP2P*, allowing it to scale to a larger number of host cores.

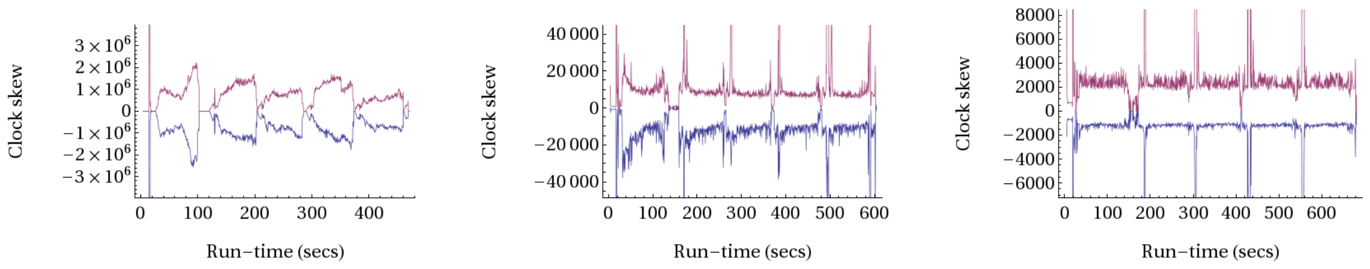
LaxBarrier performs poorly as expected. It encounters an average slowdown of 1.82× and 1.94× when compared to *Lax* on one and four host machines respectively. Although the performance improvement of *LaxBarrier* when going from one to four host machines is comparable to the other schemes, we expect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

2) Simulation error: This study examines simulation error and variability for various synchronization models. Results are generated from ten runs of each benchmark using the



(a) Normalized Run-time (b) Error (%) (c) Coefficient of Variation (%)

Fig. 5: Performance and accuracy data comparison for different synchronization schemes. Data is collected from SPLASH-2 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

Fig. 6: Clock skew in simulated cycles during the course of simulation for various synchronization models. Data collected running the `fmm` SPLASH-2 benchmark.

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 5b, Figure 5c and Table III 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 gives highly accurate results. 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 to mean, as a percentage. Error and CoV values close to 0.0% are best.

As seen in the table, *LaxBarrier* shows the best CoV (0.09%). 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%). Despite the large slack size, by preventing the occurrence of outliers *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 below, *Lax* allows thread clocks to vary significantly, giving more opportunity for the final simulated runtime to vary. For the same reason, *Lax* has the worst CoV (0.58%).

3) Clock skew: Figure 6 shows the approximate clock skew of each synchronization model during one run of the SPLASH-2 benchmark `fmm`. The graph shows the difference between the maximum and minimum clocks in the system at a given time. These 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.[1]

4) 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 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%.

[1]Spikes in the graphs, as seen in Figure 6c, are due to approximations in the calculation of clock skew. See [17].

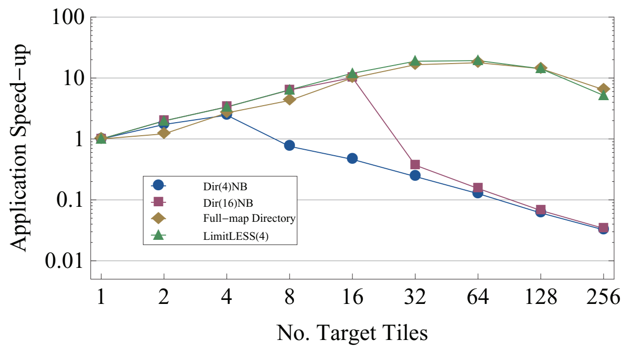


Fig. 7: Different cache coherency schemes are compared using speedup relative to simulated single-tile execution in `blackscholes` by scaling target tile count.

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.

D. Application Studies

1) Cache Miss-rate Characterization: We replicate the study performed by Woo et. al [16] characterizing cache miss rates as a function of cache line size. Target architectural parameters are chosen to match those in [16] as closely as possible. In particular, Graphite's L1 cache models are disabled to simulate a single-level cache. The architectures still differ, however, as Graphite simulates x86 instructions whereas [16] uses SGI machines. Our results match the expected trends for each benchmark studied, although the miss rates do not match exactly due to architectural differences. Further details can be found in [17].

2) Scaling of Cache Coherence: 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$_i$NB [18], is the default cache coherence protocol. Graphite also supports full-map directories and the LimitLESS protocol[2].

Figure 7 shows the comparison of the different cache coherency schemes in the application `blackscholes`, a member of the PARSEC benchmark suite [8]. `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` input. The `blackscholes` source code was unmodified.

[2]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.[19]

As seen in Figure 7, `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 heavily 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$_4$NB 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.

V. 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 [1], RSIM [2], SimOS [3], Simics [4], and QEMU [5]. Some of these are capable of simulating parallel target architectures but all of them execute sequentially on the host machine. Like Graphite, Proteus [20] is designed to simulate highly parallel architectures and uses direct execution and configurable, swappable models. However, it too runs only on sequential hosts.

The projects most closely related to Graphite are parallel simulators of parallel target architectures including: SimFlex [21], GEMS [22], COTSon [23], BigSim [11], FastMP [24], SlackSim [12], Wisconsin Wind Tunnel (WWT) [25], Wisconsin Wind Tunnel II (WWT II) [10], and those described by Chidester and George [26], and Penry et al. [27].

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 [9], FAST [6], and HASim [28] 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, implementing a new model in an FPGA is more difficult than software, making it harder to quickly experiment with different designs.

Other simulators improve performance by modeling only a portion of the total execution. FastMP [24] 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 [21] 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 [29] and used in the Georgia Tech Time Warp [30], BigSim [11], and SlackSim [12]. 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. SlackSim's "unbounded slack" mode is essentially the same as plain lax synchronization. However, its approach to limiting slack relies on a central manager which monitors all threads using shared memory. This (along with other factors) restricts it to running on a single host machine and ultimately limits its scalability. Graphite's LaxP2P is completely distributed and enables scaling to larger numbers of target tiles and host machines. Because Graphite has more aggressive goals than SlackSim, it requires more sophisticated techniques to mitigate and compensate for excessive slack.

TreadMarks [31] 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.

VI. Conclusions

Graphite is a distributed, parallel simulator for design-space exploration of large-scale multicores and applications research. It uses a variety of techniques to deliver the high performance and scalability needed for useful evaluations including: direct execution, multi-machine distribution, analytical modeling, and lax synchronization. Some of Graphite's other key features are its flexible and extensible architecture modeling, its compatibility with commodity multicores and clusters, its ability to run off-the-shelf pthreads application binaries, and its support for a single shared simulated address space despite running across multiple host machines.

Our results demonstrate that Graphite is high performance and achieves slowdowns as little as $41\times$ over native execution for simulations of SPLASH-2 benchmarks on a 32-tile target. We also demonstrate that Graphite is scalable, obtaining near linear speedup on a simulation of a 1000-tile target using from 1 to 10 host machines. Lastly, this work evaluates several lax synchronization simulation strategies and characterizes their performance versus accuracy. We develop a novel synchronization strategy called LaxP2P for both high performance and accuracy based on periodic, random, point-to-point synchronizations between target tiles. Our results show that LaxP2P performs on average within 8% of the highest performance strategy while keeping average error to 1.28% of the most accurate strategy for the studied benchmarks.

Graphite will be released to the community as open-source software to foster research on large-scale multicore architectures and applications.

Acknowledgement

The authors would like to thank James Psota for his early help in researching potential implementation strategies and pointing us towards Pin. His role as liaison with the Pin team at Intel was also greatly appreciated. This work was partially funded by the National Science Foundation under Grant No. 0811724.

References

[1] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[2] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve, "Rsim: Simulating shared-memory multiprocessors with ilp processors," *Computer*, vol. 35, no. 2, pp. 40–49, 2002.

[3] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: The SimOS approach," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 3, no. 4, pp. 34–43, Winter 1995.

[4] 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*, vol. 35, no. 2, pp. 50–58, Feb 2002.

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

[6] 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*, 2007, pp. 249–261.

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

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 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.*, vol. 2, no. 2, pp. 1–32, 2009.

[10] 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*, vol. 8, no. 4, pp. 12–20, Oct–Dec 2000.

[11] 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)*, Apr 2004, p. 78.

[12] J. Chen, M. Annavaram, and M. Dubois, "SlackSim: A Platform for Parallel Simulations of CMPs on CMPs," *SIGARCH Comput. Archit. News*, vol. 37, no. 2, pp. 20–29, 2009.

[13] 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 *Proc. of the International Symposium on Computer Architecture*, Jun. 2004, pp. 2–13.

[14] 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: Proc. of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, June 2005, pp. 190–200.

[15] 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*, vol. 27, no. 5, pp. 15–31, Sept-Oct 2007.

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

[17] J. Miller, H. Kasture, G. Kurian, N. Beckmann, C. Gruenwald III, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed simulator for multicores," Cambridge, MA, USA, Tech. Rep. MIT-CSAIL-TR-2009-056, November 2009.

[18] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *ISCA '88: Proc. of the 15th Annual International Symposium on Computer architecture*, Los Alamitos, CA, USA, 1988, pp. 280–298.

[19] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "Limitless directories: A scalable cache coherence scheme," in *Proc. of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV*, 1991, pp. 224–234.

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

[21] 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*, vol. 26, no. 4, pp. 18–31, July-Aug 2006.

[22] 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*, vol. 33, no. 4, pp. 92–99, November 2005.

[23] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi, "How to simulate 1000 cores," *SIGARCH Comput. Archit. News*, vol. 37, no. 2, pp. 10–19, 2009.

[24] 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.

[25] 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: Proc. of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, 1993, pp. 48–60.

[26] M. Chidester and A. George, "Parallel simulation of chip-multiprocessor architectures," *ACM Trans. Model. Comput. Simul.*, vol. 12, no. 3, pp. 176–200, 2002.

[27] 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*, Feb 2006, pp. 29–40.

[28] 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.

[29] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, July 1985.

[30] 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*, 1994, pp. 1332–1339.

[31] 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*, vol. 29, no. 2, pp. 18–28, Feb 1996.