Manifold: A Parallel Simulation Framework for Multicore Systems

Jun Wang, Jesse Beu, Rishiraj Bheda, Tom Conte, Zhenjiang Dong, Chad Kersey, Mitchelle Rasquinha, George Riley, William Song, He Xiao, Peng Xu and Sudhakar Yalamanchili
School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332–0250, USA
Email: {jun.wang, riley, sudha}@ece.gatech.edu, tom@conte.us,
{rbheda3, zdong30, cdkersey, wjhsong}@gatech.edu, {jesse.beu, mitch.rasquinha, xiheasas, pxu3.gatech}@gmail.com

Abstract—**This paper presents Manifold, an open-source parallel simulation framework for multicore architectures. It consists of a parallel simulation kernel, a set of microarchitecture components, and an integrated library of power, thermal, reliability, and energy models. Using the components as building blocks, users can assemble multicore architecture simulation models and perform serial or parallel simulations to study the architectural and/or the physical characteristics of the models. Users can also create new components for Manifold or port existing models. Importantly, Manifold's component-based design provides the user with the ability to easily replace a component with another for efficient explorations of the design space. It also allows components to evolve independently and making it easy for simulators to incorporate new components as they become available. The distinguishing features of Manifold include i) transparent parallel execution, ii) integration of power, thermal, reliability, and energy models, iii) full system simulation, e.g., operating system and system binaries, and iv) component-based design. In this paper we provide a description of the software architecture of Manifold, and its main elements - a parallel multicore emulator front-end and a parallel component-based back-end timing model. We describe a few simulators that are built with Manifold components to illustrate its flexibility, and present test results of the scalability obtained on full-system simulation of coherent shared-memory multicore models with 16, 32, and 64 cores executing PARSEC and SPLASH-2 benchmarks.**

I. Introduction

Multicore modeling and simulation has emerged as an important technology for future system architecture design, analysis and evaluation as well as early stage software development. While in the past, the speed of simulation tools naturally kept pace with performance growth of processors, the shift to multicore has stalled the performance growth of these simulation tools while the demand for simulation continues to grow exponentially with core scaling. This is clearly unsustainable and we argue that parallel simulation is a defacto need. This paper describes the Manifold simulation infrastructure and describes how it seeks to address this modeling and simulation challenge.

Manifold is not a simulator per se, but rather a simulation infrastructure for constructing parallel simulation models at varying degrees of fidelity and accuracy using components from a library of models. The paper describes in detail and analyzes a Manifold simulation model for a common case - a coherent shared memory multicore processor. We also briefly describe several other simulators constructed with Manifold to illustrate the range and flexibility. The distinguishing features of Manifold include i) transparent parallel execution, ii) integration of power, thermal, reliability, and energy models, iii) full system simulation, e.g., operating system and system binaries, and iv) component-based design. An early goal of the project was to have the ability to execute stock parallel application binaries and visualize in real (simulated) time the temporal variation in the thermal field. That vision captures a range of modeling scales, fidelity, and accuracy that we felt was necessary for future design space explorations.

An important influence upon Manifold is the recognition of the need to provide a path for easily integrating mature and diverse sets of existing point tools, e.g., core and cache simulators, into processor simulation models thereby leveraging many man-years of effort. Further, as the diversity of multicore processors increases, it is impractical to rebuild simulators from the ground up, or to be tied to (older) models. Support for *composable* models is important. We treat this as largely a software engineering problem, but driven by processor component modeling requirements. In particular our capability requirements are i) execute compiled binaries of multithreaded applications across multiple microarchitecture configurations without recompiling, ii) extensible support for multiple ISAs, iii) support for heterogeneous and asymmetric multicore processors, iv) interfaces to power, thermal, energy, and reliability models, v) controllable tradeoff between fidelity and speed of simulation, and vi) support for multiscale timing models.

In this paper we chose to evaluate a challenging design point, namely the parallel simulation of a cycle-level full system processor simulation model, i.e., application and operating system binaries driving cycle level models of cores, coherent caches, on-chip networks, and DRAM system. We believe this represents a significant challenge, is a desirable target, and successes here can be translated to better performance for higher level (not cycle-level) and more abstract models.

Manifold differs from other simulators in how we chose to factor the software infrastructure to support easy and rapid composition of component models into full system models. We borrow from and apply experiences from the broader parallel discrete event simulation community. The software architecture and development rules enforce the independence and interoperability to support composability. While the choices are certainly debatable we describe several example simulators and their capabilities to illustrate some of the advantages of the choices we did make.

The main contributions of this paper are the following

1) A novel, parallel full system simulation infrastructure for coherent shared memory multiprocessors.
2) The integration of power, energy, thermal, cooling, and reliability models with interactions built in, i.e., temperature with leakage power, temperature and reliability, etc.
3) Incorporation of novel domain specific synchronization algorithms for parallel simulation.
4) The ability to support diverse timing behaviors including i) discrete and time stepped simulation for components in the same model, ii) dynamic voltage frequency scaling coupled to energy and thermal models, and iii) support for multi-scale models, i.e., across wide time scales.

The following section provides a description of the overall system and software architecture and execution model. Sections III - VI describes the various library components used to construct Manifold models. The remainder of the paper provides an analysis of simulator performance, examples of alternative simulators that are easily constructed, and summary of current and ongoing work.

II. System and Software Architectures

The current status makes available a variety of components for constructing a range of models. This section describes the encompassing system architecture that guides the construction and execution of simulation models. The main elements are the execution model and the software architecture.

A. System Architecture and Execution Model

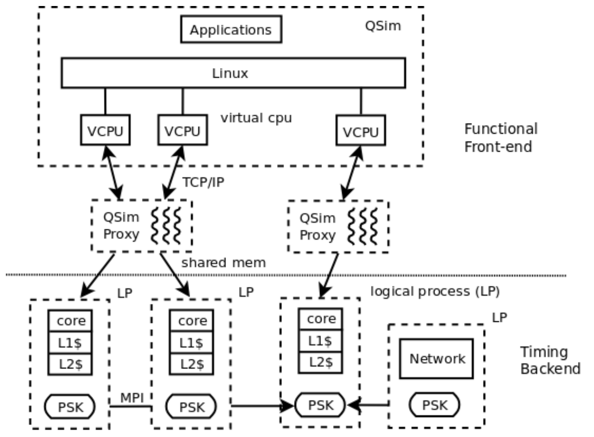


Fig. 1: Manifold execution model.

The Manifold execution model is illustrated in Figure 1. In this figure, each dashed box represents a process. A Manifold simulation is comprised of the QSim front-end parallel multicore emulator that communicates with back-end timing models. QSim boots a Linux OS and executes stock multithreaded 32-bit binaries. The QSim API drives the back-end timing models with backpressure from the timing models controlling the progress of the multicore emulator. The API supports functionality expected in the emulators for fast forward execution where the timing model is only turned on during specific periods of time with preceding warm-up periods. Furthermore, emulator state after OS boot can be checkpointed and restored, to avoid the boot times. No specific restrictions beyond what the API produces are placed on the models that can be coupled to the emulator. For example, One can easily filter out memory references to drive interconnect and memory hierarchy models.

The back-end timing model is comprised of a set of interacting *simulation components* that reflects standard software engineering practice. Each simulation component models a coarse grain physical component such as a core, switch, or cache and can send/receive events to/from other components. For example, a core model may be i) a cycle-level microarchitecture model of an out-of-order core, ii) analytic model of an in-order core, or iii) a k-CPI model (each instruction takes k cycles to execute). Simulation components communicate through ports over Manifold *links*. From the model developers perspective, simulation model construction is one of writing component timing models each of which interacts with other components via links. Adhering to the component interfaces makes components drop-in replaceable.

A model construction process describes the constituent components of a specific simulation model and their interconnection. Components are placed into *logical processes* (LPs). Each logical process manages the interacting execution of its components and executes a copy of the Manifold parallel simulation kernel (PSK). LPs execute in parallel subject to selected global synchronization protocols. Manifold links may exist between components within a LP or between components in distinct LPs. Communication between LPs is transparently handled by the PSK. The model developer need not be aware of the mapping from components to LPs, and also does not need to be aware of parallelization of the model. The same model can be transparently executed in serial or parallel.

Simulation models do not have to incorporate a front-end emulator - one can construct pure timing models. Components can incorporate trace-driven interfaces or simply contain analytic timing models. The timing behavior of a component is very flexible and all timing and event management is handled within the PSK. Components may subscribe to a clock at a specific frequency and have the PSK *tick* the component at that frequency causing the component's local tick handler to be executed and advance the state of the component. Alternatively, a component can operate in an event-driven fashion (as in discrete event simulation) responding to reception of events to advance the state and local time and possibly generate other events. A component may operate with both time-stepped and discrete event behaviors.

A Manifold simulation will execute with each LP mapped to a hardware thread. The hardware threads (*host threads*) are distributed across physical cores (*host cores*) of a cluster. This is transparent to both the developer and the simulation model itself. In its current form, we view QSim as providing a single operating system domain for the execution of blade level (multisocket) multicore architectures.

B. Software Architecture

Following a state-of-the-art practice [2], in the software architecture definition process, we start by identifying three important goals for Manifold:

1) Parallel discrete event simulation (PDES) services should be made transparent to users so that users can build parallel simulations without concerning themselves with the details of PDES.
2) The system should have an open architecture that allows the research community to make contributions either by porting existing components or writing new components, and to do so with minimal effort.
3) The system should allow users to easily mix-and-match components to build their system models.

Accordingly, we have adopted a layered software architecture [11] for Manifold. This architectural style, as pointed out in [8], promotes modifiability, portability, reuse, and separation of concerns. The software architecture consists of four layers, as shown in Figure 2. The arrows represent dependency relationships. A layer can have a dependency on one or more lower layers, but it cannot depend on any layer above it.

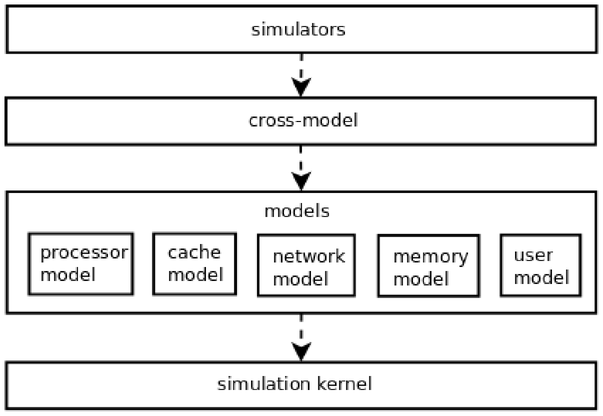


Fig. 2: Manifold software architecture.

The *simulation kernel* layer at the bottom encapsulates the parallel simulation kernel and provides PDES services such as message passing, synchronization, and data serialization/de-serialization. A later section describes the kernel in more detail. All the PDES services are provided in a transparent manner. This means creating or porting components is much easier because there are no parallel simulation specific restrictions.

The *models* layer consists of models of microarchitecture components. Manifold provides a complete set of models for simulating shared memory multiprocessors residing in a single operating system domain. Manifold only places a few requirements on components: (1) A component must be a subclass of the kernel's `Component` class. (2) For each input port, an event handler function should be defined. (3) For output, a component should use the inherited `Send` function. (4) If the component is clocked (time stepped), it should define two functions, which are called at the rising and falling edges of the clock, respectively. A typical component is shown in Figure 3.

One important restriction we place on the components is that a component should not have explicit compile-time dependence on any other component. In other words, each component is an independent unit that does not require the presence of any other component in order to compile. This restriction not only prevents a component from being tightly

```
class MyComponent : public Manifold::Component {
public:
  template<typename T>
  void handler(int port, T* data);
  void rising();
  void falling();
  ...
};
```

Fig. 3: A typical Manifold component.

coupled with another, but also enables drop-in replacement of components. Towards this end, we defined standardized interfaces for the major categories of components, e.g., cache or memory controller. Thus any compliant component model (e.g., core) can work with any compliant component model (e.g., cache) in a drop-in replacement manner. The standardized interfaces are implemented with C++ templates for data types exchanged between components (Note that the event handler in Figure 3 is a template function). A compliant component must send data types that implement the functions required by the interfaces. Otherwise, the compiler would report an error. For example, the required interface for data passed from a processor to a cache requires the implementation of two functions: *get_addr()* and *is_read()*. Any processor component can be connected to any cache component as long as the data type it sends to the cache implements these two functions.

Above the *models* layer is the *cross-model* layer, which hosts functionalities that involve two or more models in the *models* layer, for example mapping from cache-request type (in the cache component) to the network packet format used in the network component. Finally, at the top of the layers is the *simulator* layer, which uses the kernel and the models to build system models, and run parallel discrete event simulations.

III. Multicore Emulator Front-End

Manifold models utilize a multithreaded, multicore emulator front-end to drive back-end timing models. The parallel emulator is constructed using the QSim library to instantiate multiple cores and and execute a guest operating system and applications on multicore hosts.

A. QSim Implementation

QSim is derived from QEMU, an open source full system software emulator that incorporates a dynamic binary translator to support multiple instruction sets [3]. Central to QEMU is a CPU emulator wherein guest code is translated on-demand into host code and placed into a translation cache for execution and potential reuse. QSim [16] is a library that wraps the QEMU CPU emulator with a thread-safe callback-based API, as illustrated in Figure 4. QSim executes a *guest* consisting of a lightly-modified Linux kernel and a benchmark application. The design of QSim allows multiple guest threads of the application to execute simultaneously in multiple host threads, similar to the design of Coremu [26].

QSim instantiates a separate QEMU CPU emulator for each guest thread - the virtual CPU (vcpu) in Figure 1. Each of these instances has an independent translation cache and main loop, but they all share a common memory state. Instruction-level control of execution is provided using a simple user-level cooperative threading library. QSim provides instrumentation of the emulated instruction streams by adding call-backs to each translation cache. Logically, one can think of each vcpu producing a stream of instructions executed by that vcpu, accompanied by detailed information about each instruction such as virtual and physical addresses. Instruction execution progress of each instruction stream is controlled by the corresponding core timing model thereby providing some back-pressure to control progress. Instances of the core timing model call the `run()` member function of a QSim emulator object simultaneously and their interactions through the shared memory space are mediated by a global lock table as shown in Figure 4. This lock table enables multiple ordinary accesses to occur simultaneously or a single atomic read-modify-write operation, but not both at the same time; behaving identically to the readers-writer lock, with ordinary accesses replacing reads and atomic operations replacing writes.

To avoid the time necessary to boot the Linux guest OS, state files are used. These simply contain a core count, memory size, register states for every virtual core, and a zero-run compressed copy of the memory state.

Currently QSim supports x86 guests. Since the QEMU CPU emulator supports multiple instruction sets, support for other ISAs is relatively straightforward being a matter of instrumenting the corresponding translation cache with the QSim callbacks. Finally, QSim currently supports 32-bit executables with 64-bit support in progress.

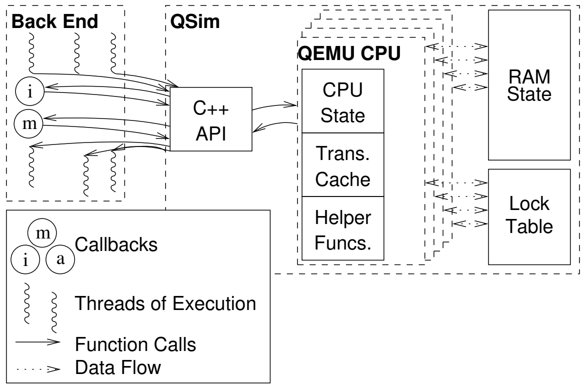


Fig. 4: QSim internals.

B. Benchmark Programs

QSim supports a special callback type for what are known as "magic" instructions. Using a technique taken from COTSon [1], x86 `CPUID` instructions for which there is no standardized definition are used as a form of out-of-band communication between the guest environment and host simulator. This allows a variety of behaviors to be instrumented, but has proven most useful for signalling the beginning and end of regions of interest in guest applications and the loading of applications into the guest after the operating system has booted.

A variety of benchmarks, including all of the SPLASH-2 [27] and PARSEC [5] programs, have been instrumented to mark their regions of interest in a way QSim can recognize. These are provided as `tar` archives which can be loaded directly by QSim, providing a convenient set of benchmark applications for use in simulations.

C. QSim Server

To facilitate the parallel execution of models across multiple blades in a cluster, Manifold supports a configuration referred to as QSIM server. LPs participating in the simulation connect to the centralized QSim server using a client library providing virtually the same API as a locally-hosted QSim library. Since QSim itself is inherently parallel, calls can be received from the various LPs in the simulation asynchronously. To reduce the adverse impact on performance of the latency of the TCP/IP communications between the LPs and the server, proxy processes are used to pre-fetch instruction streams from the server and keep the timing models (LPs) busy and never waiting on the front-end emulator - see Figure 1. A multithreaded proxy process is created for each host blade that executes the timing model. The process receives instructions for one or more QSim guest threads and places the instructions in shared memory segments to be consumed by the corresponding LPs running on the same host blade. The proxy continuously monitors the shared memory segments. When the number of instructions in a segment falls below a threshold, it communicates with the server to advance the corresponding guest thread's execution and receive more instructions.

IV. Timing Models

Timing models are encapsulated as interconnected Manifold components. As long as the component implements the Manifold component interface, the fidelity of the individual component model can range anywhere from fine-grained to coarse grained. As described in Section II, components can subscribe to a clock (time-stepped operation) or operate in an event-drive fashion (discrete event simulation) or utilize both models. A summary of components used in the exemplar simulator described in this paper are described in the remainder of this section.

A. Core Models

Several core component models are available. The most detailed is Zesto [18] - a cycle-level x86 processor simulator built on top of SimpleScalar [7], modified and integrated as part of the Manifold simulation framework. This component can be configured for out-of-order or in-order execution and tracks detailed state of a modern, Intel-style core. This model executes at about 50-200 KIPS depending on the benchmark. There are currently three front-end interfaces integrated, namely, PIN trace, QSim Library, and QSim Server.

SPX is a much faster core model that seeks to faithfully emulate timing but not the detailed microarchitecture state. It executes about 5X-10X faster than Zesto. Even faster core models have been constructed that are much less accurate, e.g., 1-CPI models, but which may suffice for certain types of analysis. These models are typically constructed with analogous front-end interfaces.

B. MCP-cache

Manifold provides a directory-based coherent cache model called *MCP-cache*, which is an implementation of the Manager-Client Pairing (MCP) [4] methodology. MCP-cache implements a cache system consisting of two levels of caches. Coherence is enforced at the L1 cache and the L2 cache serves as the directory. The most important feature of MCP-cache is its layered design that separates the implementation into a protocol-independent layer and a protocol layer. Adding a new coherence protocol only requires changing the protocol layer. MCP-cache can be used with any processor model that meets Manifold's requirements for the processor-cache interface.

C. Network Components

Manifold includes cycle-level interconnection network components. The Iris network model consists of network interface and router components. Each interface is connected to one router, and vice versa and performs flit-packet assembly and disassembly. The routers are interconnected in the way dictated by the selected topology. To use the network, components such as caches and memory controllers are connected to the network interface. These components are referred to as *terminals*. Each interface can connect to at most one terminal. To support mix-and-match of components, as discussed in Section II, the network interface defines a required interface such that any data type can be sent to the network interface as long as it implements the required interface.

D. Memory Controller and DRAM

CaffDRAM is a cycle level timing model for a DRAM module and associated memory controller that models the memory system as a hierarchy of channels, ranks, and banks. It models various timing parameters pertaining to resource contention within the DRAM memory. The memory access protocol is similar to what is described in [15]. CaffDRAM supports credit-based flow control, and can accept generic memory request conforming to the interface requirements.

V. Parallel Simulation Kernel (PDES)

Manifold's parallel simulation kernel is a software module that encapsulates the parallel discrete event simulation (PDES) functionalities of Manifold. Through its interface it provides PDES services to the upper layers and at the same time hides such PDES details as messaging passing, data serialization/de-serialization and synchronization algorithms, enabling the user of the system to easily create system models and run parallel simulations.

Important features of the simulation kernel include the following:

- Transparent support for both sequential and parallel simulations.
- Support for three time bases: ticked (integer) time, floating-point time, and mixed time, allowing link delays between components to be specified in either clock cycles or floating-point values.
- Support for three simulation paradigms based on the time base: time-stepped simulation, discrete-event simulation, and mixed simulation.

The last feature is important for efficiency in multi-scale models where components span widely different time scales. For example, the detailed core microarchitecture would be time stepped while associated models of the off-chip interconnect or I/O subsystem would be more efficiently served by discrete event support. Full system models would benefit for concurrent support for both.

The following subsections give a brief description of the kernel's simulation services and synchronization algorithms.

A. Simulation Services

Manifold's simulation kernel provides services specific to simulations of computer architectures, as well as standard discrete event simulation functions.

1) Standard PDES Functions: Manifold provides a set of standard PDES API functions including i) simulation start and termination, ii) event management & scheduling and iii) statistics collection. Other services transparently provided include data serialization/de-serialization (used in parallel simulation), and synchronization algorithms. This transparency allows users to build parallel simulations with little to no knowledge of PDES.

2) Clock Functions: The clock functions allow users to create multiple clocks, register/de-register components with clocks, query a clock's current cycle, and so on. Thus multiple clock domains are easily created (simply register a component with a clock). Moreover, the model developer is relieved from the task of managing time across multiple clock domains which can become quite complex in a parallel simulation - this is handled transparently by the kernel. In addition, the kernel supports the ability to change the clock frequency for a component. Thus, the voltage-frequency of a component can be easily changed thereby supporting dynamic voltage frequency scaling (DVFS).

3) Component Functions: Every component must be a subclass of the base class `Component`. The most important component-related functions include those for creating components and connecting components. When a component is created, it is assigned to a particular LP. When two components are connected, based on whether they are assigned to the same LP, the kernel creates appropriate data structures so that data serialization/de-serialization for the communication of events between components is handled in a transparent manner.

B. Synchronization Algorithms

In a parallel simulation, there are multiple LPs, each maintaining a local event queue and virtual clock. As events are processed, LPs generate not only local events but also events for each other. From any individual LP's point of view, since it may receive external events, ensuring events are processed in the correct order becomes a challenge. Synchronization algorithms have been created to address this problem and several standard techniques have been developed within the broader discrete event simulation community.

Manifold currently supports several synchronization algorithms, including *lower bound time-stamp (LBTS)* [13], *Null-message* (or CMB) [10], an enhanced Null-message algorithm called *Forecast Null-message* (FNM) [25], and *time quantum*.

The first three are conservative algorithms that ensure correctness by avoiding any *causality errors* [13], i.e., an LP will never advance without a guarantee that no event will be received from any other LP with a time stamp lower than its current time. Time quantum is an optimistic algorithm that does not provide such guarantees and permits causality errors to occur. By adjusting the time quantum, tradeoffs can be made between accuracy and simulation speed.

1) Lower Bound Time Stamp(LBTS): In LBTS, each LP holds a variable τ whose value is the smallest time-stamp of all of the currently scheduled events in the system. Events whose time-stamp are equal to τ can be processed. Whenever the next event to be processed has a time-stamp larger than τ, the LP cannot process the event and must instead participate in a collective computation that would determine the new value for τ. Once τ is updated, the LP will continue, and the process is repeated. This is typically overly conservative for multicore simulation and domain specific optimizations can be employed to speed up the simulation.

2) Null-message: In a conservative algorithm, events are processed only if the LP is certain it will not encounter an event in its past. This could easily lead to deadlock. The Null-message algorithm, also known as the CMB algorithm, is a deadlock avoidance algorithm that uses Null-messages to prevent deadlocks. A Null-message is a message that contains nothing but a time-stamp. Its time-stamp is the local time of the sender plus a positive value called *lookahead* and represents a lower bound for all future messages from the same sender. The lookahead is a domain-specific quantity of the simulation model, such as link delay between two components. The purpose of Null-messages is to help the receiver to advance its simulation time so that deadlocks are avoided. Additional optimizations are in place to minimize the total number of Null-messages.

3) Domain Specific Optimizations: To further improve the performance of the Null-message algorithm, we recently implemented the Forecast Null-message (FNM) algorithm from [25]. FNM advocates the use of domain-specific knowledge to improve system lookahead and significantly reduces the overhead of Null-messages. FNM consists of the following elements:

- Forecast of future inter-LP messages made by components based on their runtime state.
- Kernel functions for managing the forecast made by the components.
- Enhanced Null-messages that carry the forecast information in addition to its time-stamp.
- An algorithm that improves lookahead based on the forecast information from both sides of inter-LP links.

For example we exploit the fact that misses from the L2 cache are always separated in time by at least the hit time of the L2 cache.

4) Time Quantum: In Time Quantum, LPs execute independently and do not synchronize with each other until they reach a point that is determined by the quantum Q i.e., the LPs synchronize at time step $Q, 2Q, 3Q$, and so on. Time quantum clearly does not seek to preserve causality across LPs and the value of Q can be traded off against simulation speed. There are however, two instances in which such a tradeoff is quite useful. The first is simulations of power/energy and related thermal behaviors. In such cases, if a small percentage of events result in causality violations the perturbation in the power and thermal results can be negligible. The second case is related to the partitioning of the model across LPs. For example, consider the case where each core, L1 and L2 slice are in a distinct LPs and the network is in a separate LP. If the core receives a message from the past, this is indistinguishable from a message received at the current time but delayed in the network due to congestion.

VI. Integration of Physical Models

Modeling physical phenomena in conjunction with microarchitecture models is an essential feature of microarchitecture design space exploration. Within Manifold, this is supported through an interface to an integrated library of open source models for energy, thermal, reliability, and cooling to support transient and steady state simulation of these phenomena in conjunction with application execution on cycle level timing models.

The standardized interface with Manifold effectively removes the cross-dependency between physical and microarchitectural models; adding, updating, or removing physical models are independent of the microarchitecture and the model. The integrated physical models (including common models such as McPAT [17] and HotSpot [14]) are placed into four library classes: energy, thermal, reliability, and sensor libraries. The interface utilizes the same function calls to invoke different models that belong to the same library class, and thus replacing models does not require changes in microarchitecture simulators.

The model of a processor for physical modeling differs from the model used for simulation. In microarchitecture simulation, the processor is expressed as a hierarchical composition of functional blocks (i.e., cores, caches, network, etc.). On the other hand, in physical modeling it is represented as a hierarchy of physical components, i.e., die stack, floorplan, package and power sources. Each functional component of the timing simulation is mapped to a component in the physical hierarchy, for example, the register file occupies location on a die floorplan while a set of cores occupy a die in a stack. Each functional and physical component in turn is linked to a physical model, e.g., for energy or thermal. For example, a core is linked to an energy model while the package is linked to a thermal model.

The functional simulation captures behaviors such as access counts and pushes these through the interface to the physical models. Internal to the physical modeling library are mechanisms to coordinate measurements for energy, reliability and thermal modeling. For example, time-stamped, sampled measurements (e.g., access counts, energy, etc.) are stored in data queues so that energy, thermal and reliability measurements may be coordinated across virtual time. For instance, changing the voltage of a processor or package-level component (which is emulating dynamic voltage scaling) updates the voltage of all its sub-components in the hierarchy, and invokes a callback function of linked physical models to recompute the voltage-dependent conditions and variables. Such data synchronization capability enables the models to emulate dynamic voltage and frequency scaling (DVFS) and power gating (PG) controls during runtime simulation. Furthermore, physical effects such as the impact of temperature on leakage power, or the impact of temperature on reliability are periodically updated within the library models.

VII. Performance Analysis

This section describes the performance of a Manifold simulation for a coherent, shared memory multicore processors focusing on its scaling behavior. Using the components described in Section IV, users can construct a cycle-level chip multiprocessor (CMP) architecture timing model comprised of out-of-order cores, coherent two-level cache hierarchy, flit-level interconnection network, and DRAM memory controllers. When connected with QSim (Section III), the model can simulate a full-system CMP running multithreaded or multiprogram applications on a single Linux operating system.

For evaluation purposes, we built and tested three CMP models with 16, 32, and 64 cores, respectively. The number of memory controllers are 2, 4, and 8, respectively. The core model used is the componentized version of Zesto. Each core has a private data L1 and a unified shared L2 slice. The hierarchy is coherent using the MESI protocol. We use 5×4, 6×6, and 9×8 torus networks for the 16-, 32-, and 64-core models, respectively. Importantly, all component interactions are flow controlled using credit-based flow control to reflect as accurately as possible interactions between real-world components. Thus congestion in the memory controller can filter back to a core via backpressure in the network.

Simulation runs were performed on a Linux cluster with two Intel Xeon X5670 6-core CPUs with 24 hardware threads on each node. The operating system on the cluster is RHEL release 6.3, and the MPI version is OpenMPI 1.5.4. For the functional front-end, we use QSim server along with QSim proxy as described in Section III. The server runs on its own node, while a proxy process runs on each node that hosts the LPs. The number of nodes used by the model is 1, 2, and 3 for the 16-, 32-, and 64-core models respectively while the number of hardware threads used to execute the model is 13, 22, and 40, for the 16-, 32-, and 64-core models, respectively.

The simulation kernel uses the FNM algorithm for synchronization, and every pair of cores along with their caches are assigned to one LP. The torus network is assigned to Y different LPs, where Y is the size of the y-dimension of the torus (we empirically observed this provided the best performance). Each row of routers is assigned to one LP. For demonstration purposes, we have selected six benchmarks from the PARSEC [5] suite and six benchmarks from the SPLASH-2 [27] suite for presenting the results here. In each test we executed the simulation for 200 million cycles across the region of interest. These regions were marked in the source and compiled to 32-bit x86 executables. Table I shows the running time of the simulations.

We present the weak scaling properties of the simulation, i.e., as we increase simulator parallelism we increase the number of simulated cores. We list results for both sequential and parallel simulations. Speedup of parallel simulation is calculated as $Speedup = T_{sequential}/T_{parallel}$, where $T_{sequential}$ and $T_{parallel}$ are the running time of sequential and parallel simulations, respectively. The speedup numbers are shown in parentheses in the columns for parallel simulations. As the table shows, for the 16-core model, we achieved a speedup of 4.2X to 5.6X, for the 32-core model, 5.9X to 8.6X, and for the 64-core model, 6.7X to 12.1X. Consequently, for a 64 core simulation the timing model is executing on 40 threads and the parallel efficiency is (12.1/40) or 30%.

TABLE I: Simulation running time in minutes.

	16-core		32-core		64-core	
Bench.	Seq.	Parallel	Seq.	Parallel	Seq.	Parallel
dedup	1095.7	251.4 (4.4X)	2134.8	301.3 (7.1X)	2322.9	345.3 (6.7X)
facesim	1259.3	234.9 (5.4X)	2614.2	303.6 (8.6X)	3170.2	342.3 (9.3X)
ferret	1124.8	227.8 (4.9X)	1777.9	255.6 (7.0X)	2534.3	331.3 (7.6X)
freqmine	1203.3	218.0 (5.5X)	1635.6	245.6 (6.7X)	2718.9	337.3 (8.1X)
stream	1183.8	222.7 (5.3X)	1710.6	244.3 (7.0X)	4796.4	396.2 (12.1X)
vips	1167.0	227.3 (5.1X)	1716.3	257.2 (6.7X)	2564.6	337.9 (7.6X)
barnes	1039.9	224.3 (4.6X)	1693.0	283.3 (6.0X)	3791.8	341.4 (11.1X)
cholesky	1182.4	227.2 (5.2X)	1600.3	245.7 (6.5X)	4278.3	402.1 (10.6X)
fmm	1146.3	229.6 (5.0X)	1689.8	253.6 (6.7X)	5037.2	416.1 (12.1X)
lu	871.2	156.4 (5.6X)	1475.8	204.6 (7.2X)	4540.3	402.7 (11.3X)
radiosity	1022.3	228.8 (4.5X)	1567.5	250.4 (6.3X)	2813.5	350.3 (8.0X)
water	671.5	158.4 (4.2X)	1397.3	236.7 (5.9X)	2560.1	356.3 (7.2X)

Table II shows the number of instructions simulated per second, in thousand-instructions-per-second (KIPS). Not surprisingly, the aggregate KIPS numbers for parallel simulations are much higher than those for the corresponding sequential simulations. Note that the serial KIPS rate drops with number of simulated cores reflecting the relative increase in the number of network and memory hierarchy events that must be serially processed. We also note that many other events such as flow control events between interacting entities are also modeled contributing to the drop in serial KIPS rate. However, the drop roughly tracks the drop from ideal speedup as can be observed from Table I. Recall, that we are executing detailed, cycle-level core models with the full microarchitecture state. As stated earlier, our goal was to push the limits of parallelism with as detailed models as possible.

The KIPS rate per hardware thread (PHT) is shown in Table III. We present this metric for assessing such parallel simulations. However, its interpretation requires care. For example, it can be dependent on the partitioning. The LPs that only contain network components do not contribute to the per hardware thread KIPS rate. Larger systems involve many more non-instruction execution events such as flow control, memory accesses, network interfaces, network routers, etc. The KIPS PHT rate does not capture the whole story but we feel is still useful to capture the scaling behavior if the goal is to simulate CMPs executing applications and system software. We need to study the effects of partitioning and timing models in more depth to understand how this behavior (KIPS PHT) can be improved.

A. Integrated Power and Thermal Modeling

As an example of the capabilities for integrated power and thermal modeling, this sections reports on an example where we construct a 4-core symmetric system model for 16nm technology for power and thermal runtime evaluation. The processor model resembles the Intel Nehalem microarchitecture,

TABLE II: Simulation KIPS.

	16-core		32-core		64-core	
Bench.	Seq.	Parallel	Seq.	Parallel	Seq.	Parallel
dedup	49.28	211.98	48.56	340.61	16.88	136.82
facesim	58.66	316.42	47.90	401.72	30.01	278.94
ferret	57.77	284.81	35.41	239.59	18.10	139.54
freqmine	57.15	314.99	37.01	248.60	19.37	140.18
stream	58.77	314.34	36.73	260.04	41.42	419.77
vips	58.03	298.09	34.92	236.06	18.02	131.02
barnes	30.66	151.32	32.62	168.74	39.61	219.63
cholesky	57.95	301.47	38.87	254.54	45.68	491.46
fmm	51.01	252.46	37.90	255.86	40.94	525.51
lu	50.93	155.74	39.78	119.46	46.23	485.04
radiosity	50.87	206.96	53.15	229.02	36.29	268.70
water	27.86	95.81	29.85	132.22	25.72	179.88
Mean	50.75	242.03	39.39	240.54	31.52	284.71
Median	54.08	268.63	37.45	244.10	33.15	244.16

TABLE III: Simulation KIPS per Hardware Thread.

	16-core		32-core		64-core	
Bench.	Seq.	Parallel	Seq.	Parallel	Seq.	Parallel
dedup	49.28	16.31	48.56	15.48	16.88	3.42
facesim	58.66	24.34	47.90	18.26	30.01	6.97
ferret	57.77	21.91	35.41	10.89	18.10	3.49
freqmine	57.15	24.23	37.01	11.30	19.37	3.50
stream	58.77	24.18	36.73	11.82	41.42	10.49
vips	58.03	22.93	34.92	10.73	18.02	3.28
barnes	30.66	11.64	32.62	7.67	39.61	5.49
cholesky	57.95	23.19	38.87	11.57	45.68	12.29
fmm	51.01	19.42	37.90	11.63	40.94	13.14
lu	50.93	11.98	39.78	5.43	46.23	12.13
radiosity	50.87	15.92	53.15	10.41	36.29	6.72
water	27.86	7.37	29.85	6.01	25.72	4.50

a typical Out-of-Order core consisting of 5 major components: frontend (FE), scheduler (SCH), integer unit (INT), float-point unit (FPU) and L1 data cache (DL1), as depicted in Figure 5. The cache model is Manifold's coherent cache model with a private L1 and a shared L2 cache. With respect to physical models we use Manifold integrated physical modeling library. The specific models from that library used in this example are McPAT [17] and 3D-ICE [24] from the thermal modeling library. The generation of thermal grids and thermal fields are built into the modeling interfaces so that we can generate thermal fields directly, online from the application execution.

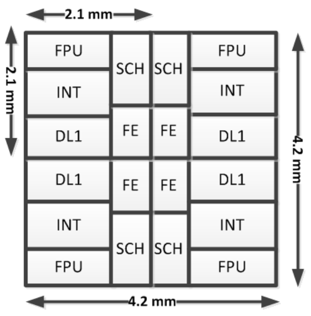
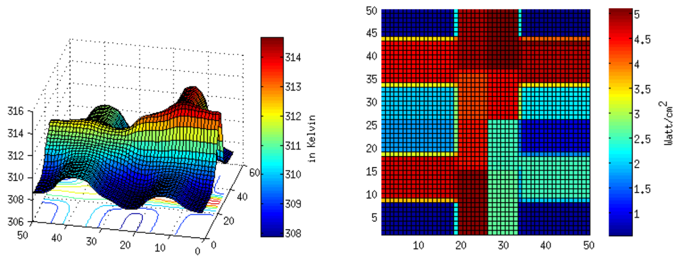


Fig. 5: Floor plan of a 4-core model.

In this example we assume the system starts at 3.0GHz with a supply voltage 1.0V, and the ambient temperature is set to 300K. Simulation runs for 500M cycles with fast forward execution across 100M instructions to skip the startup region. Figure 6 demonstrates the runtime power and thermal



(a) Processor thermal map. (b) Processor power density grids.

Fig. 6: Thermal and power grids for the 4-core model.

attributes of the 4-core system when running the sample test case *barnes* from the SPLASH-2 [27] suite. It is relatively straightforward to compute a host of additional useful metrics such as joules/instruction, ops/watt, and joules/op/mm^2. It is also easy to envision design and analysis of techniques such as power capping controllers, energy-driven thread scheduling (via feedback to the front-end) and analysis of power gating and dynamic power management techniques all driven by applications rather than off-line trace analysis. This simple example is intended to indicate the types of analysis that can be achieved with the integrated Manifold models.

B. *Modeling Asymmetric Processors*

Manifold's component-based framework simplifies the construction of asymmetric processor models of the type shown in Figure 7. Assuming a processor die area constraint, two different configurations are set up as described in Table IV: 4 homogeneous out-of-order cores vs 8 asymmetric cores (2 out-order cores and 6 in-order cores) as in Figure 7. Note that the use of a out-of-order or in-order core is a simple component replacement.

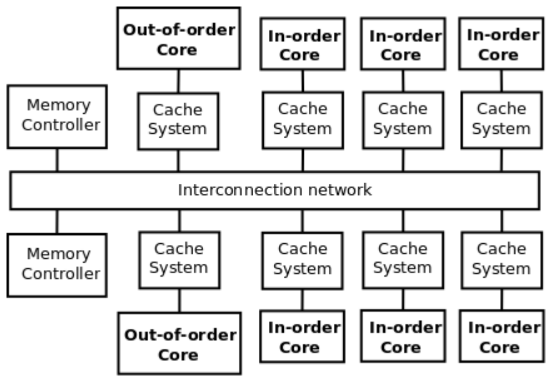


Fig. 7: Asymmetric processor composed of out-of-order and in-order cores in the Manifold framework.

Figure 8 shows an example of the type of measurements feasible - throughput differences of PARSEC [5] benchmarks between two different configurations, normalized to *blacksc-holes* with homogeneous cores. The values roughly track expectations based on relative differences in maximum IPC. One can envision other interesting capabilities with the integrated physical models.

TABLE IV: Asymmetric processor configuration

Configuration	Description	
Symmetry	Homogeneous	Asymmetric
Cores	4 out-of-order cores	2 out-of-order & 6 in-order cores
Cores	Out-of-order	In-order
Execution width	6	2 (FP/INT)
Issue width	4	1
Reorder buffer size	128	N/A
L1 cache	4-way assoc, 64-byte line, 32KB size	
L2 cache	8-way assoc, 64-byte line, 512KB size	
Clock frequency	2.0GHz	

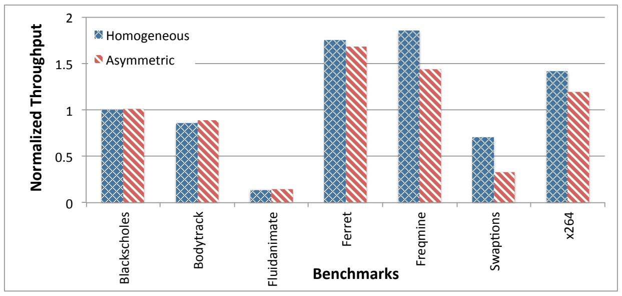


Fig. 8: Performance difference between homogeneous and asymmetric processor configurations, normalized to Blackscholes with homogeneous cores.

VIII. Lessons Learned

Integrating diverse component models for a multicore processor creates system level simulation issues that would not be apparent when developing point tools. Some of the more interesting we discovered were the following.

When a core, cache, router, and memory controller are interconnected, flow control between components is important - not just for the network. Otherwise results can correspond to infinite buffering at certain interfaces and results are incorrectly skewed. For example, memory controller models cannot simply schedule responses of memory requests on arrival, even if they are based on an accurate schedule created for prior requests. Limited buffering at the controller-network interface controls the *instantaneous* request rate that can in fact cause backpressure to percolate back to the cores (and should). Similarly, coherence messages should be buffered at the cache controller. This can eventually impact the request rate at the source core. No two hardware components interact without some form of flow control - neither should simulations.

Consider cases where component models at multiple time scales are integrated - a time stepped core integrated with a packet level end-to-end network latency model. The ratio of events in a core to events in the network can be a few orders of magnitude. Consequently, it is more efficient to have a time-stepped core model integrated with a discrete event network model. By abstracting the timing behaviors into the PSK API, components can be easily and transparently integrated without concern for core-to-core timing dependencies. Similar concerns occur for example if we wish to model application-driven power demand (voltage variations at 10s to 100s of nanoseconds) and core level power management (10s to 100s of microseconds) with thermal feedback (milliseconds).

IX. Background and Related Work

Parallel simulation for computer architecture has a long history dating back to at least the Wisconsin Wind Tunnel [21]. Over the years custom simulators for solving point problems emerged, however it has not been until the advent of multicore that major effort towards re-usable parallel simulation tools have emerged with a goal of scaling with model size. While there is a rich literature of simulation techniques for computer architecture, here we only point to those complete simulators with comparable goals, namely parallel, full system cycle-level simulation capability and do not discuss many excellent non-parallel environments such as GEM5 [6] and PTLsim [28].

The most similar to Manifold include COTSon [1] from which Manifold borrows several techniques including the basic ideas of the interface between the front-end and back-end. COTSon employed a proprietary front-end and contributed significant effort towards correctly mapping shared memory emulation to back-end timing models. More recently, Graphite [20] has emerged as a parallel multicore simulator. It is based on the use of PIN traces and implements a coherent shared memory abstraction across the host nodes. The PIN execution is instrumented to trap events that drive companion timing models while Graphite supports lax synchronization models to enable trade-offs between accuracy and simulation speed. Sniper [9] utilizes higher levels models while seeking to minimize accuracy compromise and is built on Graphite. One of the latest parallel simulators is ZSim [23]. Like Graphite, it uses PIN instrumentation for functional simulation. It runs simulation in time quanta. Each quantum is a few thousand cycles and is divided into a bound phase and a weave phase. In the bound phase, the cores are simulated without interactions with each other, but the memory access traces are recorded. Then the weave phase performs parallel simulation and uses the traces to simulate the memory accesses in order. Finally, the Structural Simulation Toolkit (SST) [22] is a component-based simulation environment that is most similar to Manifold - a component-based, parallel simulation infrastructure and accompanying simulators. SST provides an advanced system model specification infrastructure and automation in the construction of large scale models and addresses a wider range of scale and fidelity since the design point is targeted toward large scale simulations for high performance computing domain.

However, the preceding simulators occupy distinct points in space of simulator designs from Manifold. Graphite does not simulate operating system code nor is it targeted for correct, parallel, cycle-level operation (although it appears it could be). Nor do Graphite, COTSon, SST, Sniper or existing simulators integrate with physical modeling libraries - particular in a way that captures and models interactions. Manifold also supports hybrid timing models (mixture of time stepped and discrete event) and its design emphasizes composition of models - is more of an infrastructure for composing larger scale simulation models. Three standard APIs are promoted and supported in this regard - parallel simulation kernel, component, and physical modeling. In particular physical interactions between physics (e.g., thermal field) and architecture (e.g., leakage currents) and system properties (e.g., reliability) is modeled transparently. This enables a class of tradeoff analysis that has been difficult to achieve in the open source community.

With respect to the front-end, like Shade [12], an earlier single-threaded instrumented emulator, and Pin [19], a tool for instrumenting the native execution of binaries, QSim is a versatile front-end for execution-driven simulations, but it provides advantages when compared to either of these tools. Unlike these prior tools, QSim boots and runs a Linux kernel, allowing operating system code to be examined as well as user code. Further, the control provided by QSim is finer-grained. Each guest context can be advanced by the back-end timing models at the granularity of single instructions.

X. Conclusions And Future Work

This paper has described an infrastructure for creating parallel multicore full system, cycle-level simulation models. Our goal was to pick a demanding architecture simulation problem and attempt to scale it. Results were presented for the execution of standard benchmarks for up to 64 core simulations up to 200M cycles with speedups ranging from 4x-12x. Manifold enables the component-based construction of a range of parallel simulation models, ease of integration of new point tools, sharing of models, and integration of physical models. Our future work will focus on two challenges, among others. One is the incorporation of validation techniques. Even if individual components have been validated, the interactions can amplify deviations from actual behaviors. Another major challenge is to improve scalability and then extend the simulation capability to multiple operating system domains. The latter would enable simulation of future high blade count systems where Manifold's current instantiation deals with high core count within a blade or operating system domain.

Acknowledgment

This work has been supported by the National Science Foundation, under grant CNS-0855110, by Sandia National Laboratories, by HP Labs and by Oracle Labs.

References

[1] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "Cotson: infrastructure for full system simulation," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 52–61, January 2009.

[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Addison-Wesley, 2003.

[3] F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[4] J. G. Beu, M. C. Rosier, and T. M. Conte, "Manager-client pairing: A framework for implementing coherence hierarchies," 2011, pp. 226–236.

[5] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, 2009.

[6] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, and D. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, May 2011.

[7] D. Burger and T. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13–25, June 1997.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996, vol. 1.

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

[10] K. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transactions on Software Engineering*, vol. 5, no. 5, pp. 440–452, 1978.

[11] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*, 2nd ed. Addison-Wesley, 2011.

[12] B. Cmelik and D. Keppel, "Shade: a fast instruction-set simulator for execution profiling," in *1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1994, pp. 128–137.

[13] R. Fujimoto, *Parallel and Distributed Simulation Systems*. John Wiley & Sons, 2000.

[14] W. Huang, S. Ghosh, S. Velusamy, and K. Sankaranarayanan, "Hotspot: a compact thermal modeling methodology for early-stage vlsi design," *IEEE Transactions on VLSI Systems*, vol. 14, no. 5, pp. 501–513, 2009.

[15] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

[16] C. Kersey, A. Rodrigues, and S. Yalamanchili, "A universal parallel front-end for execution driven microarchitecture simulation," *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation Methods and Tools*, pp. 25–32, 2012.

[17] S. Li, J. Ahn, R. Strong, J. Brockman, D. tullsen, and N. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (Micro-42)*, 2009, pp. 469–480.

[18] G. Loh, S. Subramaniam, and Y. Xie, "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," *International Symposium on Performance Analysis of Software and Systems*, pp. 53–64, 2009.

[19] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 190–200.

[20] J. Miller, H. Kasture, G.Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[21] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood, "The wisconsin wind tunnel: virtual prototyping of parallel computers," in *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1993, pp. 48–60.

[22] A. Rodrigues, K. Hemmert, B. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B.Jacob, "The structural simulation toolkit," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, March 2011.

[23] D. Sanchez and C. Kozyrakis, "Zsim: fast and accurate microarchitectural simulation of thousand-core systems," in *40th Annual International Symposium on Computer Architecture*, 2013, pp. 475–486.

[24] A. Sridhar, A. Vincenzi, M. ruggiero, T. Brunschwiler, and D. Atienza, "3d-ice: fast compact transient thermal modeling for 3d ics with inter-tier liquid cooling," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2010, pp. 463–470.

[25] J. Wang, Z. Dong, S. Yalamanchili, and G. Riley, "Optimizing parallel simulation of multicore systems using domain-specific knowledge," in *2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, 2013, pp. 127–136.

[26] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, W. Zhang, and B. Zang, "Coremu, a scalable and portable parallel full-system emulator," in *16th ACM Symposium on Principles and Practice of Parallel Programming*, 2011, pp. 213–222.

[27] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," in *22nd Annual International Symposium on Computer Architecture*, 1995, pp. 24–36.

[28] M. Yourst, "Ptlsim: a cycle accurate full system x86-64 microarchitectural simulator," in *IEEE International Symposimum on Performance Analysis of Systems and Software (ISPASS)*, 2007, pp. 23–34.