SynchroTrace: Synchronization-aware Architecture-agnostic Traces for Light-Weight Multicore Simulation

Siddharth Nilakantan*, Karthik Sangaiah*, Ankit More*, Giordano Salvador†, Baris Taskin*, and Mark Hempstead*
*Department of Electrical and Computer Engineering
Drexel University, Philadelphia, PA USA
Email: {sn446, ks499, am434}@drexel.edu, {taskin, mhempstead}@coe.drexel.edu
†Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA USA
Email: gsalv@seas.upenn.edu

***Abstract*—Trace-driven simulation of chip multi-processor (CMP) systems offers many advantages over execution-driven simulation, such as reducing simulation time and complexity, and allowing portability, and scalability. However, trace-based simulation approaches have encountered difficulty capturing and accurately replaying multi-threaded traces due to the inherent non-determinism in the execution of multi-threaded programs. In this work, we present SynchroTrace, a scalable, flexible, and accurate trace-based multi-threaded simulation methodology. The methodology captures synchronization- and dependency-aware, architecture-agnostic, multi-threaded traces and uses a replay mechanism that plays back these traces correctly. By recording synchronization events and dependencies in the traces, independent of the host architecture, the methodology is able to accurately model the non-determinism of multi-threaded programs for different platforms. We validate the SynchroTrace simulation flow by successfully achieving the equivalent results of a constraint-based design space exploration with the Gem5 Full-System simulator. The results from simulating benchmarks from PARSEC 2.1 and Splash-2 show that our trace-based approach with trace filtering has a peak speedup of up to 18.4x over simulation in Gem5 Full-System with an average of about 7.5x speedup. We are also able to compress traces up to 74% of their original size with almost no impact on accuracy.**

I. Introduction

As chip multi-processors (CMPs) are the predominant type of architecture employed in modern systems, system designers require dependable simulation methodologies for parallel CMP-based systems. Trace-driven simulation of CMP-based systems has significant benefits over execution-driven simulation, such as reducing simulation complexity and simulation time, allowing portability, and scalability. However, execution-driven simulation is still typically used to evaluate CMPs due to the difficulty of reliably generating and accurately replaying multi-threaded traces [9].

Existing methodologies that capture traces for multi-threaded applications are currently inadequate for CMP design space exploration. PinPlay is one such methodology that captures multi-threaded traces in the form of *pinballs*. Pinplay is used for *deterministic and reproducible replay*, and it supports multi-threaded applications [19]. However, the timing associated with the execution of multi-threaded applications has inherent non-determinism, due to the presence of synchronization and other run-time factors. PinPlay's traces and replay do not model this non-determinism accurately during simulation. As a result, design space exploration of a CMP with PinPlay may lead to sub-optimal design choices. Additionally, there are currently no publicly available simulators that support pinballs of multi-threaded applications. Another trace-based solution, proposed by Rico et al., is a hybrid execution-driven and trace-driven methodology for simulation [21]. However, their methodology requires source to source transformation to interface their synchronization calls with their simulation framework. Furthermore, their simulation framework is not fully validated with a known CMP or CMP system simulator. The work presented in this paper overcomes the shortcomings of previous approaches by using *synchronization- and dependency-aware, architecture-agnostic* multi-threaded traces.

We propose SynchroTrace, a two-step methodology for trace-based simulation of multi-threaded applications: **i)** The generation of synchronization- and dependency-aware architecture-agnostic traces and **ii)** A lightweight replay mechanism that respects those dependencies, simulates synchronization actions, and handles simple scheduling for threads for playback on any target hardware platform. The tracing methodology utilizes dynamic binary instrumentation to trace through the program. Within the traces, *events* of different types are identified to separate computation, synchronization, and communication in shared memory multi-threaded programs. The replay mechanism parses these events and inserts them appropriately into the computation/memory stream during playback. We show how our methodology can be used to achieve accurate design space exploration and its flexibility in terms of speed and accuracy trade-offs. In this paper, we refer to a simulation flow that integrates SynchroTrace with a cache and NoC simulator as the "SynchroTrace simulation flow".

The rest of the paper is organized as follows: we present SynchroTrace: synchronization- and dependency-aware, architecture-agnostic multi-threaded traces, in Section II and a dependent replay mechanism (i.e. to complete the simulation flow) in Section III. We validate SynchroTrace by comparing our trace-based simulation results for a CMP design

978-1-4799-1957-4/15/$31.00 ©2015 IEEE

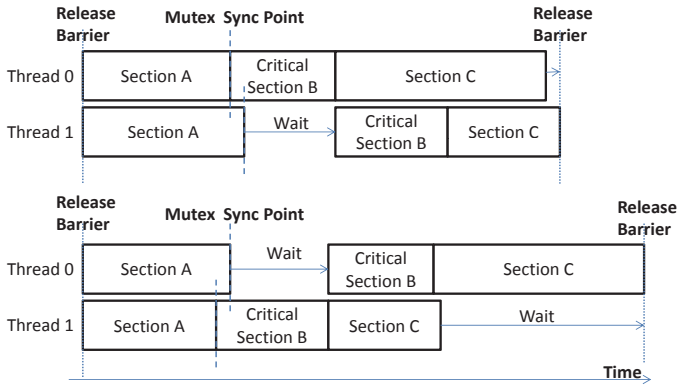


Fig. 1: **Non-Determinism in Thread Execution.** Uneven thread progress and indeterminate wait times at synchronization points cause non-determinism that potentially causes different thread interleaving for different runs.

space exploration against the Gem5 Full-System simulator results in Section IV. In Section V, we describe the performance improvement of SynchroTrace over full-system simulation and present trace-based optimizations for speedup of CMP architecture simulations. Finally, we compare SynchroTrace with related work in Section VI.

II. Synchronization- and Dependency-aware Traces

In the context of architecture simulation, *traces* refer to a record of the chronological sequence of events that occur in a program. A trace-driven simulation flow takes two passes: trace generation and trace replay. Traces can be recorded at different levels of the system depending on which subsystem is being designed. For example, an instruction trace records all the instructions in the dynamic stream in chronological order. It can be used when detailed CPU models are required. Similarly, memory traces record only the LD/ST instructions from the dynamic stream [5]. Memory traces can be used in conjunction with very simple CPU models in order to do more detailed simulation of just the "uncore" [13]. However, traditional instruction and memory traces cannot accurately model multi-threaded applications in simulation due to the non-determinism in thread execution. In this section, we describe the importance of modeling non-determinism in thread execution, our solution through synchronization- and dependency-aware traces, and implementation details on how the traces are captured.

A. Non-Determinism in Multi-Threaded Programs

Traces are convenient and portable for simulation, but due to the non-deterministic execution of multi-threaded applications, simulation using traces of multi-threaded applications has proven difficult and been attempted only a few times [17], [19], [21]. The non-determinism manifests as uneven thread progress between synchronization points and indeterminate wait time at synchronization points. Design time factors, such as CMP design configuration and static thread mapping, as well as run-time factors, such as OS load on the cores or dynamic thread mapping, can play a role in impacting thread progress differently. A particular state of relative progress between different threads is sometimes termed *thread interleaving* [19], [21]. The examples in Figure 1 are the result of two different *thread interleavings*. The non-determinism arising from the possibility of different thread interleavings can subsequently affect performance metrics, such as cycle time, core utilization, memory bandwidth, peak traffic, and energy footprint of a multi-threaded application.

An example of thread non-determinism via different thread interleavings is illustrated with an example in Figure 1. This figure depicts a portion of execution for an application containing two synchronizing threads, between two barriers, i.e. a barrier region. Each thread must complete Sections A, B, and C in sequence, and both threads must go through the Critical Section B in a mutually exclusive manner (enforced by mutex synchronization). A mutex synchronization point allows the first arriving thread to progress while the other has to wait, and a barrier only allows progress when all registered threads have arrived. Two scenarios of thread progress are shown on the top and bottom with slightly different execution times for Section A across the scenarios. This minor difference in the timing of Section A has a big effect on the wall-clock time.

The wall-clock times vary between the two scenarios. as explained below. In the scenario shown on top of Figure 1, *Thread 0* arrives at Critical Section B first due to the relative timing of Section A, and vice versa for the bottom scenario. As *Thread 0* has a longer Section C to complete, it would benefit from completing Critical Section B first. In the top scenario, *Thread 1* waits at the critical section for *Thread 0* to finish first. In the bottom scenario, *Thread 0* waits at the critical section for *Thread 1* to finish first. Since *Thread 0* is allowed to progress through the critical section first in the top scenario, both threads reach the barrier after Section C quicker.

The minor difference in the execution time of Section A represents uneven progress of execution between synchronization points in multi-threaded programs. This is one manifestation of non-determinism. The different wait times at the synchronization points in both threads is another manifestation of non-determinism. It is thus important to model the impact of non-determinism during simulation.

We propose that a trace-driven simulation flow for multi-threaded applications should not record and enforce a specific thread interleaving. Instead, a trace simulation flow must allow for thread interleaving to be determined by hardware architecture and run-time factors during replay. Thus, SynchroTrace records some architecture-independent information for each thread, which allow for correct modeling of wait times at synchronization points and uneven progress during simulation.

B. Trace Characteristics

The synchronization- and dependency-aware traces classify all information into three categories of run-time events: Computation events, Thread synchronization events, and Communication events. Each type of event is described below with the fields of each event outlined in Listings 1–3.

Computation Events represent local processing performed by a thread, completely independent of other threads. For each trace to remain **i)** ISA- and microarchitecture-agnostic, **ii)** fast, and **iii)** easily compressible, the traces only contain abstract computation events and not detailed instructions. Computation events contain counts of *Integer Operations (IOPS)*, *Floating Point Operations (FLOPS)*, *Memory Writes*, and *Memory Reads* to locations written by the same thread. The set of

unique read and written (virtual memory) addresses are stored with the event as well, with special symbols such as $ and * delimiting the lists.

Thread Synchronization Events contain the type of pthread API call and the address of the data structure used, so that a particular synchronization object can be recognized. Thread synchronization events are interpreted during simulation, and the action appropriate for the synchronization type (i.e. barrier, mutex lock, conditions etc.) is applied for participating threads. Synchronization events mediate accesses to shared resources. When the traces are replayed, the appropriate waiting time for each thread at this synchronization point is determined on-the-fly by the Replay framework described in Section III.

Communication Events represent *communication edges* between threads. A communication event is necessary for modeling communication occurring between threads that may not be fully transparent to the capture framework, such as user-level synchronization or memory traffic within the kernel as explained in Section II-C. A communication event in the consuming thread is associated with a particular *computation event* in the producing thread. Communication events can potentially hold references to data from multiple producer threads. A communication edge can generate coherence traffic when the producing event and consuming event have temporal proximity. However, since there is a possibility of different thread interleavings between capturing and replaying the trace, it is not possible to predict, ahead of simulation, whether the producing and consuming events of different threads will indeed be close in simulation time. Thus, we capture the communication event into the trace of the consuming thread, and when replaying the trace, we enforce the dependency between the consuming and producing thread.

Listing 1: **Computation Event**

```
Event Number, Integer Op Count, Floating Point
Op Count, Memory Read Count, Memory Write Count $
Unique Addresses Written * Unique Addresses Read
```

Listing 2: **Synchronization Event**

```
Event Number, pth_ty: Pthread_Call_Type ^ Address of
Synchronization Structure
```

Listing 3: **Communication Event**

```
Event Number # Producer Thread, Producer Event,
Address Range
```

An excerpt of a single thread's trace using fields from Listings 1–3 follows:

Listing 4: **Single Thread's Trace Example**

```
1774522,1,0,0,1 $ 132941440 132941447
1774523,1,0,0,1 $ 132941448 132941455
1774524 # 1 4534 7048536 7048543
1774525,1,0,1,0 * 132941388 132941391
1774526,1,0,0,0
1774527,pth_ty: 5 ^ 67113320
1774528,114,0,0,1 $ 132941456 132941463
1774529,3,0,1,0 * 132941560 132941567
1774530 # 1 5870 7048472 7048479
```

The example in Listing 4, of events *1774522* to *1774528*, shows the uncompressed version of the trace where we allow at most one memory read or write per event; the events representation also allows for multiple consecutive operations which fall under the computation or communication categories to be merged together (detailed further in Section V). The first two lines show computation events *1774522* and *1774523*. It can be observed that each event records one memory write and one integer operation with the addresses for the memory writes are shown after the $ symbol. Event *1774524* is a communication event with this thread reading from Thread 1's event *4534* through the addresses 7048536-43. Event *1774524* is a computation event that recorded one memory read and one integer operation, with the addresses read shown after the * symbol. The next event does not contain any memory operations as a synchronization operation intervened before it could record any memory operations, necessitating a synchronization event *1774527*. The synchronization event is of type 5, which represents a barrier, with the barrier address being 67113320.

C. Trace Capture Framework

SynchroTrace's capture tool is based on the Sigil workload analysis framework [16] built on top of the Valgrind Dynamic Binary Instrumentation framework. While Sigil was designed to capture communication between functions, we adapted it to register threading API calls and capture communication between threads. This capture tool monitors the execution of a program and builds sequences of computation, synchronization, and communication events for each application thread. We periodically dump the trace to a file so as to efficiently manage the amount of state held in memory during the trace gathering process. This keeps the capture tool lightweight and fast. The particular methods used to capture synchronization and communication events are discussed as follows.

1) Capturing synchronization events: Accurate modeling of non-determinism requires respecting any thread interleaving that could occur during simulation, irrespective of the interleaving encountered during capture of the trace. Two features of our capture framework allow us to model the non-determinism across various simulation runs by allowing the interleaving to be determined by the application and runtime factors mentioned earlier. The first feature is the capture

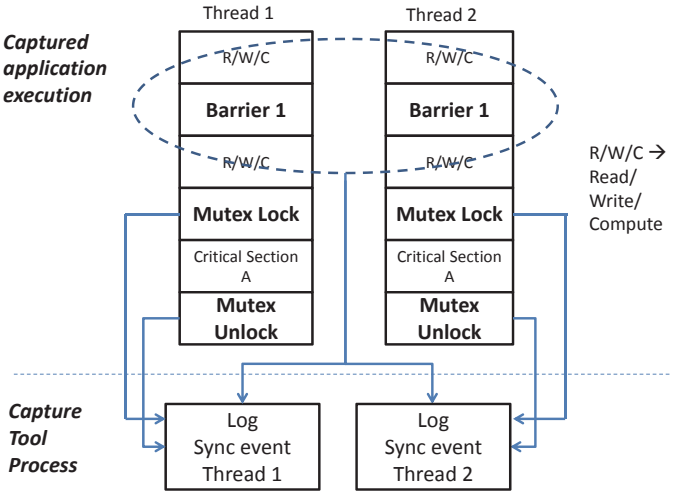


Fig. 2: Intercepting Pthread API Calls

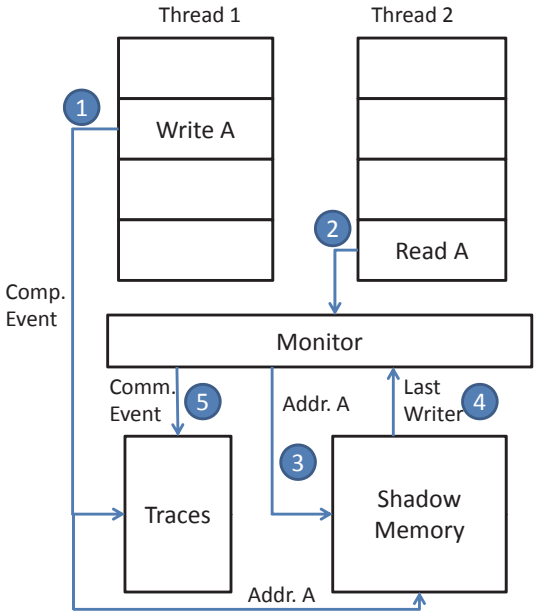


Fig. 3: Capturing Computation and Communication Events

of a separate trace for each thread which contains memory and compute operations captured in program order for that thread. The second feature is the capture and logging of synchronization events in each trace. Figure 2 illustrates an example of how we intercept pthread API calls to generate synchronization events. The trace capture mechanism uses Valgrind's function wrapping feature to intercept pthread API calls [2]. Depending on the type of synchronization encountered, a synchronization *event* is logged in the trace for one or more threads. We currently capture pthread_create/join, pthread_mutex_lock/unlock, pthread_barrier_signal/wait, and pthread_condition_wait.

SynchroTrace cannot capture threading activity when standard threading API calls are not used in the traced program, as it is not possible to infer synchronization at the assembly level in Valgrind. This can occur in cases where condition variables are explicitly written in user code, or critical sections using low-level locks are encountered in the kernel [17]. We capture communication events to handle these cases and interpret them as dependencies between the threads.

2) Capturing communication events: The capture tool monitors communication through memory addresses, with the help of a Shadow Memory [15]. Memory shadowing is an efficient way of holding an object of data for every address touched by the program. We use each object to hold the last writer of its corresponding address. Figure 3 presents an example of this process. The numbered circles indicate the sequence of dynamic steps performed in capturing the trace from an application as it runs natively. When a **store** to address A occurs in Thread 1, a computation event is emitted to the trace for Thread 1. This address is also emitted simultaneously to a Shadow Memory, which stores Thread 1 as the last writer for address A. Subsequently, in step 2, a **read** to address A occurs in Thread 2; this implies a communication edge. The address is sent to a monitor which checks against the Shadow Memory to determine the thread who last wrote to address A [15]. The last writer information is sent back to the monitor which decides if this was an inter-thread communication edge. In this example

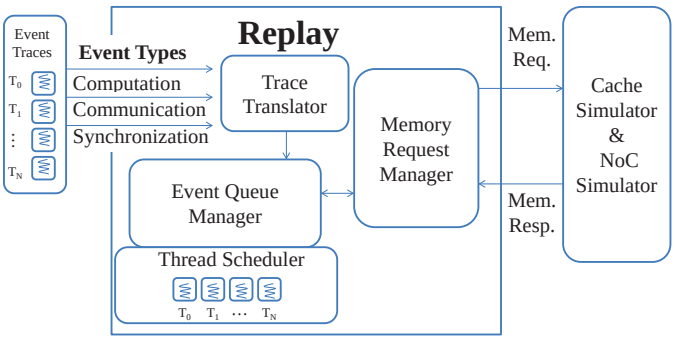


Fig. 4: Multi-Threaded Event-Trace Replay Framework

of an inter-thread communication edge, the monitor emits a communication event to the trace for Thread 2.

3) Capturing Operating System traffic: SynchroTrace's capture framework intercepts information related to Operating System (OS) actions, albeit currently in a limited fashion using communication events. Since our capture framework is built on Valgrind, SynchroTrace shares Valgrinds inability to capture any computation, communication, or synchronization events within the kernel. However, Valgrind can intercept system calls and report an aggregate of the memory addresses read and written within a system call. Thus, SynchroTrace embeds the aggregate information into computation and communication events in the trace for each thread, though the sequence of memory traffic within the kernel will not be preserved. We thus conservatively treat reads that consume from memory writes within the kernel as dependencies that a thread will be required to wait on through communication events.

Our traces are captured quicker than a full-system simulation-based trace capture as our traces are derived from native runs of the program. The *events* representation allows for more size efficient traces by only holding detailed information for the most important events. As the traces have synchronization and dependencies embedded in them, they can be used for architecture simulations and also can be post-processed to infer useful information about the workload. We will demonstrate the latter in Section V.

III. Event-Trace Replay Framework

For architecture simulations, a replay mechanism is required to process the trace and generate architectural events. The replay mechanism dynamically generates the appropriate actions for all *events* during simulation while providing light-weight thread scheduling and management. As shown in Figure 4, the captured event-trace sends computation, communication, and synchronization events for each thread into the Replay framework. Within Replay, the individual events are processed via the Trace Translator into individual Replay event data structures and passed into the Event Queue Manager (EQM). The EQM also interfaces with the Memory Request Manager (MRM) to send memory requests. The MRM interfaces with an external cache simulator and generates response back to the EQM. The Thread Scheduler handles the thread creation, deletion, scheduling, and synchronization.

Despite the theoretical capability, some simplifying decisions have been made in the SynchroTrace framework implementation as follows: the current playback mechanism with the

multi-threaded traces uses simple timing models for in-order cores. SynchroTrace currently accounts for the progression of the modeled core's cycle time using a 1-CPI timing model with detailed timing models for the uncore. The core Replay infrastructure can be connected to more detailed timing models such as out-of-order cores. Our current Replay framework processes memory requests for the Ruby/Garnet simulators. However, the multi-threaded traces and replay mechanism are portable to any cache simulator.

A. Event Queue Manager and Memory Request Manager

As detailed in Algorithm 1, the EQM handles the progression of the events for each of the threads within the *EventQueue*. During each cycle, the EQM checks if there are events ready to be processed from threads for the current cycle. If there are no available events for the current cycle across all of the threads, the *CurrentTime* progresses to the next available event's scheduled wakeup time. Events scheduled to wake up in the current time are handled by the process represented in Algorithm 2.

ProcessEvent is described in detail in Algorithm 2. For computation events, the EQM schedules the thread to wakeup after the cycle time required to complete the computation event based on the number of IOPS and FLOPS. When this thread wakes up at its scheduled clock time, the EQM will send a read or write memory request to the MRM and block the thread until the MRM triggers a memory response to the EQM. As shown in Figure 4, the MRM interfaces with the Cache and NoC simulators to obtain the correct timing for the memory request. As described by Lines 11–13 in Algorithm 1, after receiving a memory response from the MRM, the EQM will then queue the next event for the thread.

For synchronization events, the EQM sends **create** and **join** events to the Thread Scheduler. Upon processing mutex lock and barrier events, the EQM handles these events similar to the thread dependencies of the communication events; if a thread is unable to acquire a mutex lock or is waiting at a barrier, the thread will be rescheduled by the EQM to attempt again during the next cycle. If the synchronization event is successful, the thread will proceed to the next event. Synchronization events in the Replay framework generate memory requests, but these are omitted in Algorithm 1 to simplify the pseudocode.

For communication events, the EQM maintains the dependencies between consumer threads and the corresponding computation events of producer threads. While processing the communication event of a consumer thread, the EQM will check on the progress of the corresponding computation event of the producer thread. If the corresponding computation event has not been completed, the EQM will block the consumer thread from progressing. Once the corresponding computation event has been completed, the EQM will immediately issue the memory read of the communication event and block the consumer thread until the MRM triggers a memory response to the EQM.

B. Thread Scheduler

SynchroTrace can be integrated with any simulator that contains CMP architecture models. In a simulation flow that employs the SynchroTrace methodology, the Replay framework accepts the simulation parameters/configuration and configures the simulation back-end accordingly. This configuration process is independent of trace generation, so the number of threads being simulated does not necessarily correspond the number of cores. This necessitates a thread scheduler in the absence of the OS in trace-driven simulation. The Thread Scheduler handles the creation, deletion, scheduling, and synchronization of threads across any number of cores, including multiple threads per core. Currently, SynchroTrace's Thread Scheduler opportunistically swaps out stalled threads for threads ready to progress. Threads can be stalled on synchronization events, dependencies, or memory requests. Our thread scheduler performs a simple round-robin approach when multiple threads are ready to progress. While we do not currently model a cost for the scheduling actions, we intend on adding that cost to the simulation as well.

Algorithm 1 Event Queue Manager

```
1:  for all ThreadIDs in EventQueue[ThreadID] do
2:      for all Events in EventQueue[ThreadID] do
3:          if Event.TimeReady = CurrentTime then
4:              ProcessEvent                          ▷ Algorithm 2
5:          end if
6:      end for
7:  end for
8:  if AllEventsinEventQueue ≥ CurrentTime then
9:      ProgressCurrentTimetoNextEventTime
10: end if
11: if MemoryResponseTriggeredForThread then
12:     QueueNextEvent
13: end if
```

Algorithm 2 ProcessEvent

```
1:  if COMPEVENT then MemReq@(Comp.Time + CurrentTime) and WaitforResp.
2:  else if COMMEVENT then
3:      if Dep.EventCompleted then
4:          MemReq@(CurrentTime) and WaitforResp.
5:      elseScheduleThreadtoAttemptAgainNextCycle
6:      end if
7:  else if SYNCHEVENT then
8:      if Event = Create or Join then
9:          SendEventtoThreadScheduler
10:     else if MutexLockRequest then
11:         if MutexLockObtained then QueueNextEvent
12:         elseScheduleThreadtoAttemptAgainNextCycle
13:         end if
14:     else if MutexUnlockEvent then QueueNextEvent
15:     else if BarrierEvent then
16:         if LastThreadforBarrier then
17:             QueueNextEvent
18:         elseScheduleThreadtoAttemptAgainNextCycle
19:         end if
20:     end if
21: end if
```

IV. Design Space Exploration with Trace-based Simulation

SynchroTrace provides the means for accurate and efficient design space explorations ranging from low-power to highly-scaled CMPs. In this section, we demonstrate how the light-

weight SynchroTrace simulation flow can be used to select optimal CMP uncore design choices for a fixed in-order core, given uncore area and power constraints targeting CMPs. The uncore we are evaluating includes the L1 cache, L2 cache, NoC routers, and NoC links. We also show that our light-weight simulation flow yields the same result when using the cycle-accurate Gem5 Full-System simulator.

A. Experimental Methodology

Our experimental methodology consists of two sets of experiments. The focus of the first experiment is to use SynchroTrace to analyze the design space across cache sizes and network parameters for a given set of uncore area and power constraints with a fixed in-order core model. Specifically, we vary the L1 and L2 cache sizes, NoC virtual channels, NoC buffer depth, and NoC link bandwidth. The goal of this experiment is to accurately select the best performing design configuration, using the metric Cycles Per Instruction (CPI), under uncore power and area constraints. Although we capture cycles, we chose CPI as our performance metric in lieu of execution cycles so that all benchmarks simulated are weighted equally when assessed for design space exploration. To calculate CPI for both frameworks, we used the number of instructions obtained from SynchroTrace's capture tool. This kept the relative trends of CPI and cycles consistent for each individual benchmark. The focus of the second experiment is to perform an equivalent design space exploration using the cycle-accurate Gem5 Full-System simulator [5]. The goal of this experiment is to compare the cycle-accurate full-system simulator results against SynchroTrace's light-weight simulation flow for accuracy and speedup.

The base of the design configurations consists of a single 8-core chip, 2-level cache, and directory-based MESI protocol. The cache and network design parameters are detailed in Tables I and II, respectively. The CMP contains private L1 caches with an associativity of 4, a shared distributed L2 cache with an associativity of 8, and 64-byte blocks. The cores and NoC both operate at 1 GHz. The caches and NoC are designed for the 65nm technology with area and power given by Cacti 6.5 [14] for the caches and Orion 2.0 [12] for the NoC. The traces were captured on the Linux Kernel 2.6 in Red Hat Enterprise Linux 5 (RHEL5) with the standard POSIX Thread API. We benchmark the design configurations using applications from the PARSEC-2.1 [4] and Splash-2 [25] benchmark suites.

We use the SynchroTrace simulation flow illustrated in Figure 4. The traces are only generated once per benchmark and used for simulation of all 16 design points. To additionally show that the SynchroTrace simulation flow yields the same results as the Gem5 Full-System simulator, the SynchroTrace simulation flow uses the same cache and NoC simulators, Ruby and Garnet, that are used by the Gem5 framework. For the remainder of this paper, we hereby use the terminology of the "SynchroTrace simulation flow" to represent the integration of our traces and replay mechanism specifically with the Ruby cache simulator and the Garnet NoC simulator. For our comparisons, we use Gem5's TimingSimpleCPU core model which is a 1–CPI in-order model.

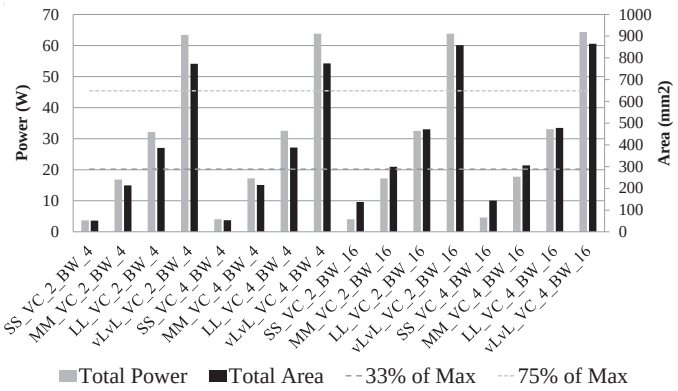


Fig. 5: Design Choices Under Area and Power Constraints

TABLE I: Cache Design Parameters

Cache Configs.	Cache Sizes
SS (Small)	L1I/D = 4kB; L2 Slice = 256kB
MM (Medium)	L1I/D = 16kB; L2 Slice = 1024kB
LL (Large)	L1I/D = 32kB; L2 Slice = 2048kB
vLvL (veryLarge)	L1I/D = 64kB; L2 Slice = 4096kB

B. Area and Power Constraints

The constraints for the pruning of the uncore design space are based on **1)** 75% and **2)** 33% of area and total power of the most resource-intensive design point (vLvL_VC_4_BW_16). Figure 5 illustrates the total uncore area and power for each design point and the corresponding constraints. Design points satisfying each of the design constraints (under respective dashed lines) are considered for further evaluation in this design space exploration.

It should be noted that the total area values calculated using Cacti 6.5 and Orion 2.0 are equivalent in both the SynchroTrace simulation flow and Gem5, as this computation is performed externally to the simulation solely using the design parameters.

In these experiments, detailed in Section IV-C, the SynchroTrace simulation flow selected the same design points under the constraints as Gem5. The consistency of the total power of the design points with both simulators is expected as the total power is largely dominated by the leakage power, which is application independent. The average difference in total power between the two simulators is roughly 1%.

C. Performance Results and Design Choices Under Constraints

Given the constraints in Section IV-B, our goal is to find the uncore hardware configuration that will yield the highest performance, which is inferred by the lowest CPI. Additionally, we investigate the accuracy of the design point selection by comparing the result against the selection of the cycle-accurate Gem5 Full-System simulator.

TABLE II: NoC Design Parameters

Network Configs.	Network Parameters
VC_2	Virtual Channels = 2, Buffer Depth = 4
VC_4	Virtual Channels = 4, Buffer Depth = 4
BW_4	Link Bandwidth = 4 Bytes
BW_16	Link Bandwidth = 16 Bytes

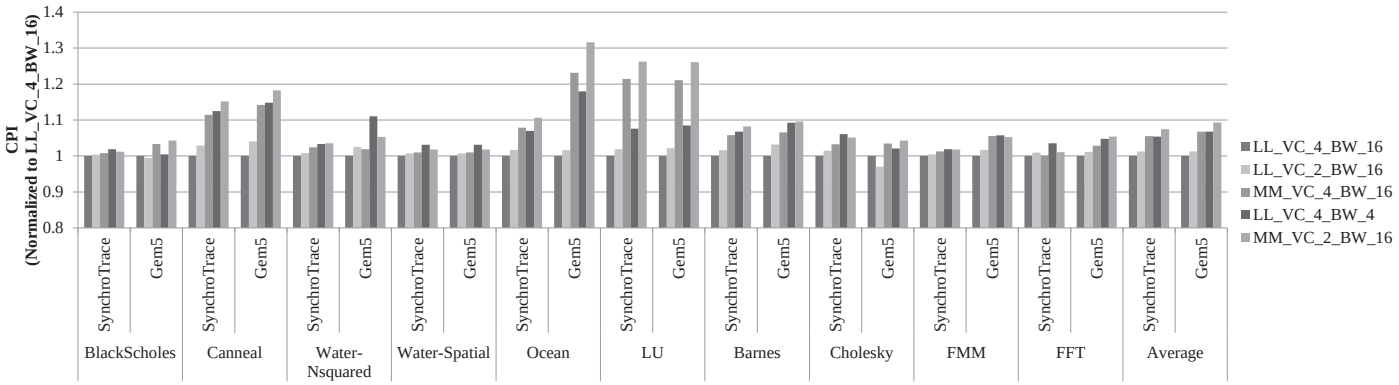
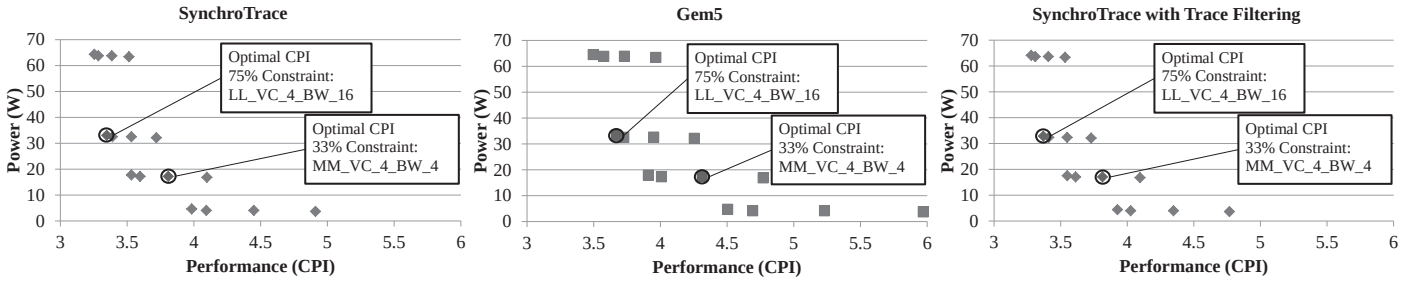


Fig. 6: CPI of Top 5 Design Points of SynchroTrace and Gem5 Under Uncore Constraints: 650 mm^2, 45 W



(a) SynchroTrace Power vs. Performance

(b) Gem5 Power vs. Performance

(c) SynchroTrace with Trace Filtering Power vs. Performance

Fig. 7: Total Uncore Power (NoC and Caches) vs. Performance (CPI)

1) Constraint 1: 75% of Max Area and Power: The design points allowed under Constraint 1 are compared for relative performance. Figure 6 summarizes the top 5 best performing design points of SynchroTrace and Gem5 across all tested benchmarks, normalized to the CPI of LL_VC_4_BW_16. Observing the average CPI of the design points in SynchroTrace and Gem5, it is evident that the LL_VC_4_BW_16 design point is the highest performing design point. The average normalized CPIs per design point of SynchroTrace are slightly skewed by up to 1.6% in comparison to Gem5. However, SynchroTrace preserves the same number of design points under the constraints with the equivalent ranking of design points by average CPI.

Furthermore, as detailed in Figure 6, SynchroTrace captures the CPI trends in all benchmarks except for stray cases where additional NoC provisioning causes slight increases in execution time (up to 4.6% difference in normalized CPI between SynchroTrace and Gem5) for Gem5. In particular, doubling the virtual channels (LL_VC_2_BW_16 to LL_VC_4_BW_16) reduces the performance of BlackScholes and Cholesky simulations with Gem5. In the case of Ocean, SynchroTrace and Gem5 both match in terms of the overall trend in CPI, but the ranges are greatly skewed between the two simulators: the overall normalized CPI range of SynchroTrace is roughly 10.6%, while the normalized CPI range of Gem5 stretches to 31.6%. This deviation is caused by the large amount of user-level synchronization within the execution of Ocean; SynchroTrace introduces dependency-based waits for communication events representing this inter-thread communication, while Gem5 executes the user-space synchronization construct specified in the program as expected. We are currently investigating how to increase the cycle-level fidelity of benchmarks that implement a large amount of user-level synchronization, but SynchroTrace is already able to maintain the normalized CPI trends of these benchmarks.

2) Constraint 2: 33% of Max Area and Power: With strict area and power constraints of 33%, the design space converges to only 4 design points. The MM_VC_4_BW_4 design point is the highest performing design point for the strict constraints for both SynchroTrace and Gem5. However, when comparing the smaller design points, the difference in average normalized CPI per design point between the two frameworks is up to 9.7%. The overall CPI trends are maintained between the two simulators, but as we show in Section IV-C3, SynchroTrace is slightly skewed towards underestimating cycles for less resource-intensive design points.

3) Design Exploration with SynchroTrace Comparison to Gem5: As shown in the design exploration above, the SynchroTrace simulation flow obtains the equivalent optimal design point under sets of constraints. Additionally, from Figures 7a and 7b, we deduce that **1)** the power estimation (as well as the area, not shown) between SynchroTrace and Gem5 are the same, and **2)** the SynchroTrace simulator skews towards underestimating the execution time in comparison to Gem5, and the skew is increased for less resource-intensive designs. This skew in absolute CPI ranges from 6.9% in vLvL_VC_4_BW_16 to 17.8% in SS_VC_2_BW_4). However, and more importantly, the ratio of CPI between any two design points in SynchroTrace (effectively the ratio of Cycles), is within 97% of the ratio of CPI for the same two design points in Gem5. Thus, the overall trend for SynchroTrace is maintained within 97% of Gem5.

We have shown that the accuracy of SynchroTrace in

uncore design space exploration and design selection experiments is 100% in comparison to the selections of full-system simulation. Furthermore, as we show in Section V, each design point is simulated up to up to 13.4× faster with SynchroTrace over Gem5.

V. Achieving Fast Design Exploration with Multi-Threaded Traces

Although our SynchroTrace simulation flow is up to 13.4x faster than Gem5 on average, our multi-threaded traces can be used to speed up simulation by trading off accuracy for speed. To this end we propose techniques including event compression ("lumped events"), "lumped-events" with hit prediction, and trace filtering. Figure 8 illustrates the speedup of the SynchroTrace simulation flow for all the trace techniques over Gem5, when simulating a modern CMP configuration most closely represented by the largest design point in Tables I and II (i.e. from [1]) for applications with 8 threads. We show up to 18.4x gains compared to Gem5 in simulation performance.

We also evaluate the accuracy in terms of design space exploration for the technique that showed the most promise: trace filtering. Due to space constraints, we mention the accuracy for each technique without showing graph-level detail.

A. Speedup using Multi-Threaded Trace Techniques

Exploring design spaces using architecture simulation can take a significant amount of time, from days to months. Our event-traces offer a significant advantage by reducing simulation time. The first bar in Figure 8 shows the speedup in simulation from using our normal trace execution through the SynchroTrace simulation flow versus the Gem5 Full-System TimingSimpleCPU based model. We executed multiple benchmarks with "simsmall" data sizes from the PARSEC 2.1 and Splash-2 benchmark suites for both the multi-threaded trace-based simulation flow and the Gem5 Full-System simulation flow as a comparison for simulation speed. Across the benchmark simulation executions, the results show that the multi-threaded trace-based simulation flow has up to a 13.4x speedup with an average of 4.6x speedup over Gem5.

B. Trace Compression

Our traces are generated by abstracting and aggregating the different classes of behavior in a program as explained in Section II; we produce Computation, Synchronization, and Communication events for multi-threaded programs that use the pthread API. This provides an opportunity to perform compression within the trace by lumping together multiple consecutive operations which fall under the computation or communication categories. When consecutive Computation events are merged together, the fields that represent counts, i.e. Integer Op Count, Floating Point Op Count, Memory Read Count, Memory Write Count are all added together. Recall the fields in each event type as shown in Listing 1 and 3. The fields that represent address ranges are merged together to keep only the unique address ranges. Consecutive Communication events can be merged by simply merging the address ranges as described above and Synchronization events cannot be merged.

When parsing a lumped event, the Replay mechanism also optimizes playback by attributing cycles for hits in a lumped-event. Lumping events together will lose some ordering information amongst operations for the benefit of compression. We

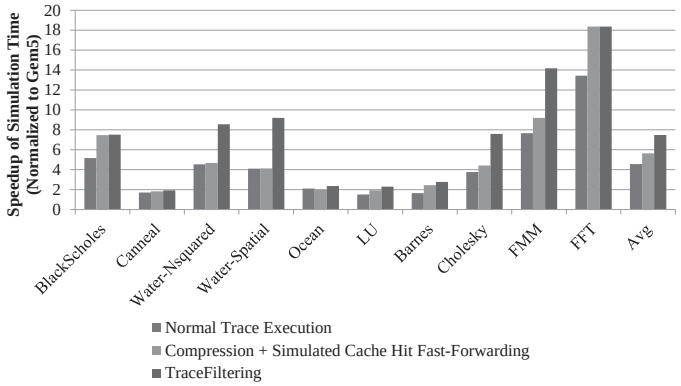


Fig. 8: SynchroTrace Speedup in Simulation using our Multi-Threaded Trace Techniques over Gem5

can set a limit on the number of events that can be lumped together in the trace, so as to maintain accuracy. For the PARSEC 2.1 and Splash-2 benchmarks tested, we found the optimal trace compression limit was 100 events per line, which produces around 10% difference in execution cycles, but shows large improvement in compression and simulation time. This compression reduces zipped file sizes by up to 74% for some benchmarks and 63% on average, while the simulation flow has up to an 18.4x speedup with an average of 5.64x speedup over Gem5 Full-System as shown in Figure 8.

C. Trace Filtering

We also studied the reduction in simulation time using a trace filtering approach inspired by prior work in the context of traces for single-threaded applications [20], [26]. Puzak's work used a direct mapped cache to filter out hits from a trace. The resulting trace only contains misses. In a multi-processor system, this will not work without modification as memory reads and writes could also potentially cause coherence actions compromising accuracy. While Wu et al. attempt to apply the technique to multi-processor scenarios, they use a multi-pass approach which was not evaluated for accuracy or the effect on coherence. Here we demonstrate the promise of this technique by filtering hits only to non-shared data (local accesses) from computation events, as filtering hits to shared data can become complex due to non-determinism.

The filtering technique we implement post-processes the trace and uses a filter cache structure to remove address ranges from computation events if they hit in the filter cache. The technique also adds a field to the trace to record the hit count, which can be used to estimate cycles by the Replay mechanism. The configuration parameters of this filter cache determine the speedup and accuracy associated with simulating filtered traces for design space exploration. We use an 8kB, fully associative structure with a line size of 8 bytes. Prior work has shown that stack distance in a fully associative structure is sufficiently representative of set-associative caches employed in modern architectures [3], [6]. Hits in the 8kB structure are very likely to hit in caches larger than 8kB during simulation, making it an effective predictor of hits. We use an 8-byte line size to conservatively allow for line size changes in the simulated configuration and to account for accesses that straddle cache line boundaries.

The speedup obtained over Gem5, shown in Figure 8 goes up to 18.4x with an average of 7.5x. Ocean has limited

speedup due to the user-level synchronization that are enforced with dependency waits in SynchroTrace. Both Canneal and LU traces are relatively large and would benefit from more aggressive compression and filtering techniques.

We also ran the same design space exploration experiment of the previous section and arrived at the same subset of designs ranked in the same order. The accuracy is shown in Figure 7c, where we plot the CPI vs Power of all 16 of the design points as in the previous section. We find that the design points with filtered traces overlap with the points from unfiltered traces in most cases, including the optimal designs. At the smaller design points, the effect of the high associativity of the filter cache causes aggressive filtering to underestimate cycles by around 2%, though the relative trends are still preserved as before.

D. Scalability

SynchroTrace is also scalable and can generate and run traces for applications with more than 128 threads. In our measured 32 thread simulations using Splash-2 benchmarks, we show significant speedup of SynchroTrace over Gem5 of up to $17x$ using the trace filtering technique discussed in Section V-C. We omit the discussion of a full comparison of the 32 thread simulations for lack of space.

VI. Background and Related Work

The most accurate solution for a simulation-based design space exploration can be obtained through execution-driven full-system simulators such as Gem5 [5] that execute entire applications. Recently, a number of scalable simulators that use parallel simulation have been released [7], [13], [22]. They allow different levels of slack in the ordering of memory accesses for multi-threaded applications and enforce synchronization between simulation threads at quanta ranging from a few 1000 cycles to entire barrier regions [7], [13], [22]. These parallel simulators have not been fully validated for relative errors and design space exploration capabilities. These prior work are orthogonal to our work in this paper, as the SynchroTrace methodology can be integrated into any of these simulators to aid in identifying synchronization points and for potential performance improvement using trace filtering.

Traces used in trace-based simulations are simply a chronological log of the various *events* (messages sent over the NoC or cache access or instructions etc.) taking place in a system. Prior trace-based simulation approaches have encountered difficulty capturing and accurately replaying multi-threaded traces due to the inherent non-determinism in the execution of multi-threaded programs [9]. SynchroTrace is able to model non-determinism by capturing and embedding synchronization events in the trace and tracking dependencies between traces during capture.

A. Comparison to Pinplay

PinPlay provides a framework, based upon dynamic instrumentation, to capture execution into traces (Pinballs) and replay the captured execution, deterministically [19]. There are clear benefits to deterministic replay, such as debugging or reduced complexity in CMP simulators for single-threaded applications. However, deterministic replay can fundamentally cause inaccuracies for design space exploration with multi-threaded benchmarks.

In the context of multi-threaded applications, Pinballs are generated for each individual thread's execution. Included in multi-threaded Pinballs is a thread dependency file that captures shared memory read and write in order and instruction dependencies among threads to deterministically replay the traces in the captured order. Deterministic replay of multi-threaded traces is useful for debugging multi-threaded applications in frameworks such as DrDebug [24]. However, deterministic replay does not allow for timing behavior to affect the critical path of multi-threaded applications and produces the same thread interleaving for every run. An example of this timing behavior is the influence of the memory system on the ordering of thread synchronization events. This potential inaccuracy in the context of design space exploration with multi-threaded benchmarks is noted by T.E. Carlson et al. [8], which includes the developers of Pinplay and a Pinplay-integrated multi-core simulator, Sniper. Pinplay's enforcement of thread event ordering can cause cycle-time inaccuracy when replaying multi-threaded Pinballs into a CMP simulator as the imposed thread ordering may differ from the native execution of multi-threaded programs on different types of CMPs. In contrast to Pinplay, SynchroTrace allows thread timing behavior to affect the critical path of multi-threaded applications with a more accurate, non-deterministic playback.

To the best of our knowledge, no Pinball-based solution has been developed for the more accurate, non-deterministic playback of multi-threaded Pinballs in the context of design space exploration. Currently, the Sniper simulator [7], which can interface with single-threaded Pinballs, is unable to playback multi-threaded Pinballs for design-space exploration.

B. Other Trace-Drive Simulation Solutions

Rico et al. [21] present a hybrid simulation methodology that uses an execution-driven component to handle threading API calls (parops, in their nomenclature) in multi-threaded applications, while a trace-driven engine handles the non-parallel portions of the application. These traces capture sequential flow of execution for each thread, somewhat similar to our methodology [21]. However, this methodology requires source to source transformations to interface the parops with their simulation framework, while SynchroTrace does not require source code changes. Also, the authors propose a simulation framework with complex interfaces, that are not fully validated against hardware or full-system simulation. They have also not characterized simulator performance and only demonstrate the methodology on a single custom application. This motivated us to write a methodology with a simple interface that works with unmodified benchmarks using standard threading libraries.

Trace-based approaches have also been employed to specifically explore the NoC design space [10], [11], [18], [23]. Most work in this space has recognized the need to establish causation between network messages in order to model the associated delays correctly. Thus, most of them attempt to annotate dependencies in their traces. Raw traces are collected, and dependencies are extracted, mostly through post-processing approaches [10], [11], [18]. YSC Huang et al. use a bloom filter inspired approach for message passing

interface (MPI) based applications but cannot handle shared-memory applications [11]. Nitta et al.'s methodology and Netrace suffer from the need for multiple full-system runs to infer true dependencies [10], [18]. In general, collecting traces through full-system simulation is not scalable to large number of threads. To the best of our knowledge, we are the first to generate reliable synchronization and dependency-aware multi-threaded traces that require no changes to application code for architecture simulation.

VII. Conclusions

In this work, we have presented SynchroTrace: Synchronization- and Dependency-Aware architecture-agnostic traces, played through an intelligent Replay mechanism for accurate, flexible, scalable, and fast design space exploration for multi-threaded applications. Our multi-threaded traces can be captured quickly, without the need for full-system simulation and have dependencies and synchronization embedded in them. Additionally, we have shown how the traces can be integrated into a simulator easily with the help of our Replay mechanism. We validate the SynchroTrace simulation flow by successfully achieving the equivalent results of a constraint-based design space exploration with the Gem5 Full-System simulator. We show how our methodology is flexible, and we can trade-off accuracy for speed by compressing and filtering traces. The results from simulating benchmarks from PARSEC 2.1 and Splash-2 show that our trace-based approach with trace filtering has a peak speedup of up to $18.4x$ over simulation in Gem5 Full-System with an average of about $7.5x$ speedup.

VIII. Acknowledgments

This material is based on work supported by the National Science Foundation including a CAREER award CCF-1350624 and grant ECCS-1232164. Karthik Sangaiah is supported by the NSF Graduate Research Fellowship under Grant No. 1002809. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Intel Xeon E5-2667. http://ark.intel.com/products/83361.

[2] Valgrind function-wrapping. http://valgrind.org/docs/manual/manual-core-adv.html#manual-core-adv.wrapping.

[3] K. Beyls and E.H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, pages 617–662, 2001.

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

[5] N. Binkert et al. The gem5 simulator. In *The ACM SIGARCH Computer Architecture Newsletter*, August 2011.

[6] M. Brehob and R. Enbody. An analytical model of locality and caching. *Michigan State University, Department of Computer Science and Engineering MSU-CSE-99-31*, 1999.

[7] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2011.

[8] T.E. Carlson, W. Heirman, H Patil, and L. Eeckhout. Efficient, accurate and reproducible simulation of multi-threaded workloads. In *REPRODUCE: Workshop on Reproducible Research Methodologies*. IEEE, 2014.

[9] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. *ACM SIGMETRICS*, June 1993.

[10] J. Hestness, B. Grot, and S. W. Keckler. Netrace: dependency-driven trace-based network-on-chip simulation. In *Proceedings of the International Wokshop on Network on Chip Architectures (NoCArc)*, pages 31–36, 2010.

[11] Y. S.-C. Huang, Y.-C. Chang, T.-C Tsai, Y.-Y Chang, and C.-T King. Attackboard: A novel dependency-aware traffic generator for exploring NoC design space. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 376–381, 2012.

[12] A. B. Kahng, B. Li, L. Peh, and K. Samadi. ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In *Proceedings of the Design, Automation Test in Europe (DATE)*, April 2009.

[13] J. E. 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 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[14] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[15] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual execution environments*, VEE '07, pages 65–74, 2007.

[16] S. Nilakantan and M. Hempstead. Platform-independent analysis of function-level communication in workloads. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, Sept 2013.

[17] S. Nilakantan, S. Lerner, M. Hempstead, and B. Taskin. Can you trust your memory trace?: A comparison of memory traces from binary instrumentation and simulation. In *International Conference on VLSI Design and 14th International Conference on Embedded System Design (VLSID ES)*, Jan 2015.

[18] C. Nitta, K. Macdonald, M. Farrens, and V. Akella. Inferring packet dependencies to improve trace based simulation of on-chip networks. In *Proceedings of the IEEE/ACM International Symposium on Networks on Chip (NoCS)*, 2011.

[19] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, 2010.

[20] T.R. Puzak. *Analysis of cache replacement-algorithms*. PhD thesis, University of Massachusetts Amherst, 1985.

[21] A. Rico, A. Duran, F. Cabarcas, Y. Etison, A. Ramirez, and M. Valero. Trace-driven simulation of multithreaded applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[22] D. Sanchez and C. Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the International Symposium on Computer Architecture*, ISCA '13, 2013.

[23] F. Trivino, F. J. Andujar, F. J. Alfaro, and J. L. Sanchez. Self-related traces: An alternative to full-system simulation for NoCs. In *International Conference on High Performance Computing and Simulation (HPCS)*, 2011.

[24] Y. Wang, H. Patil, C. Pereira, G. Lueck, R. Gupta, and I. Neamtiu. Drdebug: deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'14, New York, NY, USA, 2014. ACM.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1995.

[26] Z. Wu and W. Wolf. Iterative cache simulation of embedded cpus with trace stripping. In *Proceedings of the International Workshop on Hardware/Software Codesign*, CODES, 1999.