# SynchroTrace: Synchronization-aware Architecture-agnostic Traces for Light-Weight Multicore Simulation of CMP and HPC Workloads

KARTHIK SANGAIAH, Drexel University
MICHAEL LUI, Drexel University
RADHIKA JAGTAP, Arm Ltd.
STEPHAN DIESTELHORST, Arm Ltd.
SIDDHARTH NILAKANTAN, NVIDIA Corporation
ANKIT MORE, Intel Corporation
BARIS TASKIN, Drexel University
MARK HEMPSTEAD, Tufts University

Trace-driven simulation of chip multi-processor (CMP) systems offers many advantages over execution-driven simulation, such as reducing simulation time and complexity, allowing portability, and scalability. However, trace-based simulation approaches have 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. By recording synchronization events relevant to modern threading libraries (e.g. Pthreads and OpenMP) 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 hardware platforms and threading paradigms. Through capturing high-level instruction categories, the SynchroTrace average CPI trace replay timing model offers fast and accurate simulation of many-core in-order CMPs. We perform two case studies to validate the SynchroTrace simulation flow against the gem5 full-system simulator: 1) a constraint-based design space exploration with traditional CMP benchmarks and 2) a thread-scalability study with HPC-representative applications. The results from these case studies show that 1) our trace-based approach with trace filtering has a peak speedup of up to 18.7× over simulation in gem5 full-system with an average of 9.6× speedup, 2) SynchroTrace maintains the thread-scaling accuracy of gem5 and can efficiently scale up to 64 threads, and 3) SynchroTrace can trace in one platform and model any platform in early stages of design.

CCS Concepts: **•General and reference →Performance; •Computing methodologies →Modeling and simulation; •Networks** →*Network performance evaluation;*

*Extension of Conference Paper*: S. Nilakantan, K. Sangaiah, A. More, G. Salvador, B. Taskin, M. Hempstead, "SynchroTrace: Synchronization-aware Architecture-agnostic Traces for Light-Weight Multi-core Simulation", Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2015), March 2015, pp. 278-287. Compared to our previous publication, this work adds deeper analysis to the accuracy results of the original work, along with simulation results for additional condition variable-based benchmarks. This work also presents the trace capture and characterization of OpenMP-based HPC applications, presented in Section 2.3, as opposed to solely CMP-based Pthread applications. Additionally, a new Section 5 is added to examine SynchroTrace for performance when scaling to a large number of threads (1 to 64) with the HPC applications derived from the Department of Energy FastForward2 exascale initiative. While the original work solely models x86-based CMP platforms, the new HPC experiment models both x86- and Arm-based platforms. The additional set of experiments shows that traces captured in one platform can model any CMP platform, as we compare our simulation results of Arm-based and x86-based platforms.

DOI: 10.1145/nnnnnnn.nnnnnnn

General Terms: Modeling, Performance

Additional Key Words and Phrases: Trace-driven simulation, multi-threading, computational modeling, chip multi-processor, design space exploration

**ACM Reference format:**
Karthik Sangaiah, Michael Lui, Radhika Jagtap, Stephan Diestelhorst, Siddharth Nilakantan, Ankit More, Baris Taskin, and Mark Hempstead. 2018. **SynchroTrace**: Synchronization-aware Architecture-agnostic Traces for Light-Weight Multicore Simulation of CMP and HPC Workloads. *ACM Transactions on Architecture and Code Optimization* 1, 1, Article 1 (April 2018), 25 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

As the number of cores in future HPC-focused chip multiprocessors (CMPs) scales up, the design complexity of these future CMPs cause long simulation times. This is a challenge for the system design process, particularly of HPC systems where dependable simulation methodologies are essential. Current execution-driven simulators encounter such a simulation wall for analyzing and simulating current HPC-representative applications and systems. The simulation wall is exemplified in Figure 1, where the simulation time continues to grow as we increase the number of application threads when simulating a compute-intensive HPC kernel with a full-system execution driven simulator (gem5 [3]). It is clear that for HPC applications, it is intractable to simulate many-core CMPs with full-system execution driven simulation.

Compared to execution-driven simulation, 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, 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 [30]. 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. The traces and replay of Pinplay do not model this non-determinism accurately during simulation. As a result, a 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

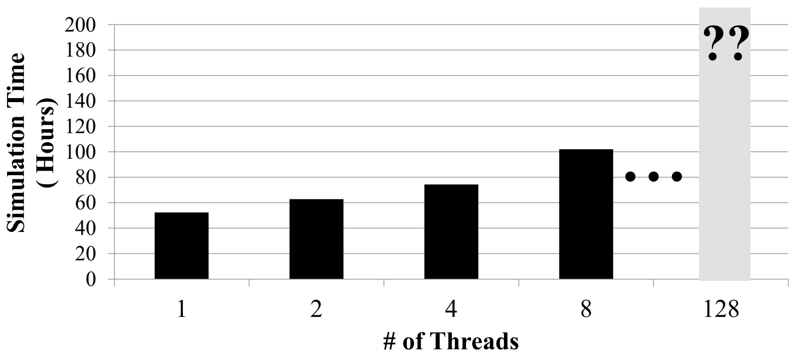


Fig. 1. Full-System Simulation of Shock Hydrodynamics HPC Application (LULESH)

solution, proposed by Rico et al. [32], is a hybrid execution-driven and trace-driven methodology for simulation. However, their methodology requires source to source transformation to interface their synchronization calls with their simulation framework. Our proposed methodology models the non-determinism of thread interleaving, while enabling fast simulations of unmodified multi-threaded applications of any in-order CMP platform.

We propose SynchroTrace, a two-step methodology for trace-based simulation of multi-threaded applications: 1) the generation of synchronization- and dependency-aware architecture-agnostic traces and 2) a light-weight replay mechanism that respects those dependencies, simulates synchronization actions, and handles simple scheduling for threads for playback on any in-order CMP platform. The tracing methodology utilizes dynamic binary instrumentation to generate the dynamic trace. 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 or memory stream during playback. In this paper, we refer to a simulation flow that integrates SynchroTrace with a cache and interconnect simulator as the "SynchroTrace Replayer", or "Replay" for short.

With a large hardware design space exploration of the memory hierarchy and interconnect sub spaces, we show how SynchroTrace provides fast (up to 18.7× faster than gem5 [3]) simulations that estimate CMP performance and power. The equivalent design space exploration is performed with gem5 as to show that SynchroTrace obtains the same best design configurations under constraints as gem5. Additionally, we present a case study analyzing the performance of HPC applications as we increase the number of threads using SynchroTrace and gem5. We show that SynchroTrace predicts thread-scalability of HPC applications within 5.8% of gem5. However, we find that SynchroTrace simulations typically do not increase in simulation time as the number of threads is increased from one to eight, while gem5 simulations requires up to a 95% increase in simulation time. We also show that SynchroTrace is tractable for simulations of 64 application threads. Lastly, we compare thread-scalability results of using traces from both x86- and Armv8-based platforms and show that traces from one processor architecture can be used to accurately model others. The results from these case studies show that our methodology has high accuracy with a peak speedup of up to 18.7× compared to simulation with gem5 full-system, can tractably scale to a system with 64 cores, and is architecture-agnostic.

The rest of the paper is organized as follows: we present SynchroTrace: synchronization- and dependency-aware, architecture-agnostic multi-threaded traces, including our new support for OpenMP-based HPC applications, in Section 2 and a dependent replay mechanism (i.e. to complete the simulation flow) in Section 3. We validate SynchroTrace by comparing our trace-based simulation results for a CMP design space exploration against the gem5 full-system simulator results in Section 4. We show in Section 5 how SynchroTrace can be used to examine HPC-focused, OpenMP-based application scalability (up to 64 threads) for Armv8 and x86-64 platforms. In Section 6, we present trace-based optimizations for speedup of CMP architecture simulations. Finally, we compare SynchroTrace with related work in Section 7.

## 2 SYNCHRONIZATION- AND DEPENDENCY-AWARE TRACES

In the context of architecture simulation, *traces* refer to the chronological event sequence of a program's execution. 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 the relevant subsystem being designed. For example, an instruction trace records all instructions dynamically executed in chronological order. This instruction trace can then be used for detailed CPU models. Similarly, memory traces record only the LD/ST instructions and addresses dynamically executed [3, 26, 30]. Memory traces can be used with simpler CPU models for more detailed simulation of just the

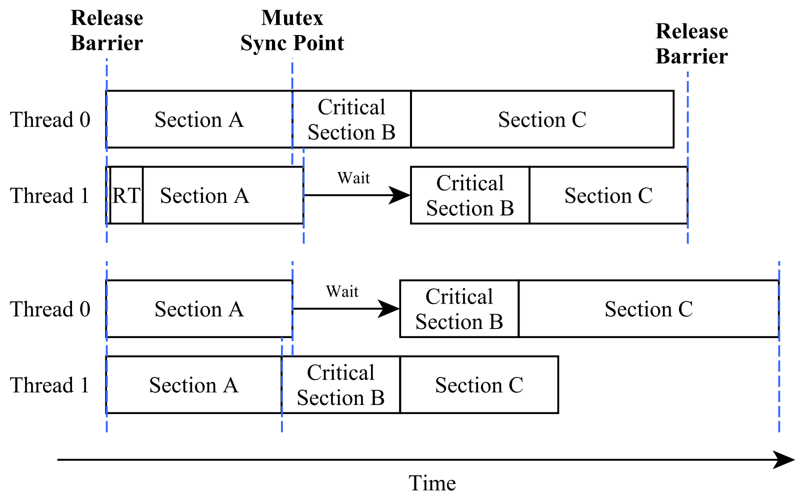


Fig. 2. **Non-Determinism in Thread Execution.** Two thread interleavings are shown with a run-time (RT) factor occurring during Section A of Thread 1. Differing thread interleavings can impact overall run-time [32].

"uncore" [23]. Additional tools, such as Ramulator [21] or gem5 [3], can produce memory traces with embedded timing information to model specific systems. However, this level of tracing requires large amounts of storage, and it is not adaptable to model system-level performance metrics, such as execution time or IPC, for multiple systems without additional tracing for each system. Additionally, traditional instruction and memory traces alone cannot accurately model multi-threaded applications in simulation due to the non-determinism of threads during execution. In this section, we describe the importance of modeling non-determinism in thread execution, our solution through synchronization- and dependency-aware *synchro*traces, and the methodology of capturing the *synchro*traces.

## 2.1 Non-Determinism in Multi-Threaded Programs

Traces are convenient and portable for simulation, but due to non-deterministic execution, simulation using traces of multi-threaded applications has proven difficult and been attempted only a few times [28, 30, 32]. The non-determinism manifests as uneven per-thread progress between synchronization points and indeterminate wait time at synchronization points. Design time factors (e.g. CMP design configuration and static thread mapping) and run-time factors (e.g. OS load on the cores or dynamic thread mapping) can impact individual thread progress differently. A particular state of relative progress between different threads is termed *thread interleaving* [30, 32]. 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 in Figure 2. This figure depicts a portion of execution for an application containing two synchronizing threads, between two barriers, i.e. a barrier region [5]. 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 required threads have arrived. Two scenarios of thread progress are shown with slightly different execution times for Section A across the scenarios. The inclusion of a dynamic run-time variable (OS) in the timing of Section A has a significant effect on the wall-clock time. 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 difference in execution time of Section A represents uneven progress between synchronization points in multi-threaded programs. This is one manifestation of non-determinism. Another manifestation of non-determinism is the variable wait times at synchronization points caused by, for example, OS scheduling, lock acquisition policies, and even cache state. It is thus important to model the impact of non-determinism during simulation.

We assert that a trace-driven simulation flow for multi-threaded applications should not record and enforce a specific thread interleaving. Instead, SynchroTrace allows for thread interleaving to be determined by hardware architecture and run-time factors during replay.

## 2.2 SynchroTrace Characteristics

The overall goal for forming a trace of a multi-threaded program is choosing a level of abstraction that balances modeling accuracy with simulation time and disk storage. Full instruction traces can enable accurate simulation but require large simulation time and disk storage. Alternatively, memory traces can enable fast simulation but do not have enough detailed information to model system-level performance metrics, such as execution time. SynchroTrace forms a fast simulation abstraction of the execution units core model (i.e. local computation of a thread), while maintaining a detailed account of memory operations, thread synchronization actions, and thread communication. Control-flow is not captured directly for this work, as the platforms investigated all use simple in-order cores that are unaffected by speculation. The additional timing cost of control-flow instructions is captured implicitly in the calculation for average instruction latency, detailed in Section 3.1.

An additional goal is to scalably generate the traces. While traces can be captured using full-system simulation [14, 29], this technique is not scalable to multi-threaded application trace capture. Synchrotraces are captured quicker than full-system simulation-based trace capture as synchrotraces are derived from native runs of the program, allowing for scalability with a marginal overhead when increasing the number of threads due to context switching handled in the dynamic binary instrumentation tool. Additionally, the *event* representation allows for more size efficient traces by only holding detailed information for the most important events.

In SynchroTrace, all operations by a multi-threaded program are classified into three categories of run-time events: Computation, Thread Synchronization, and Communication.

**Computation Events** represent local processing performed by a thread, completely independent of other threads. These events are necessary to model the timing (e.g. execution time) of each individual core in the replay of the traces. 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. IOPS and FLOPS are used to abstractly model the timing of compute operations of detailed instructions while maintaining the high fidelity memory operations for detailed memory playback; the set of unique read and written (virtual memory) addresses (i.e. the memory address of only the first read or write) are stored within the event as well. As shown in Section 4, we find this level of abstraction estimates the performance of CMPs accurately.

**Thread Synchronization Events** contain the type of synchronization API call and the address of the data structure used, so that a particular synchronization object can be recognized. Currently, OpenMP and Pthreads are supported as the synchronization API calls for SynchroTrace. Thread synchronization events are interpreted during simulation, and the action appropriate for the synchronization type (i.e. create, join, barrier, mutex lock, conditions etc.) is applied for participating threads. Synchronization events mediate accesses to shared resources and are necessary to model the non-determinism of thread interleaving.

**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 next in Section 2.3). A communication event in the consuming thread is associated with a corresponding *computation event* in the producing thread. Communication events can potentially hold references to data from multiple producer threads. A communication edge can generate cache 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. **Thread Synchronization Event**

```
Event Number, pth_ty: Sync_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 *1774530*, 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 of the computation or communication categories to be merged together (detailed further in Section 6). However, by restricting a single memory operation per computation event, the traces contain the exact issue order for each memory operation and thus, offers the highest accuracy. 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 event *4534* of Thread 1 through the addresses 7048536 to

7048543. 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.

## 2.3 SynchroTrace Capture Framework

The capture tool of SynchroTrace is based on the Sigil workload analysis framework [27], which is currently built on top of the Valgrind Dynamic Binary Instrumentation framework, but other instrumentation front-ends are possible. While Sigil [27] is originally designed to capture communication between functions, for this work, it has been adapted to capture local computation of threads and communication between threads. Another important addition of this work is capturing the synchronization behaviors of threads by wrapping prevailing Pthread and OpenMP API calls. The capture tool monitors the execution of a program and builds sequences of computation, synchronization, and communication events for each thread. We discuss the methods to capture synchronization and communication events in the following sections.

*2.3.1 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 correctly. The first feature captures a separate trace for each thread which contains memory and compute operations captured in program order for that thread. The second feature captures and logs the synchronization events in each trace.

Figure 3 illustrates an example of how we intercept synchronization API calls to generate synchronization events. The trace capture mechanism uses the function wrapping interface of Valgrind to intercept synchronization API calls [35]. Valgrind can detect when a desired function is invoked at run-time and allows a tool to redirect the original function into a user-defined function. The modified Sigil tool leverages function wrapping by redirecting threading API calls into a wrapper that includes the following operations: 1) Disable the capturing of loads and stores associated with the synchronization function, 2) run the original synchronization operation as to continue the correct progression of the application, 3) log the occurrence along with critical variables into a synchronization event, and 4) enable the capturing of the loads and stores for the

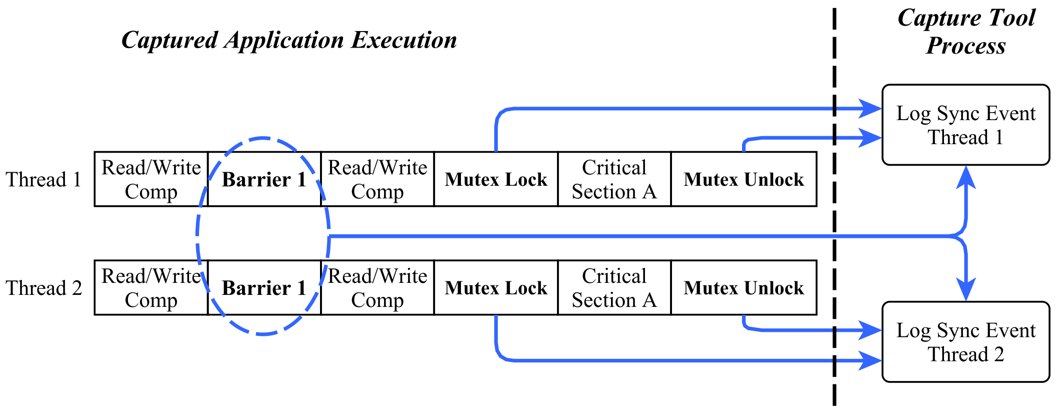


Fig. 3. Intercepting Synchronization API Calls

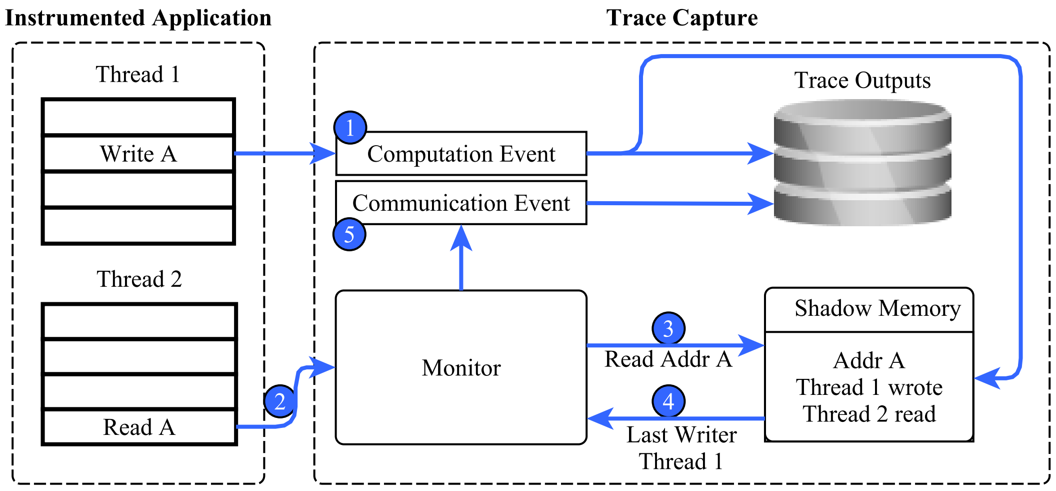


Fig. 4. Capturing Computation and Communication Events

next instruction. By disregarding instructions within synchronization functions, we abstract the underlying synchronization implementation and do not enforce any specific thread-ordering; thus, the replay platform handles the native *behavior* of the synchronization calls.

As a feature, this synchronization capture process is focused on tracing traditional CMP and HPC platforms. With regards to traditional CMP-focused benchmarks, we currently capture Pthread API calls: Pthread *create*, *mutex lock/unlock*, *barrier*, *condition signal*, and *condition wait*. Additionally, to support HPC system design focusing on large-scale parallel CMP platforms, we capture necessary compiler-level function calls for tracing OpenMP applications. The typical usage of OpenMP is in the form of common '#pragma' lines of code; we capture the compiler-generated OpenMP functions for the common OpenMP function calls. Similar to the pthread API calls we capture, these OpenMP function calls are organized into high-level categories: *thread creation/destruction*, *critical sections*, and *global synchronization*. Overall, these advances aid in fast design space exploration of HPC applications, as well as fast analysis of the thread scalability of benchmarks.

As a limitation, SynchroTrace can only capture threading activity for supported threading API calls. Examples of non-standard threading API calls include cases where condition variables are explicitly written in user code, or critical sections using low-level locks are encountered in the kernel [28]. This is the core purpose of communication events; we capture communication events to handle these cases and enforce them as dependencies between the threads.

*2.3.2 Capturing communication events.* The capture tool monitors communication through memory addresses by using Shadow Memory [25]. 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 4 presents an example of this process. When a **write** to address A occurs in Thread 1, a computation event is written to the trace of Thread 1. This address is also emitted simultaneously to a Shadow Memory, which stores Thread 1 as the last writer for address A. Subsequently, 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 [25]. The last writer information is sent back to the monitor, which detects the inter-thread communication edge. In this example of an inter-thread communication edge, the monitor emits a communication event to the trace for Thread 2.

*2.3.3 Capturing Operating System traffic.* The SynchroTrace capture framework intercepts information related to Operating System (OS) actions, albeit in a limited fashion using communication events. Since our capture framework is built on Valgrind, SynchroTrace shares the inability of Valgrind 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 is not visible to Valgrind. We thus conservatively treat reads that consume from memory writes within the kernel as dependencies enforced by replay

*2.3.4 Capturing non-determinism.* SynchroTrace captures each thread as a serial set of abstracted compute, memory, and synchronization events. The actual serial set of abstracted compute, memory, and synchronization events per thread can vary between native executions of an application. This level of non-determinism is impossible to fully capture in a dynamic trace. For example, thread behavior can diverge at synchronization points depending on the order it reaches a critical section and the state of the thread at the critical section; the state of the thread can depend on the thread ID and starting parameters, and also on the order it reached previous critical sections, which can additionally influence the order it reaches the current critical section. Indeed, depending on the given algorithm, such behavior can even cause non-deterministic outputs, not just non-deterministic execution. Another cause for non-deterministic traces is unsynchronized communication between threads, which can affect the behavior of the program. Any communication between threads is assumed to be captured correctly in-order. This work considers two traditional uses of threads: 1) to split work amongst worker threads which globally synchronize to update shared data, and 2) independent tasks that may rely on atomic operations to update and access shared data.

Such workloads can be abstractly represented as a graph, where each edge is an independent, serial set of compute and memory operations of a thread, and each node is a synchronization point between threads. As long as the edges and nodes are the same for any execution, regardless of which threads an edge belong to, then the variability of the replay simulation is determined based on the architecture and thread scheduler. For the considered applications and use of the threading models, this is indeed the case and requires just a single trace capture.

Additionally, the binary instrumentation tool, Valgrind, serializes multi-threaded workloads in a predictable manner by context switching threads either at regular intervals or when they block. In this way, the effect Valgrind has on the captured per-thread statistics is minimal if the application rationally synchronizes using the supported thread libraries.

## 3 EVENT-TRACE REPLAY FRAMEWORK

For architecture simulations, a replay mechanism is required to process the trace and generate architectural events, as is standard in event-driven simulation [14, 15, 29, 32, 34]. 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 5, 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.

The SynchroTrace Replayer uses simple timing models to abstract in-order cores. To maintain a high-level core abstraction for speed purposes, SynchroTrace uses an average CPI timing model

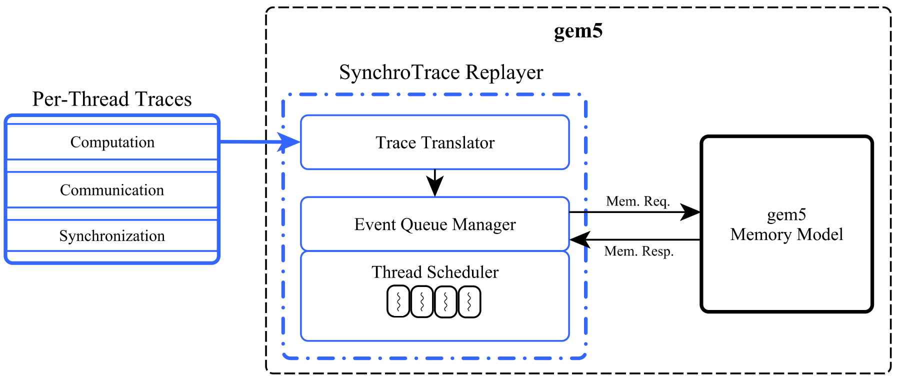


Fig. 5. Multi-Threaded Event-Trace Replay Framework

for compute instructions, while integrating with detailed timing models for a high level of accuracy in modeling the uncore. Theoretically, the core Replay infrastructure can be connected to more detailed timing models such as out-of-order cores by integrating techniques from Elastic Traces [17], although this is not performed in experimentation of this paper. Additionally, the current MRM models an infinite store queue, as to not block on write requests, but the MRM does block on read requests. Our current Replay framework processes memory requests for the classic memory system of gem5 as well as the Ruby/Garnet memory and interconnect simulators. However, the multi-threaded traces and replay mechanism are portable to any cache simulator. The SynchroTrace Replay framework, along with the Sigil tracing framework, is available as a gem5 patch on Github (https://github.com/VANDAL).

## 3.1 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*. Following a typical event-based simulation model, during each cycle the EQM checks if there are events ready to be processed from threads for the current cycle. If there are no ready events for the current cycle across all of the threads, the *CurrentTime* progresses to the scheduled wakeup time of the next available event. Events scheduled to wake up in the current time are handled by the process represented in Algorithm 2. The handling of the event is based on its type, i.e. computation, communication, or synchronization, as follows.

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. This time is calculated per-benchmark, assuming a simple in-order architecture and an average CPI timing model, as

$$\frac{total\ instructions}{IOPs + FLOPs} \times \frac{1\ cycle}{1\ instruction} \tag{1}$$

These metrics are gathered from the captured trace using Valgrind. Memory access latencies are calculated by the underlying memory model from gem5 and are not specific to SynchroTrace. Communication and Synchronization events do not have explicit latencies but affect the ordering of each thread and potential stalls. This dormant phase of the thread models the cycle time of an ALU 'computing' the instructions corresponding to the event prior to handling of memory accesses. 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 read requests of the thread until the MRM triggers a memory response to the EQM. As to exploit potential MLP, write requests will not block on the MRM as an infinite store queue is modeled. However, write requests may be blocked as a result of stalls in the memory hierarchy (e.g. the miss status holding register). As shown in Figure 5, 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 as a waiting queue for each participating thread; 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 corresponding memory requests for the synchronization variables, 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 issuing a memory read request. Once the corresponding computation event has been completed, the EQM will issue the memory read of the communication event to the MRM.

**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 $AllEventsinEventQueue \geq CurrentTime$ then
9:     $ProgressCurrentTimetoNextEventTime$
10: end if
11: if $MemoryResponseTriggeredForThread$ then
12:     $QueueNextEvent$
13: end if
```

**Algorithm 2** ProcessEvent

```
1: if $COMPEVENT$ then
2:     $IssueMemReq@(ComputationTime + CurrentTime)$
3:     if $ReadRequest$ then
4:         $WaitforResponse$
5:     end if
6: else if $COMMEVENT$ then
7:     if $DependentEventCompleted$ then
8:         $IssueMemReq@(CurrentTime)$
9:         $WaitforResponse$
10:    else
11:        $ScheduleThreadNextCycle$
12:    end if
13: else if $SYNCHEVENT$ then
14:    if $Event = Create$ or $Join$ then
15:        $SendEventtoThreadScheduler$
16:    else if $MutexLockRequest$ then
17:        if $MutexLockObtained$ then
18:            $QueueNextEvent$
19:        else
20:            $ScheduleThreadNextCycle$
21:        end if
22:    else if $MutexUnlockEvent$ then
23:        $QueueNextEvent$
24:    else if $BarrierEvent$ then
25:        if $LastThreadforBarrier$ then
26:            $QueueNextEvent$
27:        else
28:            $ScheduleThreadNextCycle$
29:        end if
30:    end if
31: end if
```

## 3.2 Thread Scheduler

The SynchroTrace Replayer accepts the simulation parameters and configures the simulation backend 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 emulating an OS thread scheduler in the absence of the OS. The Thread Scheduler handles the creation, deletion, scheduling, and synchronization of threads across any number of cores, including multiple threads per core. Currently, the Thread Scheduler opportunistically swaps out stalled threads for threads ready to progress. Threads can be stalled on synchronization events, dependencies, or memory requests. However, these scheduling actions are currently modeled with zero-cost. A simple round-robin approach is taken when multiple threads are ready to progress, though integration with more complicated thread schedulers are possible.

## 4 CMP DESIGN SPACE EXPLORATION WITH TRACE-BASED SIMULATION

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. Focusing on components of interest for an uncore designer, we evaluate design choices on the following: the L1 cache, L2 cache, NoC routers, and NoC links. We also validate that our light-weight simulation flow produces equivalent design space exploration results as the cycle-accurate gem5 full-system simulator.

## 4.1 Evaluation Strategy

Our evaluation strategy consists of two sets of experiments. The first experiment uses SynchroTrace to analyze the design space across cache 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, associativity, block size, NoC virtual channels, NoC buffer depth, and NoC link bandwidth; we simulate a pseudorandom subset of this design space, comprised of 64 design configurations. The goal of this experiment is to accurately select the best performing design configuration, in terms of execution cycles, under uncore power and area constraints. To normalize performance across benchmarks, we selected a reference CMP design: 128B block size, 16KB L1 cache size, 8 L1 cache associativity, 256KB L2 cache size, 4 L2 cache associativity, 3 flit NoC buffer depth, 3 virtual channels, 2B NoC channel bandwidth. The second experiment performs an equivalent design space exploration using the cycle-accurate gem5 full-system simulator [3]. The goal of this experiment is to compare SynchroTrace against the cycle-accurate full-system gem5 simulator results 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 2 and 3, respectively. The cores and NoC both operate at 1 GHz. The caches and NoC are designed for the 32nm technology with area and power given by Cacti 6.5 [24] for the caches and Orion 3.0 [18] for the NoC. The traces were captured on the Linux Kernel 3.10 in CentOS 7 with the standard POSIX Thread API. Table 1 summarizes the benchmark applications simulated from the PARSEC-3 [2] and Splash-2X [37] benchmark suites with input data parameters and sizes of the trace files. Table 1 also numerates the amount of integer operations, floating point operations, memory loads and stores, and synchronization operations of each benchmark application. The design space exploration case study contains a subset of the benchmarks. SynchroTrace cannot properly model the remaining unused benchmarks (BodyTrack, Ocean_CP, and Raytrace) due to limitations discussed in Section 4.3.3. All of the benchmarks are evaluated for a performance comparison in Section 4.3.3.

Table 1. PARSEC-3 and Splash-2X Benchmark Data Set and Trace Event Totals

| Benchmark | Input Data Parameter | Trace Size | OPs (Billions) | | | Sync |
|---|---|---|---|---|---|---|
| | | | IOPs | FLOPs | LD/STs | |
| Barnes | NBody = 32768 | 179MB | 0.9401 | 0.0004 | 0.5885 | 68778 |
| BlackScholes | SimLarge | 2.6GB | 6.7623 | 0.4420 | 1.9834 | 20 |
| BodyTrack | SimSmall | 219MB | 1.9866 | 0.1101 | 0.2841 | 6618 |
| Canneal | SimSmall | 962MB | 6.4254 | 0.0183 | 3.3389 | 292 |
| Cholesky | SimLarge | 195MB | 0.9413 | 0.0003 | 0.4216 | 44114 |
| FFT | $2^{20}$ complex data points | 292MB | 1.2654 | 0.0052 | 0.4272 | 130 |
| FMM | SimSmall | 197MB | 1.3080 | 0.0027 | 0.4442 | 90116 |
| LU_CB | 1500x1500 matrix | 425MB | 3.8615 | 0.0023 | 1.8034 | 352 |
| LU_NCB | 1500x1500 matrix | 487MB | 3.3021 | 0.0023 | 1.5759 | 314 |
| Ocean_CP | 130x130 | 244MB | 0.0594 | 0.0003 | 0.0481 | 10946 |
| Radiosity | SimSmall | 189MB | 2.4911 | 0.1981 | 0.4585 | 433229 |
| Radix | SimSmall | 353MB | 0.3145 | 0.0545 | 0.1115 | 346 |
| Raytrace | SimSmall | 384MB | 1.0544 | 0.0042 | 0.3330 | 422878 |
| Volrend | SimSmall | 184MB | 2.3674 | 0.2742 | 0.4599 | 121465 |
| Water-NSquared | SimMedium | 1.7GB | 24.4305 | 3.7E-06 | 8.2443 | 135450 |
| Water-Spatial | Native | 1.6GB | 17.1686 | 0.0003 | 6.0378 | 210 |

We use the SynchroTrace Replayer illustrated in Figure 5. The traces are only generated once per benchmark and used for simulation of all 64 design points. The SynchroTrace simulation flow uses the same cache and NoC simulators, Ruby and Garnet, that are used by gem5. For our analysis, we use the TimingSimpleCPU core model in gem5 which is a 1−CPI in-order pipelined model.

Table 2. Cache Design Parameters

| | |
|---|---|
| Block Size (B) | 64, 128, 256 |
| L1 Size (kB) | 16, 32, 64, 128 |
| L1 Associativity | 2, 4, 8, 16 |
| L2 Size (kB) | 128, 256, 512, 1024, 2048 |
| L2 Associativity | 2, 4, 8, 16 |

Table 3. NoC Design Parameters

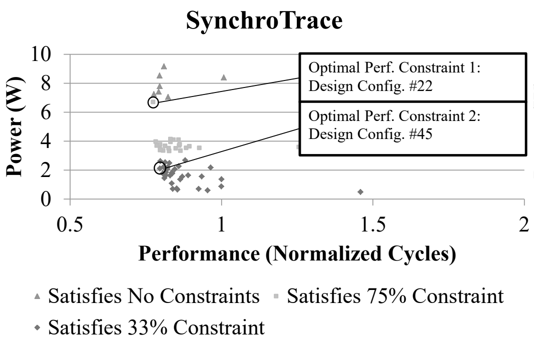
| | |
|---|---|
| Buffer Depth (Flits) | 1, 2, 3, 4, 5, 6, 7, 8 |
| # of Virtual Channels | 1, 2, 3, 4, 5 |
| Channel Bandwidth (B) | 2, 4, 8, 16, 32 |

## 4.2 Area and Power Constraints

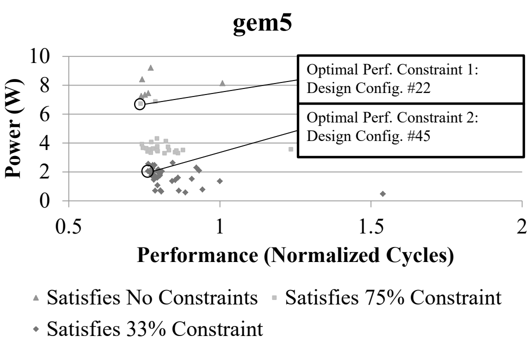
System designers often search the design space for design configurations within constrained resources. Thus, we chose two constraints for the pruning of the uncore design space are based on 1) 75% and 2) 33% of area and total power of the most resource-intensive design configuration. The 75% constraint corresponds to a limit of 6.9W and 107mm$^2$, while the 33% constraint corresponds to a limit of 3.1W and 48mm$^2$. Design points satisfying each of the design constraints 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 3.0 are equivalent in both the SynchroTrace simulation flow and gem5, as area is determined by the design configurations and not the application.

Detailed in Section 4.3, under equivalent constraints, the same design points are selected by SynchroTrace and gem5. The total power of the design points are consistent with both simulators;



(a) SynchroTrace Power vs. Performance



(b) gem5 Power vs. Performance

Fig. 6. Total Uncore Power (NoC and Caches) vs. Performance

this 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 1.4%.

## 4.3 Design Choices Under Constraints

Given the constraints in Section 4.2, our goal is to find the uncore hardware configuration that yields the highest performance. 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. Finally, we assess the accuracy of execution time of SynchroTrace against gem5 for the simulated set of benchmarks.

*4.3.1 Constraint 1: 75% of Max Area and Power.* The design space under Constraint 1 is comprised of 58 design configurations which are simulated in SynchroTrace and gem5. For illustrative purposes, Figures 6a and 6b show the normalized performance of each of the 64 design configurations for all of the simulated benchmarks in comparison to power. Applying the 75% constraints, the top remaining design is found to be design configuration #22 for SynchroTrace and gem5. Additionally, the difference of normalized execution time of design configuration #22 for the two simulation frameworks is 4.8%. Thus, under equivalent area and power constraints, SynchroTrace is able to capture the correct best design configuration as full-system simulation.

*4.3.2 Constraint 2: 33% of Max Area and Power.* With strict area and power constraints of 33%, the design space is pruned to 34 design points. As shown in Figures 6a and 6b for the given constraints, design configuration #45 is selected as the best design for SynchroTrace and gem5. Additionally, the difference of normalized execution time of design configuration #45 in each simulator is only 4.5%. Thus, even with strict area and power constraints, SynchroTrace is able to capture the equivalent top design of gem5, while providing an accurate performance result.

*4.3.3 Performance Metric Comparisons of SynchroTrace to gem5.* The comparison of execution time of SynchroTrace and gem5 across each of the 16 benchmarks are shown in Figure 7. We compare the errors of predicting execution time for each simulation in a set of boxplots. Outliers, represented by circles in Figure 7 are defined as data points with values that are larger than 150% of the interquartile range, i.e. above the upper quartile and below the lower quartile. However, outliers are considered when computing the median error of each boxplot. Shown in Figure 7, the median difference for 12 of the 16 benchmarks in the estimate of execution time falls under 5%, with a median error of 24% for Bodytrack, 7.1% for FMM, 19% for Ocean_CP, and 8.7% for

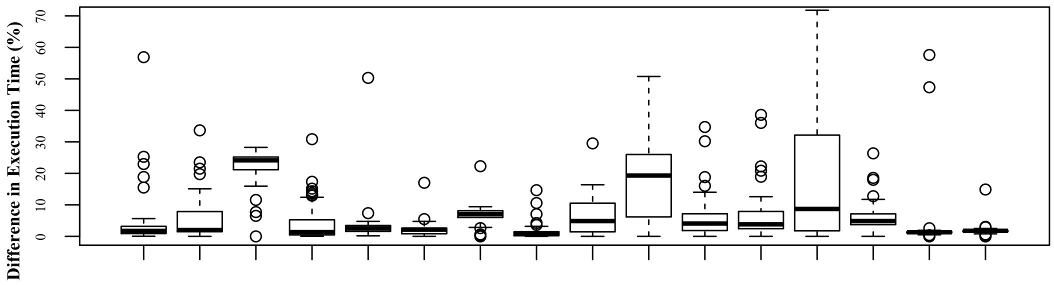


Fig. 7. Comparing Execution Time of SynchroTrace to gem5

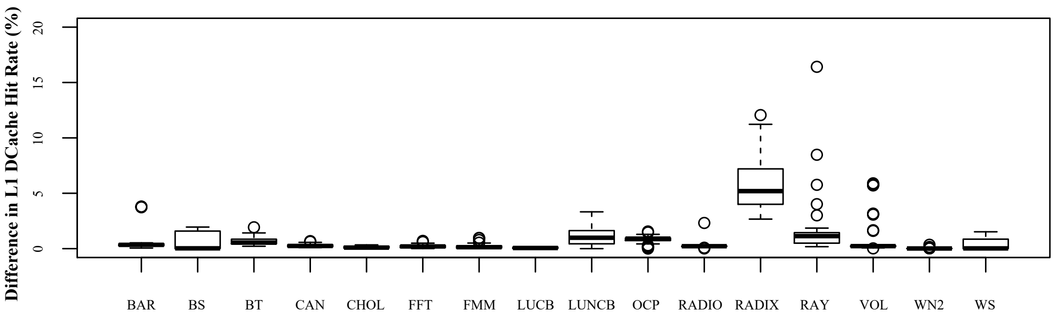


Fig. 8. Comparing L1 DCache Hit Rate of SynchroTrace to gem5

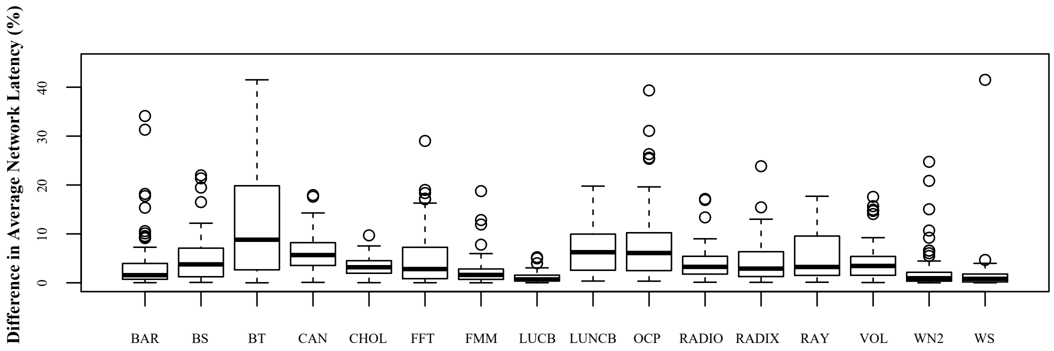


Fig. 9. Comparing Average Network Latency of SynchroTrace to gem5

Raytrace. It is clear that Bodytrack, Ocean_CP, and Raytrace have significantly higher error than the remaining benchmarks and do represent limitations to the SynchroTrace replay model. While Bodytrack uses prevailing pthread calls, this benchmark implements a custom thread pool for scheduling threads to process work. As we do not currently capture or model the user-defined synchronization function calls of the custom user-defined thread pools, the performance in error is largely attributed to the improper scheduling of threads in the SynchroTrace replay model. The error of Ocean_CP is reflective of the user-level thread dependencies detected in the tracing and replay of the benchmark. While SynchroTrace can enforce the dependencies associated with the user-level thread communication, this results in higher prediction error compared to full-system simulation. Lastly, simulations of Raytrace reveal a limitation of how the thread scheduler assigns mutex locks to threads; as Raytrace is comprised of a single synchronization barrier with a majority of the thread communication occuring through mutex locks, we discovered that two threads of the eight monopolized the mutex lock through a majority of the simulations. Thus, for a significant amount of design configurations, the performance predicted in SynchroTrace is much worse than the respective performance in gem5.

In addition to the performance comparison, the L1 DCache hit rate and average NoC latency predictions are compared in SynchroTrace and gem5 and are shown in Figures 8 and 9. As shown in Figure 8, all benchmarks have a median error of below 1% prediction error of the L1 DCache hit rate, with exception of radix (5.2%). All of the benchmarks exhibit third quartile prediction error for average network interconnect latency to be below 10% prediction, with exception to Bodytrack. Given the high accuracy for the memory and interconnect performance, it is clear that SynchroTrace matches the trends of gem5 with regards to the detailed memory and interconnect model.

It should be noted, that while the accuracy of the execution time of SynchroTrace is very high for the better performing designs, the SynchroTrace simulation flow skews slightly toward underestimating the execution time in comparison to gem5, and the skew is increased for resource-constrained designs. The largest differences in performance for each benchmark correspond to the most resource-constrained designs (e.g. the three largest differences in Blackscholes data points correspond to the smallest L2 cache, 128kB). However, from Figures 6a and 6b, we deduce that 1) the power estimation (as well as the area, not shown) between SynchroTrace and gem5 is the same, and 2) under equivalent constraints, SynchroTrace obtains the same design configurations as gem5.

## 5 ANALYZING THREAD SCALING OF HPC APPLICATIONS WITH SYNCHROTRACE

This case study demonstrates that SynchroTrace can efficiently determine the impact of thread count on applications with accuracy comparable to full-system simulation. We determine the thread scalability for several HPC applications with SynchroTrace and gem5. We also compare the cost in simulation time when the number of application threads are increased. Additionally, SynchroTrace experiments are traced on both x86 and Armv8 platforms as to compare the efficacy of using traces from one platform to model another platform.

### 5.1 Experimental Methodology

The focus of this experiment is to establish if SynchroTrace can accurately and efficiently determine how well applications scale to many threads. Toward this goal, CMP simulations of SynchroTrace and gem5 are compared from one to eight threads, with fixed data sizes. Additionally, to determine if SynchroTrace can tractably model benchmarks with a larger thread count, applications are simulated in SynchroTrace up to 64 threads. We simulate four HPC-representative benchmarks of FastForward2 [9], a Department of Energy exascale initiative; the HPC-representative benchmarks are designed to be highly parallel and strong-scaling. The benchmarks are available in the Arm

Table 4. FastForward2 Benchmark Data Sizes

| Benchmark | Description | Input Data Parameter |
|---|---|---|
| CoMD [7] | Molecular Dynamics | Number of Unit Cells in X, Y, Z = 10, Compute EAM Potentials |
| Graph500 [10] | Graph-based Kernels | RMAT Scale 15, |
| LULESH [19] | Shock Hydrodynamics | Size = 30, Iterations = 20 |
| XSBench [22] | Monte Carlo Kernel | Size = Small |

HPC Github (https://github.com/arm-hpc). The descriptions of the benchmarks and input data parameters of are listed in Table 4.

The base design of the simulated CMP consists of a clustered, two-level bus-based topology with a MOESI snooping protocol. The CMP contains private 32kB L1 caches with an associativity of four, shared 1MB L2 caches with an associativity of 16, and 64-byte blocks. Each four-core cluster contains one L2 cache. The cores and memory system operate at 1.7GHz and 1.6GHz, respectively. gem5 simulations are performed using the x86-based TimingSimpleCPU core model.

An added goal of this experiment is to show that traces generated on one platform can model the performance of any platform using SynchroTrace. Thus, Armv8 and x86 traces are collected and simulated in SynchroTrace. For Arm-based simulations, the traces are collected on an AArch64-based Arm server platform with benchmarks compiled using gcc-6.2.1. For x86-based simulations, the traces are collected on a x86-64 server platform with benchmarks compiled using gcc-4.9.2. All benchmarks used in gem5 full-system simulations are compiled using gcc-4.9.2.

## 5.2 Thread Scaling Performance Results

Using the configurations listed in Section 5.1, we evaluate the normalized performance in terms of run-time while sweeping the number of threads in both gem5 and SynchroTrace. For reference, we also show the ideal scaling curve for a program having no serial portions in which performance scales linearly with increasing thread count (Amdahl's law). As shown in Figure 10, SynchroTrace simulations with both Armv8- and x86-based traces are consistent with the scaling performance trends of gem5 in all of the four strong-scaling benchmarks. The largest discrepancy is produced by simulating the LULESH application with eight threads with x86-based traces as shown in Figure 10c. With eight threads, the normalized performance is estimated to be 6.7× with SynchroTrace, while gem5 produces a normalized performance of 6.33×. Graph500 exhibits perfect scaling in SynchroTrace and gem5, as shown in Figure 10b. The two frameworks running Graph500 have an average discrepancy of 0.7%. Given the high thread scalability accuracy of SynchroTrace for up to eight threads, we find that SynchroTrace is an accurate and efficient methodology to determine thread scalability of applications with a large number of threads.

## 5.3 Comparing Results of Armv8 and x86 SynchroTrace Simulations

It is evident from Figure 10 that the thread scaling performance of SynchroTrace simulations with x86- and Armv8-generated traces is almost identical. While the actual execution time between simulations of x86 and Armv8 traces differ by 8.9% to 19.2%, when normalized, the differences of thread-scalability between x86- and Armv8-generated SynchroTrace simulations were under 1% in Graph500, LULESH, and XSBench and under 5.5% in the worst case with CoMD. There is high correlation in x86-generated and Armv8-generated traces due to the translation to the intermediate ISA (VEX) of Valgrind and the abstraction of instructions into events. This allows the traces to be suitable for modeling multiple platforms in an early stage of design, particularly to study uncore performance characteristics and scalability across threads.

## 5.4 Scalability of Simulating HPC-focused Applications

Due to the reduced modeling overhead, SynchroTrace can tractably scale to 64 threads with modest increases in simulation time; unlike gem5, simulating additional cores does not increase the core-modeling overhead in SynchroTrace.

Figure 11 shows the relative simulation time for increasing the thread count of each application in SynchroTrace and gem5 using the x86 platform. With gem5, the worst case simulation time scaling occurs with simulations of LULESH, as 8 thread simulations require a 95% increase of simulation time over single thread simulations. However, the equivalent simulation in SynchroTrace requires

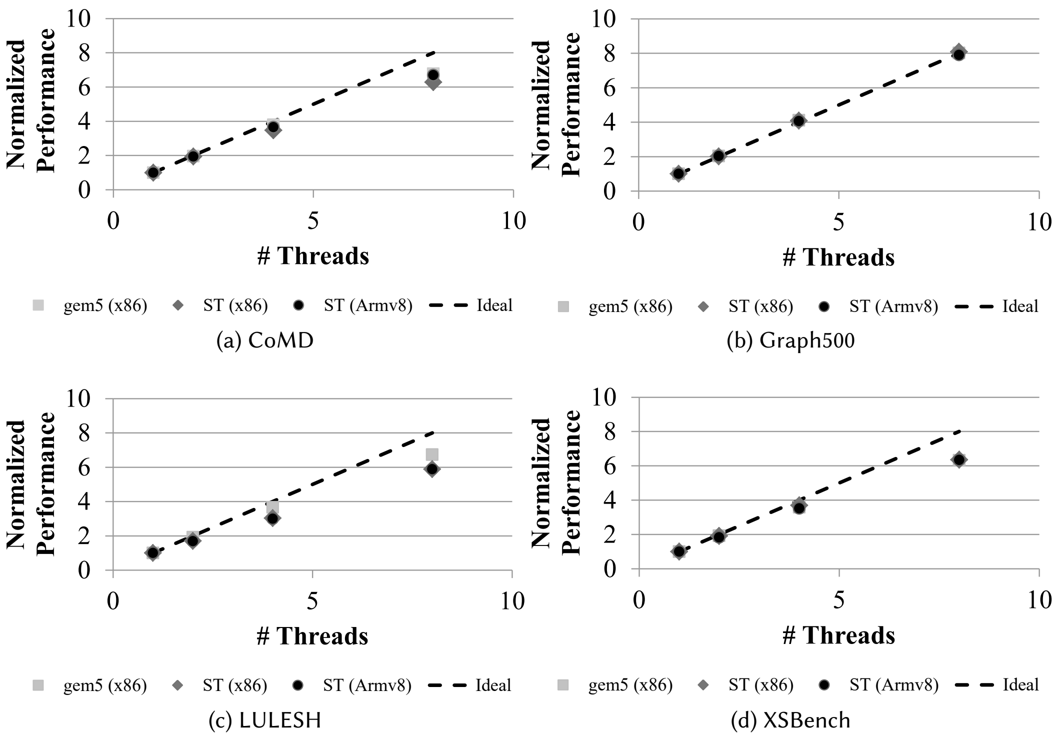


Fig. 10. SynchroTrace and gem5 Comparison in Thread Scaling

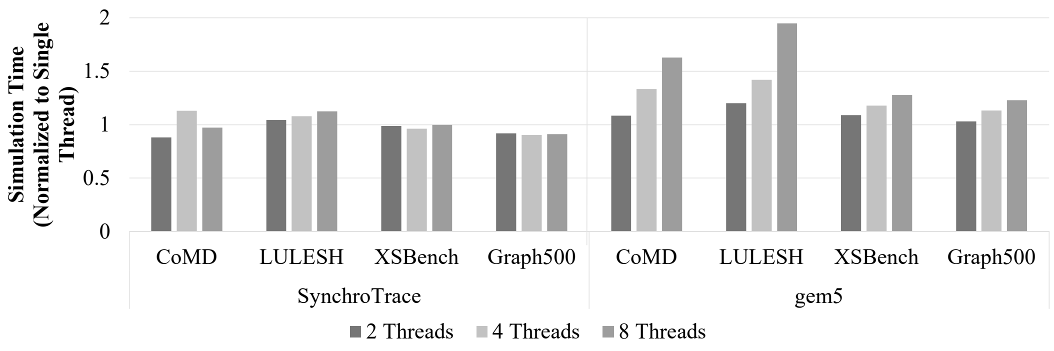


Fig. 11. HPC Simulation Time Comparison Across Threads

only a 12% increase in simulation time in comparison to a single-thread simulation. Simulation time increases less with SynchroTrace than with gem5 as the number of threads increases.

To determine if SynchroTrace is tractable for a large amount of threads, the four HPC benchmarks are simulated up to 64 threads in SynchroTrace. Figure 12 shows the simulation time of each benchmark up to 64 threads, normalized to the single thread simulation of each benchmark. For applications that match ideal strong-scaling, such as Graph500, SynchroTrace only has a slight increase in simulation time as the number of threads is increased. With SynchroTrace at 64 threads, Graph500 has a 5% increase in simulation time over a single-thread simulation; in comparison, the eight thread simulation in gem5 requires a 23% increase in simulation time. In the worst case, SynchroTrace has a 70% increase in simulation time for 64 threads for LULESH. However, it is evident that CoMD with two threads and Graph500 with 2-8 threads have a faster simulation times than their respective single threaded execution. SynchroTrace spends the majority of the simulation time processing the detailed memory model of gem5 (Ruby/Garnet or M5 classic memory model). As each memory request proceeds deeper into the memory stack (e.g. from L1 cache to L2 cache or from L2 cache to the main memory controller), there is an increase in the number memory sub-components that require computation. Thus, benchmarks and design configurations that more often access the L2 cache and main memory require higher simulation time. For example, the compute-intensive CoMD has a simulation with an L1 cache hit rate of 99%. As we increase the number of cores, we effectively increase the L1 cache size for CoMD simulation. Thus, when simulating CoMD with two threads, we achieve a speedup of 12% due to the reduced simulation time spent by processing fewer transactions in the L2 cache and NoC transactions. However, as we further increase the number of cores, we do indeed increase the overall amount of the detailed memory sub-components (e.g. cache, interconnect busses, NoC components) modeled through increased thread communication, and thus, increase the simulation time. Overall, given the limited simulation time overhead, it is observed that SynchroTrace can tractably scale to 64 threads.

## 6 TRACE OPTIMIZATION TECHNIQUES FOR FAST CMP DESIGN EXPLORATION

Although our SynchroTrace simulation flow is by default faster than gem5, our multi-threaded traces can be used to speed up simulation further by trading off accuracy for simulation speed. To this end we propose techniques including event compression, hit prediction, and trace filtering. Figure 14 illustrates the speedup of the SynchroTrace simulation flow for all the trace techniques over gem5. For this experiment, we simulate a modern CMP configuration most closely represented by the largest design point in Tables 2 and 3 (i.e. from [16]) for applications with 8 threads. We show up to 18.7× 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.

### 6.1 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 14 shows the speedup in simulation time from using SynchroTrace versus the gem5 full-system TimingSimpleCPU-based model. We simulate multiple benchmarks from the PARSEC-3 and Splash-2X benchmark suites for both the multi-threaded trace-based simulation flow and the gem5 full-system simulation flow. With the light-weight core model of SynchroTrace, the multi-threaded trace-based simulation flow has up to a 5.2× speedup with an average of 2.9× speedup over gem5 across the benchmark simulation executions.

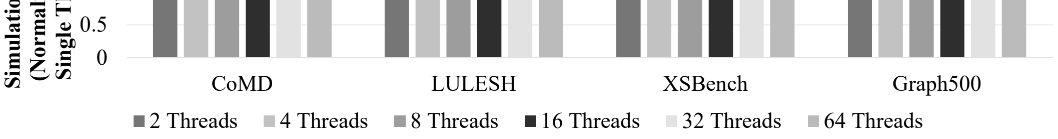
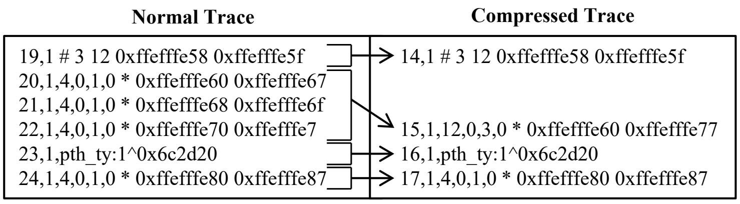


Fig. 12. SynchroTrace Simulation Time up to 64 Threads

**Algorithm 3** Event Compression

```
1: if $IOPEVENT$ then
2:     $IOPcnt$ += 1
3: else if $FLOPEVENT$ then
4:     $FLOPcnt$ += 1
5: else if $STOREEVENT$ then
6:     $STOREcnt$ += 1
7:     $AddrList = AddrList \bigcup STOREAddrs$
8:     if $STOREcnt > CompressLimit$ then
9:         $flushAndResetComputationEvent$
10:    end if
11: else if $LOADEVENT$ then
12:    if $isCommunicationEdge$ then
13:        $flushAndResetComputationEvent$
14:        $flushCommunicationEdge$
15:    else
16:        $LOADcnt$ += 1
17:        $AddrList = AddrList \bigcup LOADAddrs$
18:        if $LOADcnt > CompressLimit$ then
19:            $flushAndResetComputationEvent$
20:        end if
21:    end if
22: else if $SYNCEVENT$ then
23:    $flushAndResetComputationEvent$
24:    $flushSyncData$
25: end if
```



```
19,1 # 3 12 0xffefffe58 0xffefffe5f          14,1 # 3 12 0xffefffe58 0xffefffe5f
20,1,4,0,1,0 * 0xffefffe60 0xffefffe67
21,1,4,0,1,0 * 0xffefffe68 0xffefffe6f
22,1,4,0,1,0 * 0xffefffe70 0xffefffe7        15,1,12,0,3,0 * 0xffefffe60 0xffefffe77
23,1,pth_ty:1^0x6c2d20                       16,1,pth_ty:1^0x6c2d20
24,1,4,0,1,0 * 0xffefffe80 0xffefffe87       17,1,4,0,1,0 * 0xffefffe80 0xffefffe87
```

Fig. 13. An Example of Event Compression

## 6.2 Trace Compression

Our traces are generated by abstracting the program behavior into events and aggregating the events as explained in Section 2; we produce Computation, Synchronization, and Communication events for multi-threaded programs that use prevailing synchronization APIs. This provides an opportunity to compress the trace by merging 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. The compression is demonstrated in Algorithm 3 with an illustrated example of the merged events in Figure 13.

When parsing a merged event, the Replay mechanism also optimizes playback by attributing cycles for hits within a merged-event. Merging events together will lose some ordering information amongst operations for the benefit of compression. We can set a limit on the number of events that can be merged together in the trace, so as to maintain accuracy. For the PARSEC 3 and Splash-2X benchmarks tested, we found the optimal trace compression limit was 100 events per line, which produces around 6% difference in execution time, but shows large improvement in compression and simulation time. This compression reduces zipped file sizes by up to 93% for some benchmarks and 87% on average, while the simulation flow has up to an 12.4× speedup with an average of 7.3× speedup over gem5 full-system as shown in Figure 14.

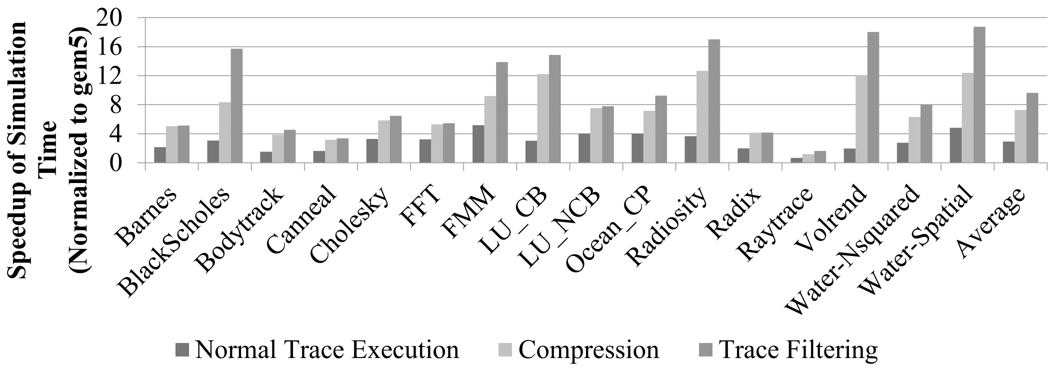


Fig. 14. SynchroTrace Speedup in Simulation using our Multi-Threaded Trace Techniques over gem5

### 6.3 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 [31, 38]. Puzak [31] uses 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. [38] 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; we do not filter hits to shared data as it can become complex due to non-determinism.

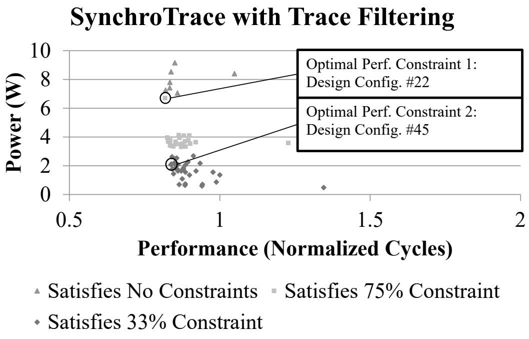
The filtering technique we implement post-processes the trace and uses a filter cache structure to remove address ranges from computation events if they hit in the filter cache. The technique also adds a field to the trace to record the hit count, which can be used to estimate cycles by the Replay mechanism. 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 [1, 4]. 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.

We show an 18.7× increase in speedup over gem5 with an average of 9.6× as detailed in Figure 14. Canneal, Water-NSquared, and Water-Spatial traces are relatively large and would benefit from more aggressive compression and filtering techniques.

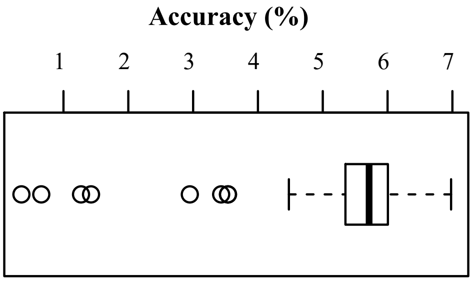
We ran the equivalent design space exploration experiment of the Section 4, and as shown in Figure 15a, we found the same optimal performing designs given the 75% and 33% power and area constraints. Additionally, while we obtain large gain in speedup, the accuracy impact is tolerable, as shown in Figure 15b, with a median error of 5.7% and a standard deviation of 1.4%.

## 7 BACKGROUND AND RELATED WORK

The most accurate solution for a simulation-based design space exploration can be obtained through full-system simulators such as gem5 [3] that execute entire applications. A number of scalable simulators that use parallel simulation have been released [6, 23, 33]. They allow different levels



(a) Design Space Exploration with Trace Filtering



(b) Comparing Execution time of SynchroTrace to SynchroTrace with Trace Filtering

Fig. 15. Accuracy of Trace Filtering Technique

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 [6, 23, 33]. These parallel simulators have not been fully validated for relative errors and design space exploration capabilities. Additionally, 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 performance improvement using the SynchroTrace Replay model and trace filtering.

The 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 [13]. SynchroTrace is able to model non-determinism by capturing and embedding synchronization events in the trace and tracking dependencies between traces during capture.

### 7.1 Comparison to Pinplay

PinPlay provides a framework, based upon dynamic instrumentation, to capture execution into traces (Pinballs) and replay the captured execution, deterministically [30]. There are clear benefits to deterministic replay, such as debugging (e.g. DrDebug [36]) 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 the execution of each individual thread. Included in Pinballs is a thread dependency file that captures shared memory reads and writes in order and instruction dependencies among threads to deterministically replay the traces in the captured order. However, deterministic replay produces the same thread interleaving for every run and does not allow for timing behavior to affect the critical path of multi-threaded applications. The enforcement of thread event ordering by Pinplay 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 of multi-threaded applications in the context of design space exploration.

### 7.2 Trace-based Model of Out-of-Order Cores

Elastic Traces [17] accurately captures the ILP and MLP of out-of-order processors with a trace-based model. Jagtap et al. [17] address several challenges of modeling out-of-order cores with trace-based simulation, including capturing and enforcing read-after-write data dependencies of instructions, capturing order dependencies of loads and stores, modeling speculative and wrong path loads, and capturing local computation timing. However, the techniques presented in Elastic Traces are specific to the target microarchitecture and only for single-threaded applications. By addressing these challenges in the architecture-agnostic tracing tool of this work, SynchroTrace could be adapted for efficiently modeling out-of-order cores in CMPs in future work.

### 7.3 Reducing Simulation Time with Proxy Benchmarking Techniques

Researchers have explored reducing simulation time of benchmarks through creating proxy applications or synthetic, representative applications. Deniz et al. [8] and Ganesan et al. [11, 12] present automated frameworks that capture characteristics of multi-threaded applications and generate portable, light-weight synthetic benchmarks for CMP simulation. The goal of these frameworks is to reduce the simulation time of representative applications as to quickly estimate pre-silicon design performance. The methodologies of these techniques are similar to tracing; the frameworks characterize the behavior of applications and create a proxy for a full benchmark simulation. Overall, the methodologies can be implemented in the SynchroTrace simulation flow for increased speedup.

### 7.4 Other Efficent Modeling Solutions

Prometheus [20] is an emulation-based framework catered toward modeling many-core systems. While Prometheus can be used to study exascale-level supercomputer nodes, the focus of the work is in platforms running task-based applications.

Rico et al. [32] 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 [32]. However, this methodology requires source to source transformations to interface the parops with their 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 two custom applications. 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 CMP design space [14, 15, 29, 34]. 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 [14, 15, 29]. Huang et al. use a bloom filter inspired approach for message passing interface (MPI) based applications but cannot handle shared-memory applications [15]. The methodology of Nitta et al. and Netrace suffer from the need for multiple full-system runs to infer true dependencies [14, 29]. In general, collecting traces through full-system simulation is not scalable to large number of threads. To the best of our knowledge, SynchroTrace is the first to generate reliable synchronization and dependency-aware multi-threaded traces that require no changes to application code for architecture simulation.

## 8 CONCLUSIONS

In this article, we have presented the SynchroTrace methodology for accurate, flexible, scalable, and fast design space exploration for multi-threaded applications. As our traces of multi-threaded applications have dependencies and synchronization embedded in them, we solve the issue of a modeling the non-determinism of thread interleaving in multi-threaded applications with a trace-driven methodology. We validate the SynchroTrace simulation flow by successfully achieving the equivalent results of a constraint-based CMP design space exploration with the gem5 full-system simulator. We show how our methodology can trade-off accuracy for speed by compressing and filtering traces. We have also presented new extensions that include support for HPC (OpenMP) applications [9], a case study with the latest CMP applications (PARSEC-3 [2], and Splash2X [37]), and a new HPC-focused thread-scalability case study, proving that SynchroTrace can capture thread scaling behavior accurately and efficiently. The results from these case studies show that our methodology is adaptable across platforms in the early design stage, has a peak speedup of up to 18.7× over simulation in gem5 full system, and tractably scales to 64 cores.

## 9 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] Beyls, K., and D'Hollander, E. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems* (2001), pp. 617–662.

[2] Bienia, C. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

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

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

[5] Carlson, T. E., Heirman, W., Craeynest, K. V., and Eeckhout, L. Barrierpoint: Sampled simulation of multi-threaded applications. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on* (March 2014), pp. 2–12.

[6] Carlson, T. E., Heirman, W., and Eeckhout, L. 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).

[7] Cicotti, P., Mniszewski, S. M., and Carrington, L. An evaluation of threaded models for a classical md proxy application. In *2014 Hardware-Software Co-Design for High Performance Computing* (Nov 2014), pp. 41–48.

[8] Deniz, E., Sen, A., Kahne, B., and Holt, J. Minime: Pattern-aware multicore benchmark synthesizer. *IEEE Transactions on Computers 64*, 8 (Aug 2015), 2239–2252.

[9] Exascale initiative. http://www.exascaleinitiative.org/.

[10] Fuentes, P., Bosque, J. L., Beivide, R., Valero, M., and Minkenberg, C. Characterizing the communication demands of the graph500 benchmark on a commodity cluster. In *2014 IEEE/ACM International Symposium on Big Data Computing* (Dec 2014), pp. 83–89.

[11] Ganesan, K., Jo, J., and John, L. K. Synthesizing memory-level parallelism aware miniature clones for spec cpu2006 and implantbench workloads. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)* (March 2010), pp. 33–44.

[12] Ganesan, K., and John, L. K. Automatic generation of miniaturized synthetic proxies for target applications to efficiently design multicore processors. *IEEE Transactions on Computers 63*, 4 (April 2014), 833–846.

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

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

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

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

[17] Jagtap, R., Diestelhorst, S., Hansson, A., Jung, M., and When, N. Exploring system performance using elastic traces: Fast, accurate and portable. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)* (July 2016), pp. 96–105.

[18] Kahng, A. B., Lin, B., and Nath, S. Orion3.0: A comprehensive noc router estimation tool. *IEEE Embedded Systems Letters 7*, 2 (June 2015), 41–45.

[19] Karlin, I., Bhatele, A., Keasler, J., Chamberlain, B. L., Cohen, J., Devito, Z., Haque, R., Laney, D., Luke, E., Wang, F., Richards, D., Schulz, M., and Still, C. H. Exploring traditional and emerging parallel programming models using a proxy application. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing* (May 2013), pp. 919–932.

[20] Kestor, G., Gioiosa, R., and Chavarra-Miranda, D. Prometheus: scalable and accurate emulation of task-based applications on many-core systems. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on* (March 2015), pp. 308–317.

[21] Kim, Y., Yang, W., and Mutlu, O. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters 15*, 1 (Jan 2016), 45–49.

[22] Liu, T., Wolfe, N., Carothers, C., Ji, W., and Xu, G. Optimizing the monte carlo neutron cross-section construction code xsbench for mic and gpu platforms. *ANS Nuclear Science and Engineering 185*, 1 (Oct. 2016), 232–242.

[23] Miller, J. E., Kasture, H., Kurian, G., Gruenwald, C., Beckmann, N., Celio, C., Eastep, J., and Agarwal, A. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2010).

[24] Muralimanohar, N., Balasubramonian, R., and Jouppi, N. 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).

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

[26] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (June 2007), 89–100.

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

[28] Nilakantan, S., Lerner, S., Hempstead, M., and Taskin, B. 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).

[29] Nitta, C., Macdonald, K., Farrens, M., and Akella, V. 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).

[30] Patil, H., Pereira, C., Stallcup, M., Lueck, G., and Cownie, J. 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* (2010), CGO '10.

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

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

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

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

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

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

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

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