Towards Cross-Framework Workload Analysis via Flexible Event-Driven Interfaces

Michael Lui*, Karthik Sangaiah*, Mark Hempstead†, Baris Taskin*
*Department of Electrical and Computer Engineering
Drexel University, Philadelphia, PA USA
Email: {mdl45, ks499}@drexel.edu, taskin@coe.drexel.edu
†Department of Electrical and Computer Engineering
Tufts University, Medford, MA USA
Email: mark@ece.tufts.edu

Abstract—**Hardware/software co-design and software profiling rest on the ability to perform a range of workload analyses. State-of-the-art tools and methods used in such analyses utilize either custom solutions or complex frameworks. There are two problems with this approach: 1) duplicated development work when moving to new and unsupported frameworks or platforms, and 2) the additional burden of in-depth knowledge required to develop the analysis tools. This work presents a methodology to solve these inefficiencies by decoupling workload analysis from the underlying techniques used to observe the workload. The interface is designed to be cross-platform and presents workloads as a set of configurable events with scalable levels-of-detail. An implementation of the methodology, PRISM, is presented which leverages two popular dynamic binary instrumentation tools, Valgrind and DynamoRIO, and additionally Intel PT via Linux perf. The goals of the methodology are three-fold: modularity, flexibility, and productivity. Three analyses are conducted using PRISM to demonstrate these properties: 1) discrepancies are assessed between workloads generated with Valgrind, DynamoRIO, and perf, 2) scalability of a complex Valgrind trace generation tool is improved, and 3) prototyping of a new dynamic loop detection and data-dependence tool is demonstrated. The average overhead of PRISM compared to in-framework analysis is 33% in the worse case and under 1% during typical analysis.**

I. Introduction

Workload analysis is used for software/hardware co-design, application profiling, debugging, and program verification. Workload analysis is achieved with a variety of techniques, including modification of software and specialized hardware to capture recent state [1]–[11]. Consistent with such variety, numerous frameworks and tools exist to streamline the development of new workload analysis tools. Yet, the status quo of creating a new analysis is still unnecessarily complex for common uses.

The general flow for creating an analysis tool can appear straightforward: 1) identify a supporting framework and 2) implement the analysis tool in the framework. However, each step is replete with risk. The nature of the analysis or its requirements can change over time, making identifying and committing to a framework dangerous. For example, it is common to eventually scale up workloads or target new workloads that require alternate platforms. Likewise, the emergence of heterogeneous architectures increases the importance of supporting alternate platforms. In the case of dynamic binary instrumentation (DBI) frameworks, Valgrind supports 32-bit and 64-bit x86, arm, and MIPS instruction sets but has no support for Windows [1]. DynamoRIO supports recent Windows releases, but has until recently had limited support for 64-bit arm and no support for MIPS [2]. Pin, another popular DBI framework, is only supported on Intel platforms [4]. Additional frameworks are developed to fill in the gap areas of support and further fraction the landscape. For example, Mambo [3] supports the low-overhead instrumentation of DynamoRIO while also providing the arm support that DynamoRIO lags. This does not even begin to address application support and transparency requirements of each framework. None of the aforementioned frameworks are able to provide analysis for GPGPU applications. A sample of the current environment of dynamic binary analysis tools is shown in Table I. The list is non-exhaustive but demonstrates the variety of approaches.

Besides choice of framework, development in these frameworks presents additional complexity. Both dynamic and static binary instrumentation (SBI) frameworks require users to modify an application in order to simply observe it. This adds a multitude of burdens, including reasoning about injected code, separation of analysis and application code, code and data reachability concerns, machine state preservation, performance implications, correctness, et al, in addition to development of the analysis. Furthermore, each framework presents its own associated constraints. For example, Valgrind serializes workloads and does not allow analysis tools to use third party libraries, including the standard C library. DynamoRIO provides more flexibility but still requires special attention to corner cases of its API, such as when analysis code is allowed to hold locks and how additional threads are implemented. Rather than specifying required metadata and having it presented for analysis, tool developers must parse through all the available capabilities and become experts within each framework to generate that data themselves. These constraints can often result in brittle tools and analyses. As such, a methodology is presented for more robust and flexible workload analysis.

As far as the authors know, PRISM is the first cross-platform, cross-framework workload analysis methodology.

TABLE I: Seminal and Recent Workload Analysis Tools

Tool Name	Implementation	Category	General Events Used
Memcheck/Massif [1]	Valgrind	Heap Profiler/Checker	memory, functions
Dr. Memory	DynamoRIO	Heap Profiler/Checker	memory, functions
Heap Check	gperftools	Heap Profiler/Checker	memory, functions
AddressSanitizer	GCC/Clang	Memory Checker	memory, functions
Gleipnir [12]	Valgrind	Memory Profiler	memory, functions
Cachegrind [13]	Valgrind	Cache Simulation	instructions, memory
Dr. Cachesim	DynamoRIO	Cache Simulation	instructions, memory
DineroIV [14]	Custom	Cache Simulation	memory
WIICA [8]	LLVM	Workload Characterization	instructions, memory, control flow
Callgrind [15]	Valgrind	Callgraph Gen. and Profiling	instructions, memory, control flow
Gprof	Custom	Callgraph Gen. and Profiling	functions, hardware sampling
CCTLib [16]	PIN	Callgraph Gen.	instructions, memory, functions
Sigil [17]	Valgrind	Callgraph Gen. and Profiling	instructions, memory, functions
SPEC-AX & PARSEC-AX [18]	LLVM	Path Profiling	basic blocks
DACCE [19]	Custom + PIN	Calling Contexts	functions, synchronization
SynchroTrace [20]	Valgrind	Trace Generation	instructions, memory, synchronization
TraceCPU [21]	gem5	Trace Generation	simulation model probes

The three main contributions of this work include:

1) a flexible workload representation to accommodate offline, latency-tolerant analyses
2) a separate event-driven interface for more productive, cross-framework workload analyses
3) an open source prototype, *PRISM*, that currently utilizes Valgrind, DynamoRIO, and Intel PT via perf [22]

Specifically, this work considers analyses that can either be performed offline or are latency-tolerant, including trace generation, characterization, heavy-weight application profiling, and record-and-replay style debugging [8], [15], [18], [20], [23], [24]. Such analyses are performed at all granularities, ranging from low-level hardware event monitoring to source-level program semantics. Other recent works target analyses that must be performed online or require quick feedback to the user, including real-time threat detection, performance profiling, debugging, dynamic parallelization, et al [25]–[27]. Furthermore, some of these works aim to modify the application in addition to observing the workload. As such, these analyses reside outside the scope of this paper and are better addressed through more direct and custom solutions.

The rest of the paper is organized as follows. A brief background on the techniques used for workload analysis are summarized in Section II. Related works that approach decoupling event generation from event analysis are presented in Section III. In Section IV, the approaches this work proposes to integrate workload analysis techniques to a condensed, flexible interface are described. A detailed look at the implementation of the PRISM framework, including performance considerations is presented in Section V. An evaluation of the performance and utility of the PRISM framework is provided in Section VI, where three case studies and a performance analysis are explored.

II. Background

Workload analysis is primarily achieved by three sampling methods: 1) dynamic binary analysis (DBA), 2) specialized hardware features and performance counters (HPCs), or 3) probing of simulation models. Each sampling method has capability benefits with varying performance costs.

1) Dynamic Binary Analysis: DBA modifies the application software either in-source, at compile-time, or at run-time to inject analysis code to record salient information for a given input and environment. *Dynamic* binary instrumentation (DBI) frameworks provide a interface to analyze an application's intermediate representation (IR) during run-time and then insert additional instructions to either change the application's functional behavior or to characterize it at runtime [1]–[4], [10]. Additionally, some DBI frameworks have the ability to perform dynamic binary translation, converting the source binary to a different target platform. *Static* binary instrumentation (SBI) tools provide similar capabilities, but insert additional instructions at or before compile-time [7], [8]. SBI has access to the full context of the source code, compared to DBI, and thus provides richer information for analysis. Higher-level languages (e.g. Python, Go, C#, Java) include profiling and analysis APIs with the language runtime and standard libraries. DBA is the most flexible and general purpose workload analysis method, at the cost of instrumentation overhead.

2) Hardware Event Sampling: Architects often add additional hardware to provide low-cost system introspection, such as hardware performance counters (HPCs) and trace solutions like arm CoreSight and Intel Processor Trace. As these techniques are implemented directly in hardware, they avoid the run-time overhead of software-based analysis. HPCs provide a rich set of real-time event counters for the system, such as instructions retired, cache misses, and branch mispredictions. Trace hardware provides highly compressed event traces that include dynamic execution and power management events. These hardware features provide real-time insight to a workload, but at the cost of extensibility and configuration. The volume and highly encoded nature of this data also make capturing and decoding large samples computationally more complex than DBA [27]. This limits the general usefulness of hardware sampling to performance tuning and debugging.

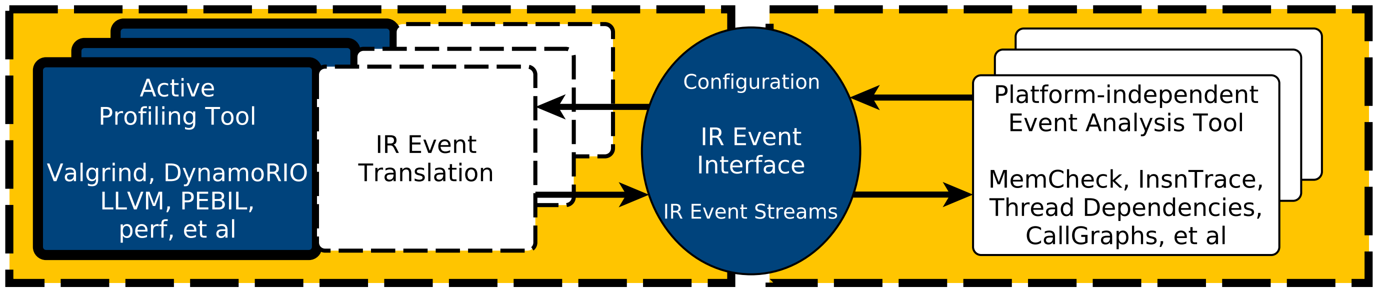

Active Profiling Tool
Valgrind, DynamoRIO LLVM, PEBIL, perf, et al
IR Event Translation
Configuration
IR Event Interface
IR Event Streams
Platform-independent Event Analysis Tool
MemCheck, InsnTrace, Thread Dependencies, CallGraphs, et al

Fig. 1: Decoupled Event Analysis Methodology

3) Simulation Probing: Hardware simulations provide the most control over granularity, allowing as much detail as available in the model without interfering with the application software. The computational cost of using full system simulators (e.g. gem5 [28], PTLsim [29], and MARSS [30]) makes this method the slowest (e.g. gem5 and MARSS operate simulation speeds of about 200 KIPS), although this can be alleviated with various sampling techniques [31], [32]. Additionally, the user is limited to the accuracy, ISA support, and application support of user workloads executing in full system simulators.

III. Related Works

As far as the authors are aware, this is the first work to target a cross-framework, cross-platform workload analysis methodology. However, this is not the first work to identify the utility in decoupling event generation from event analysis. PiPa [33] reduced analysis overhead by decoupling and parallelizing instrumented code and analysis code within DynamoRIO. A similar approach was later applied to the PIN framework [34]. This work demonstrates similar efficiency gains with the Valgrind framework in Section VI-B. Aftersight decouples the execution and analysis of an entire virtual machine to enable certain analyses that would be otherwise intractable in a deployed system [35]. The Linux perf profiler abstracts various kernel and hardware event sources to a single framework interface via perf_events. However, perf is naturally skewed toward supporting fast, low-overhead performance profiling and performance debugging tools, in contrast to the more heavyweight, latency-tolerant analyses targeted in this work. OpenTracing aims to standardize an interface to higher-level events in production systems, motivated by the similarly fractured nature of popular distributed tracing subsystems and frameworks [36]. Lastly, one of the first published instrumentation frameworks, ATOM, was motivated by 1) the inefficiencies of having users manage low-level code instrumentation and 2) many custom solutions being generated for a common class of problems [6].

IV. Modular Event-Driven Analysis

In this section, a methodology to improve workload analysis by decoupling workload generation from workload analysis is outlined. This decoupling gives rise to a more amenable and robust event-driven interface for analysis. Presenting workloads as a streams of events has several benefits:

1) Analysis code is modular; it is developed only once and can then be deployed cross-platform and cross-framework.
2) Analysis code does not require the user to modify a framework or application.
3) Analysis development is more productive; users can focus on developing and running analyses, while framework experts can focus on improving support and performance.
4) Analysis can request a set of event-types; if the underlying framework cannot support the set then the user is immediately notified.
5) The analysis tool is unconstrained by the underlying profiling framework; it can be developed and tested with a standalone project stub.

Figure 1 shows an overview of this methodology. It has three components:

1) an event generation *frontend* that feeds the backend via a framework or tool.
2) an event stream represented as versatile and extensible event primitives to decouple workload analysis from event generation,
3) an event analysis *backend* that processes the workload via an event-driven interface,

A. Workload Event Stream

Because of the variety of analyses being targeted, workloads are represented as discrete, generalized event *primitives*—tuples of related attributes—in ordered streams. Event primitives are naturally dependent on both the capabilities of the event generation frontend and the nature of the event analysis backend. For example, platform-agnostic synchronization points, e.g. locks or condition variables, can be more useful than the low-level details of their implementation in the workload. As such, the primitives are intentionally abstract to support a sliding level-of-detail. Common use-cases, as in Table I, were used to guide the choice of primitives.

Each event primitive and its description are as follows:

- **Memory:** retrieves or stores resources.
- **Compute:** performs some transformation on data.

TABLE II: Event Granularity

Event	Granularity	Attribute
Memory	Coarse	read/write
	Medium	address
	Fine	size
Compute	Coarse	IOP/FLOP
	Medium	add/sub/mult/div
	Fine	size of operands
Synchronization	Coarse	spawn/join/sync
	Medium	thread id
	Fine	pthread function
Context	Coarse	instruction/loop/basic block/function
	Medium	address
	Fine	opcode
Control Flow	Coarse	jump
	Medium	destination
	Fine	conditional

- **Synchronization:** orders independent streams of consecutive memory and compute events.
- **Context:** groups memory, compute, and/or synchronization events. For example, an instruction can have multiple memory events; a function or basic block can have multiple instructions, memory events, and compute events.
- **Control Flow:** a potential divergence in the event stream. Control flow was not implemented for the case studies in this paper.

A subset of varying metadata detail is shown in Table II. Fundamentally, an implementation can augment this in deployment with alternative IR. For example, LLVM IR has been used to lift binary programs and in both DBI and SBI frameworks, however these tools are still restricted to specific platforms [8], [10], [37], [38].

Figure 2 demonstrates the structure of the event stream. Event primitives are sequentially presented to the event analysis backend. Threads can be merged into one or more event streams to 1) avoid overwhelming the backend with events and 2) allow scaling of event stream resources, such as event buffers and OS-level threads. Workload threads in the same stream are separated by *thread marker* context events, although ordering amongst these threads are specified within separate *synchronization* events. For example, a thread's event stream would include mutex lock and unlock synchronization events, given that the backend requests these events and the frontend supports generating the events.

B. Event Analysis Backend

Event primitives are processed by an analysis *backend* via an event-driven interface. Figure 1 shows the interface between the frontend and backend. In the proposed methodology, backend tools 1) configure the frontend by requesting event primitive attributes and 2) subscribe analysis routines to specific event primitives. In this way, the analysis tool developer works independently of the underlying event generation frontend. If a given frontend cannot support an analysis, the developer can either choose a different frontend, consult a frontend expert, or contribute by adding support for additional future works.

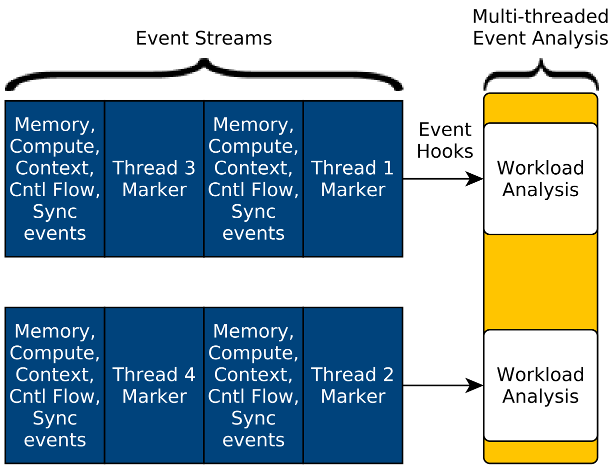

Event Streams
Multi-threaded Event Analysis
Memory, Compute, Context, Cntl Flow, Sync events
Thread 3 Marker
Memory, Compute, Context, Cntl Flow, Sync events
Thread 1 Marker
Event Hooks
Workload Analysis
Memory, Compute, Context, Cntl Flow, Sync events
Thread 4 Marker
Memory, Compute, Context, Cntl Flow, Sync events
Thread 2 Marker
Workload Analysis

Fig. 2: Multithreaded PRISM Event Interface

In the case of multithreaded workloads, threads can be split into independent event streams for improved throughput with parallel analysis. This is completely transparent to the event analysis backend and requires no additional development, except normal multithreaded considerations.

C. Event Generation Frontend

The methodology enables modular event generation by adding two components to a given framework or tool:

1) a component to dynamically translate a workload into event primitives, and
2) an interface to forward event primitives to event analysis.

The development time and effort for these additions is required once to enable a range of analyses. Because the backend is decoupled, there is great flexibility in how a workload can be generated. As with any IR, workloads can be synthetically generated, sent over a network, dynamically or statically instrumented, or captured across distinct platforms.

V. Implementation

In this section, the open-source implementation of the proposed methodology, PRISM [22], is described.

A. Event Analysis Interface

The event analysis backend interface is written in C++14 to provide a balance of user abstraction, flexibility, and performance. Conceptually, any language with sufficient support can serve as the backend interface. An analysis tool implements a *backend* interface with virtual functions to process event primitives, as shown in Listing 1. Level-of-detail, or event attributes, are requested via another callback, which is processed when the backend registers itself, as shown in Listing 2. At runtime, when both the analysis backend and generation frontend are chosen, event attribute support is resolved and an error is reported if there is a mismatch.

```
class Analysis : public BackendIface
{
        virtual void onMemEv(const MemEv &ev) {/**/}
        virtual void onCompEv(const CompEv &ev) {/**/}
        virtual void onSyncEv(const SyncEv &ev) {/**/}
        // ...
};
```

Listing 1: Analysis Backend Interface

```
Requirements request()
{
        Requirements req = initReqs();
        // ...
        req[MEMORY_ADDRESS] = enabled;
        req[SYNC] = enabled;
        req[CONTEXT_INSTRUCTION] = enabled;
        req[CONTEXT_FUNCTION] = enabled;
        // ...
        return req;
};
```

Listing 2: Event Request Interface

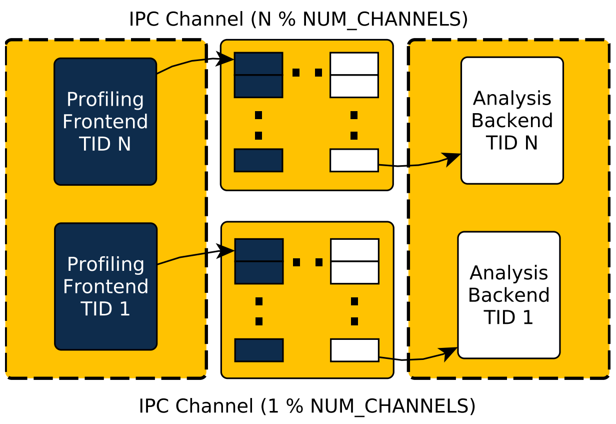


Fig. 3: PRISM IPC Buffering

B. Event Primitive Channels

Because the event analysis backend is a separate process, inter-process communication (IPC) is leveraged to forward events between frontend and backend. Shared memory buffers are implemented in each framework, as they have the lowest overhead, comprising of cache and translation-lookaside buffer (TLB) misses.

A configurable scheme was developed that multi-buffers events in shared memory buffers. The frontend and backend both reserve an available buffer to asynchronously produce and consume events, as shown in Figure 3. Multiple buffers are used to give slack for burstiness in the analysis tool. Asynchronous event generation and event analysis has been shown to have substantial performance benefits within DBI frameworks [33], [34]. Each set of shared memory and synchronization constructs is called a *channel*, and each channel represents one event stream. Recall in Section IV-A that each event stream represents one or more workload threads. That is, a workload with two threads can be processed serially with one event stream, or in parallel with two event streams. The number of event streams is configurable at run time. This is important in workloads with 10's to 100's of threads, where the host machine can become resource starved. Lockfree buffers were also tested, but found to produce too much overhead from frequent context switching in backend event analysis tools.

This achieves three goals: 1) support multiple producers and consumers to take advantage of multithreaded event generation and analysis, 2) minimize the overhead of buffer arbitration, and 3) allow users to easily scale buffer resources. The overhead of this architecture is discussed in Section VI.

C. Event Generation & Translation

Two popular DBI frameworks, Valgrind and DynamoRIO, and the Linux perf system, with Intel PT (IPT) are used. Each framework is integrated into PRISM and invoked with a simple command line switch. Table III shows an overview of event translation for each framework. To support modern multithreaded applications, all frameworks support synchronization events from pthread library API calls. When a pthread call is detected, a single abstract IR event is generated, and event generation is disabled until the call returns.

1) Valgrind [1]: Valgrind is a disassemble-and-resynthesize DBI framework available on Unix-like platforms. Instructions are disassembled into internal *VEX IR*, which are grouped into single-entry multiple-exit basic blocks. The basic block is then instrumented and passed back to the Valgrind core, which performs a just-in-time compilation of the basic block back into the target ISA. Multithreaded applications running under Valgrind are serialized as if running on a single-core architecture. In this implementation, Valgrind is configured to use a ticket lock implemented with futex syscalls. Each thread is run for a given set of basic blocks (100,000 by default) before giving up the big lock. Threads will also give up the lock for requisite reasons such as during blocking syscalls.

The Valgrind frontend leverages a modified Callgrind tool [15]. Callgrind was chosen because it provides facilities for detecting thread changes, function entry/exit, and its own custom event profiling infrastructure. The Valgrind tool attaches to the shared memory buffer and flushes events when the Callgrind's event infrastructure flushes. Because Valgrind serializes workloads, only one event stream is supported.

2) DynamoRIO [2]: DynamoRIO is a copy-and-annotate DBI framework, providing lower instrumentation overhead than Valgrind, which recompiles its IR. The IR interface offered by DynamoRIO is not a decomposed internal IR, but metadata, such as address and opcode, accessed via an internal API; instrumentation acts directly on architecture-specific instructions. This comes at the cost of increased instrumentation complexity, such as validating inserted instructions and hardware state. Like Valgrind, single-entry, multiple-exit basic blocks are analyzed and instrumented by DynamoRIO clients. Unlike Valgrind, native threads are used for multi-threaded applications, and clients are allowed to create additional client-space threads with caveats, with support for many third party standard libraries.

TABLE III: Example PRISM IR Event Translation

	Memory	Compute	Synchronization	Context
Valgrind	leverage Callgrind	VEX IR properties	wrap thread library API	leverage Callgrind
DynamoRIO	instrument memory operands	opcode filter	wrap thread library API	instrument call/return branches
perf	XED library; no address support	XED instruction class filter	check for thread API symbol every instruction	provided by perf

A custom client was created for this implementation that translates instructions into event primitives by filtering instruction opcodes and inserting instrumentation to query memory operands. Translated event primitives are directly written into the shared memory from the code cache. To support varying number of event streams, a custom serialization subsystem was developed, which is not supported by default in DynamoRIO. Valgrind's serialization is used as a reference implementation. Each application thread is assigned to an IPC channel, or event stream, which a thread attempts to lock, with a ticket lock, before generating events. Threads that do not hold the channel lock block until the channel becomes available. A thread that holds the channel reserves it for a predefined number of events to avoid contention in the frontend and to avoid unreasonable context switching in the backend.

3) Perf with Intel Process Trace (PerfPT): Perf is a powerful profiling tool for Linux platforms that offers an interface to a rich set of both hardware and kernel event counters and probes. Recent additions to perf, starting Linux 4.1, have included Intel Process Trace (IPT) support. IPT is supported for 5th generation Intel Core architectures and newer. IPT compresses a workload trace into a set of packets, which can describe dynamic execution with conditional branch results. Perf can reconstruct a full instruction trace with an IPT trace and other side-channel information collected both external to IPT and in additional IPT packets. Additionally, portable analysis is enabled as the recorded trace can be analyzed on different platforms.

The *perf script* tool was modified to fully decode and translate the instruction trace. This required the Intel X86 Encoder Decoder (XED) library to be integrated further into the perf IPT subsystem. The PerfPT frontend first requires that a *perf.data* trace file be created. Then, when the frontend is called, the trace file is used as input into the modified perf tool. Because the trace file is parsed in a single thread, only one event stream is created.

VI. Evaluation

In this section, the targeted goals—modularity, flexibility, and productivity—of the proposed methodology are evaluated with PRISM. Each workload analysis is accomplished with a *single* backend analysis tool, instead of separate implementations in each framework. No Valgrind-, DynamoRIO- or perf-specific code development was required. Instead, the analyses were developed with the event-driven workload interface specified in Listings 1 and 2.

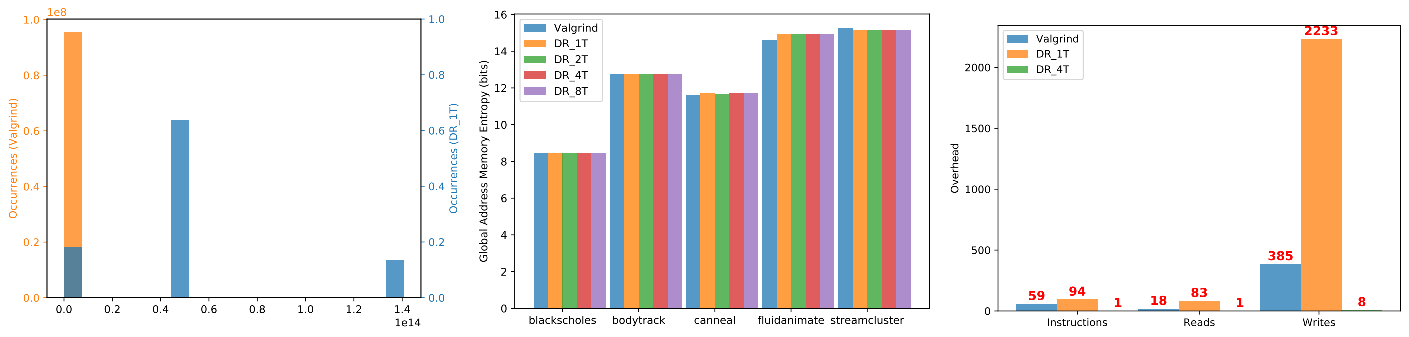
A subset of benchmarks from the Parsec-3.0 benchmark suite—*blackscholes*, *bodytrack*, *canneal*, *fluidanimate*, and *streamcluster*—is used throughout the evaluation [39]. The benchmark subset was configured to run with 4-threads and simsmall, which was found to be representative of the event behavior in larger configurations. The subset was chosen to present a mix of event generation and event analysis behavior in the form of serialization, parallelization, synchronization contention, and compute. The DynamoRIO client is executed four times, with different levels of configured serialization to allow for comparison with Valgrind's serialized instrumentation. DR_1T, DR_2T, DR_4T, and DR_8T represent workloads generated by DynamoRIO being serialized to 1, 2, 4, and 8 threads respectively, as described in Section V-C2. Table IV lists the platforms used in this evaluation. All DynamoRIO and Valgrind experiments were performed on the Xeon E5-2470, which supports mapping each workload analysis thread to a physical core, while perf experiments were performed on the Core i7-6700K machine, as the Xeon server did not support IPT.

TABLE IV: Evaluation Platform

Number of Cores	20	4
CPU Model	2x Intel Xeon E5-2470 v2	Intel Core i7-6700K
Memory Size	64 GB	64 GB
Operating System	64-bit CentOS7 Linux 3.10	Ubuntu 16.10 Linux 4.8

A. Event Generation Parity

In this section, PRISM is used to compare workloads generated between Valgrind, DynamoRIO, and PerfPT frontends, without having to manually instrument or modify each individually. A simple backend tool was created to aggregate event counts, and each event generation source was modularly inserted as a frontend. Only minor disparities were found between each frontend, making platform support, application support, and scalability the primary reason to choose one frontend over another. When Valgrind and DynamoRIO were compared, load, store, and instruction events reported an average error of 1.08%, 10.5%, and 0.7%, respectively. When Valgrind and PerfPT were compared, load, store and instruction events reported an average error of 9.29%, 31.05%, and 9.3%, respectively. Larger discrepancies were expected with PerfPT traces because they were collected on a different platform than the Valgrind and DynamoRIO traces, as listed in Table IV. Discrepancies in stores were attributed to differences in event translation. For example, string, conditional load/store, and certain atomic instructions were handled differently between frontends, further highlighting the importance of expertise and separation of duties in event generation.



(a) Virtual Memory Address Space (b) Global Memory Address Entropy Size (c) Overhead with Parsec Spinlock

Fig. 4: Characterization of Frontend Workload Generation

Global and local memory entropy of all generated addresses were additionally measured for Valgrind and DynamoRIO to ensure reasonable similarity in the memory signatures of the generated workloads. Global memory address entropy represents the temporal locality of memory accesses, computed in terms of bits of information, while local memory address entropy represent spatial locality, i.e. bits required to characterize how often memory is accessed within regions [8]. The address space layout of applications profiled with dynamic binary instrumentation tools may differ depending on the profiling framework because the framework is responsibility for loading the binary and any shared libraries. This is shown in Figure 4a, where DynamoRIO notably uses a larger address range. Zooming into the address range used by Valgrind shows a similar separation of data, heap, and stack, albeit within a smaller range. As these ISA- and platform-independent metrics analyze the locality of the access pattern, the expectation is such that, while the memory access patterns are largely varied in DynamoRIO and Valgrind, the inherent memory entropy of the application should be approximately the equivalent across frontend tools. Figure 4b represents the bits of information required to measure the global memory entropy of five Parsec benchmarks. Local memory entropy also produced similar findings between frontends. As such, analyses that depend on memory traces would not be skewed between either tool. DynamoRIO and Valgrind compute almost equivalent values for global and local memory entropy, with an average difference between Valgrind and DynamoRIO of 0.76% for global memory entropy and 1.6% for local memory entropy.

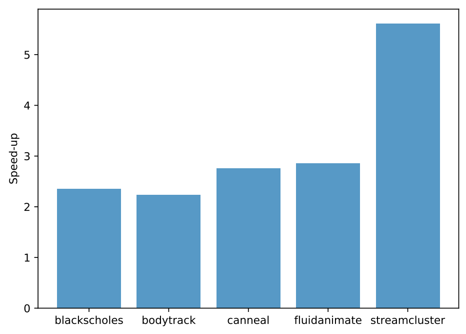
The memory trace logging was written once for both frameworks, does not require intimate knowledge of either framework, does not depend on a specific platform or framework, and can be written with only 5 lines of code. PerfPT was not included in this study because the modified *perf script* tool was not able to dynamically generate addresses.

Synchronization events measured from each frontend were identical, when Parsec was configured to use native pthread libraries. By default, Parsec uses a user-level spinlock implementation for some benchmarks. Figure 4c demonstrates the effect when the user-level spinlock implementation is used for the streamcluster benchmark. Each bar represents the additional events generated by a frontend with an unsupported spinlock barrier implementation compared to a supported synchronization library. The larger the bar, the more divergent the workload is from its ideal representation. For example, Valgrind generates 385x more write events due to its serialization implementation. DR_4T is less affected because thread are natively run and not unnecessarily spinning. In contrast, DR_1T, which serializes the workload, suffers up to four orders of magnitude error over the pthread implementation. This is because its implementation gives each thread a shorter window than Valgrind. This is to reduce skew that may occur from one thread dominating. In this situation, the thread with the lock is overwhelmed by allowing all *other* threads run, and is unable to make sufficient progress. Furthermore, Valgrind and DR_1T experience slowdowns of 29.3x and 63.1x, respectively. These inconsistencies demonstrate the effect of choosing a frontend with inadequate workload support and also the value in modular event generation.

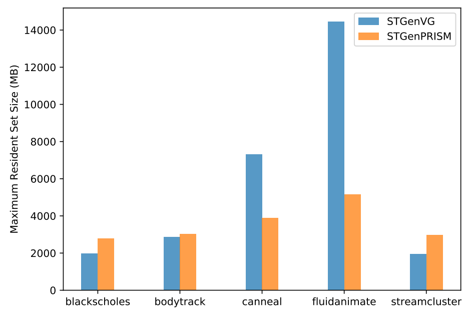
B. *Synchronization-Aware Trace Generation*

In this section, PRISM is used to demonstrate the utility of flexible tool design. Specifically, the complex trace generation for the SynchroTrace simulator is recreated with improvements to speed, memory usage, and storage. SynchroTrace is a trace-driven simulator that targets multithreaded workloads on multi-core architectures [20]. SynchroTrace uses a heavyweight, custom Valgrind tool to generate traces embedded with synchronization and inter-thread dependency information. Compute and memory events are dynamically recorded in a per-thread log. When a dependency is detected between threads, as unique communication edges, a special communication event is inserted. All unique communication edges are tracked with a custom shadow memory [40].

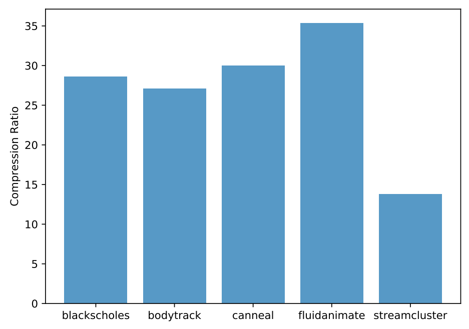
The same trace generation functionality was developed as an event analysis backend, *STGenPRISM*. By separating Valgrind from the event analysis, multiple constraints were removed that allowed for a more powerful trace generation tool. For example, Valgrind does not allow for native tools to use standard or third party libraries, which precludes many salient tool features without substantial development time and



(a) Trace Generation Speed-up



(b) Trace Generation Memory Usage



(c) Trace Size

Fig. 5: Runtime Improvements with Decoupled Analysis

effort. While other frameworks relax some of these constraints, important restrictions still exist. For example, DynamoRIO can support third party libraries and even C++ clients, however creation of native threads in the client is still not allowed. Furthermore, there is the cost of additional research, familiarization, and development for the new framework.

Three new features are enabled in STGenPRISM: 1) asynchronous text logging, 2) binary serialization and 3) zlib compression. Figure 5 shows the improvements of the new trace generation. Figure 5a shows a speed-up of at least 2x for each workload, with a maximum of 5.6x. Speed up is consistently achieved due to asynchronicity between event generation and analysis, and parallelization of the analysis, which is also shown for DynamoRIO and PIN [33], [34]. Variations in speed up is due to differences in the event mix of each workload. Figure 5b shows memory usage of STGenPRISM to be more scalable as well. This is caused by complexities in Valgrind's internal memory allocation [41]. The default allocator used in the GNU C++ Library appears to have more favorable characteristics. The higher base memory usage in STGenPRISM is due to differences in the shadow memory implementation; namely, a larger target address space is supported, and multiple reader-edges are supported for each memory address. Finally, Figure 5c shows an average compression ratio of 26.95, or an average of 95.9% intermediate storage space saved, by enabling open-source serialization and compression libraries. These optimizations were implemented with minimal effort because of flexible PRISM tool development.

C. Dynamic Loop-Detection and Dependency Graphs

In this section, a new analysis backend, to explore dynamic loop and dependency detection, is easily prototyped to demonstrate productivity. Sometimes the compiler misses opportunities for vectorization due to conservative analysis or behavior that depends on run time information, such as potential pointer aliasing. In such cases, dynamic analysis of applications has been leveraged to characterize the *vectorizability* of an application by constructing data-dependence graphs of loop constructs [42]. This analysis can, generally, be separated into loop detection, data-dependence graph generation, and further data-dependence analyses. Specifically, parts of the zlib library that have recently been the target for recent SIMD acceleration are investigated.

Zlib has recently been the target of vectorization due to its ubiquity. However, official support has been slow, compelling organizations like Google to maintain patched versions. Specifically, string hashing, substring matching, and cyclic redundancy check (CRC) calculation optimizations have been suggested for zlib's compression to the DEFLATE data format [43], [44]. All standard zlib code referenced uses deflate version 1.2.11.

A dynamic loop-detection and data-dependence graph generation tool was developed to compare the official zlib library to the customized version used in the Chromium project. Results of a characterization of loops within *deflate_slow* invocations, the default compression strategy, are shown in Table V. Loops were ranked by both the number of iterations and total instruction count, however this is not a direct proxy for percentage of total time spent in a loop. *% of instructions* is the ratio of instructions measured within the loop, compared the aggregate instruction count in all measured loops. *% of loops* is the ratio of loop iterations, compared the aggregate loop iterations in all measured loops. Loops that were identified as vectorizable also list the stride length found via the generated dependency graph.

A demonstrative application is used, which compresses a stream of bytes from the /dev/urandom pseudo-device for Unix-like systems. To simplify analysis, the application was compiled without optimizations. This helps ensure loop dependencies are captured in memory accesses, as locally-scoped variables are typically written back to the stack with low optimization. Each function in Table V encapsulates a region of zlib's decompression algorithm that has been targeted for optimization. *Slide_hash* is patched in Chromium to use SIMD vectorization. Assembly optimized versions of *longest_match* are provided with the standard zlib distribution and is the subject of still further optimizations. *Pqdownheap* and *bi_reverse* are used in Huffman tree construction. Dynamic Huffman tree construction is known to have performance implications, and is completely discarded in some performance-oriented variants [44]. *Crc_fold_copy* is a new function patched into zlib

TABLE V: Zlib Deflate Loops

Function	FILE:LINE	% of Instrs	% of Total Loops	Stride
slide_hash	deflate.c:210	28.3	8.9	2
slide_hash	deflate.c:217	28.3	8.9	2
deflate_slow	deflate.c:1934	16.8	56.7	N/A
pqdownheap	trees.c:458	9.5	7.6	N/A
longest_match	deflate.c:1283	5.8	2.4	N/A
bi_reverse	trees.c:1163	5.8	1.0	N/A
crc_fold_copy	N/A	0.3	0.4	N/A

that leverages CRC hardware. Enabling these optimizations in the Chromium zlib build elicits a 7.4% speed-up.

The tool uses event primitives to build up the data-dependence graph dynamically. Memory reference metadata, such as data width, is easily leveraged by requesting increased detail in memory events. Function context events trigger a heuristic loop detection that identifies loop iterations by observing instruction context events. Instruction address patterns are monitored and the callstack is tracked to maintain relationships between loop entities and other functions. PRISM already provides support for object-file function contexts, which can be complex to capture and depend on the source language and profiling frontend. Loop contexts in general are also difficult to detect in a program binary due to compiler transformations, but the heuristic makes an effort to detect a single loop header, or entry point, while allowing multiple back edges. Previous works have used basic blocks to detect loops via PIN and LLVM IR via static instrumentation to track both memory and register dependencies for data-dependency analysis [42], [45]. Notably, such instrumentation would enable generation of new *loop context events* in PRISM, further simplifying analysis by enabling the single tool to target higher-level abstractions, freeing up the researcher to focus on substantive workload characterizations. Once again, this characterization was prototyped easily and quickly by choosing and subscribing to salient events and did not require any framework knowledge or modification.

D. Framework Overhead

Finally, this section quantifies the IPC overhead of sending events to an external process. Figure 6 shows the worst case overhead for each frontend and multiple DynamoRIO configurations. On average, this overhead is a 33% slowdown of wall clock execution. A *null* analysis tool is used, which simply throws away events. The run time is compared between 1) the null tool and 2) event generation in each frontend with a stubbed out backend. Valgrind and DynamoRIO are configured to generate instruction addresses, memory addresses, compute IOP/FLOP events, and synchronization events. This overhead is acceptable when compared to development and debug time that would otherwise be required for each framework. PerfPT overhead, measured in the same way, is under 1%, because of the heavyweight decoding step required which dominates event generation time.

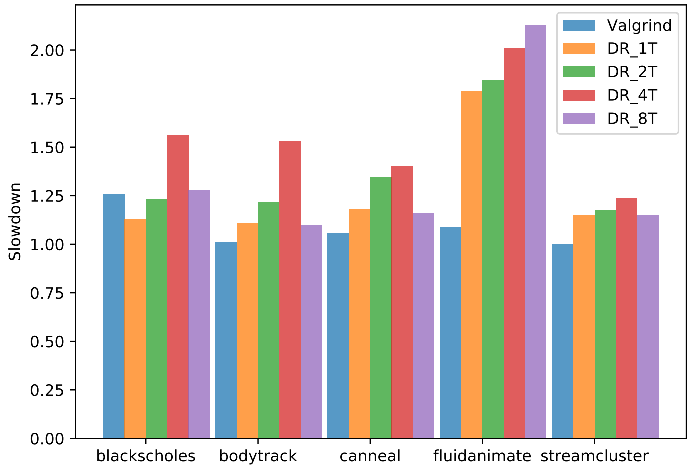


Fig. 6: Worse Case IPC Overhead

VII. Conclusion

In this work, PRISM, a decoupled, cross-framework, cross-platform workload analysis framework was presented. To demonstrate its utility, three analyses were developed independently of the underlying workload profiling framework. These studies highlight the division of labor between event capture and characterization. Normally, both duties are required, but PRISM allows researchers to focus on the important characterization aspects. Analyses using PRISM benefit from a versatile, efficient interface and the ability to intrinsically support a broad spectrum of workloads.

References

[1] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’07. New York, NY, USA: ACM, 2007, pp. 89–100.

[2] D. Bruening, “Efficient, transparent, and comprehensive runtime code manipulation,” Ph.D. dissertation, MIT, September 2004.

[3] C. Gorgovan, A. d’Antras, and M. Luján, “Mambo: A low-overhead dynamic binary modification tool for arm,” *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, pp. 14:1–14:26, Apr. 2016.

[4] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’05. New York, NY, USA: ACM, 2005, pp. 190–200.

[5] M. Kambadur, S. Hong, J. Cabral, H. Patil, C.-K. Luk, S. Sajid, and M. A. Kim, “Fast computational gpu design with gt-pin,” in *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*, ser. IISWC ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 76–86.

[6] A. Srivastava and A. Eustace, “Atom: A system for building customized program analysis tools,” in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI ’94. New York, NY, USA: ACM, 1994, pp. 196–205.

[7] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, “Pebil: Efficient static binary instrumentation for linux,” in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, March 2010, pp. 175–183.

[8] Y. S. Shao and D. Brooks, “Isa-independent workload characterization and its implications for specialized architectures,” in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2013, pp. 245–255.

[9] B. P. Railing, E. R. Hein, and T. M. Conte, "Contech: Efficiently generating dynamic task graphs for arbitrary parallel programs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 2, July 2015.

[10] Y.-H. Lyu, D.-Y. Hong, T.-Y. Wu, J.-J. Wu, W.-C. Hsu, P. Liu, and P.-C. Yew, "Dbill: An efficient and retargetable dynamic binary instrumentation framework using llvm backend," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '14. New York, NY, USA: ACM, 2014, pp. 141–152.

[11] N. C. Doyle, E. Matthews, G. Holland, A. Fedorova, and L. Shannon, "Performance impacts and limitations of hardware memory access trace collection," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 506–511.

[12] T. Janjusic and K. Kavi, "Gleipnir: A memory profiling and tracing tool," *SIGARCH Comput. Archit. News*, vol. 41, no. 4, pp. 8–12, Dec. 2013.

[13] N. Nethercote, "Dynamic binary analysis and instrumentation," Ph.D. dissertation, University of Cambridge, November 2004.

[14] J. Edler and M. D. Hill. Dinero iv trace-driven uniprocessor cache simulator. University of Wisconsin. [Online]. Available: http://pages.cs.wisc.edu/ markhill/DineroIV/

[15] J. Weidendorfer, M. Kowarschik, and C. Trinitis, "A tool suite for simulation based analysis of memory access behavior," in *ICCS 2004: 4th International Conference on Computational Science*. Springer, 2004, pp. 440–447.

[16] M. Chabbi, X. Liu, and J. Mellor-Crummey, "Call paths for pin tools," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 76:76–76:86.

[17] 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, pp. 196–206.

[18] S. Kumar, W. N. Sumner, and A. Shriraman, "Spec-ax and parsec-ax: extracting accelerator benchmarks from microprocessor benchmarks," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sept 2016, pp. 1–11.

[19] J. Li, Z. Wang, C. Wu, W.-C. Hsu, and D. Xu, "Dynamic and adaptive calling context encoding," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 120:120–120:131.

[20] S. Nilakantan, K. Sangaiah, A. More, G. Salvadory, B. Taskin, and M. Hempstead, "Synchrotrace: Synchronization-aware architecture-agnostic traces for light-weight multicore simulation," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, Philadelphia, PA, March 2015, pp. 278–287.

[21] R. Jagtap, S. Diestelhorst, A. Hansson, M. Jung, and N. When, "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.

[22] Prism. Drexel VLSI and Architecture Lab (VANDAL). [Online]. Available: https://github.com/VANDAL/prism

[23] U. Milic, A. Rico, P. Carpenter, and A. Ramirez, "Sharing the instruction cache among lean cores on an asymmetric cmp for hpc applications," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 3–12.

[24] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieres, and M. Rosenblum, "Towards practical default-on multi-core record/replay," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 693–708.

[25] T. Zhang, C. Jung, and D. Lee, "Prorace: Practical data race detection for production use," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 149–162.

[26] P. Roy and X. Liu, "Structslim: A lightweight profiler to guide structure splitting," in *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, March 2016, pp. 36–46.

[27] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and efficient cfi enforcement with intel processor trace," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 529–540.

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

[29] M. T. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator," in *2007 IEEE International Symposium on Performance Analysis of Systems Software*, April 2007, pp. 23–34.

[30] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: A full system simulator for multicore x86 cpus," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 1050–1055.

[31] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '03. New York, NY, USA: ACM, 2003, pp. 318–319.

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

[33] Q. Zhao, I. Cutcutache, and W.-F. Wong, "Pipa: Pipelined profiling and analysis on multi-core systems," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '08. New York, NY, USA: ACM, 2008, pp. 185–194.

[34] D. Upton, K. Hazelwood, R. Cohn, and G. Lueck, "Improving instrumentation speed via buffering," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09. New York, NY, USA: ACM, 2009, pp. 52–61.

[35] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling dynamic program analysis from execution in virtual environments," in *USENIX 2008 Annual Technical Conference*, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404014.1404015

[36] OpenTracing. [Online]. Available: http://opentracing.io

[37] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[38] K. Kirchner and S. Rosenthaler, "Bin2llvm: Analysis of binary programs using llvm intermediate representation," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ser. ARES '17. New York, NY, USA: ACM, 2017, pp. 45:1–45:7.

[39] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[40] 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*, ser. VEE '07. New York, NY, USA: ACM, 2007, pp. 65–74.

[41] P. Waroquiers and S. Nilakantan, "Custom modified callgrind tool uses too much memory," Valgrind Developers Mailing List. [Online]. Available: http://valgrind.10908.n7.nabble.com/Custom-modified-Callgrind-tool-uses-too-much-memory-td49927.html

[42] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Dynamic trace-based analysis of vectorization potential of applications," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 371–382.

[43] L. P. Deutsch, "Deflate compressed data format specification," IETF Requests for Comments, Tech. Rep., May 1996. [Online]. Available: https://www.ietf.org/rfc/rfc1951.txt

[44] J. T. Kukunas, V. Gopal, J. Guilford, S. Gulley, A. van de Ven, and W. Feghali, "High performance zlib compression on intel architecture processors," Tech. Rep., April 2014. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/zlib-compression-whitepaper-copy.pdf

[45] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri, "Loopprof: Dynamic techniques for loop detection and profiling," 2006.