Simutrace: A Toolkit for Full System Memory Tracing

Marc Rittinghaus Thorsten Groeninger Frank Bellosa
Operating Systems Group
Karlsruhe Institute of Technology (KIT)
os@itec.kit.edu

ABSTRACT

The ongoing increase in software and hardware complexity poses a challenge for researchers, system architects and software developers who need to understand a system's runtime behavior. Traces of executed code paths and memory references as well as interesting events for example in the area of resource management are frequently used to support development and debugging as they provide a valuable insight into the execution. Memory traces place an extraordinary demand on the tracing components due to the fact that the rate at which the processor accesses main memory is inherently higher than the rate of function calls, MPI messages or other system events. While many tracing frameworks for memory traces have been proposed in the past, a major limitation of these frameworks is their restriction to track only selected processes and their inability to monitor privileged system components. Profiles generated with these tools therefore do not encompass memory references performed by the operating system (OS), system daemons or (kernel-mode) drivers. In this white paper, we present **Simutrace**, a tracing framework for efficient full system memory tracing. Simutrace captures memory accesses at the hardware level, using functional full system simulation for holistic memory traces. Simutrace incorporates an aggressive but fast compressor, making full length, no-loss memory traces of long-running workloads possible.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Simulation*

Keywords

Full System Simulation, Memory Tracing, Operating Systems, Performance Analysis

1. INTRODUCTION

Traces are frequently used in the development and evaluation of software components and operating systems. During execution, events and state information of interest are captured in traces for later offline analysis. Traces can provide valuable insight into the dynamic behavior of a software and deliver empirical support to focus optimization and debugging efforts. Depending on the intended use of a trace, the type and number of collected events and properties vary. Prominent types of traces are call graphs [28], which retain information about a program's executed function hierarchy, or domain specific traces such as communication profiles in MPI applications [23, 35]. *Memory traces*, that is recordings of a processor's memory accesses, leap out from these types of traces. While they proved to be very effective for driving memory hierarchy simulations [15, 17, 22, 26, 30, 31, 48, 50] or gathering statistics about an application's memory access patterns [37, 51], memory traces pose an extraordinary demand on the tracing components. This is due to the fact that the rate at which new events are generated—i.e., the rate at which the processor accesses main memory—is inherently higher than the rate of function calls, MPI messages or other system events. Memory traces thus quickly become very large in size, consuming gigabytes of storage. A Linux virtual machine (VM) running a minimal Linux kernel build for instance produces approximately 145 billion trace entries. The same system completing the Postmark benchmark from Phoronix Test Suite [5] even generates around 175 billion write events alone. Memory traces therefore heavily depend on an efficient encoding, a scalable trace format and a tracing mechanism that is capable of dealing with a high rate of incoming events.

Over the years, various memory tracing frameworks have been developed [8, 15, 31, 39, 46, 54]. A major limitation of these frameworks is however their restriction to track only selected processes and their inability to monitor privileged system components. Profiles generated with these tools therefore do not encompass memory references performed by the operating system (OS), system daemons or (kernel-mode) drivers. That raises questions concerning the accuracy of results obtained through such narrow traces as the interaction of tracked processes and the system is completely left out [13]. Further distortions can originate from the incurred slowdown through instrumentation and tracing, which influences the relative timing between processes and the system. Tracing tools able to capture events in the OS kernel [6, 19, 52] on the other hand do not offer memory tracing capabilities. These limitations make current tracing frameworks not applicable to memory centric *operating system research.*

In this white paper, we present **Simutrace**, a novel tracing toolkit, which has been conceived with full length, no-loss memory tracing in mind. Simutrace captures memory accesses at the hardware level using *functional full system simulation*, thus including all user-space programs, the operating system, drivers and direct memory access (DMA) operations. Through the use of full system simulation, Simutrace fully supports tracing of just-in-time (JIT) as well as self-modifying code and requires no special modification or preparation of the target system. Tracing is non-intrusive and free of any side-effects, as the simulation preserves the timing between running workloads and the underlying OS.

Another difference to existing solutions lies in the choice of recorded data. To be conducted, some analyses in the field of operating system research (e.g., a study on the characteristics of redundant memory pages to improve memory deduplication mechanisms [21, 33, 40]) require knowledge about the content of memory pages. Existing solutions typically constrain traces to the referenced virtual or physical addresses. Simutrace instead also tracks for each write operation the actual written data. That enables analysis tools to reconstruct the content of the examined system's physical memory for any given point in time within the trace interval.

Although Simutrace is conceived with memory tracing in mind, its design is not restricted to this type of data. Instead the architecture has been kept as general, flexible and modular as possible to allow Simutrace to adapt to new tracing scenarios and interface with different full system simulators. We leveraged this ability in an evaluation of page usage characteristics, in that additional OS introspection events have been captured and included in the traces. That allowed us correlating page usage to processes and threads. In this white paper, we will therefore frequently refer to introspection information as a valuable source of trace events to supplement memory traces and describe where including such data in traces resulted in certain design decisions.

The remainder of this white paper is organized as follows: In Section 2, we present Simutrace, describe its design and explain the concepts behind key components. In Section 3, we provide an overview of Simutrace's native trace format. Related work is summarized in Section 4. We finally conclude and give a prospect on future plans in Section 5.

2. SIMUTRACE

The primary tasks of a tracing framework are the *collection*, *reduction* and *storage* of traces and the ability to provide access to the trace data for analysis [22]. In the literature, various criteria have been suggested to further qualify these tasks [48]:

- **Flexibility**: The framework should not restrict the type of data that it is able to capture.
- **Speed**: The slowdown through tracing should be low.
- **Accuracy**: The collected trace should be free of any distortions, it should be complete and include the information of interest in full detail.
- **Portability**: The tracing framework should be runnable on different host architectures and platforms and easily interface with different simulators.
- **Ease-of-Use**: The amount of effort required by the end-user to use the framework should be low. A concise interface is a prerequisite for that.
- **Expense**: The cost of any hardware or software required solely for the purpose of collecting traces should be low.

Simutrace has been designed with these criteria in mind to address the shortcomings for operating system research of previous approaches. In the next sections we highlight, in what decisions these metrics have manifested. Although Simutrace is capable of tracing arbitrary events to fulfill the *flexibility* criterion, we focus in this white paper on memory tracing due to its particularly high demands on a tracing framework. We start by explaining the general architecture laid out for Simutrace. We continue by illustrating how trace data in Simutrace is logically organized and how events are collected, submitted and eventually reduced. We close Section 2 with an overview on multi-threaded tracing.

2.1 General Architecture

Existing tracing frameworks have been based on various designs. SIGMA [15] encapsulates all tracing logic in a dedicated library and links it into the application that is to be examined. A similar approach is taken by METRIC [31]. The authors of ScalaMemTrace [8], a recent work in the area of application memory tracing, opted for a more modular design. While they implemented the collection of events in a library, they moved the compression and storage of the gathered data into a separate process. In Simutrace, we have adopted this approach and extended it to a full client-server architecture as found in many database applications. The general design of Simutrace is depicted in Figure 1. A tracing extension in a full system simulator collects

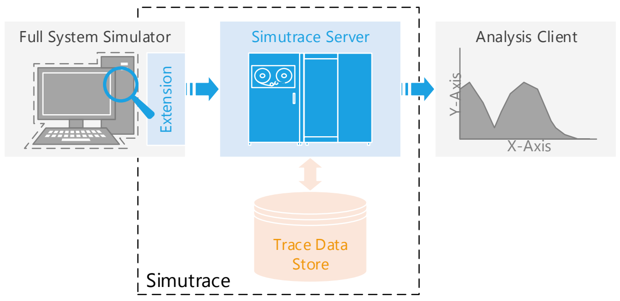

Figure 1: Basic architecture. An extension in a full system simulator collects events. The Simutrace server receives the data, stores it and provides access for later analysis and inspection.

information on memory accesses (and other events) in the simulated machine–the *target*–and transfers these information to the Simutrace *server*. A simulator as well as peers connecting to the server to analyze recorded data, thus are Simutrace *clients*. The server is the central component in Simutrace, managing the storage of and access to trace data by receiving and presenting traces from and to clients. In that role, it is also responsible for processing traces (e.g., compression/decompression) before they are written to or read from persistent storage. The server thus carries out the most compute and memory intensive operations in the course of memory tracing. After trace reduction, analysis clients can connect to the server and read the whole or only specific time spans of the trace, filtered by the type of event (e.g., only memory writes).

This design has several advantages over a pure library-based approach. From a software engineering perspective, strictly separating the trace processing and storage management from the collection facilitates development and testing. It also increases the *portability* of the tracing framework and

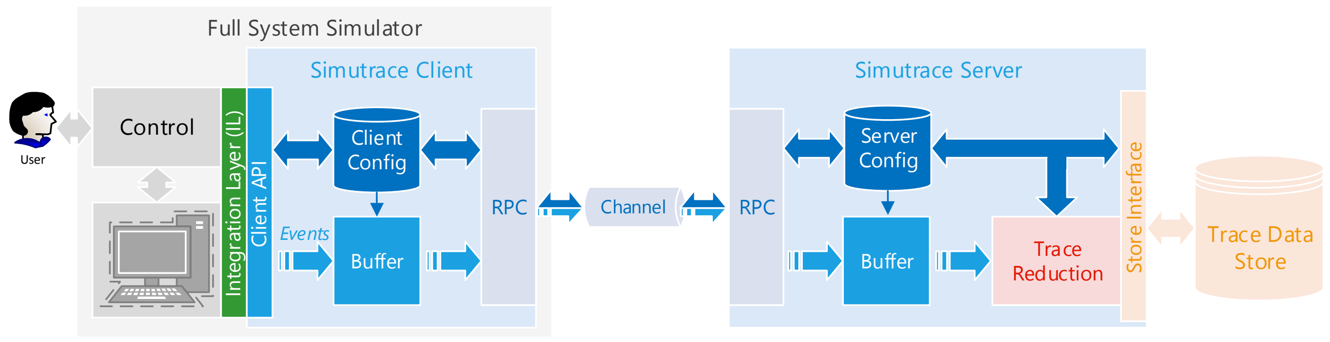

Figure 2: Tracing with Simutrace. The client runs within the simulator and stores events received by the VM in a buffer. On repletion, the buffer is transferred to the server for processing (e.g., compression). The server finally saves the reduced data in a store (e.g., trace file). Configurations regarding the organization of traced data are set by the simulator and are mirrored between the client and server.

eases interfacing different simulators. From a user's perspective, another advantage comes to play in *multi-tracing*. Since most simulations are single-threaded (even when simulating a multi-core machine) and take considerable time to complete, it is common to leverage the physical parallelism of today's hosts by running multiple independent simulations in parallel (e.g., to examine different workloads or configurations). A single server process with a *shared* worker pool and a *centralized* job management can efficiently use the remaining host resources without requiring any balancing or synchronization between multiple competing trace processors.

Figure 2 illustrates the design of Simutrace's client and server components in more detail:

Client Library and Configuration. The trace collection is performed in the *client library*, which is loaded into a full system simulator. The interface of the library has been designed with a focus on *ease-of-use*. It encompasses around 20 methods to manage the connection with the server, create or open traces, configure the tracing session and submit or retrieve individual trace events. Before a client can start tracing, it has to connect to a storage server, create a new tracing session and specify the organization and type of data that will be submitted as well as the type and location of persistent storage that should be used. The client library then propagates the configuration to the server.

Integration Layer. The simulator needs to be extended with a thin *integration layer* (IL). This layer comprises hooks, which communicate information on events in the simulation to Simutrace. To implement the integration layer the full source code of the simulator is usually not required as most closed-source simulators already provide methods to hook into operations carried out by the simulation (e.g., memory accesses, instruction fetches, etc.). The integration layer then only needs to generate a trace event from the information supplied by the simulator and write the event to a specifically allocated buffer. On repletion, the client library sends the buffer to the server for asynchronous processing and storage.

Communication. While in a pure library-based design, communication between components of a tracer works on the level of function calls and shared memory, transferring data across process boundaries requires an explicit mechanism. In Simutrace, the client and server follow a uni-directional *remote procedure call* (RPC) model, where the client invokes methods in the server and calls can carry arbitrary payloads. To mitigate the overhead caused by the communication, Simutrace dynamically selects the communication channel, based on the platform and the location of the server process:

Local. Usually the server process runs on the same machine as the client. In that case, a local pipe is used to perform RPC calls. Although pipes already provide decent performance, they still incur extra copies to transport data. That is especially undesired when transferring large payloads such as a buffer of collected trace entries. Simutrace therefore utilizes shared memory for all buffers containing trace data. This way, only information on *where* in the buffer new data resides needs to travel the channel and the server can access the trace data through its mapping directly, thus eliminating all copies.

Remote. In some scenarios, for example if a single consistent trace file is desired in a distributed simulation or if a trace should be analyzed on a dedicated machine while it is still being recorded, it is favorable to run the server on a different host than the client. In these cases, Simutrace establishes a socket-based TCP/IPv4 or IPv6 communication channel. The client and server then copy any payloads to and from the socket implementation and explicitly transfer them over the channel. We are currently investigating to integrate support for *remote direct memory access* (RDMA) found in modern network adapters. With RDMA, trace buffers could be directly transferred into the main memory of the peer, thereby saving any local copies.

Trace Processing and Storage. When the client successfully submitted trace data, the server asynchronously processes it through a worker pool. The number of workers is determined at the start of the server and is fixed throughout its runtime. If the user did not supply any custom setting, the server creates one thread per logical CPU. The operations executed as part of the trace processing are defined through the storage format and depend on the type of events submitted. They may range from a simple generic compression, over data re-arrangement and information extraction to facilitate analysis, to optimized event compression schemes.

In any case, the trace data is eventually written onto persistent storage. Simutrace comes with a storage provider for a custom format that we developed to fully leverage the features of our software and which is particularly suited for traces with a high amount of entries. The provider saves the trace to disk as a regular file, which can optionally be moved or copied to other systems for inspection.

Accessing Traces. The steps involved in accessing traces are almost identical to the ones taken to create the trace. An analysis client makes use of the same library to connect to the server, but *opens* instead of creates a trace. The server then reads the trace's organization and replicates it to the client. When the client requests certain time spans, the server decompresses the data and places it in the same (shared) buffer that is utilized during recording. To keep the latency low the server integrates caching and pre-fetching of trace segments. However, since the size of traces is usually significantly larger than the available system memory, only short durations of the trace can be present in memory at any time. The client needs to be designed with this constraint in mind.

2.2 Trace Organization

The *flexibility* criterion suggested by Uhlig et al. requires a tracing framework to place no restriction on the type of events recorded [48]. That, however, introduces a new dimension of complexity compared to a design that has to deal with a flow of uniform data only.

For each simulated event of interest, Simutrace has to record the properties relevant for the anticipated analysis. For a memory write issued by a simulated CPU, these might be the (physical) address to which the operation was targeted at, the size of the data written and potentially the data itself. One use-case of the suggested type flexibility is to supplement the recordings of memory writes with semantic information from an operating system (OS) introspection layer. Such a layer would for instance generate events on process creation and context switches to allow correlating memory operations to processes. Naturally, the properties to record for such events will greatly differ from the ones stored for memory writes. In consequence, events of different types usually are of different size. That quickly leads to multiple inefficiencies, when not considered early in the design of a tracing framework:

- **Accessibility**: Trace events are typically recorded in chronological order by simply appending entries to the end of the already written trace. If the type and therefore the size of each event may vary, addressing or seeking to a certain entry becomes difficult. A reader has to scan through all previous entries to find the right offset in the trace, degenerating the access to a costly $O(n)$ operation.

- **Storage Efficiency**: The effectiveness of compression schemes applied to the trace depends on the actual distribution of event types in the recorded flow. This is rooted in the fact that most general purpose compression algorithms (e.g., the LZ77-family [55]), although able to work on arbitrary data and thus on mixed traces, work with a sliding window in which the schemes try to identify repeating patterns. The compression ratio, however, is likely to decrease when the variability of entry types (i.e., data layouts) within the sliding window increases. Coping with different types further complicates the application of custom compression methods that are specialized to deal with a certain type of event.

- **Locality**: When examining a trace, it is often desired to only inspect events of a certain type at a time. As with the storage efficiency, the degree of locality for a certain entry type depends on the actual collected data and the ratio at which entries for the involved types are generated. The locality for a certain type is low, if the relative amount of events recorded for this type is low and the entries are regularly distributed over the full length of the trace. Such a characteristic applies for instance to introspection events in a memory trace. Accessing such events becomes a time-consuming operation as an unknown number of other entries have to be skipped.

To mitigate these effects, Simutrace introduces the concept of *streams*. Streams group semantically connected events and present these as an independent flow of entries, while each stream is restricted to contain data of a *single* type only. Additionally, every stream is stored in its own region in memory and on disk. When utilizing streams as abstraction, each source of events in the simulation such as a memory access hook creates one or more dedicated streams to which it submits entries. Figure 3 illustrates the approach.

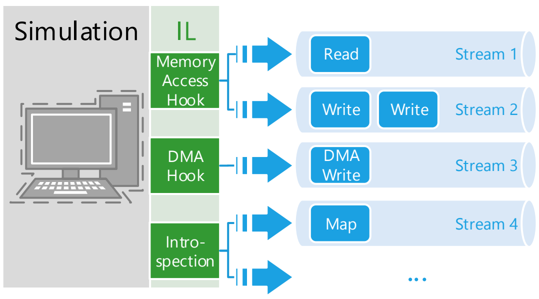

Figure 3: Trace organization. Each event is assigned to a stream of semantically connected entries. A stream is limited to a single type.

Separating event records of different types logically and spatially avoids all of the aforementioned inefficiencies. Due to the restriction to a single type per stream, all events within a stream have a fixed size, enabling a reader to address an entry by index in $O(1)$. Type information per entry can be omitted, saving storage space and avoiding interpretation.

The single-type semantic of streams makes it easy for the server to apply type-specific processing, for instance specialized compression schemes. To that end, the storage format can associate streams of a certain type to an *encoder*, which the server will use to process the streams' data. The logic of the encoder is supplied by the implementation of the storage format and invoked by the server in the context of a worker thread. To define entry types, Simutrace employs 128 bit globally unique identifiers (GUIDs). Accordingly, when a client registers a new stream, it has to supply the GUID of the desired stream's data type.

Variable-sized Entries. While a fixed size for entry types improves data accessibility, it also limits what data can be stored in a trace. In general, the approach forbids any data to be traced whose size is not known in advance and at the same time is too variably sized to just reserve space in an entry without wasting space in memory and on disk as well as producing superfluous processing overhead. Taking introspection events as an example, any strings such as the environment or command line of a process fall into this category. Another example are recordings of the simulation's screen output. The resolution might change (e.g., during the boot phase) and is not necessarily known when designing the tracing hook in the simulator. To solve this conflict without compromising accessibility, Simutrace utilizes a combination of referencing and specific encoding for variable-sized data.

An entry as a whole has a variable size, if one or more fields in the entry (i.e., properties of the event) are of variable size. The basic idea in Simutrace is thus to move any variable-sized data to a dedicated stream and instead store a fixed-size reference to the new location in the entry. That recovers accessibility for the stream containing the recorded events and which should provide optimal access characteristics.

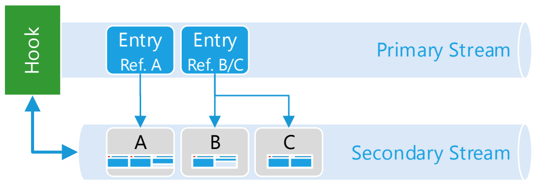

Figure 4: Handling of variable-sized data. When tracing, the hook writes variable-sized data fields to a separate stream, storing only fixed-size references in the actual entry.

However, it also breaks the accessibility for the *secondary stream*, which holds the variable-sized data. Moreover, without using the references in the primary stream to identify structure, the secondary stream is not even readable. Although in practice, this would not be a problem, because the data in the secondary stream can be accessed with a byte-offset as reference in $O(1)$ and the secondary stream does not need to be read by its own, we extended the concept to retain independent readability for the secondary stream.

To that end, Simutrace utilizes a special encoding in the secondary stream (see Figure 5). The data is segmented into equally sized blocks. Each block is preceded by a marker that indicates if the subsequent entry is a continuation of the current one, and a length field, which specifies how many bytes in the block hold valid content. Any spare room in a block is padded using zero-bytes. From the perspective of the stream, each block represents an independent entry with a fixed size. Accordingly, the reference in the original entry points to the corresponding data's first entry in the secondary stream. Although a reader has to interpret

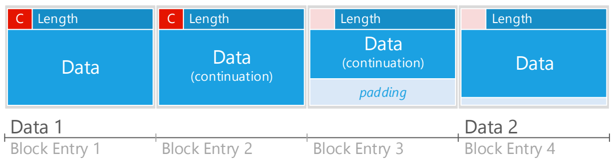

Figure 5: Encoding of variable-sized data through chunking and padding. Continuation is specifically marked (here in red).

the marker and length fields to retrieve the original data (if it spans multiple blocks), splitting the data into blocks preserves the stream semantic—i.e., streams only contain entries of equal size, giving architectural consistency across all types of streams. The optimal block size depends on the length of the expected data, where a large input prefers bigger blocks and vice versa.

2.3 Buffer Management

To be able to submit entries to a stream each stream needs to be backed by a buffer, which on repletion is send to the server for processing and storage. Since the server usually runs on the same machine as the client, the buffer can be shared between the client and server via shared memory to avoid expensive transfers. In consequence, the client must not overwrite the buffer of a stream, that is submit new entries, until the server finishes processing the buffer. That restriction, however, causes the simulation to stall if new events for the same stream are generated during the phase of processing. This is usually the case for memory tracing.

To comply with the *speed* criterion, a memory tracer should provide an asynchronous processing model, which allows the client submitting entries during processing. That can be accomplished by swapping a stream's underlying buffer. Considering that a tracing session including supplementing introspection information can easily encompass over 30 streams, a certain degree of buffer management is required.

Figure 6 illustrates the buffer allocation in Simutrace. The client and server create a (shared) *stream buffer*, which is divided into equally sized *segments*. These segments are uti-

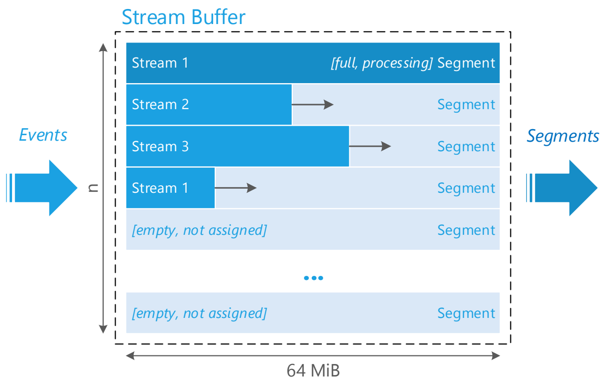

Figure 6: Stream buffer. Each stream allocates a segment in a stream buffer to store incoming trace entries. If a segment is full, it is send to the server for asynchronous processing. To receive further events, the stream allocates a new segment.

lized to back streams. When the client tries to write to a stream for the first time, Simutrace transparently allocates a segment from the stream buffer and assigns it to the stream. The attempted write can then be completed, while any subsequent writes within the segment directly succeed. When a segment has been fully consumed, the client sends it to the server and swaps it with a newly allocated one, thereby allowing new events to be generated and submitted to the stream. When the server finishes processing a segment, it releases the segment, thereby making it eligible to allocation again.

During the entire tracing session, the server stays in control of the allocation of segments. Although that introduces additional communication overhead, it keeps memory management tied at one place. In practice, extra overhead is avoided by combining the submission of a full segment and the request of a new one into a single RPC call.

Another factor determining the communication overhead is the size of the segments in the stream buffer. Larger segments can accommodate more events until they have to be replaced and thus reduce the frequency at which the server needs to be contacted for memory allocation and processing. Furthermore, as the processing—in most cases compression—of traced events is done at the granularity of segments, the segment size also sets the bounds for the effectiveness of most general purpose compression schemes. Larger segments, which allow a larger sliding window, usually tend to give a higher compression ratio. However, large segments also have their shortcomings. Since Simutrace allocates at the granularity of entire segments, large segments come with an increased memory consumption. That in turn may limit the number of streams that a system is capable of serving. The optimal configuration for the stream buffer in terms of segment number and size thus depends on the available system resources and the anticipated number of streams. We empirically found 64 MiB to provide a good balance between compression effectiveness and memory consumption.

2.4 Multi-Threaded Tracing

A fundamental challenge for full system simulators is the slowdown incurred due to the extensive emulation and the inspection functionality provided by such simulators. To facilitate the emulation and to offer fully deterministic execution, system simulation is usually performed in a single thread, which emulates each core in a round-robin fashion. The slowdown for multi-core simulations thus increases linearly with the number of simulated cores. Recent work in the field of full system simulation has shown that a viable solution to accelerate multi-core simulation is to run each virtual CPU core on a dedicated hardware thread [16, 29, 49]. While achieving good speedups (3.8x for a quad-core ARM simulation [16]), the approach creates at the same time new requirements for tracing frameworks.

When multiple CPUs are present in the simulated system, one often wants to correlate memory accesses with the CPU that invoked them. A possible solution to track this relationship is to express it through the *structure* of the trace, that is separating memory access traces on a per-core basis. Simutrace supports this approach through the use of one or more dedicated streams per CPU. Moreover, the API exposed by Simutrace is fully thread-safe, thereby allowing multiple threads to trace events simultaneously, that is allocate and submit stream segments. Another solution is to create only a single stream, but include the originating device in the trace entry (e.g., through a CPU number). In this configuration, the threads driving the simulation of the virtual CPUs write to the *same* stream in parallel. Simutrace supports thread-safe writes to a single stream with a special set of tracing functions, which utilize atomic operations to synchronize writes to the backing segment. However, such a setup should be used with care. That is grounded in the fact that frequent writes by multiple CPUs within the bounds of the same cache line (typically 64 bytes) are likely to thrash caches. Moreover, the interlocked operations required for synchronization are expensive on today's CPUs.

In Simutrace, to enable multiple threads in a tracing session to safely participate in the recording of events, each thread has to explicitly connect to the session. The benefit of this semantic is that the client and server can establish a dedicated RPC channel per thread. For each channel in turn the server creates a new worker thread, which executes RPC requests in behalf of the respective client thread.

2.5 Conclusion

The criteria proposed by Uhlig et al. have manifested in the design of Simutrace in multiple ways. Simutrace follows a client-server design that allows efficient trace processing in multi-tracing scenarios and eases the porting to new platforms and simulators through modularization and functional separation. A client—i.e., a full system simulator when tracing or an analysis software when inspecting previously written traces—utilizes an easy-to-use library to manage sessions with the server, configure or retrieve the structure of a trace and submit or read recorded events (e.g., memory accesses).

Traces in Simutrace are organized through streams that separate events according to their semantic background and type. This concept provides a high degree of flexibility and allows Simutrace to maintain fast $O(1)$ (random) access on read, without placing any restrictions on the type of events captured in a trace. Moreover, the support for variable-sized entries makes it possible to efficiently trace data whose size is not known in advance without wasting storage space and processing capacity. To prepare for emerging multi-core parallel simulators, Simutrace is fully capable of multi-threaded tracing.

As speed is one of the most important criteria for a tracer, segment buffers keep the overhead for submitting events to a stream at the cost of filling a data structure in memory. Processing tasks such as the compression of recorded events are done asynchronously by the server, utilizing the full hardware parallelism available in the host. For local tracing sessions, the use of shared memory, pipes and a concise RPC interface keeps the overhead for communication with the server low.

3. SIMUTRACE STORAGE FORMAT

When a tracer receives new events it eventually has to write them to persistent storage as the amount of data is usually too high to keep in memory. Moreover, traces are often inspected over a long period of time (e.g., weeks), regularly revisited as research progresses and new questions arise. That, too, makes holding traces in memory unfeasible. To store a trace on disk, every tracing framework has to either support an existing trace format, or define its own format. Which format a framework uses heavily depends on the type and amount of data it has to cope with as most trace formats are optimized for a certain scenario and may not satisfy the specific requirements. In that sense, a format developed to encode message passing interface (MPI) packets is typically unsuited to store memory traces and vice versa.

Simutrace uses a modular storage approach, which allows both solutions—i.e., integrating existing formats or adding custom ones—to be taken. To that end, Simutrace defines an interface that a format implementation has to adhere to. The user can then choose the format and storage location with the help of a *storage specifier* in the form `format:path` as part of the tracing session's configuration. In the current version, Simutrace includes a flexible custom format, named **simtrace** after the 8-byte magic in its header, which we developed to support all features exposed by Simutrace. When designing the format we laid out the following criteria:

- **Flexibility**: The storage format should not restrict the type of data that it is able to store.
- **Scalability**: Traces should be able to grow as desired. In particular, a trace should be partially readable and the format should allow fast seeking to a specified time span or a certain event (e.g., by index or timestamp).
- **Storage Efficiency**: The format should be capable of employing a mixture of generic and specialized compression schemes to achieve high compression ratios.
- **Structuring**: The format should provide means to organize recorded events according to their source, semantic background or type.
- **Compatibility**: Traces are sometimes archived to comprehend results at a later time. The format should be able to evolve while maintaining compatibility with previous versions.

To embrace these criteria we devised a generic trace format that is able to store arbitrary data and types of events. It employs Simutrace's concept of *encoders* in the processing stage to apply generic or specialized compression schemes depending on the entry type reported for a stream. The current version includes a special encoder for memory access traces and defaults to a generic compression for all other data types. As *streams* are Simutrace's native abstraction for trace organization, we adopted this primitive in the storage format. To maintain backward compatibility with previous versions of the format in the future, a trace file includes a version identifier in the header.

In the next sections we describe in more detail what measures are taken in the format to preserve fast access even for traces with hundreds of gigabytes of size. We will further elaborate on the specialized memory encoding integrated in Simutrace's storage format.

3.1 File Organization

Detailed memory traces can easily grow up to hundreds of gigabytes in size, even when compressed. As traces that huge need to be compressed before they are written to disk, working with them requires a preceding decompression; at least from the beginning to the point of interest. In practice, that makes accessing information contained in contiguously compressed traces unfeasible.

Wu et al. proposed a *scalable log file format* (SLOG) for their MPI tracing facility on IBM SP systems [53]. The format is tailored to allow fast random access to trace data. The key idea behind SLOG is to divide the trace data into small chunks and compress these *frames* individually. A *directory* serves as a fast index into frames.

The same concept builds the foundation for the simtrace format. Figure 7 (a) denotes its anatomy. A trace file is a collection of independently readable frames where each frame is owned by a particular stream and holds the data of a single (compressed) 64 MiB stream buffer segment. Due to the multi-threaded processing model, there is no guarantee that the storage layer writes the frames in the correct order. The server therefore assigns each frame a *sequence number* specifying the chronological order in which the client filled the segments. That allows the server to reproduce the original flow. The sequence number is stored in a *header* which the storage layer prepends to each frame. In addition, a directory at the beginning of the trace gives quick access to individual frames without having to discover their positions in the trace file first. The directory can store up to 1024

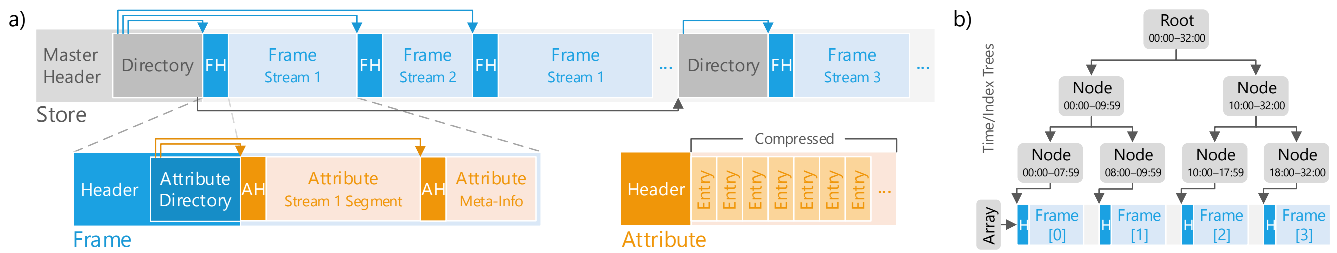

Figure 7: (a) File organization. Trace files are built from variable-sized frames. Each frame is associated with a stream and encompasses a set of attributes, one attribute being the compressed trace entries for the time span covered by the frame. Directories allow fast frame discovery. (b) In-memory directories. Simutrace builds an entry index based, per stream tree directory. Additional time-based indices are constructed for streams with timing information. For fast sequential access, frames are hold in an array.

frame references (worth 64 GiB of trace data). On repletion further directories may be allocated, while for the first 448 directories a direct link is included in the file header. In contrast to the SLOG format, a frame does not only hold actual compressed trace data, but instead comprises a list of attribute-value pairs. Accordingly, the trace data itself is capsuled within a *data attribute*. This flexibility makes it easy to add encoder specific meta-data to a frame or to enrich a frame with additional information (e.g., summary data as generated by some tracers [14]). Simutrace also employs this feature to store a stream's properties such as type and name in a special frame.

3.2 Data Access

Although the directory helps to quickly find the location of frames within a trace file, having to access each frame to retrieve information on the time span it covers would be a costly operation. Especially if the reader wants to jump right to events generated at a certain point in the simulation. Simutrace solves this problem by (a) including information on what time span a frame holds in its header and mirroring the header in the directory and (b) building for each stream a set of in-memory red-black tree based indices to quickly locate the right sequence number.

In addition to various meta-data, each frame header includes the numeric *index* of the first entry, the *number of entries* contained, the start- and end *simulation time* (measured in executed instructions) as well as the start- and end *wall-clock time*. The latter can be useful if one for example knows that a certain phenomenon occurred 30 min after starting the tracing session but has no knowledge on the amount of simulation time passed up to this point. Each one of these *index properties* can be used to address ranges in a trace. Since they are, as part of the header, mirrored in the directory, it is sufficient to read all directories within a trace (each one occupying 128 KiB) to retrieve all relevant range information, thereby keeping the time to open a trace file low. The server can then build the respective index trees. Figure 7 (b) illustrates the in-memory layout of the index structures. An array stores the meta-data for each frame. This information includes all of the aforementioned index properties as well as the position of the frame within the trace file. For each of the index properties (if provided) an interval-tree is constructed to allow fast random access to an arbitrary index or point in time. The underlying array provides cheap sequential access. Note that these structures are built in-memory per stream. Furthermore, they are not explicitly saved in the trace file as creating them when opening a file does not introduce a noticeable delay, even for large traces.

3.3 Memory Trace Encoding

For memory traces, Simutrace comes with a built-in memory encoder, which accepts trace entries with the following information:

- **Cycle Count**: A 48 bit monotonically increasing timestamp used to correlate entries chronologically.
- **Meta-Data**: A 16 bit field to save additional information about each memory access. Currently, only one bit is allocated to indicate if the CPU performed the access with the architecture's full register width.
- **Instruction Pointer (IP)**: The 32/64 bit virtual address of the instruction that issued the memory access. On the x86(-64) architecture this is the value of the `eip` or `rip` register respectively. The IP adds control flow information to memory traces.
- **Data Address**: The 32/64 bit virtual or physical memory address, depending on the chosen entry type.
- **Data**: An optional 32/64 bit field for the read or written value. For accesses below the architecture's native width, this field also includes the access's size. In a write trace, this field allows reconstructing the full main memory contents of the target, including executed code as well as all application- and OS data structures such as page tables.

Note that it is not necessary to include the type or originating device (e.g., CPU number in a multi-core simulation) in the trace entry, because different streams can be created to separate accesses by type or device. To encode a single memory access including the data, Simutrace thus generates a data entry of 20 bytes for 32 bit and 32 bytes for 64 bit architectures.

When working with memory traces, the amount of captured data is usually so high that traces must be compressed. With the presented format, tracing a minimal Linux kernel build generates over 1.5 TiB of raw data for 53 billion write entries. While general purpose compressors already provide decent reductions in size, over the years, various schemes specifically for memory traces have been developed [8, 11, 18, 24, 25, 27, 32, 34, 36, 42]. These methods better take advantage of the locality and repetition in memory traces and thus achieve higher compression ratios. With VPC4 [11], Burtscher et al. proposed one of the leading compressors for extended traces, that is traces, which contain hard-to-compress values such as the read or written data. These values repeat less often, exhibit less patterns and span larger ranges than instruction pointers and data addresses. In Simutrace, we want to be able to capture these values. We thus chose to use a modified version of VPC4 to form the heart of Simutrace's built-in memory encoder. Before we highlight our modifications, we start with a short overview of VPC4.

VPC4. At its core, VPC4 utilizes a set of value predictors to identify patterns in value sequences (e.g., memory access addresses) and to forecast the likely next value. The algorithm assigns each predictor a unique id. During compression, each traced value is compared with all predictions. If one or more predictors are right, the id of the successful predictor with the highest usage count is written to a *predictor id stream.* Otherwise, if all predictors are wrong, VPC4 writes a special failure id to the predictor id stream and notes the (unpredicted) value in a separate *predictor data stream.* A single byte is used to encode the predictor id, while 4 or 8 bytes are required for the extended data, depending on the simulated architecture's width (i.e., 32 or 64 bit). After a prediction has been retrieved, VPC4 updates all predictors with the current value and proceeds to the next trace entry. Decompression works analogously.

Since the prediction rate is usually between 82% and 98% [10]

most values can be expressed by a shorter predictor id, thus compressing the value sequence. In addition, the selection rule that is applied when multiple predictors are right, shifts the output to a small set of ids. That makes the id and data streams eligible for further compression through a second-stage compressor. Early versions of VPC included a custom compressor, but the authors found widely used general purpose algorithms to achieve better results. From version 3 on, VPC thus employs bzip2 [1, 9] as second-stage compressor.

VPC4 uses a mixture of *global prediction* for instruction pointers and *local prediction* for extended data (ED) such as addresses. For the latter, the algorithm uses predictors, which take the current IP as key to localize the prediction. To forecast the likely next value, VPC4 implements three types of predictors:

Last Value Predictor (LV). The last value predictor maintains a list of the last n seen values in least recently used (LRU) order. The predictor provides all n values as predictions and is consequently assigned a respective number of ids. The last value predictor is particularly suited to forecast simple sequences of repeating or alternating n values. For local prediction, the history of the last n values is extended to a *prediction table* (PT) with 2^t lines (see Figure 8). The instruction pointer modulo 2^t serves as index into the table. We denote a last value predictor with the symbol $\mathrm{LV}[n]_t$.

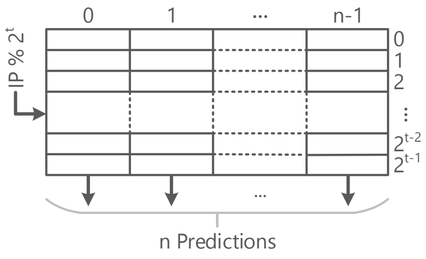

Figure 8: $\mathbf{LV}[n]_t$ predictor with n predictions and 2^t value lines. The index is built from the IP [12].

Finite Context Method Predictor (FCM). The finite context method predictor [43, 44] predicts the next value based on a finite number of r preceding values–the *context*–with the assumption that the next value will be equal to the one that followed last time the same context. The history length r is called the *order* of the predictor. To make a prediction, the FCM predictor hashes the context and uses the hash as index into a prediction table. FCM predictors can accurately forecast constant sequences of r arbitrary values, however they must see a respective number of values before they get matches. This phase is known as *learning time* [44]. For local prediction, the context is extended to a *context table* (CT) with 2^s lines. VPC4 uses FCM predictors for IPs and ED and maintains n predictions per context in LRU order, similarly to the LV predictor (see Figure 9). The hash is computed with the *select-shift-fold-xor* function [43]. We use $\mathrm{FCM}r[n]_t^s$ to describe a FCM predictor.

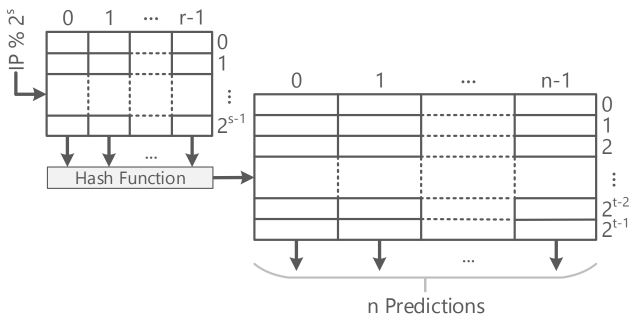

Figure 9: $\mathbf{FCM}r[n]_t^s$ predictor with order r, n predictions, 2^s history lines and 2^t value lines. The index is built from the IP [12].

Differential FCM Predictor (DFCM). A DFCM predictor [11, 20] is essentially a FCM predictor, which stores and is updated with *strides*, that is differences between consecutive values, instead of the absolute values. To form its predictions, the predictor adds the last absolute value to the predicted strides. In consequence, DFCM predictors are able to forecast values that they have not seen previously. Another key benefit of the differential predictor is a reduced aliasing in the prediction table. Due to these properties, DFCM predictors often outperform FCM predictors [20]. To denote a DFCM predictor we write $\mathrm{DFCM}r[n]_t^s$.

SVPC. To enable VPC4 to efficiently compress entries of the previously presented structure and to improve compression ratio as well as compression and decompression time, we modified VPC4 in multiple ways. From herein, we refer to our variant of VPC4 as Simutrace VPC or simply *SVPC*. SVPC differs from VPC4 in the following ways:

Transparency. VPC4 writes predictor ids and unpredicted values into separate files on disk. SVPC instead leverages the stream abstraction provided by our storage format. For each memory stream to compress, SVPC creates two additional *hidden* streams to store the predictor ids and data. The original memory stream is left empty in the trace file. However, when the user accesses the empty memory stream later, SVPC restores the content of the requested time span by decompressing the respective ranges in the hidden streams. The compression is thus kept transparent to the user.

Multi-Threaded Processing. To seamlessly integrate into the trace processing model in Simutrace, SVPC has been extended to perform multi-threaded (de-)compression by allowing multiple submitted or requested time spans to be processed in parallel.

Partial Decompression. VPC4 employs a tool to create the final compressor based on configurable parameters such as the structure of the data to process [12]. While these generated VPC4 compressors always process the whole trace, SVPC is capable of partial decompression at the granularity of frames.

Second-Stage Compression. The compressors built with VPC4 use pipelining to send the predictor ids and unpredicted values to the second-stage compressor. In contrast, SVPC fully integrates the second-stage compressor, which has been shown to improve compression and decompression speed [36]. In addition, we replaced the second-stage com-

Field	Size (Bit)	Predictors	Key/Base
Cycle Count	48	DCFCM1$[2]^{0}_{17}$, DCFCM3$[2]^{0}_{19}$	IP
Meta-Data[1]	16	-	-
Instruction Pointer (IP)	32/64	FCM1$[2]^{0}_{17}$, FCM3$[2]^{0}_{19}$	-
Data Address[2]	32/64	FCM1$[2]^{16}_{19}$, DFCM1$[2]^{16}_{17}$, DFCM3$[2]^{16}_{19}$, LV$[4]_{16}$	IP
Data[3]	32/64	FCM1$[2]^{16}_{19}$, DFCM1$[2]^{16}_{17}$, DFCM3$[2]^{16}_{19}$, LV$[4]_{16}$	IP

[1] Only one bit used; indicates full-size data access.
[2] Either physical or virtual address, depending on entry type.
[3] Optional. If not full-size, Data also encodes the access's size.

Table 1: Simutrace memory entry and SVPC predictor configurations.

pression algorithm. Burtscher et al. selected bzip2 [1] as default compressor and evaluated gzip [2] as an less effective, but faster alternative [11]. We empirically found that for trace data LZMA [4] usually achieves better compression in less time than bzip2.

Differential Cycle FCM Predictor (DCFCM). SVPC uses the same predictors and predictor configurations as VPC4 (see Table 1) with one exception. Simutrace adds a 48 bit monotonically increasing timestamp to every trace entry to correlate entries chronologically. Since we are using a simulation on functional level, we do not have access to the actual number of CPU cycles spent to reach a certain event. We hence chose to approximate the cycle count with the number of executed instructions and assume a cycles-per-instruction (CPI) ratio of 1:1, which is sufficiently precise for event correlation on instruction granularity. The compression of the cycle count takes advantage of this definition as the cycle count will show similar patterns as the instruction pointer. Because the cycle count is monotonically increasing, we need to use DFCM predictors to extract the patterns. When running benchmarks with the plain DFCM predictors, we however found that adding the instruction pointer to the stride further improved compression by reducing aliasing.

4. RELATED WORK

Tracing is a widely discussed topic in the literature and many solutions have been proposed [8, 15, 3, 31, 39, 45, 46, 54].

Sigma [15] is a framework for memory analysis with a focus on cache performance optimization. The traces recorded with Sigma are constrained to memory addresses and control flow information for correlation with program code. Sigma utilizes static binary rewriting for program instrumentation, that is instructions for capturing the memory access information are added to the compiled binary ahead of execution. For full system analyses, however, this technique is not suited.

With METRIC [30, 31], Marathe et al. presented a more recent memory tracing tool, which uses dynamic binary rewriting. METRIC thus allows recording more complex program behavior such as self-modifying or just-in-time generated code. However, the authors designed METRIC for collecting and processing partial access traces from user-space applications, only. That makes the tool as well unsuited for operating system research. Nonetheless, METRIC employs a sophisticated scheme for compressing memory access traces. The algorithm encodes memory accesses as a sequence of power regular section descriptors (PRSDs) [30]. Each descriptor comprises a base address and information to compactly express a (constant) stride pattern. PRSDs are linked through control flow information, which METRIC compresses with SEQUITUR [38]. For scientific applications, PRSDs yield a higher compression ratio than VPC [11]—the family of compression schemes we have chosen as basis for SVPC in Simutrace—because the algorithm has been specifically tailored to compactly encode memory access patterns originating from tight (nested) loops in a single thread and executable [31]. VPC on the other hand is targeted towards efficiently compressing traces of general-purpose programs. For whole system tracing, where memory accesses are issued by potentially hundreds of threads spread across a mix of applications as well as the operating system and drivers, VPC is hence a more adequate choice. Compared to VPC, PRSDs are also missing the capability to efficiently compress extended data such as the read or written values.

ScalaMemTrace [8] is a recent memory tracer for MPI applications. The tool is able to monitor multiple threads or processes and uses an extended version of PRSDs, called EPRSDs, for compression. EPRSDs further reduce trace size by leveraging recurring sequences of memory accesses across the monitored scheduling entities. However, they are still specifically tailored to compress memory traces of dense algebraic kernels and are thus not suited for whole system trace compression. ScalaMemTrace uses a similar architecture to Simutrace in that the trace reduction is performed asynchronously and in parallel. However, the degree of parallelization in ScalaMemTrace is limited to the number of traced MPI processes. Simutrace, in contrast, parallelizes the processing of stream segments and thus scales with the rate of submitted or requested data.

To capture memory references from *all* (user-space) processes on a system Kaplan et al. published a patch for the Linux 2.4 kernel, which integrates a memory tracing facility called kVMTrace [3]. The tool detects memory references by write-protecting all but a small, fixed portion of each process's virtual memory mappings. Accesses to any protected pages trigger a page fault that kVMTrace handles by logging the reference. The tool then unprotects the page to let the memory access succeed, potentially missing further accesses to the page. To reactivate monitoring, kVMTrace periodically resets the write-protection. The principle has been originally proposed by Uhlig et al. in Tapeworm II [47] as an alternative to trace-driven cache simulation. A benefit of this approach is its smaller slowdown compared to dynamic binary rewriting and full system simulation. kVMTrace thus trades execution speed for precision. As a kernel patch, kVMTrace is architecturally kept simple and provides no compression or scalable storage format.

The presented memory tracing frameworks share a limited applicability to operating system research grounded in their restriction to monitor user-mode code only. Simutrace in turn also records memory accesses issued by the OS kernel or other kernel-mode code such as device drivers. Retrace [45] is the work, which is closest to ours in that it also allows full system tracing in VMware Workstation 6.0[1]. The tool splits tracing into two separate phases. The workload is first run in a regular virtual machine (VM) with hardware-assisted virtualization. During this phase non-deterministic events such as interrupts are captured. The second phase is called the expansion phase, where the VM's execution is deterministically replayed on the basis of the recorded events. The hypervisor then runs in a simulation mode, which allows tracing of memory accesses. In the presented form, Retrace uses the expansion phase to record instruction traces in a plain text format with gzip [2] compression. Retrace thus misses most of the advanced mechanisms available in Simutrace such as a multi-threaded trace processing engine or an advanced storage backend. In contrast to Retrace, Simutrace can also be used independently from any particular full system simulator. In fact, reading previously recorded traces requires no simulator at all.

A drawback of the full system simulation in Retrace as well as in Simutrace is the immense slowdown of multiple orders of magnitude. We are currently investigating an acceleration method for scalable parallelization based on checkpoints and deterministic replay [41]. Sheldon et al. suggested a similar approach to mitigate the slowdown during their work on Retrace [45].

An alternative to software-driven full system memory tracing is the use of specialized hardware. HMTT [7] adopts a DIMM-snooping mechanism that utilizes hardware boards plugged into DIMM slots to capture whole system memory accesses. As in Simutrace, HMTT correlates traced operations with processes with the help of additional introspection information. A benefit of a hardware-based solution such as HMTT is, that it does not impose a relevant overhead on normal execution. However, the hardware has to advance steadily to keep up with increasing memory bandwidth and new memory standards. Hardware tracers are thus not only bound to costs for purchasing the hardware, but also for maintaining compatibility with new host technology. Simutrace instead can run on any standard PC and directly benefits from faster CPUs, increased hardware parallelism, or improved memory subsystems. As a software solution, Simutrace is also more flexible in what data is traced and how it is encoded for later access.

5. CONCLUSION

Memory trace analysis provides a valuable insight into the dynamic behavior of a system and delivers empirical support to focus optimization and debugging efforts. Memory traces are, however, not trivial to attain and pose high demands on the tracing components. In this white paper, we presented Simutrace, a framework for full-length, no-loss full system memory trace recording. Simutrace incorporates a fully parallelized processing engine and combines fast, but aggressive compression with a flexible and scalable storage format. Pre-compiled packages for Windows and Linux as well as the full source code and documentation of Simutrace are available for download at `http://simutrace.org`.

[1]Retrace is no longer supported in recent versions.

6. REFERENCES

[1] bzip2 compression library. `http://bzip.org/`.

[2] gzip compression tool. `http://www.gzip.org/`.

[3] kvmtrace. `http://www.cs.amherst.edu/~sfkaplan/research/kVMTrace/`.

[4] Lzma sdk. `http://www.7-zip.org/sdk.html`.

[5] Phoronix test suite. `http://www.phoronix-test-suite.com/`.

[6] Solaris dtrace. `http://wikis.oracle.com/display/DTrace/DTrace`.

[7] Y. Bao, M. Chen, Y. Ruan, L. Liu, J. Fan, Q. Yuan, B. Song, and J. Xu. Hmtt: a platform independent full-system memory trace monitoring system. In *ACM SIGMETRICS Performance Evaluation Review*, volume 36, pages 229–240. ACM, 2008.

[8] S. Budanur, F. Mueller, and T. Gamblin. Memory trace compression and replay for spmd systems using extended prsds. *The Computer Journal*, 55(2):206–217, 2012.

[9] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

[10] M. Burtscher. Vpc3: a fast and effective trace-compression algorithm. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, pages 167–176. ACM, 2004.

[11] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. The vpc trace-compression algorithms. *Computers, IEEE Transactions on*, 54(11):1329–1344, 2005.

[12] M. Burtscher and N. B. Sam. Automatic generation of high-performance trace compressors. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 229–240. IEEE, 2005.

[13] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti. Precise and accurate processor simulation. In *Workshop on Computer Architecture Evaluation using Commercial Workloads, HPCA*, volume 8, 2002.

[14] A. Chan, W. Gropp, and E. Lusk. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, 16(2):155–165, 2008.

[15] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. Sigma: A simulator infrastructure to guide memory analysis. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 1–1. IEEE, 2002.

[16] J. Ding et al. Pqemu: A parallel system emulator based on qemu. ICPADS. IEEE, 2011.

[17] J. Edler and M. D. Hill. Dinero iv trace-driven uniprocessor cache simulator, 1998.

[18] E. Elnozahy. Address trace compression through loop detection and reduction. In *ACM SIGMETRICS Performance Evaluation Review*, volume 27, pages 214–215. ACM, 1999.

[19] P.-M. Fournier, M. Desnoyers, and M. R. Dagenais. Combined tracing of the kernel and applications with lttng. In *Proceedings of the 2009 linux symposium*, 2009.

[20] B. Goeman, H. Vandierendonck, and K. De Bosschere. Differential fcm: Increasing value prediction accuracy by improving table usage efficiency. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 207–216. IEEE, 2001.

[21] T. Gröninger. On statistical properties of duplicate memory pages. Diploma thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, Oct.31 2013. `http://os.itec.kit.edu/`.

[22] M. Holliday. Techniques for cache and memory simulation using address reference traces. *International journal in computer simulation*, 1(1):129–151, 1991.

[23] H. Jagode, A. Knüpfer, J. Dongarra, M. Jurenz, M. S. Müller, and W. E. Nagel. Trace-based performance analysis for the petascale simulation code flash. *International Journal of High Performance Computing Applications*, 25(4):428–439, 2011.

[24] E. E. Johnson. Pdats ii: Improved compression of address traces. In *Performance, Computing and Communications Conference, 1999 IEEE International*, pages 72–78. IEEE, 1999.

[25] E. E. Johnson, J. Ha, and M. Baqar Zaidi. Lossless trace compression. *Computers, IEEE Transactions on*, 50(2):158–173, 2001.

[26] H. Kang and J. L. Wong. vcsimx86: a cache simulation framework for x86 virtualization hosts. 2013.

[27] S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson. Trace reduction for virtual memory simulations. In *ACM SIGMETRICS Performance Evaluation Review*, volume 27, pages 47–58. ACM, 1999.

[28] A. Knupfer and W. E. Nagel. Construction and compression of complete call graphs for post-mortem program trace analysis. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 165–172. IEEE, 2005.

[29] R. Lantz. Fast functional simulation with parallel embra. Citeseer, 2008.

[30] J. Marathe, F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, and A. Yoo. Metric: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 289–300. IEEE, 2003.

[31] J. Marathe, F. Mueller, T. Mohan, S. A. Mckee, B. R. De Supinski, and A. Yoo. Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(2):12, 2007.

[32] A. Milenkovic and M. Milenkovic. Stream-based trace compression. *Computer Architecture Letters*, 2(1):4–4, 2003.

[33] K. Miller. *Efficient Main Memory Deduplication Through Cross Layer Integration.* PhD thesis, Karlsruhe, Karlsruher Institut für Technologie (KIT), Diss., 2014, 2014.

[34] C. Mills, A. Snavely, and L. Carrington. A tool for characterizing and succinctly representing the data access patterns of applications. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 126–135. IEEE, 2011.

[35] S. Moore, F. Wolf, J. Dongarra, S. Shende, A. Malony, and B. Mohr. A scalable approach to mpi application performance analysis. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 309–316. Springer, 2005.

[36] T. Moseley, D. Grunwald, and R. Peri. Seekable compressed traces. In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pages 129–138. IEEE, 2007.

[37] R. C. Murphy and P. M. Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *Computers, IEEE Transactions on*, 56(7):937–945, 2007.

[38] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2 and 3):103–116, 1997.

[39] M. Payer, E. Kravina, and T. R. Gross. Lightweight memory tracing.

[40] M. Rittinghaus. Runtime benefits of memory deduplication. Diploma thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, July5 2012.

[41] M. Rittinghaus, K. Miller, M. Hillenbrand, and F. Bellosa. Simuboost: Scalable parallelization of functional system simulation. In *Proceedings of the 11th International Workshop on Dynamic Analysis (WODA 2013)*, Houston, Texas, Mar. 16 2013.

[42] A. D. Samples. *Mache: No-loss trace compaction*, volume 17. ACM, 1989.

[43] Y. Sazeides and J. E. Smith. Implementations of context based value predictors. Technical report, Citeseer, 1997.

[44] Y. Sazeides and J. E. Smith. The predictability of data values. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 248–258. IEEE, 1997.

[45] M. Sheldon and G. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2007)*, 2007.

[46] A. Snavely, N. Wolter, and L. Carrington. Modeling application performance by convolving machine signatures with application profiles. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 149–156. IEEE, 2001.

[47] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest. Trap-driven simulation with tapeworm ii. In *ACM SIGPLAN Notices*, volume 29, pages 132–144. ACM, 1994.

[48] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170, 1997.

[49] K. Wang et al. Parallelization of ibm mambo system simulator in functional modes. *SIGOPS*, 42(1), 2008.

[50] J. Weidendorfer, M. Kowarschik, and C. Trinitis. A tool suite for simulation based analysis of memory

access behavior. In *Computational Science-ICCS 2004*, pages 440–447. Springer, 2004.

[51] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely. Quantifying locality in the memory access patterns of hpc applications. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 50. IEEE Computer Society, 2005.

[52] R. W. Wisniewski and B. Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 3–3. IEEE, 2003.

[53] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 50–50. IEEE, 2000.

[54] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. Ubiquitous memory introspection. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 299–311. IEEE Computer Society, 2007.

[55] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.