Weir: A Streaming Language for Performance Analysis

Anton Burtsev Nikhil Mishrikoti * Eric Eide Robert Ricci

University of Utah
Salt Lake City, UT, USA

*Cisco Systems, Inc.
San Jose, CA, USA

aburtsev@cs.utah.edu nik.mys@gmail.com eeide@cs.utah.edu ricci@cs.utah.edu

Abstract

For modern software systems, performance analysis can be a challenging task. The software stack can be a complex, multi-layer, multi-component, concurrent, and parallel environment with multiple contexts of execution and multiple sources of performance data. Although much performance data is available, because modern systems incorporate many mature data-collection mechanisms, analysis algorithms suffer from the lack of a unifying programming environment for processing the collected performance data, potentially from multiple sources, in a convenient and script-like manner.

This paper presents Weir, a streaming language for systems performance analysis. Weir is based on the insight that performance-analysis algorithms can be naturally expressed as stream-processing pipelines. In Weir, an analysis algorithm is implemented as a graph composed of stages, where each stage operates on a stream of events that represent collected performance measurements. Weir is an imperative streaming language with a syntax designed for the convenient construction of stream pipelines that utilize composable and reusable analysis stages. To demonstrate practical application, this paper presents the authors' experience in using Weir to analyze performance in systems based on the Xen virtualization platform.

1. Introduction

The performance analysis of modern software systems is a challenging task. A modern enterprise system is a complex assembly of numerous software components that operate at multiple levels of the software stack and that use multiple levels of scheduling, data caching and request buffering, and various forms of parallelism including asynchronous and preemptive execution. Seeking to meet the growing performance and scalability requirements, modern systems go even further and reimplement functionality of existing operating system modules in custom runtimes that are tuned for their specific needs [4, 16, 22, 23]. A configuration error, modification to the system, change in a hardware setup, or even a change in a workload can result in unpredictable system behavior [9, 13].

The systems community has addressed the needs of performance analysis by concentrating on deploying lightweight data-collection mechanisms as part of the system stack. Significant progress has been made in this area. Dynamic instrumentation [7, 18] and hardware-based statistical sampling [14], for example, enable on-demand, dynamic collection of performance information and fine-grain attribution of performance events to specific parts of a system's code. The existing system stack provides a rich set of data *collection* mechanisms, which covers virtually every component in the system [10]. Performance of individual components can be well tested, profiled, analyzed, and understood.

What is lacking, however, is generic infrastructure for *analyzing* this data to understand the complex performance relationships between systems components. As systems scale out to run a combination of multiple software components and custom runtimes, with a hierarchy of schedulers that orchestrate execution on multiple (and many-core) CPUs and GPU coprocessors, the performance of systems becomes much less dependent on a single bottleneck. The performance of such systems is largely determined by the availability of data, latency of communication, overheads of synchronization, and efficiency of scheduling algorithms. The main task of performance analysis is to correlate performance information from multiple sources and reason about the behavior of overlapping, transient performance bottlenecks.

To address the need for unifying analysis infrastructure, we have created Weir, an imperative, streaming programming language environment in which analysis algorithms can operate on multiple streams of data. Weir reflects the nature of the performance domain: the need to run queries, analyze, and correlate events across multiple streams of performance data. The tracing technologies that already exist in the systems stack provide the power to collect the streams of performance events. Weir is designed to provide a convenient interface for querying and processing those streams.

In Weir, an analysis algorithm is expressed as composition of "stages" that operate on streams of performance data. Source stages read raw performance data, which is collected by different tracing mechanisms throughout the system stack, and convert each data source into a stream of events. Events flow through the assembly of stages, which is called a pipeline. The logic within the stages and the topology of the pipeline encodes the meaning of the analysis algorithm. Weir provides convenient language syntax for encoding analysis algorithms as pipelines.

Traditionally, the work on streaming languages has concentrated on two problems: (1) efficiently mapping a pipeline onto many-core hardware [5, 20, 27] and (2) online processing of massive amounts of data that is infeasible to store offline [1, 2]. The main goal for Weir, however, is delivering a set of programming abstractions that support the rapid and script-like development of systems performance-analysis algorithms. Weir is designed so that analyses can be implemented simply, intuitively, conveniently, and composably. Furthermore, Weir is aimed to become a part of the

* Nikhil Mishrikoti contributed to the research described herein while he was a graduate student at the University of Utah.

The definitive version was published in *Proceedings of the 7th Workshop on Programming Languages and Operating Systems (PLOS)*, Farmington, PA, Nov. 2013, http://dx.doi.org/10.1145/2525528.2525537

systems stack; its abstractions are designed to be familiar to systems engineers.

The contributions of this paper are threefold. First, it argues that a streaming programming language is a suitable and useful basis for the development of performance analyses for modern systems software stacks. Second, it presents Weir, a streaming language for constructing performance-analysis algorithms. Weir allows an analyst to implement new algorithms over multiple data sources in a script-like manner, making it possible to define and execute new analyses "on demand" as performance anomalies arise. Third, this paper presents evidence that Weir can be practically useful. It presents a case study in which we used Weir to debug performance problems in a virtualized environment based on Xen.

2. Weir

Weir is an imperative, streaming programming language designed to provide flexible, convenient, and intuitive mechanisms for constructing analysis pipelines. Weir's syntax is aimed at both scripting and command-line use.

Weir is a stream-based language because stream processing offers several advantages for the development of performance analysis algorithms. First, streaming involves the notion of repeatedly processing discrete "events," and events are a suitable and general abstraction for representing performance data. General operators, e.g., sorting, filtering, and caching, can be parametric and configured to operate on any event irrespective of the information it carries and the source it was collected from. Second, an event interface provides a flexible foundation for building analysis algorithms. Because filters both consume and produce events, they can be composed straightforwardly: the output of one filter becomes the input to another. Third, performance data is naturally time-ordered. Typical performance-analysis algorithms perform join operations on the timestamps of events, e.g., what was the CPU utilization when the queue reached saturation? Such operations are easily expressed and efficiently implementable in streaming languages. This eliminates the need for complex database technology, which might otherwise be required to run join queries in an efficient manner.

A Weir program defines a *pipeline*: a directed graph of processing nodes, called *stages*. A stage can have multiple incoming and outgoing edges, and Weir's runtime routes events from stage to stage along these edges. Each stage has a *filter*, a function that is invoked every time an event is delivered to the stage by the Weir runtime. When invoked by Weir's scheduler, a stage is responsible for consuming newly delivered events and producing zero or more output events. Each edge represents an unlimited queue of events, which is managed by Weir's runtime. Weir allows arbitrary pipeline topologies, including loops.

Weir is implemented in two layers, one for pipelines and another for filters. The pipeline layer is a domain-specific language for defining analysis pipelines. The filter layer consists of the stream operators that are the logic elements in a pipeline; these operators are written in C and C++. This design choice is driven by the observation that typical system-analysis problems are already solved by combining two layers: basic programs that collect and/or process data, and a scripting language that combines basic tools to achieve larger goals. For Weir, we chose to implement filter functions in C and C++ because these languages provide good integration with system libraries that are often required for comprehending behavior of the system: e.g., ELF and virtual-machine introspection (VMI) libraries. (In addition, C and C++ are widespread in the systems community, which is the target audience for Weir.) The purpose of Weir's domain-specific pipeline language is to allow users to quickly find the answer to a specific performance problem; this goal is met by allowing a user to "script" an appropriate assembly of stages to solve the problem at hand. The syntax of Weir's pipeline language is simple because it is tailored to composing stages only; the work of implementing the filter functions within stages is performed in C and C++.

2.1 Events

Events in Weir are objects; they are instances of types that extend a basic event type with domain-specific fields. The basic type provides two methods that access an event's type and timestamp, both of which are set by an event's source stage. By examining an event's type at run time, a filter can decide how the event should be processed: e.g., passed through, dropped, or downcast to a more specific event type so that domain-specific fields can be accessed. Event timestamps are measured in nanoseconds relative to a global clock. Source stages that read external inputs, such as log files, are responsible for translating from an external input's "local time" to Weir's global time.

2.2 Pipeline Construction

The simplest program in Weir defines a one-stage pipeline that invokes a filter. The example below instantiates a stage that invokes the *count()* filter to count the number of events in an event stream:

```
count()
```

Weir provides operators for static and dynamic construction of stream pipelines. The "|" (pipe) operator connects two stages with an edge. An output of the stage on the left side of pipe will flow into the input of the stage on the right. The example below uses a pipe to connect three stages into a pipeline that counts the number of hypercalls performed by a specific VM:

```
vm(id) | match(HYPERCALL) | count()
```

Two filters, *vm()* and *match()*, are combined to select events that (1) happen on behalf of a specific VM and (2) are instances of hypercall-invocation events.

Weir uses curly braces to introduce a scope. A scope is a composite stage that is defined by one or more internal pipelines. Individual pipelines within a scope are separated with a semicolon and conceptually run in parallel. The first stage of each pipeline is connected to the input-event point of the scope with a split connection. Similarly, the last stage of each pipeline is connected to the output-event point of the scope with a union connection. The example below constructs a pipeline that counts the number of page faults and hypervisor calls for a specific VM:

```
vm(id) | {
    match(PAGEFAULT);
    match(HYPERCALL);
} | count()
```

The outputs of both *match()* stages are combined to form the input to the *count()* stage, which counts the number of events it sees.

The "+" (plus) operator is syntactic sugar for the above notation. Plus creates a scope and connects all the stages in the plus expression to the entry and exit points of the scope. Parentheses provide a natural way to delineate parallel stages. Using the plus operator, the example above can be written as:

```
vm(id) | (match(PAGEFAULT) + match(HYPERCALL)) | count()
```

The former notation provides a more readable representation of the pipeline in large scripts, while the latter is suited to short pipelines invoked from the command line.

In many cases, it is convenient to control the flow of events and enable parts of a pipeline when a certain condition is met. Weir provides a logical *if()* operator that routes events to one of two pipeline branches depending on a Boolean test. The example below computes how much time a particular virtual machine spends inside the hypervisor when CPU utilization is greater than 30%:

```
if (utilization() > 0.3) { vm(id) | time_in_hypervisor(); }
```

The operators that form the test expression are regular streaming operators that can be used in any part of the pipeline, except that they are required to return an event that represents a value that can be used to form a proper logical expression.

To support analysis algorithms that require dynamic construction of the pipeline depending on the content of the performance trace, Weir provides a *foreach()* operator. *foreach()* creates a new pipeline from a template, which is described by an associated scope, every time it sees a previously unseen value returned by its selection expression. The example below shows all of the VMs that were running in a trace:

```
foreach (id = vm_id()) { take(1) | printf("VM id:%", id); }
```

In each of the scopes created by *foreach()*, the *id* variable is bound to a unique value returned by *vm_id()*. The *take()* filter passes one event "downstream" and drops all subsequent events; the *printf()* filter implements formatted output.

2.3 Named Pipes

Complex analysis logic often requires a pipeline graph that is impossible to construct with the plus and pipe operators. To enable the construction of arbitrary graphs, Weir relies on the concept of *named pipes*. A named pipe can appear at the beginning or end of a pipeline and refers to a unique connection point that can be attached to another part of the pipeline. The example below uses three named pipes, *s*, *s1*, and *s2*, to compute the wait time on CPU 2 when the utilization of CPU 1 is above 30%.

```
read("xentrace.dat") −> s;

s −> cpu(1) | s1;
s −> cpu(2) | s2;

s1 −> if (utilization() > 0.3) { emit(OPEN) | s2; }
      else { emit(CLOSE) | s2; }
s2 −> gate(CLOSED) | wait_time();
```

By default, all pipelines in a scope are connected to the scope entry point. The source operator ("->") allows one to define a specific source for a pipeline. Named pipes serve as sinks to source connectors. For example, events that appear on the incoming edges of *s2* (i.e., when *s2* appears at the end of a pipeline) are routed to the outgoing edges of *s2* in all places in which it serves as a source.

The above example also introduces gates. The *gate()* filter implements a control point that can be either "open" or "closed." An open gate passes the events it receives; a closed gate drops them. The state of a gate is changed by control events (Section 2.5).

Weir also uses named pipes to implement a traditional (blocking) *join* operator. By default, a scope connects the outputs of all its interior pipelines with a union operator: that is, it merges their outputs, and events that appear on any incoming edge of the union flow into the joining node at the moment they become available. To implement a blocking version of join, Weir relies on the *join* operator and named pipes. Named pipes are used to identify the incoming edges and pass them to the joining operator. The example below computes the amount of time that a guest virtual machine spends inside the hypervisor. The example uses two named pipes, *start* and *end*, to identify the edges for the joining function *time()*, which takes two events, one from each named pipe, as arguments:

```
{
    match(EXIT_FROM_GUEST) | start;
    match(EXIT_TO_GUEST) | end;
} join time(start, end) | sum()
```

The join operator blocks until events are available on each incoming edge. The join operator invokes the join function, *time()*, passing the events from the specified named pipes.

Combining named pipes and the assignment operator, Weir provides a notion of a traditional scalar variables. The example below uses named pipes *counter* and *id* as variables to report a sorted histogram of all event types encountered in a trace:

```
foreach (id = event_id()) {
    counter = count() | match(FLUSH) | cons(counter, id);
} join sort() | {
    car() | ctr;
    cdr() | event_descr() | str;
} join printf("% : %\n", ctr, str);
```

Inside *foreach()*, *count()* accumulates the number of events of the type associated with a particular template instance (i.e., a particular value of *event_id()*). The *match()* drops events from the pipeline until the *FLUSH* event is generated at the end of the event stream. At that point, *cons()* is used to create an event that is a pair of values: counter and event id. The pair from each template instance reaches *sort()*, which orders the pairs by their first (*car*) values.

The assign operator assigns a value returned by its right-hand-side stage to its left-hand-side named pipe, and pushes the input event down the pipeline. Effectively, the assign operator implements the following construct:

```
{ rhs() | lhs; tmp; } join cons(lhs, tmp) | cdr()
```

2.4 Modules

Modules serve two purposes in Weir. First, modules allow the user to extend the language with new filters. A typical module implements a trace-reading filter, which converts a domain-specific trace of performance data into an event-stream representation, and a set of domain-specific filters. Second, a module provides a namespace for event types and filters. Each module has a unique name. A tuple of a module name and event-type name ensures that each event type has a unique name. This allows Weir to work with performance data from multiple sources and dispatch event streams based on the name of the module that creates them (e.g., reads them from a trace file). The example below uses two modules "xentrace" and "oprofile" to read two trace files; this enables a combined analysis of events produced by a hypervisor and the guest operating system it is hosting. This example prints the events in time-sorted order to the console:

```
xentrace::read("xentrace.dat") −> r1;
oprofile::read("oprofile.dat") −> r2;

(r1 + r2) −> merge_sort(r1, r2) | {
    xentrace::match(_any) | xentrace::print();
    oprofile::match(_any) | oprofile::print();
}
```

The "_any" construct provides a way to specify a set of events created by a specific module. The *merge_sort()* filter merges two streams of temporally ordered events. Multiple named pipes can appear on the left-hand side of the arrow. The filter can reference them by name; in this case, the named pipes are plumbed as separate edges to the stage on the right-hand side. Alternatively, a stage can receive a merged union stream as a single union edge.

2.5 Control Events

Control events provide a mechanism to send control messages across the stages of a pipeline. Control events are frequently used to flush window stages like caches, to open and close gates, to restart counter stages, and so on. The example below uses CLOSE, FLUSH, and OPEN control events from Weir's *std* namespace. The analysis algorithm prints 100 events around the time when CPU utilization jumps to 30%, allowing the user to concentrate on behavior around a specific time period of interest:

```
use std; use xentrace;
cache_size = 100;
read("xentrace.dat") -> r;

r -> if (utilization() > 0.3) { emit(OPEN) | start_counter; }

(r + start_counter) -> gate(CLOSED) | every(cache_size/2) | {
                                emit(FLUSH) | flush_cache;
                                emit(CLOSE) | start_counter; }

(r + flush_cache) -> window_cache(cache_size) | print();
```

Two named pipes, *flush_cache* and *start_counter*, serve as dedicated channels that deliver events to the *window_cache()* and *gate()*. The *emit()* stages insert new control events of the specified types into the pipeline. The *window_cache()* buffers the last *N* events it has received, passing them downstream only when told to *FLUSH*.

3. Implementing the Language

The Weir runtime implements the abstractions required for constructing and running pipelines: pipeline-construction operators, stages, edges, scopes, modules, and variables. The Weir parser uses the functions provided by the runtime to construct and run pipelines.

3.1 Filters

Filters—the functions that define the logic within stages—are implemented by Weir modules and are registered with the language runtime. A filter is required to implement a single entry point, which is its *invocation handler*. The handler is invoked by the language runtime every time a new event needs to be processed by the filter. Optionally, each filter can implement and register an initializer and a finalizer. The initializer is invoked when a stage is created by the language runtime to use the filter; initializers provide a way to allocate storage that is private to a stage, and a pointer to this private storage is passed to the invocation handler every time it is invoked. The finalizer is called only once, when the operator is destroyed, and is typically used to compute and/or output aggregate statistics.

A filter's invocation handler takes one or more edges as an input. Two methods are provided by the runtime for the operators to dequeue and queue events from an edge: *pop()* and *push()*. The runtime uses reference counting as a primitive form of garbage collection to manage the life cycle of events. If a filter wants to drop an event, it dequeues it from the edge and releases its reference.

3.2 Scheduling

The Weir runtime allows a pluggable implementation of a scheduler. The current implementation uses a depth-first, round-robin scheduler. It tries to schedule the execution of all events on all outgoing edges of the current node, and while running each, it will follow the edge and will try to schedule the node at the end of the edge. This simple scheduler has the nice property that the oldest events are scheduled first, which results in a natural semantics for performance-analysis algorithms—a time-ordered input is processed in order, and results in time-ordered output. Source nodes, e.g., trace readers, do not have incoming edges. The runtime schedules them in a round-robin fashion when there are no runnable nodes left in the pipeline. A source node is scheduled until it reaches the end of file, at which time it becomes unrunnable.

4. Performance Analysis with Weir

We now present a case study to show how one can use Weir to analyze a performance anomaly in a version of Xen [3] that we enhanced for deterministic replay [6]. This example illustrates the way we expect Weir to be used in practice: it requires collating data from multiple sources in a complex, multi-layered environment,

```
use std; use xentrace;
read("xentrace.dat") -> time::skip(10) | time::take(1) | r;

r -> {
        first;
        event::skip(1) | second;
    } join time(second, first) | time;

second -> xentrace::print() | str;
(str + time) -> join printf("%:%", time, str);
```

Listing 1. Print the trace with times between adjacent events.

and follows an exploratory path, with simple initial observations pointing the way to more in-depth analysis. The performance issue described in the study is one that we actually encountered before we had implemented Weir. Diagnosing issues like the one in this study was one of our main motivations for developing the language.

Our XenTT system [6] adds an interposition layer to Xen that records all nondeterministic events flowing into a target VM. This layer adds some overhead, but it can be very difficult to find the specific causes of overhead from end-to-end observations alone. In one particular case, we found that the Phoronix Apache benchmark [17] suffered a much larger drop in performance than we expected [6]: when run in XenTT with recording enabled, this benchmark was able to serve only 69% of the requests that were served when recording was disabled. We set out to find the reason.

4.1 Integrating with Xen

We use two sources of data for analyzing performance in our replay-enabled version of Xen. The first is Xentrace, a data-collection framework included with Xen. Xentrace uses compile-time instrumentation of the hypervisor code, a lightweight trace communication mechanism, and a user-level daemon to collect a variety of hypervisor-level events. The second is the log created by XenTT's replay-interposition layer. The log stores a trace of all nondeterministic events, e.g., interrupts, results of nondeterministic instructions, reads from I/O ports, inputs from virtual device drivers, and so on. Another powerful feature of the replay log is its integration with the Branch Tracing Store (BTS) facility provided by Intel CPUs. BTS allows one to configure the CPU to record all branches taken by the system in a memory buffer. Our tracing infrastructure saves the memory buffer in a trace and implements a symbol-resolution tool that resolves machine addresses to human-readable function names and source line numbers. To integrate Xentrace and our replay log with Weir, we implemented a source reader for each facility that converts a raw trace into a stream of Weir events.

4.2 Analyzing Apache Performance

We start our analysis by implementing a Weir script that computes a sorted histogram of events in the trace collected by Xentrace. The script is similar to the third example in Section 2.3. Upon running the script, we find a clear anomaly in the output: the trace contains a high number of guest enter events in comparison to guest exits (10,301 versus 1,169), but one would expect those numbers to be the same. To verify our understanding of the trace, we implement another script that prints the trace of events while attributing each event with the time passed since the previous event was recorded (Listing 1). The script uses *read()* from the *xentrace* module to read the log collected by Xentrace. The *skip()* and *take()* filters select one second of execution starting ten seconds into the trace.

The script from Listing 1 confirms that Xentrace records several consecutive enters into the guest with no exits. This leads us to conclude that Xentrace instrumentation is incomplete. For some unknown event, it fails to record an exit from the guest system. We verify this conclusion by running a similar script against a log

```
1  use std; use xentrace;
2  read("xentrace.dat") -> xentrace;
3
4  xentrace -> cpu(2) | vm(1) | time::skip(10) | time::take(1) | r;
5
6  r -> match(FLUSH) | eof;
7
8  (expect_exit_guest + r) -> gate(OPEN) | {
9            match(EXIT_GUEST) | {
10                potential_start_pattern;
11                emit(CLOSE) | expect_exit_guest;
12                emit(OPEN) | expect_rdtsc;
13           }; }
14
15 (expect_rdtsc + r) -> gate(CLOSED) | {
16           match(RDTSC) | {
17                emit(CLOSE) | expect_rdtsc;
18                emit(OPEN) | expect_enter_guest;
19           };
20           match(ENTER_GUEST) | {
21                emit(CLOSE) | expect_rdtsc;
22                emit(OPEN) | expect_exit_guest;
23           }; }
24
25 (expect_enter_guest + r) -> gate(CLOSED) | {
26           match(ENTER_GUEST) | {
27                emit(FLUSH) | potential_start_pattern;
28                end_pattern;
29                emit(CLOSE) | expect_enter_guest;
30                emit(OPEN) | expect_exit_guest;
31           }; }
32
33 potential_start_pattern -> window_cache(1) | start_pattern;
34
35 (start_pattern + end_pattern) ->
36           join time(end_pattern, start_pattern) | sum;
37
38 (sum + eof) ->
39           sum() | time_in_hypervisor;
40
41 r -> {
42             event::take(1) | first;
43             window_cache(1) | last;
44       } join time(last, first) | total_time;
45
46 (time_in_hypervisor + total_time) ->
47       join printf("Total time %, time in hypervisor %\n",
48                   total_time, time_in_hypervisor);
```

Listing 2. Compute the amount of time spent in the hypervisor due to TSC accesses.

collected by the deterministic replay infrastructure. We find that the deterministic log is correct: it contains an identical number of enter and exit events, and it reveals that the events that cause lost exits are accesses to the timestamp counter (TSC) register via the `rdtsc` instruction. Normal Xen guests may directly invoke the `rdtsc` instruction, which provides them with a high-precision source of time. Our deterministic replay layer, however, must interpose on every TSC access, to record the value that is returned to guest.

We extend the Xentrace tracing layer to record exits from the guest system that are caused by a trapped instruction, which requires emulation by the hypervisor. We then create the script in Listing 2, which computes the total time that our guest system spends in Xen due to the emulation of `rdtsc` instructions. The script implements a state machine, illustrated in Figure 1, that recognizes a sequence of three events: guest exit, read TSC, and guest enter. Any number of other events can appear between the events of the triple.

The three pipelines at lines 8–13, 15–23, and 25–31 cooperate to implement the state machine. The current state is determined by

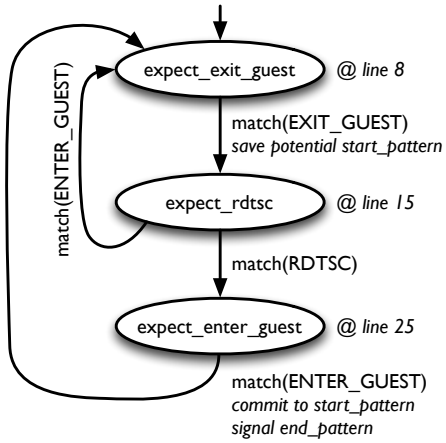

Figure 1. State machine implemented by the script in Listing 2.

which one of pipelines is unblocked, i.e., processing events rather than discarding them. The first pipeline, at lines 8–13, is the initial state and starts unblocked. When it sees an *EXIT_GUEST* event, it forwards the event—a potential start of an event triple—to the *window_cache()* at line 33, which buffers the event. The pipeline then triggers a state change by blocking further events to itself (line 11) and unblocking events to the second pipeline (line 12).

When the second pipeline detects an *RDTSC* event, it enables the third pipeline, which looks for an *ENTER_GUEST* event. When that event is found, the third pipeline signals that an exit-rdtsc-enter triple has been found. It "commits" the previously found *EXIT_GUEST* event by flushing the *window_cache()*, where it was previously held. The cache forwards its event downstream to the *start_pattern* named pipe. At the same time, the *ENTER_GUEST* event is forwarded to the *end_pattern* named pipe (line 28). In addition, the state machine returns to the initial state to find more exit-rdtsc-enter triples.

The output of the script's state machine is thus contained in the *start_pattern* and *end_pattern* named pipes, and the remainder of the script computes and prints the total time spent in the hypervisor due to TSC accesses. The join operator at line 36 takes paired start and end events and computes the time elapsed between them. The elapsed times are summed at line 39; when *sum()* receives a *FLUSH* on the *eof* named pipe, it outputs a single event containing the accumulated total time. The pipeline at lines 41–44 computes the elapsed time over all events in the trace, and the final pipeline produces the script's output.

The last step of the investigation relies on the BTS information from the deterministic log and clarifies the reason that the `rdtsc` instruction is invoked. A simple Weir script that prints several BTS events immediately preceding the TSC access reveals that the `rdtsc` instruction is invoked as part of the network packet-receive function (*netif_receive_skb()*) in the Linux kernel.

Using Weir, we were able to start with a high-level observation of poor performance and trace its cause to the frequent use of a specific x86 instruction in the Linux kernel.

5. Related Work

The concept of a streaming language takes roots in multiple application and research domains [24]. One side of this spectrum, digital signal processing, emphasizes the ability of a streaming language to express multiple forms of parallelism and efficiently map computation onto multi-node hardware [5, 20, 27]. Another side, database and event-processing systems, views streaming languages as a mechanism to express a computation over an infinite stream of data [1, 2]. Despite the central commonality—the computation is performed on streams of data—the diverging goals of each application domain

drive the design of each domain's languages in its own direction. A variety of streaming language "dialects" exist, including frameworks and libraries that provide streaming constructs [21, 28], extensions to existing languages [5, 11, 26], imperative streaming languages that provide visual or textual primitives for constructing stream pipelines [1, 27], functional reactive programming [8], and declarative extensions to relational SQL and logical languages [2, 12]. None of these existing dialects exactly fits the goals of Weir.

Closest to Weir, imperative streaming languages like StreamIt [27] implement only static scheduling of the stream: i.e., the rate at which operators produce events is fixed at compile time. Static scheduling enables a variety of performance-critical optimizations but harms the ability of a language to express many stream-processing tasks outside of the small set of domains in which the computation is inherently static, e.g., graphics algorithms and signal processing. Weir seeks to provide a simple, predictable programming model and full flexibility of programming over streams. Dynamic scheduling, the ability to construct arbitrary pipelines, and convenient syntax with explicit streams are critical for representing algorithms from the performance-analysis domain. Being dynamically scheduled, Weir pays a performance price. Recent research, however, argues that a general intermediate representation for dynamic streaming languages can enable many optimizations, which traditionally were possible only for statically scheduled streaming languages [25].

6. Conclusion

The principal barrier to understanding the performance of a modern systems software stack is often not a lack of data. Rather, it is the difficulty of reasoning over multiple sources of data within a single analysis framework. Weir is a new stream-based programming language that supports whole-system analyses by providing an environment for script-like implementations of analysis algorithms over multiple data sources. Weir is evolving, and its authors intend to expand the breadth and depth of the analyses that can be expressed in Weir. In particular, they plan to apply Weir to multi-data-source analyses within the A3 adaptive security environment [15].

Acknowledgments

We thank Jon Rafkind, who implemented the parser generator utilized by Weir [19], and we thank and the anonymous PLOS '13 reviewers for their comments on drafts of this paper. This material is based upon work partially supported by the National Science Foundation under Grant No. 1059440. This work was also partially supported by the Air Force Research Laboratory and DARPA under Contract No. FA8750–10–C–0242.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug. 2003.

[2] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, Oct. 2003.

[4] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: safe user-level access to privileged CPU features. In *OSDI*, pages 335–348, Oct. 2012.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graphics*, 23(3):777–786, Aug. 2004.

[6] A. Burtsev. *Deterministic Systems Analysis*. Phd dissertation, University of Utah, May 2013.

[7] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX ATC*, pages 15–28, June–July 2004.

[8] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *PLDI*, pages 411–422, June 2013.

[9] B. Ford. Icebergs in the clouds: the other risks of cloud computing. In *HotCloud*, June 2012.

[10] B. Gregg. The USE Method: Linux performance checklist, Mar. 2012. `http://dtrace.org/blogs/brendan/2012/03/07/`.

[11] G. Hong, K. Hong, B. Burgstaller, and J. Blieberger. AdaStreams: a type-based programming extension for stream-parallelism with Ada 2005. In *Reliable Software Technologies — Ada-Europe 2010*, volume 6106 of *LNCS*, pages 208–221. Springer, 2010.

[12] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, pages 75–90, Oct. 2005.

[13] J. C. Mogul. Emergent (mis)behavior vs. complex software systems. In *EuroSys*, pages 293–304, Apr. 2006.

[14] OProfile, July 2013. `http://oprofile.sourceforge.net/`.

[15] P. Pal, R. Schantz, A. Paulos, B. Benyo, D. Johnson, M. Hibler, and E. Eide. A3: An environment for self-adaptive diagnosis and immunization of novel attacks. In *SASO Workshops*, Sept. 2012.

[16] Y. Park, E. Van Hensbergen, M. Hillenbrand, T. Inglett, B. Rosenburg, K. D. Ryu, and R. W. Wisniewski. FusedOS: Fusing LWK performance with FWK functionality in a heterogeneous environment. In *IEEE 24th Intl. Symp. on Computer Architecture and High Performance Computing*, pages 211–218, Oct. 2012.

[17] Phoronix Test Suite: An automated, open-source testing framework. `http://www.phoronix-test-suite.com/`.

[18] V. Prasad, W. Cohen, F. Ch. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Linux Symp.*, volume 2, pages 49–64, July 2005.

[19] J. Rafkind. Vembyr - multi-language PEG parser generator written in Python, Nov. 2011. `http://code.google.com/p/vembyr/`.

[20] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, pages 519–530, June 2013.

[21] E. C. Reed, N. Chen, and R. E. Johnson. Expressing pipeline parallelism using TBB constructs: a case study on what works and what doesn't. In *SPLASH '11 Workshops*, pages 133–138, Oct. 2011.

[22] B. Rhoden, K. Klues, D. Zhu, and E. Brewer. Improving per-node efficiency in the datacenter with new OS abstractions. In *SOCC*, Oct. 2011.

[23] J. Sacha, J. Napper, S. Mullender, and J. McKie. Osprey: Operating system for predictable clouds. In *IEEE/IFIP 42nd Intl. Conf. on Dependable Systems and Networks Workshops (DSN-W)*, June 2012.

[24] R. Soulé. *Reusable Software Infrastructure for Stream Processing*. PhD dissertation, New York University, May 2012.

[25] R. Soulé, M. I. Gordon, S. Amarasinghe, R. Grimm, and M. Hirzel. Dynamic expressivity with static optimization for streaming languages. In *DEBS*, pages 159–170, June–July 2013.

[26] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: high-throughput stream programming in Java. In *OOPSLA*, pages 211–228, Oct. 2007.

[27] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction: 11th Intl. Conf.*, volume 2304 of *LNCS*, pages 179–196. Springer, 2002.

[28] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable Internet services. In *SOSP*, pages 230–243, Oct. 2001.