Weir: A Streaming Language for Performance Analysis

Anton Burtsev  Nikhil Mishrikoti  Eric Eide  Robert Ricci

University of Utah
Salt Lake City, UT USA
{aburtsev, nikmys, eide, 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. A configuration error, modification of the topology or a specific context of execution and multiple sources of performance data. 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 a 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 manycore hardware and (2) online processing of massive amounts of data that is infeasible to store offline. 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 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 streaming 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 route 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 systems 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 tailored to composing stages only; the work of implementing the filter functions within stages is performed in C and C++.

### 2.1 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 the log:

```
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 hypervisors 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 hypervisor 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 higher 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 relies on `foreach()` to create a new counter for each previously unseen event type. The example below shows all VMs that were running in the 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 `printf()` function implements formatted output.

### 2.2 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 at the 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 CPU utilization on the CPU 1 is above 30%.

```cpp
read("xentrace.dat") -> s;

t1 -> cpu(1) | s1;

t2 -> cpu(2) | s2;

s1 -> if (utilization() > 0.3) { emit(UNBLOCK) | s2; } else { emit(BLOCK) | s2; }
s2 -> block() | 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.

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 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:

```cpp
{ 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. An example below uses named pipes `counter` and `id` as variables to report a sorted histogram of all event types encountered in a trace:

```cpp
foreach (id = event_id()) {

counter = count() | match(FLUSH) | cons(counter, id);

join sort() |

car() | cdr() | event_descr() | str;

} join print("% : %
", 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, `const` 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 the right hand side operator to the named pipe, and pushes the input event down the pipeline. Effectively the assign operator implements the following construct:

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

### 2.3 Modules

Modules serve two purposes in Weir. First, modules allow the user to extend the language with new filter operators. A typical module implements a trace-reading operator, which converts a domain-specific trace of performance data into a general event 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:

```cpp
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 operator 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.

### 2.4 Control Events

Control events provide a mechanism to send control messages across stages of a pipeline. Control events are frequently used to flush window operators like caches, block and unblock parts of the pipeline, restart counter operators, and so on. The example below uses BLOCK, FLUSH, and UNBLOCK control events from Weir’s `std` namespace. The analysis algorithm prints out one hundred events around the time when CPU utilization jumps to 30%, allowing the user to concentrate on behavior around specific time period of interest:

```cpp
use std; use xentrace;

xentrace::read("xentrace.dat") -> r;

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

(r + start_counter) -> block() | every(cache_size/2) |
{ emit(FLUSH) | flush_cache;
emit(BLOCK) | start_counter; }

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

Two named pipes, `flush_cache` and `start_counter`, serve as dedicated control channels that deliver events to the `window_cache()` and `block()`. The `emit()` stages insert a new event of the specified type into the pipeline. The `window_cache()` stage keeps the history of last `N` events.

### 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 the pipeline.

#### 3.1 Filters

Filters—the functions that define the logic within stages—are implemented by Weir modules and are registered with the language
We use two sources of data for analyzing performance in our replay. 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 implemented Weir. Diagnosing issues like the one in this study and follows an exploratory path, with simple initial observations pointing the way to more in-depth analysis. The performance issue we analyze is 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 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 are allowed to directly invoke 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 trace exits from the guest system that are caused by a trapped instruction, which requires emulation by the hypervisor. We then implement the script in Listing 2 which computes the amount of time that our guest system spends in Xen due to the emulation of rdtsc instructions.

The script implements a state machine 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 state machine is implemented with pipelines (lines 8–22 and 25–31). Two of these start in the “blocked” state and receive no events until they become unblocked. The first, at lines 8–13 starts unblocked. When the first pipeline sees the EXIT_GUEST event, it saves it as a potential start of the event sequence in the window_cache() and unblocks the pipeline that detects the RDTSC event. If the last pipeline detects the ENTER_GUEST event, it routes the last event in the sequence to the end_pattern named pipe. When both start_pattern and end_pattern become available, the join function at line 36 computes the elapsed time.

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 second example in Section 2.2. 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 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 are allowed to directly invoke 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 trace exits from the guest system that are caused by a trapped instruction, which requires emulation by the hypervisor. We then implement the script in Listing 2 which computes the amount of time that our guest system spends in Xen due to the emulation of rdtsc instructions.

The script implements a state machine 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 state machine is implemented with pipelines (lines 8–22 and 25–31). Two of these start in the “blocked” state and receive no events until they become unblocked. The first, at lines 8–13 starts unblocked. When the first pipeline sees the EXIT_GUEST event, it saves it as a potential start of the event sequence in the window_cache() and unblocks the pipeline that detects the RDTSC event. If the last pipeline detects the ENTER_GUEST event, it routes the last event in the sequence to the end_pattern named pipe. When both start_pattern and end_pattern become available, the join function at line 36 computes the elapsed time.
use std; use xentrace;
read("xentrace.dat") -> xentrace;
xentrace -> cpu(2) | vm(1) | time::skip(10) | time::take(1) | r;
r -> match(FLUSH) | eof;
\( (\text{expect\_exit\_guest} + r) \rightarrow \text{unblocked} \) |
  \{ 
    match(EXIT\_GUEST) |
    \{ 
      emit(UNBLOCK) | expect_rdtsc;
      emit(BLOCK) | expect_exit_guest;
      \} 
    \} 
  \} ; 
\} 
\} 
\} 
\} 
\} 
ansism 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 streaming language in its own direction.
A variety of streaming language “dialects” exist, including frame-
works and libraries that provide streaming constructs [19][25], exten-
sions to existing languages [5][11][24], imperative stream languages
that provide visual or textual primitives for constructing stream
pipelines [1][25], functional reactive programming [8], and declar-
ative extensions to relational SQL and logical languages. [2][13].
None of these existing dialects exactly fits the goals of Weir.

Closest to Weir, imperative streaming languages like StreamIt [25]
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 [23].

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 an evolving language, and in
the future, its authors intend to expand the breadth and depth of
the analyses that can be expressed in Weir.

References
and architecture for data stream management. The VLDB Journal,
language: semantic foundations and query execution. The VLDB Journal,
[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neuge-
bauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In
[4] A. Belay, A. Bittau, A. Moshizadeh, D. Terei, D. Mazières, and
K. Kozyrakis. Dune: safe user-level access to privileged CPU features.
In OSDI, pages 335–348, 2012.
and P. Hanrahan. Brook for GPUs: stream computing on graphics
sity of Utah, 2013.
tion of production systems. In USENIX Annual Technical Conference,
[8] E. Czaplicki and S. Chong. Asynchronous functional reactive program-


