Fay: Extensible Distributed Tracing from Kernels to Clusters

Úlfar Erlingsson
Google Inc.*

Marcus Peinado
Microsoft Research
Extreme Computing Group

Simon Peter
ETH Zurich*
Systems Group

Mihai Budiu
Microsoft Research
Silicon Valley

ABSTRACT

Fay is a flexible platform for the efficient collection, processing, and analysis of software execution traces. Fay provides dynamic tracing through use of runtime instrumentation and distributed aggregation within machines and across clusters. At the lowest level, Fay can be safely extended with new tracing primitives, including even untrusted, fully-optimized machine code, and Fay can be applied to running user-mode or kernel-mode software without compromising system stability. At the highest level, Fay provides a unified, declarative means of specifying what events to trace, as well as the aggregation, processing, and analysis of those events.

We have implemented the Fay tracing platform for Windows and integrated it with two powerful, expressive systems for distributed programming. Our implementation is easy to use, can be applied to unmodified production systems, and provides primitives that allow the overhead of tracing to be greatly reduced, compared to previous dynamic tracing platforms. To show the generality of Fay tracing, we reimplement, in experiments, a range of tracing strategies and several custom mechanisms from existing tracing frameworks.

Fay shows that modern techniques for high-level querying and data-parallel processing of disaggregated data streams are well suited to comprehensive monitoring of software execution in distributed systems. Revisiting a lesson from the late 1960's [15], Fay also demonstrates the efficiency and extensibility benefits of using safe, statically-verified machine code as the basis for low-level execution tracing. Finally, Fay establishes that, by automatically deriving optimized query plans and code for safe extensions, the expressiveness and performance of high-level tracing queries can equal or even surpass that of specialized monitoring tools.

Categories and Subject Descriptors

D.4.8 [**Performance**]: Monitors; D.2.5 [**Software Engineering**]: Testing and Debugging—*Tracing*

General Terms

Design, Languages, Measurement, Experimentation, Performance

*Work done while at Microsoft Research Silicon Valley.

To appear in SOSP 2011: the 23rd ACM Symposium on Operating Systems Principles, October 23-26, 2011, Cascais, Portugal.

1. INTRODUCTION

Fay takes a new approach to the collection, processing, and analysis of software execution traces within a machine or across a cluster. The dictionary definition of Fay is "a fairy," as a noun, or "to join tightly or closely," as a verb. In our work, Fay is a comprehensive tracing platform that provides both expressive means for querying software behavior and also the mechanisms for the efficient execution of those queries. Our Fay platform implementation shows the appeal of the approach and can be applied to live, unmodified production systems running current x86-64 versions of Windows.

At its foundation, Fay provides highly-flexible, efficient mechanisms for the inline generation and general processing of trace events, via dynamic instrumentation and safe machine-code execution. These mechanisms allow pervasive, high-frequency tracing of functions in both kernel and user-mode address spaces to be applied dynamically, to executing binaries, without interruption in service. At the point of each trace event generation, Fay safely allows custom processing of event data and computation of arbitrary summaries of system state. Through safe execution of native machine code and through inline code invocation (not using hardware traps), Fay provides primitives with an order-of-magnitude less overhead than those of DTrace or SystemTap [11, 45].

At its topmost level, Fay provides a high-level interface to systems tracing where runtime behavior of software is modeled as a distributed, dynamically-generated dataset, and trace collection and analysis is modeled as a data-parallel computation on that dataset. This query interface provides a flexible, unified means for specifying large-scale tracing of distributed systems. High-level queries also allow the Fay platform to automatically optimize trace event collection and analysis in ways that often greatly reduce overhead.

Below is an example of a complete high-level Fay query that specifies both what to trace and also how to process and combine trace events from different CPUs, threads, and machines:

```
from io in cluster.Function("iolib!Read")
   where io.time < Now.AddMinutes(5)
   let size = io.Arg(2)   // request size in bytes
   group io by size/1024 into g
   select new { sizeInKilobytes = g.Key,
                countOfReadIOs = g.Count() };
```

This query will return, for an entire cluster of machines, an aggregate view over 5 minutes of the read sizes seen in a module `iolib`, for all uses of that module in user-mode or in the kernel. In our Fay implementation, such declarative queries are written in a form of LINQ [29]. From these queries, Fay automatically derives efficient code for distributed query execution, optimizing for factors such as early trace data aggregation and reduced network communication.

Fay can also be accessed through other, more traditional means. In particular, in our implementation, Fay can be used through scripts in the PowerShell system administration scripting language [55], as well as directly through standard command-line tools. However it is used, Fay retains the best features of prior tracing systems, such as efficient trace event collection, low overhead—proportional to tracing activity, and zero by default—and stateful *probes* that can process event data directly at a tracepoint. Fay also provides strong safety guarantees that allow probes to be extended in novel ways with new, high-performance primitives.

1.1 Implementation and Experience

For now, Fay has been implemented only for the current x86-64 variants of Windows. However, the Fay approach is generally applicable, and could be used for distributed software execution tracing on most operating systems platforms. In particular, a Fay implementation for Linux should be achievable by modifying existing mechanisms such as Ftrace [48], Native Client [64], and the FlumeJava or Hadoop data-parallel execution frameworks [2, 13].

Although the specifics will vary, any Fay implementation will have to overcome most of the same challenges that we have addressed in our implementation for Windows. First, Fay must preserve all the relevant software invariants—such as timing constraints, reentrancy and thread safety, locking disciplines, custom calling conventions, paging and memory access controls, and the execution states of threads, processes, and the kernel—and these are often hard-to-enumerate, implicit properties of systems platforms.

Specifically, Fay must correctly manage tracepoints and probes and reliably modify machine code to invoke probes inline at tracepoints—which is made especially challenging by preemptive thread scheduling and hardware concurrency [1]. As described in Section 3, Fay meets these challenges with generally-applicable techniques that include machine-wide code-modification barriers, non-reentrant dispatching, lock-free or thread-local state, and the use of time-limited, safe machine code to prevent side effects. In particular, Fay offers the lesson that reliable machine-code modification is a good basis for implementing platform mechanisms, as well as to install tracepoints.

Second, Fay must provide mechanisms for safe machine-code extensibility, in a manner that balances tradeoffs between simplicity, performance, high assurance, applicability to legacy code, compatibility with low-level runtime environments, debuggability, ease-of-use, etc. As described in Section 3.3, the safety of our Fay extensions is based on XFI mechanisms, which are uniquely well suited to low-level, kernel-mode machine code [18]. We have developed several variants of XFI, over a number of years, and applied them to different purposes. Our experience is that specializing mechanisms like XFI to the target application domain, and its constraints, results in the best tradeoffs. Thus, Fay's XFI variant is relatively simple, and is tuned for thread-local, run-to-completion execution of newly-written, freshly ported, or synthesized Fay extensions, either in user-mode processes or the kernel.

Third, as the last major hurdle, to efficiently support high-level queries, a Fay tracing platform must correctly integrate with new or existing query languages and data-parallel execution frameworks. In particular, Fay query-plan generation, optimizations, and task scheduling must correctly consider the difference between persistent, redundantly-stored trace event data and tracepoint-generated data—which is available only at an online, ephemeral source, since a tracepoint's thread, process, or machine may halt at any time. Section 4.2 describes how our Fay implementation meets this challenge, by using a simple, fixed policy for scheduling the processing of ephemeral trace events, by using explicitly-flushed, constant-size (associative) arrays as the single abstraction for their data, and by applying incremental-view-update techniques from databases to query planning and optimization.

We have applied Fay tracing to a variety of execution monitoring tasks and our experience suggests that Fay improves upon the expressiveness and efficiency of previous dynamic tracing platforms, as well as of some custom tracing mechanisms. In particular, we have found no obstacles to using data-parallel processing of high-level queries for distributed systems monitoring. Although Fay query processing is disaggregated—collecting and partially analyzing trace events separately on different CPU cores, user-mode processes, threads, and machines—in practice, Fay can combine collected trace events into a sufficiently global view of software behavior to achieve the intended monitoring goals. We have found no counterexamples, ill-suited to Fay tracing, in our review of the execution tracing literature, in our searches of the public forums and repositories of popular tracing platforms, or in our experiments using Fay tracing to reimplement a wide range of tracing strategies, described in Section 5. Thus, while data-parallel processing is not a natural fit for all computations, it seems well-suited to the mechanisms, strategies, and queries of distributed systems tracing.

Our experiences also confirm the benefits of extensibility through safe, statically-verified machine code—benefits first identified four decades ago in the Informer profiler [15]. Safe extensions are key to the flexibility of Fay tracing, since they allow any untrusted user to utilize new, native-code tracing primitives without increased risk to system integrity or reliability. As described in Section 4.2, they also enable practical use of high-level, declarative Fay tracing queries, by allowing Fay to synthesize code for efficient, query-specific extensions that it can use for early aggregation and processing in optimized Fay query plans.

In the rest of this paper we outline the motivation, design, and high-level interfaces of Fay tracing and describe the details of its mechanisms. We report on benchmarks, measurements, and use cases in order to establish the scalability, efficiency, and flexibility of Fay tracing and to show its benefits to investigations of software behavior. In particular, we show that Fay tracing can replicate and extend a variety of powerful, custom strategies used on existing distributed software monitoring platforms.

2. GOALS AND LANGUAGE INTERFACES

Fay is motivated by an idealized model of software execution tracing for distributed systems, outlined in Figure 1. The goals can be summarized as follows: The tracing platform should allow arbitrary high-level, side-effect-free *queries* about any aspect of system behavior. At each *tracepoint*—i.e., when the traced behavior occurs at runtime—the platform should allow arbitrary processing across all current system state. Such general processing *probes* should be allowed to maintain state, and used to perform early data reduction (such as filtering or aggregation) before emitting *trace events*.

Ideally, tracing should incur low overhead when active and should have zero overhead when turned off. The total overhead should be proportional to the frequency of tracepoints and to the complexity of probe processing. Tracing should be optimized for efficiency, in particular by favoring early data reduction and aggregation; this

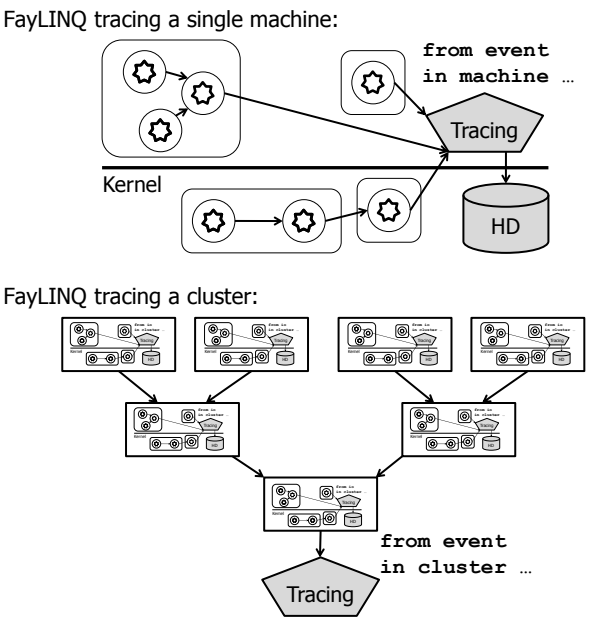

Figure 1: Tracing of an operating system and a machine cluster, as implemented in FayLINQ. Stars represent tracepoints, circles are probes, rounded rectangles are address spaces or modules, rectangles are machines, and pentagons denote final aggregation and processing. Arrows show data flow, optimized for early data reduction within each module, process, or machine; redundant copying for fault tolerance is not indicated.

optimization should apply to all communication, including that between probes, between traced modules, and between machines in the system. Finally, trace events may be ephemeral, since software or hardware may fail at any time; however, once a trace event has been captured, further trace processing should be lossless, and fault-tolerant.

To achieve these goals for Fay tracing, our implementation integrates with two high-level-language platforms: PowerShell scripting [55] and the DryadLINQ system for distributed computing [66]. Figure 2 and Figure 3 show examples of how Fay tracing can be specified on these platforms.

FayLINQ is a high-level interface to Fay tracing that allows analysis of strongly-typed sequences of distributed trace events. FayLINQ is implemented by extending DryadLINQ and derives its expressive programming model from Language INtegrated Queries, or LINQ [29]. FayLINQ's programming model allows a flexible combination of object-oriented, imperative code and high-level declarative data processing [65, 66]. A FayLINQ query can simultaneously express trace collection, trace event analysis, and even the persisting of trace event logs.

FayLINQ queries operate on the entire dataset of all possible tracepoints, and their associated system state, but hide the distributed nature of this dataset by executing as if it had been collected to a central location. In practice, queries are synthesized into data-parallel computations that enable tracing only at relevant tracepoints, and perform early data selection, filtering, and aggregation of trace events. FayLINQ makes use of modified mechanisms from DryadLINQ—described in Section 4.2—to handle query optimization, data distribution, and fault-tolerance [65, 66]. In particular, analysis and rewriting of the query plan allows FayLINQ to automatically derive optimized code that runs within the finite space and time constraints of simple probe processing, and can be used even in the operating system kernel.

```
$probe = {
   process {
     switch( $([Fay]::Tracepoint()) ) {
       $([Fay]::Kernel("ExAllocate*"))
         { $count = $count + 1; }
     }
   }
   end { Write-FayOutput $count; }
}
Get-FayTrace $probe -StopAfterMinutes 5 `
   |  select count `
   |  measure -Sum
```

Figure 2: A Fay PowerShell script that counts the invocation of certain memory-allocation functions in a 5-minute interval, on all CPUs of a Windows kernel. Here, `$probe` uses a `switch` to match tracepoints to `awk`-like processing (counting) and specifies the output of aggregated data (the count). A separately-specified pipeline combines the outputs (into a final sum).

```
cluster.Function(kernel, "ExAllocate*")
.Count(event => (event.time < Now.AddMinutes(5)));
```

Figure 3: An example FayLINQ query to perform the same count as in Figure 2 across an entire cluster. From this, Fay can generate optimized query plans and efficient code for local processing (counting) and hierarchical aggregation (summing).

There is little room for optimization in script-based tracing systems such as Fay PowerShell, or the popular DTrace and SystemTap platforms [11, 45]. These scripting interfaces share inefficiencies that can also be seen in Figure 2. Trace events are generated by executing imperative probes that are specified separately, in isolation from later processing, and this barrier between event generation and analysis prevents most automatic optimizations. Furthermore, by default, for final analysis, trace events must be collected in a fan-in fashion onto a single machine.

In comparison, FayLINQ is able to give the illusion of tracing a single system, through a unified, coherent interface, even when multiple computers, kernels, or user-level processes are involved. Only a few limitations remain, such as that tracing may slightly perturb timing, and that probes can access only state in the address space they are tracing.

Fay tracing may sometimes be best done directly on the command line, or through a PowerShell script, despite the limited opportunity for optimization, In particular, PowerShell is part of the standard Windows monitoring toolset, and is well suited to processing and analysis of object sequences such as trace events [55]. Furthermore, PowerShell exposes Windows secure remote access features that allow Fay scripts to be executed even across machines in heterogeneous administrative domains.

Even so, the benefits of FayLINQ over PowerShell are made clear by the example query of Figure 3. This query shows how simple and intuitive tracing a cluster of machines can be with FayLINQ—

especially when compared against the more traditional script in Figure 2, which applies to one machine only. Using FayLINQ, this query will also be executed in an efficient, optimized fashion. In particular, counts will be aggregated, per CPU, in each of the operating system kernels of the cluster; per-machine counts will then be aggregated locally, persisted to disk—redundantly, to multiple machines for fault-tolerance—and finally aggregated in a tree-like fashion for a final query result.

3. FUNDAMENTAL MECHANISMS

At the core of Fay tracing are safe, efficient, and easily extensible mechanisms for tracing kernel and user-mode software behavior within a single machine.

3.1 Tracing and Probing

The basis of the Fay platform is dynamic instrumentation that adds function tracing to user-level processes or the operating system kernel. Fay instrumentation is minimally intrusive: only the first machine-code instruction of a function is changed, temporarily, while that function is being traced.

Notably, Fay instrumentation uses inline invocations that avoid the overhead of hardware trap instructions. However, such inline invocations, and their resulting state updates, are necessarily confined to a single process, or to the kernel, forcing each address space to be traced separately. Therefore, Fay treats even a single machine as a distributed system composed of many isolated parts.

3.1.1 Tracepoints

Fay provides *tracepoints* at the entry, normal return, and exceptional exit of the traced functions in a target address space. All Fay trace events are the result of such function boundary tracing. Fay can also support asynchronous or time-based tracepoints, as long as they eventually result in a call to an instrumentable function.

When a tracepoint is triggered at runtime, execution is transferred inline to the Fay *dispatcher*. The dispatcher, in turn, invokes one or more probe functions, or *probes*, that have been associated with the tracepoint. A probe may be associated with one or more tracepoints, and any number of probe functions may be associated with each tracepoint. Further details of the Fay dispatcher are described in Section 3.2 and illustrated in Figure 5.

To enable tracing of an address space, the base Fay platform module must be loaded into the address space to be traced. This platform module then installs probes by loading *probe modules* into the target address space.

3.1.2 Probe Modules

Fay probe modules are kernel drivers or user-mode libraries (DLLs). For both FayLINQ and PowerShell, source-to-source translation is used to automatically generate compiled probe modules. (Our implementation uses the freely available, state-of-the-art optimizing C/C++ compiler in the Windows Driver Kit [36].)

Figure 4 outlines how Fay probe modules are used for tracing in the kernel address space. A high-level query is evaluated and compiled into a safe probe module; then, that driver binary is installed into the kernel. At a kernel function tracepoint, Fay instrumentation ensures that control is transferred to the Fay dispatcher, which invokes one or more probes at runtime. Finally the probe outputs (partially) processed trace events for further aggregation and analysis.

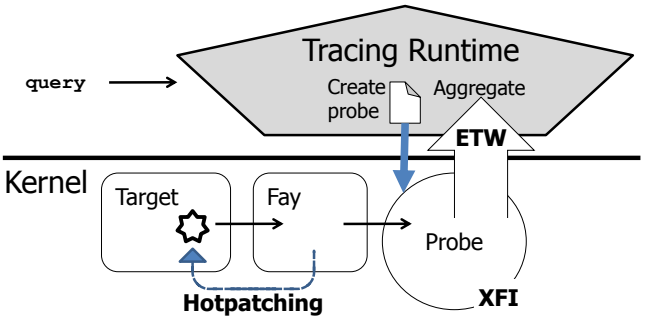

Figure 4: Overview of how Fay makes use of probes when tracing the kernel address space. Visual representations are as in Figure 1—e.g., the star is a tracepoint. Kernel arrows show probe module installation (going down), dynamic instrumentation (going left), the dispatch of a tracepoint to a probe function (going right), as well as the flow of trace event data (going up).

Probe modules are subject to the standard Windows access control checks. In particular, only system administrators can trace the kernel or other system address spaces, and kernel probe modules must be cryptographically signed for the x86-64 platform. However, this is not enough: bad compiler setup, malicious input data, or other factors might easily lead to the creation of a flawed probe that would impair system security and reliability. Therefore, subsequent to their generation, probe module binaries are rewritten and processed to establish that they can be safely loaded and used within the traced address space. This processing is based on a variant of XFI: a Software-based Fault Isolation (SFI) technique that is uniquely applicable to both kernel-mode and user-mode code [61, 64, 18]. Section 3.3 gives the details of the simplified XFI mechanisms used in our Fay platform.

Fay probe modules can be written from scratch, in C or C++, ported from legacy code, or even hand-crafted in assembly code. Fay can also be extended with new computations or data structures, similarly specified as low-level or native code. Such Fay probe *extensions* might, for example, include hash functions for summarizing state, or code for maintaining representative samples of data. Extensions allow enhancing Fay with new primitives without any changes to the platform—and can be used even from FayLINQ or other high-level queries. Extensions are compiled with probes, and are subject to the same safety checks; therefore, they raise no additional reliability or security concerns.

Fay resolves symbolic target-module references by making use of debug information emitted at compile time for executable binaries. (Much the same is done in other tracing systems [11, 45].) On the Windows platform, such "PDB files" are available, and easily accessible through a public network service, for all components and versions of Windows.

3.1.3 Probe Processing

When triggered at a tracepoint, a probe will typically perform selection, filtering, and aggregation of trace data. For instance, a probe may count how often a function returns with an error code, or collect a histogram of its argument values. However, probes are not limited to this; instead, they may perform arbitrary processing.

In particular, probes might summarize a large, dynamic data structure in the traced address space using expensive pointer chasing—

but do so only when certain, exceptional conditions hold true. Fay probe *extensions* for such data traversal may even be compiled from the same code as is used in the target system. Thus, Fay tracing can make it practical to perform valuable, deep tracing of software corner cases, and to gather all their relevant system state and execution context when they occur.

Fay probes can invoke an *accessor* support routine to examine the state of the system. Multiple accessors are available in a runtime library and can be used to obtain function arguments and return values, the current CPU, process, and thread identity, CPU cycle counts, etc. A `TryRead` accessor allows attempted reading from any memory address, and thereby arbitrary inspection of the address space. All accessors are simple, and self-contained, in order to prevent probe activity from perturbing the traced system.

3.1.4 Probe State

For maintaining summaries of system behavior, Fay provides each probe module with its own local and global memories. This mutable state is respectively private to each thread (or CPU, in the kernel), or global to each probe module. These two types of state allow efficient, lock-free, thread-local data maintenance, as well as communication between probe functions in the same address space—globally across the CPUs and threads of the target system.

Both types of mutable probe state are of constant, fixed size, set at the start of tracing. However, probes may at any time send a *trace event* with their collected data, and flush mutable state for reuse, which alleviates the limitations of constant-size state. To reduce the frequency of such trace event generation, probes can make use of space-efficient data structures (e.g., our Fay implementation makes use of cuckoo hashtables [19]).

To initialize global and local state, probe modules can define special *begin* and *end* probe functions, invoked at the start and end of tracing. These "begin" and "end" probe functions are also invoked at thread creation and termination, e.g., to allow thread-local state to be captured into a trace event for higher-level analysis.

In combination, the above mechanisms allow Fay probes to efficiently implement—from first principles—tracing features such as predicated tracing, distributed aggregation, and speculative tracing [11]. In addition, they make it easy to extend Fay tracing with new primitives, such as sketches [7]. These features are exposed through the high-level Fay language interfaces, and can be considered during both the optimization of Fay tracing queries and during their execution. Section 5 describes some of our experiences implementing such extended Fay tracing features.

3.1.5 Limitations of Fay Tracepoints and Probes

Compared to popular, mature tracing platforms, our Fay implementation has some limitations that stem from its early stage of development. For example, while Fay tracing can be used for live, online execution monitoring (e.g., as in Figure 7), the batch-driven nature of the Dryad runtime prevents streaming of FayLINQ query results. Also, currently, users of Fay tracing must manually choose between `call` and `jmp` dispatchers, and whether trace events are logged to disk, first, or whether per-machine analysis happens in a real-time, machine-local Fay aggregation process.

On the other hand, the Fay primitives in our implementation are fundamentally limited to function-boundary tracing of specially-compiled binary modules, for which debug information is available. Other tracing platforms also rely on debug information to offer full functionality, and are applied mostly to properly-compiled or system binaries. Less common is Fay's lack of support for tracing arbitrary instructions. However, although supported by both DTrace and SystemTap, per-instruction tracing can affect system stability and is also fragile when instructions or line numbers change, or are elided, as is common in optimized production code. Thus, this feature is not often used, and its omission should not greatly affect the utility of Fay tracing.

To confirm that per-instruction tracing is rarely-used, we performed an extensive review of the public discussion forums and available collections of tracing scripts and libraries for both DTrace and SystemTap. Typical of the per-instruction tracing we could find are examples such as counting the instructions executed by a process or a function [17], or the triggering of a tracing probe upon a change to a certain variable [57]. This type of tracing is not likely to be common, since it requires extensive instrumentation and incurs correspondingly high overhead, and since its goals are more easily achieved using hardware performance counters or memory tracepoints. Programmer addition of new debugging messages to already-compiled code is the one example we could find where per-instruction tracing seemed practical [56]; however, the same can also be achieved by running under a debugger or, if recompilation is an option, by the addition of calls to empty functions, which Fay could then trace. Therefore, we have no current plans to extend Fay beyond function-boundary tracing.

Fay supports only disaggregated tracing, even within a single machine: Fay probes have only a disjoint view of the activity in different address spaces, i.e., the kernel or each user-mode process, which is then combined by higher-level Fay trace-event processing. Existing tracing platforms such as DTrace [11] support imperative operations on per-machine shared state, and use hardware-trap-based instrumentation to access this shared state from both the kernel and any user-mode address space. We have considered, but decided against, adding Fay support for machine-global probe state, accessible across all address spaces, implemented via memory mapping or a software device driver. So far, the distributed nature of Fay tracing has made it sufficiently convenient to get visibility into user-mode activity by combining trace events from user and kernel address spaces.

3.2 Dispatching Tracepoints to Probes

Fay tracing uses inline invocations to a Fay probe dispatcher, through a call or jump instruction inserted directly into the target machine code. Some other platforms dynamically insert a kernel transition, or faulting instruction, to perform tracing [11, 45]. Compared to this alternative, Fay inline tracing offers greater efficiency, by avoiding hardware traps; similarly, the Ftrace facility recently added to Linux also uses inline tracing for kernel functions [48].

Fay repurposes Windows *hotpatching* in a novel manner to modify the machine code at a function entry point, so that control is transferred to the Fay probe dispatcher. Windows function hotpatching is an existing operating systems facility, designed to allow incorrect or insecure functions to be replaced on a running system, without a reboot or process shutdown [34]. Hotpatching performs reliable, atomic code modification with all CPUs in a well-defined state (e.g., not executing the code being hotpatched). Previously, hotpatching has been rarely used: since its introduction in 2003, we are not aware of a generally-available software update from Microsoft that makes use of hotpatching.

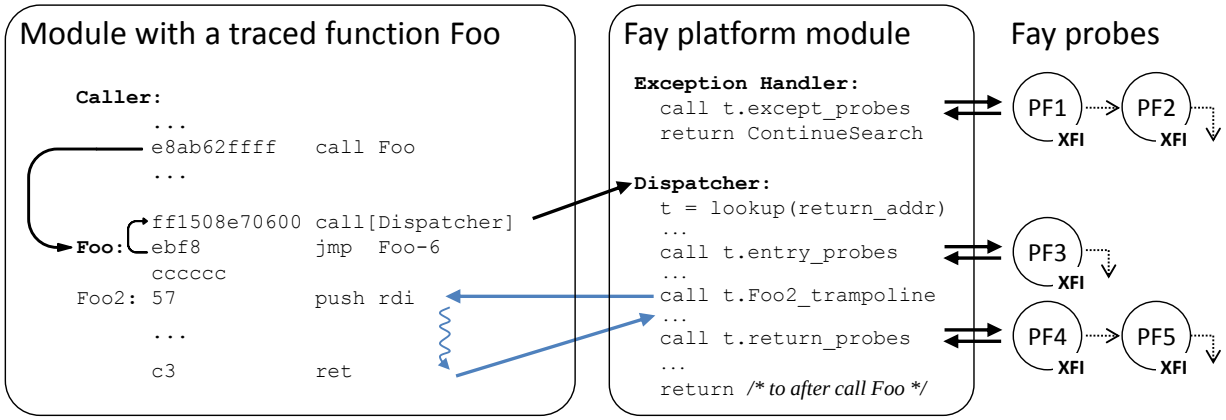

Figure 5: Fay dynamic instrumentation of a function `Foo`, with five separate, safe probe functions invoked at its entry, return, and exception tracepoints. Rounded rectangles show the relevant binary modules in the traced address space (the kernel, or a user-mode process). Arrows indicate control flow, starting at the original call to `Foo` (no arrow is shown for the return to that original call site). The lighter arrows, colored blue, show the nested call to `Foo` from the Fay dispatcher—via a trampoline that executes `Foo`'s original first instruction and then jumps to its second instruction.

Fay uses the hotpatching mechanism to insert, at the start of functions, inline invocations to the Fay probe dispatcher. This permitted, but unintended use of hotpatching allows Fay to be used for the pervasive tracing of existing, unmodified production systems.

All currently supported Windows binaries are hotpatch enabled. Hotpatching constrains machine-code at function entry: six unused bytes must be present before the function, and its first instruction must be at least two bytes long, and be drawn from a small set of opcodes. Each binary must also contain a number of *hotpatch data slots* for pointers to new function versions; a normal binary module has only 31 such slots, while the kernel has 127. In Fay, these constraints on hotpatch data slots do not limit the number of tracepoints: Fay tracing is scalable to an arbitrary number of functions.

Figure 5 shows the machine code of a function `Foo` after Fay has used hotpatching to modify `Foo` to enable its entry, return, and exceptional exit tracepoints. The first instruction of `Foo` has been replaced with a two-byte instruction that jumps backwards by six bytes. At the six-bytes-earlier address, a new instruction has been written that calls the Fay dispatcher. The call is indirect, through one of the hotpatch data slots of the target module being traced (this indirection allows loading the Fay platform module anywhere in the 64-bit address space).

As Figure 5 indicates, upon entry the Fay dispatcher looks up a descriptor for the current tracepoint (shown as `t` in the figure). Tracepoint descriptors control what probes are triggered and provide the crucial first instruction that allows the dispatcher to call the traced function. Fay looks up these descriptors in a space-efficient hashtable [19], and the use of a simpler hashtable, with significantly more memory, could reduce the cost of this lookup. For threads not being traced, the lookup and use of descriptors might even be eliminated by using a Fay dispatcher with multiple entry points—one for each possible first instruction—since different preamble code at each distinct entry point could instruct the Fay dispatcher how to emulate the effects of a traced function's first instruction before passing control to the rest of the function. Fay does not yet implement such elaborations, since we have found the current lookup efficient enough (about 40 cycles in our measurements).

A Fay tracepoint descriptor contains lists of probe functions to be invoked, as well as other relevant information—such as the global and local state to be used for each probe. Dispatching is lock free, but runs with (most) interrupts disabled; descriptor updates are atomically applied at an all-CPU synchronization barrier.

If the current thread is to be traced, the Fay dispatcher will invoke probe functions both before and after the traced function as indicated in the tracepoint descriptor lists—subjecting the execution of each probe to the necessary safety and reliability constraints.

The Fay dispatcher also invokes the traced function itself. For this, the dispatcher creates a new stack frame with copies of the function's arguments. Then, the dispatcher uses a pointer from the tracepoint descriptor to transfer control to a function-specific, executable trampoline that contains a copy of the traced function's first instruction, followed by a direct jump to its second instruction.

The Fay dispatcher also registers an exception handler routine, for capturing any exceptional exit of the function being traced. Fay invokes exceptional exit probes when an exception is unwound past this handler; once the probes have executed, Fay forwards the exception on to higher stack frames.

Actually, Fay provides multiple dispatcher implementations whose performance and scalability differs. In particular, depending on the traced function, Fay can save different sets of registers: functions synthesized through whole-program optimizations require preserving all registers, while stable, externally-accessible functions require saving only a small, non-volatile set of registers.

Figure 5 shows the slowest and most scalable version of the Fay dispatcher. This version hotpatches a `call` instruction before the traced function. That `call` pushes `Foo`'s address on the stack for descriptor lookup. This dispatcher is scalable since it requires only one hotpatch data slot (out of the very limited number of slots). However, the `call` places a superfluous return address on the stack, which the dispatcher must eliminate before returning (at the /**/ comment). Unfortunately, on modern CPU architectures, such stack manipulations can have an adverse performance impact

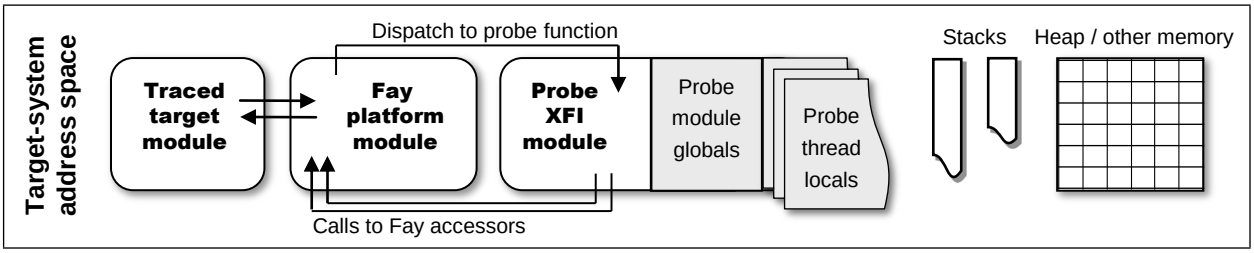

Figure 6: The layout of a traced address space, with a Fay probe XFI module. Probe functions may invoke only a restricted set of Fay accessor support routines. Probe functions may write only to the shaded memory areas—and only to the thread-local memory of the current thread. A probe may attempt to read any memory address via a Fay accessor that prevents faults due to invalid addresses. XFI safeguards the integrity of the execution stacks, privileged hardware registers, and other critical host-address-space state.

by disrupting dynamic branch prediction [51]. Therefore, when only a limited number of functions are traced, Fay will use a faster dispatcher, where hotpatching places a `jmp` instruction to a dispatch trampoline. Both dispatchers have low overheads; Section 5 compares their performance.

3.3 Reliability and Safety

Reliability is the paramount goal of the Fay dispatcher and other Fay mechanisms; these must be correct, and are designed and implemented defensively, with the goal of allowing target systems to always make progress, and fail gracefully, in the worst case. However, Fay relies crucially on the safety of probe processing: to the rest of the system, probes must always appear as (almost) side-effect-free, pure functions—whether written by hand, compiled in an uncertain environment, or even when crafted by a malicious attacker. To ensure probe safety, previous tracing systems have used safe interpreters or trusted compilers [11, 45].

Fundamentally, Fay ensures probe safety through use of XFI: one of the recently-developed, low-overhead SFI mechanisms that are suitable to x86-64 CPUs [18, 61, 64]. XFI is the only SFI mechanism to be applicable even to machine code that runs as part of privileged, low-level systems software. Thus, Fay can rely on XFI to provide comprehensive constraints on machine code probes, including flexible access controls and strong integrity guarantees, and yet allow probes to be utilized in any address space, including the kernel. As in all SFI systems, safety is enforced through a combination of inline software guards and static verification of machine code. Below, we outline the characteristics of the Fay variant of XFI; more details about its underlying policies and mechanisms can be found in the original XFI paper [18].

Like previous variants, Fay XFI is implemented using Vulcan [54]. However, Fay XFI aims for simplicity, and avoids complexities—such as "fastpath guards" [18]—as long as doing so retains acceptable performance. Instead of being fully inlined, Fay XFI guards reside in separate functions, but are invoked inline with arguments pushed on the stack. While slightly less efficient, this style leads to minimal code perturbation, which both simplifies XFI rewriting and also facilities debugging and understanding of probe machine code.

Fay XFI is also customized to its task of enforcing safety properties for Fay probes. Figure 6 shows a Fay XFI probe module in a target address space (cf. Figure 1 in [18]). Fay probes should be side-effect-free, and execute only for short periods—to completion, without interruption, serially on each (hardware) thread—using only the fixed-size memory regions of their local and global state, and making external invocations only to Fay accessor routines. Thus, upon a memory access, Fay XFI memory-range guards can compare against only one thread-local and one static region, and need not consult slowpath permission tables—and, similar, fixed tables can be consulted upon use of a software call gate.

Fay probes are not unmodified legacy code—they are either newly written, newly ported, or automatically generated. Therefore, Fay XFI does not allow arbitrary C, C++, or assembly code, but imposes some restrictions on how probes are written. Fay probes may not use recursive code, dynamically allocate memory on the stack frame, or make use of function pointers or virtual methods; these restrictions make XFI enforcement of control-flow integrity trivial, and also reduce the number of stack-overflow guards necessary, by allowing worst-case stack usage to be computed statically. Also, Fay probes may not use code that generates or handles exceptions, or use other stack context saving functionality; such probe code would be very difficult to support at low levels of the kernel and we have removed the associated XFI host-system support. Finally, Fay probes may not access stack memory through pointers, so probe code must be converted to use thread-local probe state instead of stack-resident variables; this simplifies XFI rewriting and verification, and eliminates the need for XFI allocation stacks. These restrictions do not prevent any functionality, and although they may result in greater porting efforts for some Fay probe extensions, this is not onerous, since Fay probes necessarily execute relatively small amounts of code and this code is often automatically generated.

Despite the above simplifications, Fay XFI still enforces all the safety properties of XFI [18]—for instance, constraining machine-code control flow, preventing use of dangerous instructions, restricting memory access, and thwarting violations of stack integrity.

3.3.1 Thread-local Tracking for Reliability

To ensure reliability, the interactions between Fay and the software it is tracing must always be benign. Thus, the operation of the Fay dispatcher, probes, and accessors must be self-contained, since Fay's invocation of an external subsystem might adversely affect the integrity of that subsystem, or result in deadlock. For example, while Fay accessor routines may read system state, they must never invoke system functions with negative side effects.

A thread that is performing Fay dispatching must be treated differently by both the Fay platform and the system itself. In particular, Fay tracing must not be applied recursively, such as might happen if Fay were used to trace system functions that are themselves used by code in a Fay accessor routine. This scenario might happen, e.g., if Fay tracing was applied to mechanisms for trace event transport.

To prevent recursive tracing, Fay maintains a thread-local flag that is set only while a probe is executing, and that is checked during dispatching. (In the kernel, a small amount of thread-local storage is available in the CPU control block; in user mode, arbitrary thread-local storage is available.) A similar flag allows Fay to efficiently support thread-specific tracing: the common scenario where some threads are traced, but not others. Depending on the state of these flags for the current thread, the Fay dispatcher may skip all probes and invoke only the traced function. Fay keeps a count of lost tracing opportunities due to the Fay dispatcher being invoked recursively on a flagged thread.

Fay does not enforce any confidentiality policy: no secrets can be held from kernel probes. Even so, Fay kernel probes are subject to an unusual form of memory access control. A probe may write only to its global or local state, and may only read those regions when dereferencing a memory address. In addition, probes may use a special `TryRead` accessor to try to read a value from any (potentially invalid) memory address; this functionality can be used by probes that perform pointer chasing, for example. The `TryRead` accessor sets a thread-local flag that changes pagefault behavior on invalid memory accesses and prevents the kernel from halting (Section 3.5 gives further details on its implementation). However, Fay will prevent even `TryRead` from accessing the memory of hardware control registers, since such accesses could cause side effects.

Finally, probes must be prevented from executing too long. In the kernel, a special tracing probe is added by Fay to one of the Windows kernel functions that handles timer interrupts, to detect runaway probes. This special probe maintains state that allows it to detect if a hardware thread is still running the same probe as at the previous timer interrupt—and will trigger an exception if a Fay probe runs for too many timer interrupts in a row.

3.4 Transporting Trace Events

Fay uses Event Tracing for Windows, (*ETW*) [41] to collect and persist trace events in a standard log format. ETW is a high-functionality Windows system mechanism that provides general-purpose, structured definitions for trace events, efficient buffering of trace events, support for real-time trace consumers as well as efficient persistent logging and access to tracelog files, support for dynamic addition and removal of producers, consumers, and trace sessions, as well as the automatic provisioning of timestamps and other metadata.

ETW tracing is lock free and writes trace events to CPU-local buffers. Also, ETW is lossless, in that the number of outstanding buffers is dynamically adjusted to the rate of event generation—and in the unlikely case that no buffer space is available, an accurate count of dropped events is still provided. Finally, the standard, manifest-based ETW tracelog formats allows Fay trace events to be consumed and processed by a wide range of utilities on the Windows platform.

3.5 Practical Deployment Issues

Our Fay implementation has been crafted to ensure that it can be installed even on production systems, without a reboot. In particular, we have carefully (and painfully) avoided dependencies on system internals, and on features that vary across Windows versions. For this, our Fay implementation sometimes makes use of side-effect-free tracing of system functions such as in our support for asynchronous tracepoints. In one case we had to change the behavior of Windows: Fay hotpatches the kernel page fault handler with a new variant that throws an exception (instead of halting execution) when invalid kernel-mode addresses are accessed during execution of the `TryRead` accessor.

The use of Fay tracing is subject to some limitations. In particular, Fay requires that target binary modules have been compiled with hotpatching support; while this holds true for binaries in Windows and Microsoft server products, it is not the case for all software. Also, kernel tracing with the more scalable Fay probe dispatcher will require rebooting with kernel debugging automatically enabled; otherwise, PatchGuard [35] will bugcheck Windows after detecting an unexpected `call` instruction, which it disallows in machine-code hotpatches.

Finally, even for Windows system binaries, Fay is currently not able to trace variable-argument functions—since the Fay dispatcher would then have to create a stack frame of unbounded size for its invocation of the traced function.

4. LANGUAGES FOR FAY TRACING

We have integrated Fay with PowerShell to provide a traditional scripting interface to tracing, and also created FayLINQ to provide a LINQ query interface and a declarative, data-parallel approach to distributed tracing. Both these popular high-level language platforms provide flexible, efficient means of specifying tracing, in a manner that feels natural—thereby removing the need to introduce a domain-specific language, as done in other dynamic tracing platforms [11, 45].

We have implemented several Fay support mechanisms that can be utilized both in PowerShell and FayLINQ, since both are managed code platforms. In particular, these provide for optimized compilation of probe modules, their installation into the kernel, or injection into a user-mode process. These mechanisms also give access to debug information (from PDBs) for currently executing software—e.g., to allow symbolic identification of tracepoints in a target binary module, as well as the global variables, types, enums, etc., of that module. Finally, these mechanisms allow real-time consumption of ETW trace events, and the custom, type-driven unmarshalling of their contents.

4.1 Fay PowerShell Scripting

Here we give a brief outline of Fay PowerShell scripting. PowerShell is structured around *cmdlets*, which are similar to awk scripts operating on streams of objects, and augmented with administration and monitoring features. In PowerShell, Fay probes are just regular cmdlets, with a few natural changes in semantics: `begin{}` blocks execute at the start of tracing, `process{}` executes at each tracepoint, variables such as `$global:var` live in global state, whereas regular variables are thread local, etc.

When used with Fay support cmdlets, such as `Get-FayTrace`, tracing scripts are converted to C code, using source-to-source translation, and compiled and processed into binary XFI probe modules. Fay makes use of partial evaluation to resolve symbolic reference in PowerShell scripts, as well as to identify tracepoints and define a specialized probe function for each tracepoint. We have used PowerShell mostly as a convenient means for ad hoc Fay tracing, like that in Figure 7.

4.2 FayLINQ Queries

FayLINQ integrates the fundamental Fay mechanisms with the LINQ language, as well as the optimizations and large-scale data

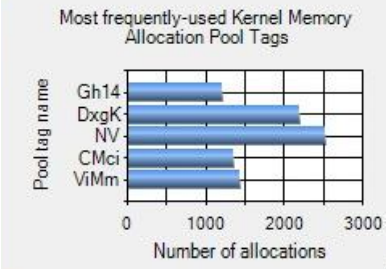

Figure 7: Output of a 20-line Fay PowerShell script that every second updates a visual histogram of the five most common types (or "tags") of memory allocations from non-paged kernel memory. The greatest number of memory allocations are of type 'NV', indicating they are due to the NVidia display driver.

processing capabilities of DryadLINQ [66]. This combination allows high-level queries about distributed systems behavior to be applied to—and executed on—the same cluster of computers.

On both a single machine, and on a cluster, FayLINQ input is naturally modeled as operations on a concatenated set of trace event streams. Fundamentally, Fay tracing generates multiple, disjoint streams of ordered trace events, with a separate trace event stream output by each thread in each address space. Therefore, FayLINQ tracing consists of the execution of LINQ queries on an unordered, merged collection of these ordered streams.

Concretely, the FayLINQ implementation proceeds from a single, high-level query to generate an efficient set of tracepoints, and code for Fay probes that perform extraction, processing, and early aggregation of trace event data. FayLINQ also produces optimized DryadLINQ query plans and processing code for both machine-local and cluster-level aggregation and analysis.

The example in Figure 8 helps explain how FayLINQ operates, and give an overview of query execution. In the query, `kernelAllocations` constrains the set of tracepoints to those at the entry of the primary kernel memory allocation function—with the `Function` extension method operating like a `Where` clause. Then, from each tracepoint, the query retrieves the time property and the size of the allocation, which is the second argument of `ExAllocatePool` (unfortunately, PDB files do not contain symbolic argument names). Then, `allocIntervalSizePairs` is used to collect, for each tracepoint, which `period`-length interval it fell into, and integer $\log_2$ of its allocation size. These events are then grouped together into `results`, and a separate count is made of each group where both the time and $\log_2$ allocation sizes are equal, with these triples output as strings. Importantly, this final grouping applies to events from all machines, and is implemented in two phases: first on each machine, and then across all cluster machines.

Distributed tracing can be straightforwardly implemented by emitting trace events for each tracepoint invocation and collecting and processing those events centrally. One approach would be to use a flat, wide schema (the union of all possible output fields) to allow the same trace events to be output at any probe and at any tracepoint. Probes may be very simple, and need only fill out fields in the schema. Unfortunately, this is not a very viable strategy: flattened schemas lead to large trace events, and the output of trace events at

```
// Get the disaggregated set of kernel allocation trace events.
var kernelAllocations =
   PartitionedTable<FayTracepoint>
   .Get("fay://clustername)")
   .Function(kernel,"ExAllocatePool");

// For the next 10 minutes, map each allocation to a coarser period-based
// timeline of intervals and to log2 of the requested allocation size.
var allocIntervalSizePairs =
   from event in kernelAllocations
   where event.time < Now.AddMinutes(10)
   let allocSize = event.Arg(2)  // NumberOfBytes
   select new { interval = event.time/period,
                size = log2(allocSize)) };

// Group allocations by interval and log2 of the size and count each group.
var results =
   from pair in allocIntervalSizePairs
   group pair by pair into reduction
   select new { interval = reduction.Key.interval,
                logsize = reduction.Key.size,
                count = reduction.Count() };

// Map each interval/log2size/count triple to a string for output.
var output =
   results.Select( r => r.ToString() );
```

Figure 8: A FayLINQ query that summarizes the rate of different-sized kernel memory allocation requests over 10 minutes. The output indicates, for each `period`-length interval, how often allocation sizes of different magnitude were seen.

high-frequency tracepoints will incur significant load, which may easily skew measurements or even swamp the system.

Instead of the above, naive implementation approach, FayLINQ performs a number of steps to optimize the execution of queries like that in Figure 8. At a coarse granularity, these steps are:

Generic Optimizations. First, FayLINQ performs basic DryadLINQ query optimizations, like dead code removal—notably moving filtering and selection to the leaves of the query plan—i.e., towards the source of trace event data, the tracepoints.

Second, since a `fay://` data source is used, FayLINQ creates an optimized query plan, which collects trace events from Fay probes. Like with PowerShell, the query is analyzed (using a form of partial evaluation) to discover what machines, address spaces, processes, and threads, and what functions should be traced by Fay.

Greedy Optimizations. Third, the query plan optimizer greedily tries to move operations into Fay probe functions—as many as possible. For the query in Figure 8, nearly all work can be pushed into Fay probes at the query plan leaves, since the `GroupBy` operator can be decomposed into a local and a global aggregation [65].

Fourth, by default, the plan is modified to materialize Fay probe output, to make trace events persistent and fault tolerant. Fifth, a DryadLINQ plan is built for all remaining query parts. For Figure 8, this is the final, global aggregation and the computation of the `output` strings. Sixth, the code for the Fay probes, and their installation and use, is emitted as a synthetic Dryad input vertex [26].

Query Execution. Figure 9 shows how FayLINQ will efficiently execute the example in Figure 8. Figure 10 shows the term-rewriting rules used to generate this optimized query plan.

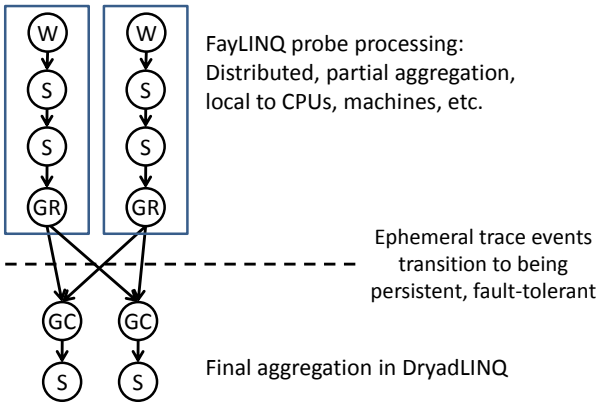

Figure 9: Optimized plan for the query in Figure 8. Symbols are as in Figure 10 and its legend (e.g., arrows show data flow).

The dotted line in Figure 9 marks the separation between Fay and DryadLINQ. At runtime, Dryad input vertices execute, start Fay tracing, and then enter a loop processing the trace events output by Fay probes. The Fay probes will usually perform some aggregation. The results of the aggregation are periodically encapsulated in ETW events and flushed to the cluster-level aggregation pipeline. Normally the aggregation results are flushed when the internal fixed-size hashtables are filled. However, the user can control the message frequency by specifying that aggregated event statistics should be flushed at least every *k* probe invocations. The payload of the ETW events is unmarshalled and decoded into .NET objects, which are further transported using the standard DryadLINQ transport mechanisms using reliable Dryad channels. The DryadLINQ part of the query runs on the cluster, taking full advantage of the fault-tolerance, scheduling and optimizations of the Dryad runtime, which is proven to scale to large clusters.

4.2.1 Probe Code Generation

The FayLINQ implementation optimizes the query plan to move data filtering, transformation, and aggregation (including GroupBy-Aggregate) from the LINQ query into Fay probes. Currently, the following LINQ statements can be executed by Fay probes: `Where`, `Select`, `Aggregate`, and `GroupBy`—as well as the many special cases of these operators, such as `Sum`, `Count`, `Average`, `Distinct`, `Take`, etc. Query parts that cannot be executed by probes are executed by DryadLINQ, on the cluster. This includes the aggregates of data from multiple machines—which, DryadLINQ will automatically perform in a tree-like fashion, when that improves performance [65].

In our current implementation, not all uses of the above LINQ operators can be transformed to execute in Fay probes. The operators must use only values (basic types and structures) and must only call static methods for which a Fay accessor or extension is available. However, Fay probes can invoke any lambda expression that uses only these basic primitives. For example, sketch-based tracing (similar to that in Chopstix [7]) can be expressed simply as

```
clusterTraceEvents
.Where(event => HCA(event.time/period, event))
```

where `HCA` is a function in an optimized, native-code Fay extension that sketches all events in each distinct time period, by updating mutable probe state. Section 5.2.3. describes further how Fay extensions can implement tracing primitives such as sketching.

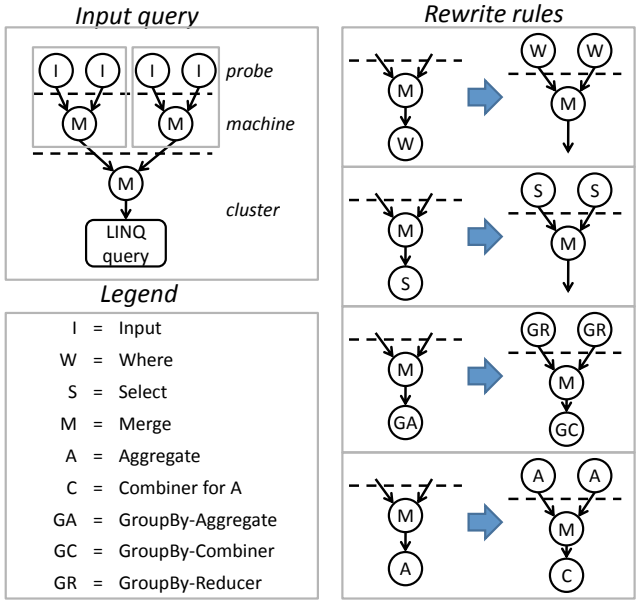

Figure 10: Term-rewriting from LINQ to Fay probes, with circles and arrows representing operators and data flow. The input operation merges the trace events from tracepoints and performs user-specified computations on that merged stream. Term-rewriting optimizations push operations closer to data sources. The first and second rewrite rules push filtering and selection ahead of merging. The last rewrite rule transforms counting into a sum of partial counts; the third rule generalizes this last one and rewrites aggregations on partial groups.

FayLINQ generates C code using syntax-directed translation from the optimized LINQ query plan. The translation proceeds naturally—e.g., `Where` translates to `if` statements, etc. More interestingly, at each tracepoint invocations, the C code may modify probe state to perform incremental updates. Since LINQ queries are essentially database views, this implementation of FayLINQ query evaluation is much like an optimized incremental view update, and makes use of database-like mechanisms [24, 65]. For example, in the query of Figure 8, at each probe invocation, the time interval and log(allocation size) are computed immediately and used to update counts in a hashtable.

Notably, `GroupBy`, when followed by aggregation, can often be translated using Fay hashtable updates. As mentioned above, this pattern—often known as map-reduce [14]—can often be decomposed into a local and global `GroupBy` operations [65]. The local reduction may then be Fay hashtable updates, while the global reduction remains in DryadLINQ. Importantly, fixed-size hashtables may be used for local aggregation, since the global aggregation can "fix" incomplete aggregations. When the hashtables are full, and insertion fails, Fay probes can output a trace event containing the hashtable data and clear it out for reuse.

5. EXPERIMENTS AND EVALUATION

We have used Fay to diagnose system behavior on both single machines and on medium-size clusters. For example, as we started using Fay we immediately noticed a performance issue where the built-in Windows command shell was CPU-bound doing continuous system calls for no good reason. Below, we retell our diagnosis of this issue as a detailed, anecdotal case study of using Fay tracing.

Windows System Call	Count	Callers
NtRequestWaitReplyPort	1,515,342	cmd.exe conhost
NtAlpcSendWaitReceivePort	764,503	CSRSS
NtQueryInformationProcess	758,933	CSRSS
NtReplyWaitReceivePort	757,934	conhost

Table 1: The processes in the command shell case study, and a count of how often they made the relevant system calls. The two calling `NtRequestWaitReplyPort` did so about equally often.

The utility of tracing and monitoring platforms has long since been established through both published results as well as through previous anecdotal case studies. In many cases, such as in the DTrace study in Section 9 of [10], an issue is first raised by some external monitoring tool that can be applied continuously to live production systems (such as an offline log analysis tool or a low-overhead, statistical profiler [9]). After such initial identification by other means, dynamic tracing may be used for detailed, manual or semi-automatic behavior analysis. Even then, tracing overheads may be too high for production systems, which often forces the issue to be reproduced on non-critical systems before it can be analyzed.

Fay tracing can be efficient enough to overturn the above paradigm and allow continuous dynamic tracing of live production systems, both before and during the analysis of any detected issues. With this in mind, the Fay primitives have been used to extend the existing tracing mechanisms in one of Microsoft's mature, scalable enterprise transaction platforms. This platform performs transactions on separate threads and, during normal operation, Fay tracing allows the properties of a random sample of transactions to be closely monitored with very low overhead. Fay tracing has little global performance impact (e.g., it does not force kernel traps), and threads that are not being traced spend few extra CPU cycles at each tracepoint, thanks to thread-specific Fay dispatching. If an issue arises, and needs to be analyzed, Fay tracing can be dynamically directed to detailed behavior analysis, and more functions and threads, usually at only a modest, acceptable increase in overhead.

The rest of this section starts off with a Fay case study, presented in the informal, anecdotal style of studies in the literature [10]. Instead of enumerating further tracing applications, we subsequently examine the flexibility of Fay tracing through the implementation of a variety of different distributed software monitoring strategies. Finally, we present experimental measurements that establish the efficiency of the Fay tracing primitives, the scalability of the Fay platform to fully-loaded clusters, and the benefits of FayLINQ query optimizations.

5.1 A Fay Performance-diagnosis Case Study

In some of our earliest Fay tracing experiments, we interactively used the Windows command shell (cmd.exe) while observing a live, real-time chart of machine-wide system-call frequencies much like that in Figure 7. Surprisingly, we observed very high frequencies for some tasks where we expected to see few system calls, such as `copy * NUL`, or `type large.txt` in a minimized window, or `dir /S >NUL`. We used Fay to investigate, as described below, and to ensure reproducibility we used only public information available outside Microsoft, such as public symbol files.

Outputting a 16 MB file of ASCII text in a minimized console window, using `type`, produced around 3.75 million system calls, and was CPU bound for a significant amount of time. We used a Fay query to aggregate by calling process, with Table 1 showing the dominant four system calls. To see how these three processes interacted, we combined their system calls and arguments into a single view, using a Fay query for a temporal join (see Section 5.2.4 and [5]). The query showed a repeated pattern: cmd.exe blocks on a port request to conhost; then, conhost blocks on a port request to the CSRSS service, which queries for process information; then, CSRSS blocks on a port send to conhost, which unblocks it; finally, conhost makes a request back to cmd.exe, unblocking it. These were clearly Windows Local Procedure Calls (LPC) spanning the three processes [49].

Fay tracing showed some LPC rounds to be a result of the well-documented `WriteConsole` function outputting a line (of 80 characters or less) to the console, However, we saw an even greater number of LPC rounds caused by a function `FileIsConsole`. By Fay tracing of arguments, we could establish that, for every single line of output, the command shell would check twice whether `stdout` was directed to the console window, or not, at the cost of two LPC rounds and many context switches and system calls. Even more surprisingly, we saw those checks and LPC rounds continue to occur when output was directed to a file—causing nearly a million system calls when we used `type` to output our 16 MB text file to the special file `NUL`, for example.

We also used Fay tracing to investigate other frequent system calls, by collecting and counting their distinct arguments, return values, and user-mode stack traces. This data indicated that the calls to `NtQueryInformationProcess` in Table 1 were due to an internal CSRSS function, `IsConhost`, inspecting an undocumented property (number 49) of the cmd.exe process. The arguments and return values strongly indicated that CSRSS was retrieving this property, on every LPC round, to verify that an intermediary conhost was still hosting the console for an originating cmd.exe.

The above behavior also occurs for commands run in shell scripts, which often redirect large amounts of output to files or to `NUL`. The most frequent system calls simply retrieve information from the kernel, and user-mode processes can typically cache such data or read it via a "shared user data page" (like the one exposed by the Windows kernel) that gives a read-only, up-to-date view of data maintained elsewhere [49]. Thus, concretely, our Fay case study identified potential reductions in the LPC rounds and context switches required for each line of command shell output, which could eliminate most of the system calls in Table 1. However, command shell output is usually not a critical performance issue, and its implementation in Windows appears tuned for reliability and simplicity; thus, while insightful, our observations are not sufficient to justify immediate changes to user-mode or kernel-mode code.

5.2 Reimplementing Tracing Strategies

To stress the generality of Fay tracing, we reimplemented several existing, custom tracing strategies on top of the Fay tracing platform. This reimplementation was done with minimal effort, by leveraging Fay extensions and the high-level queries of FayLINQ. We used two DryadLINQ clusters: one with 12 machines with dual 2GHz AMD Opteron 246 processors and 8GB of memory, and another with 128 machines with two 2.1GHz quad-core AMD Opteron 2373EE processors and 16GB of memory, both running Windows Server 2008 R2 Enterprise. Below we describe our implementations and (in some cases) the results of applying these monitoring strategies to our clusters.

5.2.1 Distributed Performance Counters

A common strategy for distributed monitoring is to count the events generated across all machines of a cluster. Fay tracing can trivially implement this strategy by applying the appropriate aggregation operations to any metrics on the trace events available to probes on a single machine. Unlike traditional performance counters, Fay tracing allows both user-controllable and efficient aggregation. For instance, with small changes, the query shown on page 1 can provide per-process, per-thread, and per-module statistics on all cluster activity in both user-mode and the kernel. Such monitoring of memory allocation cannot be achieved with traditional Windows performance counters, even on a single machine.

5.2.2 Automatic Analysis of Cluster Behavior

Several recent systems have applied automatic machine-learning techniques to extract useful information from activity signatures collected across a cluster [30, 63]. We used FayLINQ to perform an analysis similar to that of Fmeter [30] on our cluster, while it executed an unrelated map-reduce workload (N-gram generation).

A single FayLINQ query sufficed to express the entire trace collection, the k-means clustering of the collected traces, and the analysis of the traced workload using those machine-learning results. This query collects periodic system-call-frequency histograms for the 402 system calls in the Windows kernel, at a granularity of around 1 second. Collecting this information does not measurably affect CPU utilization or machine performance, since FayLINQ synthesizes efficient, stateful kernel probes that maintain counts per system call. The data-analysis part of the FayLINQ query reduced the dataset dimensionality by applying k-means clustering (with k set to 5) on the histograms, using published distributed machine-learning techniques for DryadLINQ [33]. Then, the FayLINQ query associated the workload activity in each period with the closest of the five centroids resulting from the k-means clustering. Finally, the FayLINQ query output results into a visualization tool to produce the chart in Figure 11.

Figure 11 shows activity on all machines, during execution of the map-reduce workload. All activity periods are associated with their most similar k-means centroid, each of which has a unique color and a (manually-added) label: io, idle, memory, cpu, or outlier. By comparing against the map-reduce job plan, it can be seen that Figure 11 precisely captures the workload's different processing stages, as annotated at the bottom of the figure—including the use of five machines in the first stage, and ten machines for the second, and the final stages of io-intensive data reduction. Here, we compared against ground truth from a known map-reduce job plan. However, in most cases, no such explicit plan exists, and similar FayLINQ analysis could clarify the processing phases of even complex, opaque distributed services.

5.2.3 Predicated and Windowed Trace Processing

Some systems implement stateful or non-deterministic tracing primitives that are not so easily expressed as pure, functional LINQ queries. Nonetheless, FayLINQ can utilize Fay's extensibility to provide such primitives and incorporate their results into tracing queries. Concretely, users of FayLINQ can implement any probe extension by providing an arbitrary C function, or make use of our library of such extensions.

Fay extensions can use optimized machine code to evaluate the state of a traced system in any manner, whether complex or stateful. Thus, Fay can offer a efficient, general form of predication and

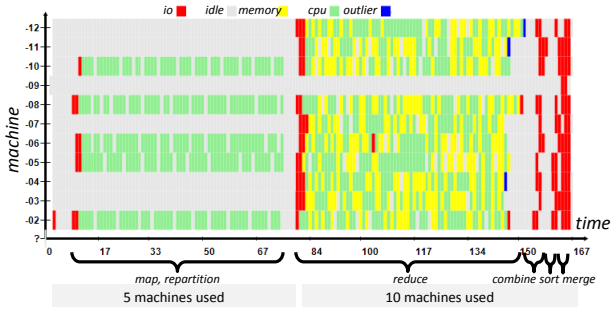

Figure 11: The result of FayLINQ analysis of cluster behavior while executing a map-reduce job. This 2D plot shows the results of automatic k-means clustering of system-call histograms collected periodically across all machines. The X axis shows time, machines are on the Y axis, and each period is colored according to its representative k-means centroid.

speculation, and support tracing that cannot even be expressed in language-restricted platforms like DTrace [11]. To achieve similar functionality, other tracing platforms require the evaluation code to be fully trusted—thereby leaving the traced system fully exposed to any reliability and security issues in that code.

In particular, we have implemented Fay probe extension functions for Chopstix sketches [7], to provide statistical, non-uniform sampling of low- and high-frequency events with low overhead. FayLINQ sketching uses a hashtable of counters to ensure that trace events are output in logarithmic proportion to the total number of occurrences. While our sketching library implementation hides some complexity, FayLINQ users need only invoke a simple `HCA` function to use the library, much as in the code on page 10.

We have also implemented probe extensions for temporal processing on trace event streams, such as windowed (sliding or staggered) computations. For example, our simple `MovingAverage` extension for computing moving averages is used in the below query, which emits all kernel memory allocations that are 10 times larger than the current local moving average:

```
cluster.Function("ExAllocatePoolWithTag")
.Select(event => GetArg(2))  // allocation size
.Select(sizeArg => new {
                  average = MovingAverage(sizeArg),
                  size = sizeArg })
.Where(alloc => alloc.size > 10*alloc.average);
```

5.2.4 Tracking Work Across Distributed Systems

Several distributed monitoring platforms track all the activity performed for work items, as those items are processed by different parts of the system [5, 50]. Often, such tracking is done via passive, distributed monitoring, combined with "temporal joins" to infer dynamic dependencies and flow of work items. Fay tracing can easily support such monitoring, by encoding temporal joins as recursive queries that transitively propagate information, and by iterating to convergence. We have used FayLINQ to track work in a distributed system by monitoring and correlating sent and received network packets, to analyze the traffic matrix of DryadLINQ workloads.

5.2.5 Tracing Across Software Abstractions

We used Fay to redo a study of the Windows timer interfaces and mechanisms; the original study [43] was done by modifying Win-

Experiment	Fay	Solaris DTrace	OS X DTrace	Fedora STap
km	220	1717	1805	1129
um `call`	197	1557	2565	9009
um `jmp`	155			
um `call` **deep**	431	1683	2813	9384
um `jmp` **deep**	268			

Table 2: Overhead in CPU cycles per call to a traced function. Here, km is kernel mode, um is user mode, and deep builds a 20-deep stack before each call. Fay dispatches using inline `call` or `jmp` instructions; other platforms trap to the kernel.

dows source code. Starting with the low-level, kernel timer interfaces `KeSetTimer`, `KeSetTimerEx`, and `KeCancelTimer`, we used FayLINQ to trace timer usage. For each use, we grouped by return addresses on the call stack and sorted to identify common callers, thereby identifying the small number of modules and functions that are the primary users of `KeSetTimer`, etc. We then iterated, by creating a larger, recursive FayLINQ query, predicated to generate trace events only in certain contexts, and discovered 13 sets of timer interfaces in Windows, such as `ZwUserSetTimer`. Close, manual inspection revealed that those interfaces were based on five separate timer wheel implementations [59].

5.3 Performance Evaluation

To assess the efficiency and scalability of our Fay implementation, we measured the performance of Fay tracing and its mechanisms for instrumentation, inline dispatching, and safe probe execution. The experiments ran on an iMac with a 3.06GHz Intel E7600 CPU and 8GB of RAM. We configured this machine to run 64-bit versions of Windows 7 Enterprise, Mac OS X v10.6, Fedora 15 Linux (kernel version 2.6.40-4.fc15), and Oracle Solaris 11 Express, in order to directly compare Fay tracing against DTrace, on two platforms, and against SystemTap (version 1.5/0.152) on Linux.

5.3.1 Microbenchmarks

To measure the cost of dispatching and executing an empty probe, we created a user-mode microbenchmark that contains an empty function `foo`, which it calls in a tight loop. We measured its running time both with, and without, Fay tracing of `foo` using an empty probe. We also created a microbenchmark that invokes a trivial system call in a tight loop, and where we traced the kernel-mode system call handler. (We used the `getpid` system call, except on Windows where we used `NtQuerySystemInformation` with an invalid parameter to minimize the work it performed.)

We also wanted to measure the effects of branch-misprediction caused by the stack manipulation of the Fay `call` dispatcher (see Section 3.2). Therefore, we created variants of the microbenchmarks that call `foo` via a sequence of 20 nested functions—forcing 20 extra stack frames to be unwound at each `foo` tracepoint.

Table 2 shows the results of our microbenchmarks, with time measurements converted to CPU cycle counts. Fay takes around 200 cycles per call and, as expected, dispatching using `jmp` is noticeably faster than Fay `call` dispatcher. If a thread is not being traced, this work can be cut in half, and the Fay `call` dispatcher adds only about 107 cycles per call. In both of these cases, the hashtable lookup of tracepoint descriptors accounts for roughly 40 cycles. The experiments for DTrace and SystemTap were run using

	MD5	lld	hotlist
Measured Fay XFI slowdown	184%	552%	1387%
XFI slowdown from [18]	101%	346%	798%

Table 3: Slowdown due to XFI for three benchmarks. The Fay XFI variant is much simpler, but has nearly twice the overhead.

	Fay	Solaris DTrace	OS X DTrace
Traced functions	8001	31998	9341
Function calls (millions)	60	253	306
Running time w/tracing	28.0	103.2	149.6
Slowdown	2.8x	17.2x	26.7x

Table 4: Instrumenting all kernel functions to test scalability.

function boundary tracing and per-CPU collection and aggregation. Compared to Fay, the other tracing platforms generally required a bit less than an order-of-magnitude more cycles.

Next, we compare the execution time of three benchmark probes with and without XFI rewriting, summarizing the results in Table 3. This experiment replicates parts of Table 1 in [18] (slowpath with read and write protection). Our overhead is larger than that in [18], which is not surprising, since we targeted simplicity in our implementation. However, Fay XFI performance still compares favorably to that of safe interpreters like those used in DTrace [47].

5.3.2 Scalability and Impact of Optimizations

We have used Fay to trace all the 8,001 hotpatchable functions in the Windows kernel and increment a per-CPU counter at each tracepoint, to count the total kernel function invocations. Such tracing does not occur often, but can be useful. An example application, that has seen practical use in other tracing platforms, is the tracing of all kernel activity due to a specific kernel module, such as a network driver, or a specific interrupt handler [17], and the generation of function call graphs for later visualization [16].

Table 4 displays the results of tracing a workload that copied all the RFC text files between ramdisk directories, deleted the new copies, and repeated this a fixed number of times. Fay scales very well, and using it to trace the vast majority of Windows kernel functions leaves the machine perfectly responsive and about 2.8 times slower on a benchmark that spends 75% of its time executing kernel code. Notably, the scale of this experiment creates a worst-case scenario for Fay performance: the Fay `call` dispatcher adds an extra stack frame on every kernel function invocation, and suffers a branch-prediction miss on every function return.

The slowdown factors for DTrace are significantly higher, on both Solaris and Mac OS X. However, slowdown factors are not directly comparable, since Fay and DTrace are instrumenting different operating systems. Trying to repeat the experiment with SystemTap resulted in a hung Linux kernel, apparently due to a long-standing, well-known SystemTap bug [58].

We tested the scalability, robustness, and optimizations of Fay tracing by utilizing our 128-machine, 1024-core cluster for a benchmark that makes 50 million memory allocations per machine. In the benchmark, each thread allocates and clears 10 heap-memory regions, of a random size between 1 byte and 16 kilobytes, yields with a `Sleep(0)`, clears and frees the 10 regions, and then loops.

We measured all configurations of partitioning per-machine work over 1, 2, 5, or 10 processes and 1, 5, 10, 50, 100, 500, or 1000 concurrent threads in each process. These configurations ran on the entire, dedicated cluster, spreading 6.4 billion allocations between 128 to 1,280,000 threads, each at 100% CPU utilization when running. The benchmark took between 30 seconds and 4 minutes to run, depending on the configuration—not counting unpredictable delays and high variance caused by the cluster's job scheduler.

Using a FayLINQ query to measure total allocated memory added an overhead of 1% to 11% (mean 7.8%, std.dev. 3.8%) to the benchmark running time. The numbers matched our expectation: per allocation, the benchmark spent approximately a couple of thousand cycles, to which Fay tracing added a couple of hundred cycles, as per Figure 2—but, as the number of processes and threads grew, increased context switches and other costs started masking some of Fay's overhead. The time to initialize tracing, and install Fay probes, grew as processes increased from 1 to 10, going from 1.5 to 7 seconds. Whether or not Fay tracing was enabled, the benchmark had similar variance in CPU time (mean std.dev. 2%, max std.dev. 6%) and wall-clock time (mean std.dev. 10%, max std.dev. 33%), both per-process and per-thread.

We exercised the fault-tolerance of Fay tracing by randomly killing threads, processes, or machines running the benchmark. When a thread dies, all its thread-local Fay probe state is lost, if it has not already been sent as a trace event. Machine-local Fay aggregation continued unimpeded by failure of benchmark threads or processes. Even upon the failure of machines, the Dryad fault-tolerance mechanisms would ensure that cluster-level aggregation continued. Thus, the results of our FayLINQ query were perturbed in proportion to our violence. In addition, the data lost for any thread could be bounded by having Fay probes periodically send off their data as ETW trace events. For our benchmark FayLINQ query, probe state was sent as trace events every 100 memory allocations, at the cost of 1% extra Fay tracing overhead.

In the limit, a trace event might need to be sent at every tracepoint invocation, if the work of a tracing query was completely unsuited to Fay probe processing. To assess the benefits of early aggregation and FayLINQ optimizations, we modified our benchmark to measure such high-frequency trace events. With nearly half-a-million Fay trace events a second, and no probe processing, the benchmark's tracing overhead increased to between 5% and 163% (average 67%, std.dev. 45%). However, most of those trace events were lost, and not accounted for in the result of our FayLINQ query.

These lost trace events were surprising, since our Fay implementation minimizes the risk of data loss, both by dynamically tuning ETW buffer size, and also by running time-critical Fay activity like trace-event processing on Windows threads with high enough priority. Upon inspection, we discovered that the real-time, machine-local FayLINQ aggregation process that converts ETW trace events to .NET objects—rather slowly, on a single thread—was completely unable to handle the high event rate. FayLINQ can be manually directed to stream trace events directly to disk, into ETW log files, processed by later, batch-processing parts of the query plan. We attempted this next, but failed again: each ETW log file record is about 100 bytes, which at 50 million events, in less than four minutes, exceeded our disk bandwidth. Even though consuming data at high rates is intrinsically difficult, these results clearly indicated that FayLINQ was lacking in its support for high-event-rate tracing. So, we enhanced Fay with a custom, real-time ETW consumer thread that efficiently streams just the Fay payload of ETW events (4 bytes in our benchmark) directly to disk. After this, FayLINQ could return correct query results, by generating a plan that processes the disk files subsequent to the benchmark run.

To further evaluate the benefits of FayLINQ query-plan optimizations, we reran the experiment from Section 5.2.2 with the term-rewriting in Figure 10 turned off. While Fay tracing previously had no measurable performance effects, unoptimized tracing significantly increased the workload completion time, e.g., due to the addition of (a near-constant) 10% of CPU time being spent on kernel-mode trace event processing. Also, the lack of early-aggregation optimizations lead to a high event rate (more than 100,000 events/second, for some phases of the workload). Thus, we again had to direct FayLINQ to create query plans that stored trace events first on disk, and finished processing later. Even then, several times more data was received and processed at the higher-levels of the FayLINQ aggregation pipeline.

6. RELATED WORK

Fay is motivated by the many attractive benefits of the DTrace platform [11], while Fay's fundamental primitives are more like those of SystemTap [45] and Ftrace [48].

Fay makes use of, and integrates with a number of technologies from Microsoft Windows [49], including Event Tracing for Windows [41], PowerShell [55], Vulcan [54], Hotpatching [34], Structured Exception Handling [44], and the Driver Model [40].

Dynamic Instrumentation Systems Fay is related to several systems that perform dynamic instrumentation: KLogger [20], PinOS [8], Valgrind [39], scalable tracing on K42 [62], Ftrace and SystemTap on Linux [45, 48], Solaris DTrace [11], the NTrace prototype [42], and Detours for the Win32 interface [25].

The Fay probe dispatcher is related to new tracing tools that make use of inline mechanisms, not traps. On Linux, Ftrace [48] provides tracing based on fast, inline hooks placed by compiling the kernel with special flags. On Windows, the NTrace research project leverages hotpatching [42], but does so via a custom, modified kernel. Compared to Fay, the Ftrace and NTrace mechanisms offer more limited functionality, are likely to be less efficient, and provide neither safe extensibility nor a high-level query interface.

Safe Operating Systems Extensions Fay is an example of a system that implements safe operating systems extensions using software-based techniques [6]. This is not a new idea. Indeed, Fay has striking similarities to the SDS-940 Informer profiler developed at the end of the 1960's [15]. Other systems and techniques for providing safe system extensibility include Typed Assembly Language [37], Proof-Carrying Code [38], as well as Software-based Fault Isolation (SFI) [61], and its implementations in MiSFIT [52], Native Client [64], and similar systems [18].

Declarative Tracing and Debugging The Fay integration with DryadLINQ is related to several prior efforts to support declarative or relational queries of software execution traces. In particular, Fay is related to declarative tracepoints [12], PQL [31], and PTQL [23], and also to work in aspect-oriented programming [3].

In the trade-off between creating a domain-specific language and using a generic language, such as LINQ, we have opted towards the latter. Embedded knowledge about the semantics of traces (e.g.,

time, procedure nesting, etc.) can make the evaluation of some queries more efficient. Probes should be able to aggregate and reduce data as much as possible, while relegating expensive computations to external systems. Here, we believe that FayLINQ strikes a good balance.

Large-scale, Distributed Tracing Large-scale, distributed tracing, data collection and debugging [28, 53] is a highly active area, with several existing, attractive systems, and one deployed across a billion machines [22]. Of particular relevance are recent systems, like Chopstix [7], and Flight data recorder [60], as well as their predecessor DCPI [9] and its recent distributed analogue GWP [46]. Similarly, earlier work such as Magpie [5] on tracing requests across activities has recently been extended to the datacenter [50]. Finally, also highly relevant is work from the high-performance community for tracing in parallel systems [27, 32], and the techniques of stream-processing platforms [4]. Flume [21] is a log collection system that allows the transformation and filtering of log data, similar in some aspects to simple FayLINQ queries.

7. CONCLUSIONS

Fay is a flexible platform for the dynamic tracing of distributed systems. Fay is applicable to both user- and kernel-mode activity; our Fay implementation for x86-64 Windows can be applied even to live, unmodified production systems. Users can utilize Fay tracing through several means, which include traditional scripting. Fay users can also safely extend Fay with new, efficient tracing primitives, without affecting the reliability of traced systems.

Distinguishing Fay from previous tracing platforms is its disaggregated execution, even within a single machine, as well as its safe, efficient extensibility, and its deep integration with a high-level language and distributed runtime in FayLINQ—all of which facilitate large-scale execution trace collection and analysis.

Building on the above, FayLINQ provides a unified, declarative means of specifying what events to trace, as well as the aggregation, processing, and analysis of those events. As such, FayLINQ holds the potential to greatly simplify the investigation of performance, functionality, or reliability issues in distributed systems. Through benchmarks and experiments, we have demonstrated the efficiency and flexibility of Fay distributed tracing, and also shown how a few simple FayLINQ queries can offer the same functionality as that provided by custom mechanisms in other tracing platforms.

Acknowledgments

We thank the SOSP PC for their useful comments and our shepherd, Greg Ganger, for his gracious help. Fay's hotpatch-based instrumentation was designed and implementated in collaboration with the Microsoft Windows Fundamentals group: Anshul Dhir, Haifeng He, Bradford Neuman, and Dragoş Sâmbotin contributed directly to the implementation, starting from a prototype by Neeraj Singh. Gloria Mainar-Ruiz helped with the Fay XFI implementation and experiments. Jacob Gorm Hansen and Jorrit Herder worked on previous XFI variants and applications in Windows, respectively, for KMDF device drivers and for safe kernel-mode interrupt handlers for UMDF device drivers. The DryadLINQ integration benefited from work by Pradeep Kumar Gunda. Jon Currey helped with cluster-experiment infrastructure. Michael Vrable provided camera-ready support. The up-to-date version of SystemTap that we measured was provided by William Cohen, of Red Hat, Inc.

8. REFERENCES

[1] J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *PLDI*, 2011.

[2] Apache. Hadoop project. `http://hadoop.apache.org/`.

[3] P. Avgustinov, J. Tibble, E. Bodden, L. Hendren, O. Lhotak, O. de Moor, N. Ongkingco, and G. Sittampalam. Efficient trace monitoring. In *OOPSLA*, 2006.

[4] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD*, 2005.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, 2004.

[6] B. N. Bershad, S. Savage, P. Pardyak, D. Becker, M. Fiuczynski, and E. G. Sirer. Protection is a software issue. In *HotOS*, 1995.

[7] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *OSDI*, 2008.

[8] P. P. Bungale and C.-K. Luk. PinOS: A programmable framework for whole-system dynamic instrumentation. In *VEE*, 2007.

[9] M. Burrows, Ú. Erlingsson, S.-T. A. Leung, M. T. Vandevoorde, C. A. Waldspurger, K. Walker, and W. E. Weihl. Efficient and flexible value sampling. In *ASPLOS*, 2000.

[10] B. Cantrill. Hidden in plain sight. *ACM Queue*, 4, 2006.

[11] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conf.*, 2004.

[12] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo. Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks. In *SenSys*, 2008.

[13] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *PLDI*, 2010.

[14] J. Dean and S. Ghemawat. MapReduce: A flexible data processing tool. *Comm. ACM*, 53(1), 2010.

[15] P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. In *IFIP*, 1971.

[16] Eclipse. Callgraph plug-in. `http://wiki.eclipse.org/Linux_Tools_Project/Callgraph/User_Guide`.

[17] F. C. Eigler. Systemtap tutorial, Dec. 2010. `http://sourceware.org/systemtap/tutorial/`.

[18] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI*, 2006.

[19] Ú. Erlingsson, M. Manasse, and F. McSherry. A cool and practical alternative to traditional hash tables. In *Workshop on Distributed Data and Structures*, 2006.

[20] Y. Etsion, D. Tsafrir, S. Kirkpatrick, and D. G. Feitelson. Fine grained kernel logging with KLogger: Experience and insights. In *EuroSys*, 2007.

[21] Flume: Open source log collection system. `http://github.com/cloudera/flume`.

[22] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt.

Debugging in the (very) large: Ten years of implementation and experience. In *SOSP*, 2009.

[23] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA*, 2005.

[24] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM Intl. Conf. on Management of Data*, 1993.

[25] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *USENIX Windows NT Symposium*, 1998.

[26] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[27] G. L. Lee, M. Schulz, D. H. Ahn, A. Bernat, B. R. de Supinskil, S. Y. Ko, and B. Rountree. Dynamic binary instrumentation and data aggregation on large scale systems. *Intl. Journal on Parallel Programming*, 35(3), 2007.

[28] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *PLDI*, 38(5), 2003.

[29] F. Marguerie, S. Eichert, and J. Wooley. *LINQ in action*. Manning Publications Co., 2008.

[30] T. Marian, A. Sagar, T. Chen, and H. Weatherspoon. Fmeter: Extracting Indexable Low-level System Signatures by Counting Kernel Function Calls. Technical Report `http://hdl.handle.net/1813/23568`, Cornell University, Computing and Information Science, 2011.

[31] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: A program query language. In *OOPSLA*, 2005.

[32] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia distributed monitoring system: Design, implementation and experience. *Intl. Journal on Parallel Computing*, 30, 2003.

[33] F. McSherry, Y. Yu, M. Budiu, M. Isard, and D. Fetterly. *Scaling Up Machine Learning*. Cambridge U. Press, 2011.

[34] Microsoft Corp. Introduction to hotpatching. *Microsoft TechNet*, 2003.

[35] Microsoft Corp. Kernel patch protection: Frequently asked questions. *Windows Hardware Developer Central*, 2006. `http://www.microsoft.com/whdc/driver/kernel/64bitpatch_FAQ.mspx`.

[36] Microsoft Corp. WDK and developer tools. *Windows Hardware Developer Central*, 2010. `http://www.microsoft.com/whdc/DevTools/default.mspx`.

[37] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *POPL*, 1998.

[38] G. C. Necula. Proof-carrying code. In *POPL*, 1997.

[39] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[40] W. Oney. *Programming the Microsoft Windows Driver Model*. Microsoft Press, 2002.

[41] I. Park and R. Buch. Improve debugging and performance tuning with ETW. *MSDN Magazine*, April 2007.

[42] J. Passing, A. Schmidt, M. von Lowis, and A. Polze. NTrace: Function boundary tracing for Windows on IA-32. In *Working Conference on Reverse Engineering*, 2009.

[43] S. Peter, A. Baumann, T. Roscoe, P. Barham, and R. Isaacs. 30 seconds is not enough!: A study of operating system timer usage. In *EuroSys*, 2008.

[44] M. Pietrek. A crash course on the depths of Win32 structured exception handling. *Microsoft Systems Journal*, 1997.

[45] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Ottawa Linux Symposium*, 2005.

[46] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4), 2010.

[47] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy. The structure and performance of interpreters. In *ASPLOS*, 1996.

[48] S. Rostedt. Debugging the kernel using Ftrace. *lwn.net*, 2009.

[49] M. E. Russinovich, D. A. Solomon, and A. Ionescu. *Microsoft Windows Internals*. Microsoft Press, 2009.

[50] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report 2010-1, Google Inc., 2010.

[51] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *MICRO*, 1998.

[52] C. Small and M. I. Seltzer. MiSFIT: Constructing safe extensible systems. *IEEE Concurrency: Parallel, Distributed and Mobile Computing*, 6(3), 1998.

[53] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: Global views of distributed program execution. In *SenSys*, 2009.

[54] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[55] W. Stanek. *Windows PowerShell(TM) 2.0 Administrator's Pocket Consultant*. Microsoft Press, 2009.

[56] M. Strosaker. Sample real-world use of SystemTap. `http://zombieprocess.wordpress.com/2008/01/03/sample-real-world-use-of-systemtap/`.

[57] SystemTap. Examples. `http://sourceware.org/systemtap/examples/`.

[58] SystemTap. Bug 2725: function("*") probes sometimes crash & burn, June 2006. `http://sources.redhat.com/bugzilla/show_bug.cgi?id=2725`.

[59] G. Varghese and A. Lauck. Hashed and hierarchical timing wheels. *IEEE/ACM Transactions on Networking*, 5(6), 1997.

[60] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev. Flight data recorder: Monitoring persistent-state interactions to improve systems management. In *OSDI*, 2006.

[61] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[62] R. W. Wisniewski and B. Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Supercomputing*, 2003.

[63] D. B. Woodard and M. Goldszmidt. Model-based clustering for online crisis identification in distributed computing. Technical Report TR-2009-131, MSR, 2009.

[64] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Comm. ACM*, 53(1):91–99, 2010.

[65] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP*, 2009.

[66] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. G. Kumar, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.