Events Can Make Sense

Maxwell Krohn (*MIT CSAIL*), Eddie Kohler (*UCLA*) and M. Frans Kaashoek (*MIT CSAIL*)
`{krohn,kaashoek}@csail.mit.edu, kohler@cs.ucla.edu`

ABSTRACT

Tame is a new event-based system for managing concurrency in network applications. Code written with Tame abstractions does not suffer from the "stack-ripping" problem associated with other event libraries. Like threaded code, tamed code uses standard control flow, automatically-managed local variables, and modular interfaces between callers and callees. Tame's implementation consists of C++ libraries and a source-to-source translator; no platform-specific support or compiler modifications are required, and Tame induces little runtime overhead. Experience with Tame in real-world systems, including a popular commercial Web site, suggests it is easy to adopt and deploy.

1 INTRODUCTION

This paper introduces Tame, a system for managing concurrency in network applications that combines the flexibility and performance of events with the programmability of threads. Tame is yet another design point in a crowded space, but one that has proven successful in real-world deployments. The system is, at heart, an event-based programming library that frees event developers from the annoyance of "stack ripping" [1]. We have implemented Tame in C++ using libraries and source-to-source translation, making Tame deployable without compiler upgrades.

Threads are the more popular strategy for managing concurrency, but some situations (and programmers) still call for events [7, 13, 24, 28, 34, 39]. Applications with exotic concurrency, such as multicast, publish/subscribe, or TCP-like state machines, might find threads insufficiently expressive [37]. Certain contexts do not support threads or blocking [6, 21]. On new platforms, portability can favor events, which require only a `select` call and no knowledge of hardware-specific stack or register configuration [11]. Finally, some event-based servers perform better and use less memory than threaded competitors [18, 24–26].

But a key advantage of events—a single stack—is also a liability. Sharing one stack for multiple tasks requires stack ripping, which plagues the development, maintenance, debugging and profiling of event code [1]. The programmer must manually split (or "rip") each function that might block (due to network communication or disk I/O), as well as all of its ancestors in the call stack. Ripping a function obscures its control flow [6] and complicates memory management.

However, the right abstractions can capture events' expressivity while minimizing the headaches of stack ripping [30]. The Tame system introduces powerful abstractions with implementation techniques suitable for high-performance system programming. The specific contributions of the Tame system are:

1. A high-level, type-safe API for event-based programming that frees it from the stack-ripping problem but is still backwards compatible with legacy event code.
2. A new technique to incorporate threads and events in the same program.
3. A maintainable and immediately deployable implementation in C++, using only portable libraries and source-to-source translation.
4. An automated memory management scheme for events that does not require garbage collection.

Our experience with Tame has shown the interface sufficient to build and run real systems. Programmers other than the authors rely on Tame in educational assignments, research projects [36], and even a high-traffic commercial Web site [16].

2 RELATED WORK

The research systems most closely related to Tame are Capriccio [38] and the work of Adya et al. [1]. Capriccio is a cooperative threading package that exports the POSIX thread interface but looks like events to the operating system: it uses sophisticated stack management to make one stack appear as many, saving on cycles and memory. However, the Capriccio system strives to equal events only in terms of *performance* and not in terms of *expressivity*; its authors note that the thread interface is less flexible than that of events [37].

Adya et al.'s system is a way to combine event-based and threaded code in the same address space. The key insight is that a program's style of stack management (automatic or manual) is orthogonal to its style of task management (cooperative or preemptive) and that most literature on events and threads mistakenly claims they are linked. As in Adya's system, a Tame program can be expressed in a syntax that has readable automatic stack management (like threads) yet has explicit cooperative task management (like events). Tame differs because it extends automatic stack management to *all* event code, while "hybrid" event code in Adya's system still requires manual stack management. Other systems like SEDA [40] use threads and events in concert to achieve flexible scheduling and intraprocess concurrency. Tame is complementary to such hybrid systems and can be used as an implementation technique to simplify their event code.

Many other systems attempt to improve threads' scalability and efficiency. NPTL in Linux [9] and I/O completion ports in Windows [22] improve the performance of kernel threads; we compare Tame with NPTL in our evaluation. Practical user-level cooperative threading packages include Gnu PTH, which focuses on portability [12], and StateThreads, which focuses on performance [31].

Existing practical event libraries fall into several categories. The most primitive, such as `libevent` [27], focus exclusively on abstracting the interface to OS events (i.e., `select` vs. `poll` vs. `epoll` vs. `kqueue`), and don't simplify the construction of higher-level events, such as RPC completions. The event libraries integrated with GUI toolkits, such as Motif, GTK+, and Qt, support higher-level events, but are of course tuned for GUIs rather than general systems programming. The type-safe *libasync* event library for C++ is the basis of our work [21, 41].

The protothreads C-preprocessor library [11] gives the illusion of threads with only one stack. Protothreads are useful in resource-constrained settings such as embedded devices and sensor networks, but lacking stacks or closures, they must use global variables to retain state and therefore are not suited to building composable APIs. The Tame system shares implementation techniques with protothreads and similar C coroutine libraries [10, 11], as well as the `porch` program checkpointer [29].

The Tame language semantics draw from a rich body of previous work on parallel programming [32]. Like condition variables [14], Tame's `events` allow signaling and synchronization between different parts of a program, but unlike condition variables, `events` do not require locks (or threads, for that matter). Many parallel programming languages have constructs similar to Tame's `twait`: Occam has `PAR` [17], and Pascal-FC has `COBEGIN` and `COEND` [8].

Tame also borrows ideas such as closures and function currying from functional languages like Lisp [33] and Haskell [15]. Previous work in modeling threads and concurrency in functional languages, such as Haskell and ML, has noted a correspondence between continuations and threads. A user-level thread scheduler essentially chooses among a set of active continuations; blocking adds the current continuation to this set and invokes the scheduler. For instance, Claessen uses monads in Haskell to implement threading [5]. Li and Zdancewic extend Claessen's technique to combine threads and events [20]. Concurrent ML (CML) uses continuations to build a set of concurrency primitives much like those of Tame [30]. Tame and CML have similar `events`, Tame's `rendezvous` shares some properties with CML's `choose` operator, and Tame's `twait` is analogous to CML's `sync`. There are differences in performance and function. CML `events` are effectively continuations and preserve the equivalent of an entire call stack, while Tame `events` preserve only the top-level function's closure, and CML has no direct equivalent for Tame's user-supplied event IDs—instead the CML user must manipulate `event` objects directly. Tame's constructs have similar power but are efficiently implementable in conventional systems programming languages like C++.

```
// Threads
void wait_then_print_threads() {
   sleep(10);         // blocks this function and all callers
   printf("Done!");
}

// Tame primitives
tamed wait_then_print_tame() {
    tvars { rendezvous<> r; }
    event<> e = mkevent(r);      // allocate event on r
    timer(10, e);    // cause e to be triggered after 10 sec
    twait(r);      // block until an event on r is triggered
                // only blocks this function, not its callers!
    printf("Done!");
}

// Tame syntactic sugar
tamed wait_then_print_simple_tame() {
   twait { timer(10, mkevent()); }
   printf("Done!");
}
```

Figure 1: Three functions that print `Done!` after ten seconds. The first version uses threads; the second Tame version is essentially as readable.

3 Tame Semantics

Tame makes easy concurrency problems easy to express in events (as they were easy to express in threads). Figure 1 shows three implementations of a trivial function; the second Tame version is indeed close to the threaded version in code length and readability. The rest of this section describes the Tame primitives and syntactic sugar. We also show through examples how the full power of Tame simplifies the expression of *hard* concurrency problems, and how Tame allows users to develop composable solutions for concurrency problems (harder to express correctly in threads).

3.1 Overview

Tame introduces four related abstractions for handling concurrency: *events*, *wait points*, *rendezvous*, and *safe local variables*. They are expressed as software libraries whenever possible, and as language extensions (via source-to-source translation) when not.

First, each **event** object represents a future occurrence, such as the completion of a network read. When the expected occurrence actually happens—for instance, a packet arrives—the programmer *triggers* the event by calling its `trigger` method.

The `mkevent` function allocates an event of type `event`<T>, where T is a sequence of zero or more types.

This event's `trigger` method has the signature `void trigger(`*T*`)`. Calling `trigger(`*v*`)` marks the event as having occurred, and passes zero or more results *v*, which are called *trigger values*, to whomever is expecting the event. For example:

```
rendezvous<> r;  int i = 0;
event<int> e = mkevent(r, i);
e.trigger(100);
assert(i == 100);                    // assertion will succeed
```

When triggered, `e`'s `int` trigger value is stored in `i`, whose type is echoed in `e`'s type.

The **wait point** language extension, written `twait`, blocks the calling function until one or more events are triggered. Blocking causes a function to return to its caller, but the function does not complete: its execution point and local variables are preserved in memory. When an expected event occurs, the function "unblocks" and resumes processing at the wait point. By that time, of course, the function's original caller may have returned. Any function containing a wait point is marked with the `tamed` keyword, which informs the caller that the function can block.

The first, and more common, form of wait point is written "`twait {` *statements*`; }`". This executes the *statements*, then blocks until *all* events created by `mkevent` calls in the *statements* have triggered. For example, code like "`twait { timer(10, mkevent()); }`" should be read as "execute '`timer(10, mkevent())`', then block until the created event has triggered"—or, since `timer` triggers its event argument after the given number of seconds has passed, simply as "block for 10 seconds". `twait{}` can implement many forms of event-driven control flow, including serial and parallel RPCs.

The second, more flexible form of wait point explicitly names a **rendezvous** object, which specifies the set of expected events relevant to the wait point. Every `event` object associates with one `rendezvous`. A wait point `twait(r)` unblocks when *any one* of `rendezvous r`'s events occurs. Unblocking consumes the event and restarts the blocked function. The first form of wait point is actually syntactic sugar for the second: code like "`twait {` *statements*`; }`" expands into something like

```
{ rendezvous<> __r;
  statements;   // where mkevent calls create events on __r
  while (not all __r events have completed)
    twait(__r);                                 }
```

The `twait()` form can also return information about *which* event occurred. A rendezvous of type `rendezvous<`*I*`>` accepts events with *event IDs* of type(s) *I*. Event IDs identify events in the same way thread IDs identify threads, except that event IDs have arbitrary, programmer-chosen types and values. A `twait(r,` *i*`)` statement then sets *i* to the ID(s) of the unblocking event.

Although wait points are analogous to blocking a thread until a condition variable is notified, blocking in Tame has a different meaning than in threads. A blocked threaded function's caller only resumes when the callee explicitly `return`s. In Tame, by contrast, a `tamed` function's caller resumes when the called function *either returns or blocks*. To allow its caller to distinguish returning from blocking, a `tamed` function will often accept an event argument, which it triggers when it returns. This trigger signals the function's completion. Here is a function that blocks, then returns an integer, in threads and in Tame:

```
int blockf() {          tamed blockf(event<int> done) {
   ... block ...           ... block ...
   return 200;             done.trigger(200);
}                       }

i = blockf();           twait { blockf(mkevent(i)); }
```

In Tame, the caller uses `twait` to wait for `blockf` to return, and so must become `tamed` itself. Waiting for events thus trickles up the call stack until a caller doesn't care whether its callee returns or blocks. This property is related to stack ripping, but much simpler, since functions do not split into pieces. Threaded code avoids any such change at the cost of blocking *the entire call stack* whenever a function blocks. Single-function blocking gives Tame its event flavor, increases its flexibility, and reduces its overhead (only the relevant parts of the call stack are saved). We return to this topic in the next section.

When an event `e` is triggered, Tame queues a *trigger notification* for `e`'s event ID on `e`'s rendezvous `r`. This step also unblocks any function blocked on `twait(r)`. Conversely, `twait(r)` checks for any queued trigger notifications on `r`. If one exists, it is dequeued and returned. Otherwise, the function blocks at that wait point; it will unblock and recheck the rendezvous once someone triggers a corresponding event. The top-level event loop cycles through unblocked functions, calling them in round-robin order when unblocking on file descriptor I/O and first-come-first-served order otherwise. More sophisticated queuing and scheduling techniques [40] are possible.

Multiple functions cannot simultaneously block on the same rendezvous. In practice, this restriction isn't significant since most rendezvous are local to a single function. A Tame program that needs two functions to wait on the same condition uses two separate events, triggering both when the condition occurs. Tame-based read locks (see Section 7.5) are an example of such a pattern.

Finally, **safe local variables**, a language extension, are variables whose values are preserved across wait points. The programmer marks local variables as safe by enclosing them in a `tvars {}` block, which preserves their values in a heap-allocated closure. (Function parameters are always safe.) Unsafe local variables have indeterminate values after a wait point. The C++ compiler's uninitialized-

Classes	Keywords & Language Extensions	Functions & Methods

`event<>`
• A basic event.
`event<`*T*`>`
• An event with a single *trigger value* of type T. This value is set when the event occurs; an example might be a character read from a file descriptor. Events may also have multiple trigger values of types $T_1 \dots T_n$.
`rendezvous<`*I*`>`
• Represents a set of outstanding events with event IDs of type *I*. Callers name a rendezvous when they block, and unblock on the triggering of any associated event.
	`twait(`*r*`[`,*i*`]);`
	• A wait point. Block on explicit rendezvous *r*, and optionally set the event ID *i* when control resumes.
	`tamed`
	• A return type for functions that use `twait`.
	`tvars { ... }`
	• Marks safe local variables.
	`twait {` *statements*`; }`
	• Wait point syntactic sugar: block on an implicit rendezvous until all events created in *statements* have triggered.
		`mkevent(`*r*`,`*i*`,`*s*`);`
		• Allocate a new event with event ID *i*. When triggered, it will awake rendezvous *r* and store trigger value in slot *s*.
		`mkevent(`*s*`);`
		• Allocate a new event for an implicit `twait{}` rendezvous. When triggered, store trigger value in slot *s*.
		e`.trigger(`*v*`);`
		• Trigger event *e*, with trigger value *v*.
		`timer(`*to*`,`*e*`); wait_on_fd(`*fd*`,`*rw*`,`*e*`);`
		• Primitive event interface for timeouts and file descriptor events, respectively.

Figure 2: Tame primitives for event programming in C++.

variable warnings tell a Tame programmer when a local variable should be made safe.

Type signatures Events reflect the types of their trigger values, and rendezvous reflect the types of their event IDs. The compiler catches type mismatches and reports them as errors. Concretely, `rendezvous` is a conventional C++ template type, defined in a library. All events associated with a rendezvous of type `rendezvous<I>` must have event IDs of type `I`. The `mkevent` function has type:

```
event<T1,T2,...> mkevent(rendezvous<I> r, const I &i,
    T1 &s1, T2 &s2, ...);
```

The arguments are a `rendezvous`, an event ID `i`, and slot references `s1`, `s2`, ... that will store trigger values when the event is later triggered. C++'s template machinery deduces the appropriate event ID and slot type(s) from the arguments, so `mkevent` can unambiguously accommodate optional event IDs and arbitrary trigger slot types. The `event::trigger` method has type:

```
void event<T1,T2,...>::trigger(const T1 &v1,
    const T2 &v2, ...);
```

When called, this method assigns the trigger values `v1`, `v2`, ... to the slots given at allocation time, then unblocks the corresponding rendezvous. Wait points have type `twait(rendezvous<I> r, I &i)`; when the wait point unblocks, `i` holds the ID of the unblocking event.

Primitive events Three library functions provide an interface to low-level operating system events: `timer()`, `wait_on_fd()`, and `wait_on_signal()`. Each function takes an `event<>` *e* and one or more extra parameters. `timer(`*to*`,`*e*`)` triggers *e* after *to* seconds have elapsed; `wait_on_fd(`*fd*`,`*rw*`,`*e*`)` triggers *e* once the file descriptor *fd* becomes readable or writable (depending on *rw*); and `wait_on_signal(`*sig*`,`*e*`)` triggers *e* when signal *sig* is received. The base event loop that understands these functions is implemented in terms of `select()` or platform-specific alternatives such as Linux's epoll or FreeBSD's kqueue [19].

Like all programs based on events or cooperative threads, a tamed program will block entirely if any portion of it calls a blocking system call (such as `open`) or takes a page fault. Tame inherits *libasync*'s non-blocking substitutes for blocking calls in the standard library (such as `open` and `gethostbyname`). For tamed programs to perform well in concurrent settings, they should use only non-blocking calls and should not induce swapping.

Figure 2 summarizes Tame's primitive semantics.

3.2 Control Flow Examples

Common network flow patterns like sequential calls, parallel calls, and windowed calls [37] are difficult to express in standard event libraries but much simplified with Tame. As a running example, consider a function that resolves addresses for a set of DNS host names. An initial design might use the normal blocking resolver:

```
void multidns(dnsname name[], ipaddr a[], int n) {
   for (int i = 0; i < n; i++)
      a[i] = gethostbyname(name[i]);
}
```

Of course, this function will block all other computation until all lookups complete. An efficient server would allow other progress during the lookup process. The event-based solution would use a *nonblocking* resolver, with a signature such as:

```
tamed gethost_ev(dnsname name, event<ipaddr> e);
```

This resolver uses nonblocking I/O when contacting local and/or remote DNS servers. (Alternately, Tame's threading support makes it easy to adapt a blocking resolver for nonblocking use; see Section 4.) Since `gethost_ev`'s caller

```
void multidns_nasty(dnsname name[], ipaddr a[], int n,
                    event<> done) {
   if (n > 0) {
      // When lookup succeeds, gethost_ev will call
      // "helper(name, a, n, done, RESULT)"
      gethost_ev(name[0], wrap(helper, name, a, n, done));
   } else // done, alert caller
      done.trigger();
}
void helper(dnsname *name, ipaddr *a, int n,
            event<> done, ipaddr result) {
   *a = result;
   multidns_nasty(name+1, a+1, n-1, done);
}
```

Figure 3: Stack-ripped *libasync* code for looking up `n` DNS names without blocking. A simple `for` loop has expanded into two interacting functions, obscuring control flow; all callers must likewise split.

can regain control before the lookup completes, the lookup result is returned via a trigger value: once the address `a` is known, the resolver calls `e.trigger(a)`. The trigger simultaneously exports the result and unblocks anyone waiting for it. Here's how to look up a single name with `gethost_ev`:

```
tvars { ipaddr a; }
twait { gethost_ev(name, mkevent(a)); }
print_addr(a);
```

Without Tame, adapting `multidns` to use `gethost_ev` is an exercise in stack-ripping frustration; for the gory details, see Figure 3. Tame, however, makes it simple:

```
1  tamed multidns_tame(dnsname name[], ipaddr a[],
                       int n, event<> done) {
2     tvars { int i; }
3     for (i = 0; i < n; i++)
4        twait { gethost_ev(name[i], mkevent(a[i])); }
5     done.trigger();
6  }
```

`multidns_tame` keeps all arguments and the local variable `i` in a closure. Whenever `gethost_ev` looks up a name, it triggers the event allocated on line 4. This stores the address in `a[i]` and unblocks `multidns_tame`, after which the loop continues. Though the code somewhat resembles threaded code, the semantics are still event-driven: `multidns_tame` can return control to its caller before it completes. Thus, it signals completion via an event, namely `done`. Any callers that depend on completion must use Tame primitives to block on this event, and thus become `tamed` themselves. The `tamed` return type then bubbles up the call stack, providing the valuable annotation that `multidns_tame` and its callers may suspend computation before completion.

`multidns_tame` allows a server to use the CPU more effectively than `multidns`, since other server computation can take place as `multidns_tame` completes. However, `multidns_tame`'s lookups still happen in series: lookup *i* does not begin until lookup $i-1$ has completed. The obvious latency improvement is to perform lookups in parallel. The tamed code barely changes:

```
tamed multidns_par(dnsname name[], ipaddr a[],
                   int n, event<> done) {
   twait {
      for (int i = 0; i < sz; i++)
         gethost_ev(name[i], mkevent(a[i]));
   }
   done.trigger();
}
```

The only difference between the serial and parallel versions is the ordering of the `for` and `twait` statements (and that `i` doesn't need to be in the closure). Since both versions have the same signature, the programmer can switch implementation strategies without changing caller code. With threads, however, the serial version could use one thread to do all lookups, while the parallel version would use as many threads as lookups. Tame preserves events' flexibility while providing much of threads' readability.

A generalization of serial and parallel control flow is *windowed* or *pipelined* control flow, in which *n* calls are made in total, and at most $w \leq n$ of them are outstanding at any time. For serial flow, $w = 1$; for parallel, $w = n$. Intermediate values of *w* combine the advantages of serial and parallel execution, allowing some overlapping without blasting the server. With Tame, even windowed control flow is readable, although the simplified `twait{}` statement no longer suffices:

```
1  tamed multidns_win(dnsname name[], ipaddr a[],
                      int n, event<> done) {
2    tvars { int sent(0), recv(0); rendezvous<> r; }
3    while (recv < n)
4       if (sent < n && sent - recv < WINDOWSIZE) {
5          gethost_ev(name[sent], mkevent(r,a[sent]));
6          sent++;
7       } else {
8          twait(r);
9          recv++;
10      }
11    done.trigger();
12 }
```

The loop runs until all requests have received responses (`recv == n`). On each iteration, the function sends a new request (lines 5–6) whenever a request remains (`sent < n`) and the window has room (`sent - recv < WINDOWSIZE`). Otherwise, the function harvests an outstanding request (lines 8–9). Again, the signature is unchanged, and the implementation is short and clear. We have not previously seen efficient windowed control flow expressed this simply.

3.3 Typing and Composability

Tame's first-class events and rendezvous, and its distinction between event IDs and trigger values, improve its flexibility, composability, and safety.

First, Tame preserves safe static typing without compromising flexibility by distinguishing event IDs from trigger values. Event IDs are like names. They identify events,

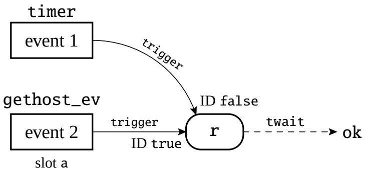


Figure 4: Relationships between events (boxes) and rendezvous (round box) for a DNS lookup with timeout.

and are known when the event is registered; all events on the same rendezvous must have the same event ID type. Trigger values, on the other hand, are like results: they are not known until the event actually triggers. Examples include characters read from a file descriptor, RPC replies, and so forth. Event IDs and trigger values are related, of course; when a `twait` statement returns event ID *i*, the programmer knows that event *i* has triggered, and therefore its associated trigger values have been set. In contrast, several other systems return trigger values as part of an event object; the `twait` equivalent returns the object, and extracting its values requires a dynamic cast. The Tame design avoids error-prone casts while still letting a single rendezvous handle events with entirely different trigger value types.

To demonstrate Tame's composability, we'll add timeouts to the following event-based DNS lookup:

```
tvars { ipaddr a; }
twait { gethost_ev(name, mkevent(a)); }
```

We want to cancel a lookup and report an error if a name fails to resolve in ten seconds. The basic implementation strategy is to wait on *two* events, the lookup and a ten-second timer, and check which event happens first.

```
tvars { ipaddr a;  rendezvous<bool> r;  bool ok; }
timer(10, mkevent(r, false));
gethost_ev(name, mkevent(r, true, a));
twait(r, ok);
if (!ok) printf("Timeout");
r.cancel();
```

The event ID `false` represents timeouts, while `true` represents successful lookup. The `twait` statement sets `ok` to the ID of the event that triggers first[1], so `ok` is false if and only if the lookup timed out. The `r.cancel()` call cleans up state associated with the event that did not trigger. Figure 4 diagrams the relevant objects.

This code is verbose and hard to follow. Supporting timeouts on *every* lookup, or on other types of event, would require adding `rendezvous` and `timer` calls across the program and abandoning the `twait{}` syntactic sugar.

```
1  template <typename T> tamed
   __add_timeout(event<T> &e_base, event<bool, T> e) {
2     tvars { rendezvous<bool> r; T result; bool rok; }
3     timer(TIMEOUT, mkevent(r, false));
4     e_base = mkevent(r, true, result);
5     twait(r, rok);
6     e.trigger(rok, result);
7     r.cancel();
8  }

9  template <typename T> event<T> add_timeout(event<bool, T> e) {
10    event<T> e_base;
11    __add_timeout(e_base, e);
12    return e_base;
13 }
```

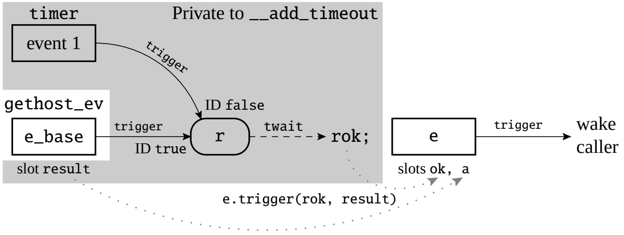


Figure 5: Code and object relationships for a composable timeout event adapter, and its use in a DNS lookup.

Tame can do better. Its programmers can write an adapter that can add a cancellation timeout to *any* event. The adapter relies on C++'s template support and on Tame's first-class events, and resembles adapters from higher-level thread packages such as CML. Many other event libraries could not express this kind of composable abstraction, which was a main motivator for Tame's design. The adapter simplifies the lookup code to:

```
tvars { ipaddr a;  bool ok; }
twait { gethost_ev(name, add_timeout(mkevent(ok, a))); }
if (!ok) printf("Timeout");
```

The caller generates an event with *two* trigger slots, one for the base trigger value and one for a boolean that indicates success or failure. Either a successful lookup or a timeout will trigger the event. Success will set the boolean trigger value to true, timeout will set it to false. Thus, after waiting for the event, callers can examine the boolean to check for timeout. This pattern works with `twait{}` statements as well as explicit rendezvous.

Unfortunately, the `gethost_ev` function requires an event that takes a *single* trigger value, namely the IP address. It will not supply an extra trigger value unless we change its signature and implementation, which would make it specific to our timeout adapter—something we'd like to avoid. But Tame's abstractions let us *transparently interpose* between `gethost_ev` and its "caller". The adapter will set the extra trigger value.

Figure 5 shows the code and a diagram of the object relationships. The real work takes place in `__add_timeout`, which creates *two* events: `e_base`, which is returned (and eventually passed to `gethost_ev`), and an internal event passed to the timer function on line 6. The two

[1] In other libraries we have examined, such as CML, the `twait` function would return an opaque, system-chosen ID that the programmer would compare with return values from `mkevent`. Though this works, event IDs are far more convenient, particularly when many events are outstanding.

created events associate with the rendezvous r local to __add_timeout. This is the interposition. When the timeout triggers, or when e_base triggers (due to a successful DNS lookup), __add_timeout will unblock, set the ok slot appropriately, and then trigger e. Only this last step unblocks the caller. The caller observes that ok and a have been set, but is oblivious to __add_timeout's intercession; it is as if gethost_ev set ok itself.

It would be trivial to add other types of "timeout", such as signal receipt, to add_timeout; its signature would not change, and neither would its callers. Similarly, one-line changes could globally track how many events time out. We've added significant additional concurrency semantics with only local changes: the definition of composability.

3.4 Future Work

Tamed processes do not currently run on more than one core or CPU. The production Tame-based applications we know of consist of multiple concurrent processes cooperating to achieve an application goal. OkCupid.com, for instance, uses exclusively multicore and SMP machines. Its Web front-ends run no fewer than fifty site-specific Tame-based processes, all of which simultaneously answer Web requests. When traffic is high, all CPUs (or cores) are in use. Nevertheless, few changes to Tame would be required for true simultaneous threading support. Tame already supports event-based locks to product data structures from unwanted interactions (Section 7.5). As in *async-mp* [41], multiple kernel threads could draw from a shared pool of ready tasks, as restricted by Tame's current atomicity assumption: at most one thread of control can be active in any given closure at a time. Locks enforced by the kernel, or any equivalent technique, could ensure this invariant.

Tame does not currently interact well with C++ exceptions: an exception raised in a Tamed function might be caught by the event loop.

Some of Tame's limitations are not implementation-dependent but rather consequences of its approach and semantics. As mentioned in Section 3, changing a function from a regular C++ function to a tamed function involves signature changes all the way up the call stack. Some developers might object to this limitation, especially those who export libraries with fixed interfaces.

4 THREADS

Tame can interoperate with threads when a thread package is available, suggesting that the Tame abstractions (wait points, events and rendezvous) apply to both programming models. With thread support, Tame simplifies the transition between threaded and event-style programming, for instance allowing event-based applications to use threaded software in the C library (e.g. gethostbyname) and database client libraries (e.g. libmysqlclient [23]). We have only experimented with cooperative user-level threading packages, though kernel-level threads that support SMPs are also compatible with our approach.

The key semantic difference between threaded and event-based operation is how functions return. In event-based Tame, functions can either return a useful value via a return statement or block via twait, but not both. Threaded functions *can* both block and return a value, since the caller regains control only when the computation is done.

With thread support, Tame exposes both event and thread return semantics. In Tame, a threaded function is one that calls twait but does not have a tamed return type. When such a function encounters twait(r), it checks for queued triggers in r as usual; if none are present, it asks for a wakeup notification when a trigger arrives in r, and then *yields* to another thread. During the yield, the threading package preserves the function's entire call stack (including all of its callers), while running other, more ready computations. When the trigger arrives, the blocked thread awakes at the twait call and can return to its caller.

A trivial example using threads in Tame is a reimplementation of the sleep call:

```
1  int mysleep(int d) {
2     twait { timer(d, mkevent()); }
3     return d;
4  }
```

As usual, the call to timer registers an event that will be triggered after a d second delay. The function then calls twait on an implicit rendezvous at line 2, yielding its thread. After d seconds have elapsed, the main thread triggers the event allocated on line 2, waking up mysleep and advancing control to the return statement on line 3. Since mysleep is threaded (i.e., does not have a tamed return value), it returns an actual value to its caller.

Blocking the current thread uses Tame's existing twait syntax, but starting a new thread requires a new tfork function:[2]

```
tfork(rendezvous<I> r, I i, threadfunc<V> f, V &v);
```

The semantics are:

1. Allocate e = mkevent(r, i).
2. Fork a new thread. In the new thread context:
   (a) Call f() and store its return value in v.
   (b) Trigger event e.
   (c) Exit the thread.

When the function f completes, the rendezvous r receives a trigger with event ID i. This unifies the usually separate concepts of event "blocking" and joining on a

[2] threadfunc<R> is an event whose trigger method yields a return value of type R. Given the function int f(), we can create a threadfunc<int> from a function pointer to f. From the function int g(int a), we can create a threadfunc<int> by *wrapping* g with an integer argument, as in function currying [15].

thread. Code like the following uses `tfork` and `twait{}` syntactic sugar to call a blocking library function from an event-based context:

```
tamed gethost_ev(const char *name,
                 event<struct hostent *> e) {
   tvars { struct hostent *h; }
   twait { tfork(wrap(gethostbyname, name), h); }
   e.trigger(h);
}
```

This starts `gethostbyname(name)` in a new thread, then blocks in the usual event-driven way until that thread exits. At that point, the caller is notified via an event trigger of the `struct hostent` result.

5 Memory Management

Tame hides most details of event memory management from programmers, protecting them at all costs from wild writes and catching most memory leaks. For the large majority of Tame code that uses the `twait{}` environment, correct program syntax guarantees correct leakless memory management. For more advanced programs that use explicit `rendezvous`, Tame uses reference counting to enforce key invariants at runtime. The invariants are:

I1 A function's closure lives at least until control exits the function for the last time.

I2 Some of an `event`'s trigger slots may be safe local variables, and triggering it assigns values to those variables. Thus, a function's closure must live as least until `events` created in the function have triggered.

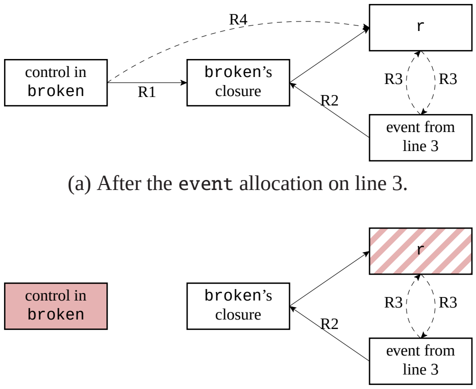
I3 `Events` associated with a `rendezvous` *r* must trigger exactly once before *r* is deallocated. The programmer must uphold I3 by correctly managing `rendezvous` lifetime and triggering each `event` exactly once.

A closure should be deallocated as soon as so doing does not violate I1 or I2.

Of these invariants, I3 depends the most on program correctness. Some cases are easy to handle. Tame ignores attempts to trigger an event multiple times (or aborts, depending on runtime options), and forgetting to trigger an event in a `twait{}` environment will cause a program hang and is thus easily observable. The difficult case involves managing the lifetimes of explicit `rendezvous`. Consider the function `broken`:

```
1  tamed broken(dnsname nm) {
2     tvars { rendezvous<> r;  ipaddr res; }
3     gethost_ev(nm, mkevent(r, res));
4     // Whoops: forgot to twait(r)!
5  }
```

The `event` created on line 3 uses the trigger slot `res`, a safe variable in `broken`'s closure. The function then exits without waiting for `r` or examining `res`. This is a bug—an *event leak* in violation of I3. If Tame deallocated `broken`'s closure eagerly, right after it exited, then the `event`'s eventual trigger would write its value into the deallocated memory where the closure used to be.



(a) After the `event` allocation on line 3.

(b) After control exits `broken` on line 5.

Figure 6: Memory references for the `broken` function. Weak references are shown as dotted lines, and strong references as solid lines. Solid fill indicates a function exit, and striped fill indicates cancellation.

Tame's answer is a careful reference-counting scheme; its runtime keeps track of events and closures with C++ "smart pointer" classes. For example, `event<>` objects are actually smart pointers; the `event` is stored elsewhere, in a private object only accessible by Tame code. If necessary, Tame keeps the `event` around even after the user frees it. There can be circular references among these three types of objects—for example, a closure contains a local `event`, which names a different closure-local variable as a trigger slot. Tame uses two different types of reference counts to break the circularity: *strong* references, which are conventional reference counts, and *weak* references, which allow access to the object only if it hasn't been deallocated.

In outline, Tame keeps the following reference counts:

R1 Entering a tamed function for the first time adds a strong reference to the corresponding closure, which is removed only when the function exits for the last time. This preserves I1.

R2 Each `event` created inside a closure holds a strong reference to that closure, preserving I2. The reference is dropped once the `event` is triggered.

R3 A `rendezvous` and its associated `events` keep weak references to each other. The references a `rendezvous` keeps to its `events` allow it to cancel `events` that did not trigger before the `rendezvous`'s deallocation. Canceling an `event` clears its R2 reference; any future trigger attempt on the `event` will be ignored, preserving I3. The weak references the other way enable an `event` to update its `rendezvous` upon a trigger.

Figure 6a shows these references in the `broken` function following line 3's `event` allocation. The most important problem introduced by this reference counting scheme is due to R2: an untriggered `event` can cause a closure leak.

Such a leak can be caught by checking the associated rendezvous upon deallocation for untriggered events. A rendezvous' deallocation is up to the programmer, but there is an important and common case in which Tame can intervene. If a `rendezvous` was declared as a local variable in some closure, and that closure has exited for the last time, then no future code will call `twait` on the `rendezvous`, *even if the closure cannot be deallocated yet* because of some stray reference. Thus, Tame amends the reference counting protocol as follows:

R4 Exiting a tamed function for the last time cancels any `rendezvous` directly allocated in that function's closure. Canceling a `rendezvous` cancels all `events` associated with it. Actual deallocation occurs only when the closure is deallocated, which might be some time later.

Figure 6b shows how Tame's reference counting protocol solves `broken`'s leak. Control exits the function immediately, forcing `r`'s cancellation by R4. Upon cancellation, `r` checks that all of its `events` have triggered. In this case, the `event` allocated on line 3 has not triggered, but Tame cancels it, clearing its R3, which releases the closure, and in turn, releases `r`. Any eventual trigger of the `event` is ignored.

6 Implementation Details

Tame is implemented as a C++ preprocessor (or source-to-source translator). The difficulty of parsing C++ is well known [2]. Tame avoids as much C++ parsing as possible at the cost of several semantic warts, which could be avoided with fuller compiler integration.

6.1 Closures

Each `tamed` function has one closure with a flat namespace, restricting C/C++'s scoping. Internally, the Tame translator writes a new C++ structure for each tamed function, containing its parameters and its `tvars` variables. This structure gets an opaque name, discouraging the programmer from accessing it directly.

Programmers are free to use arbitrary C++-stack allocation, as long as no wait points come between the declaration and use of stack-allocated variables. When they do, the underlying C++ compiler generates a warning due to `goto` branches in the emitted code (see the next section).

Maintaining Section 5's R4 requires that each closure know which `rendezvous` it directly contains, so it can cancel them appropriately. This knowledge is unavailable without fully parsing C++: a closure might contain an object of type `foo`, that contains an object of type `bar`, that contains a `rendezvous`, which will in turn share fate with the closure. As a first-order heuristic, Tame marks the beginning and the end of the new closure in memory using simple pointer arithmetic and associates with the closure all `rendezvous` that fall between the two fence posts.

6.2 Entry and Exit Translation

The translation of a tamed function adds to the function one new entry and exit point per `twait` statement. A translated `twait` statement first checks whether a trigger is pending on the corresponding `rendezvous`. If so, control flow continues past the `twait` function as usual; but if not, the function records the current wait point, adds a function pointer for this wait point to the `rendezvous`, and returns to its caller. Later, a trigger on an `event` in the `rendezvous` invokes the recorded function pointer, which forces control to reenter the function and jump directly to the recorded wait point. The Tame translation shifted the function's parameters and safe local variables to a closure structure, so the function can access these values even after reentry.

The Tame preprocessor adds an extra "closure pointer" parameter to each `tamed` function. The closure pointer is null when the function is called normally, causing the translated function body to allocate and initialize a new closure. The closure pointer is non-null when the function is reentered at a later wait point. The names of parameters and safe local variables are changed to opaque identifiers to hide them from the function body; instead, local variables with reference types make these names point into the closure. This strategy reduces the extent to which Tame must understand C++ name lookup, since the translation preserves the function implementation's original namespace. Multiple entry points are simulated with a switch statement at the beginning of the function; each case in the switch jumps to a different label in the function. There is one label for the initial function entry and one for each wait point.

Internally, `mkevent` is a macro that fetches some specially named variables (such as the current closure, or the current implicit `rendezvous` in the case of a `twait` environment). An input of `mkevent(rv, w, t1, t2)` generates a call of the form:

```
_mkevent(__cls, rv, w, t1, t2);
```

for some closure `__cls`. `_mkevent` heap-allocates a new `event` object, packing it with references to all of the supplied arguments. The resulting `event` object provides one method, `trigger`, which takes trigger value parameters with the types of `t1` and `t2`. All of these operations are type-safe through use of C++ templates.

Putting these pieces together, the translation of:

```
tamed A::f(int x) {
  tvars { rendezvous<> r; }
  a(mkevent(r)); twait(r); b(); }
```

looks approximately like:

```
1  void A::f(int __tame_x, A_f_closure *__cls) {
2    if (__cls == 0)
3      __cls = new A__f__closure(this, &A::fn, __tame_x);
4    assert(this == __cls->this_A);
5    int &x = __cls->x;
6    rendezvous<> &r = __cls->r;
7    switch (__cls->entry_point) {
8    case 0: // original entry
9      goto __A__f__entry__0;
10   case 1: // reentry after first twait
11     goto __A__f__entry__1; }
12 __A__f__entry__0:
13   a(_mkevent(__cls,r));
14   if (!r.has_queued_trigger()) {
15     __cls->entry_point = 1;
16     r.set_reenter_closure(__cls);
17     return; }
18 __A__f__entry__1:
19   b();
20 }
```

Lines 5–6 set up the function body so that references to `x` and `r` refer to closure-resident values. Lines 7–11 direct traffic as it enters and reenters the function after `twait` points. Lines 12–19 are the translation of the user code. Lines 14–17 are the translation of the `twait(r)` call from the original function. If no trigger is queued on `r`, the translation bumps the entry point (line 15) and tells the `rendezvous` to reenter `__cls` via the method `A::fn` when a trigger arrives (line 16). Once that happens, `f` will jump to entry point 1 (line 18) and call `b()`.

6.3 Backwards Compatibility

Our implementation of Tame borrows its event loop and event objects from the *libasync* event library [21]. The key compatibility feature is to implement `events` as *libasync* `callbacks`, allowing legacy functions to interface with tamed functions, and consequently, legacy projects to incrementally switch over to tamed code.

The Tame prototype implements thread support with the Gnu PTH library [12]. PTH supplies stubs for blocking network calls such as `select`, `read` and `write`. Thus the `select` call in *libasync*'s select loop transparently becomes a call to PTH's scheduler. Similarly, blocking network calls in third party libraries like `libmysqlclient` drop into the scheduler and later resume when the operation completes. We also had to make *libasync* call a modified, Tame-aware `select`. This `select` returns early when another thread in the same process triggers an event that should wake up the current thread (something that never happens in single-threaded Tame).

7 Experience With Tame

Like any other expressive synchronization system, Tame requires some mental readjustment and ramp-up time. In most cases, developers need only the `twait{}` environment, which is designed to be simple to learn and comparable to thread programming. With only this subset of Tame, programmers become much more productive relative to vanilla-event coders, and hopefully as productive as thread programmers.

7.1 Web Server

The latest version of OKWS [18], a lightweight Web server for dynamic Web content, uses the Tame system. Its most obvious applications are serial chains of asynchronous function calls, such as startup sequences that involve IPC across cooperating processes. These chains are common in OKWS; Tame lets them occupy a single function body, making the code easier to read.

A more specialized Tame application is in OKWS's templating system, which allows OKWS Web developers to separate their application logic from the HTML presentation layer. In a manner similar to Flash [24], OKWS uses blocking helper processes to read templates from the file system; the main server calls the helper processes asynchronously. However, since templates can be arbitrarily nested, reading one template may require many helper calls. The previous version of OKWS, written without Tame, sacrificed expressiveness for programmability. Web site developers had to request all template files they would ever need when their Web service started up, so that a call to publishing a template in response to a Web request would not block and force a stack rip. In the new version of OKWS, publishing a template is an asynchronous operation, and site developers can therefore publish any file in the `htdocs` directory, at any time. Tame saves developers from the stack ripping problem that previously discouraged this feature.

7.2 An Event-Based Web Site

OkCupid.com [16] is a dating Web site that uses OKWS as its Web server. For several years, its programmers wrote code in the *libasync* idiom to manage concurrency, but in early 2006 switched over to Tame to simplify debugging and to improve productivity. Currently seven programmers, none of whom are the authors, depend on Tame for maintaining and developing site features. The system is easy enough to use so that the first programming project new employees receive is to convert code from the old event-based system to Tame syntax.

OkCupid.com has found Tame's parallel dispatch particularly useful when programming a Web site. When a user logs into the site, the front-end Web logic requests data from multiple databases to reconstruct the user's preferences and server-resident state. To minimize client-perceived latency from disk accesses, these queries can happen in parallel. With just *libasync* primitives, parallelism was hidden in stack-ripped code and caused bugs. Tame's solution is the parallelism inherent in the `twait` environment. To call f and g in parallel, then call h once they both complete, a Tame programmer simply writes:

```
twait { f(mkevent()); g(mkevent()); }
twait { h(mkevent()); }
```

7.3 An NFS Server

A graduate distributed systems class at MIT requires its students to write a simple Frangipani-inspired [35] file server that implements the NFS Version 3 protocol [4]. In spring 2006, the students had the option to write their assignment with the Tame tool. Four out of 22 students used Tame, most successfully on the source file that implements the file system semantics (about two thousand lines long). Consider, for example, the `CREATE` RPC, for creating a new file on the server. When given this RPC, the server must acquire a lock, lookup a file handle for the target directory, read the contents of the directory, write out new directory contents, then write out the file, and finally release the lock, all the while checking for various error conditions. The solutions with legacy *libasync* involve code split up over no fewer than five functions, with a stack rip at every blocking point. Students who used Tame accomplished the same semantics with just one function. Quantitatively, the students who used Tame wrote 20% less code in their source files, and 50% less code in their header files. Qualitatively, the students had positive comments about the Tame system and semantics, and strongly preferred writing in the Tame idiom to writing *libasync* code directly.

7.4 Debugging

Tame's preprocessor implements source-code line translation, so debuggers and compilers point the programmer to the line of code in the original Tame input file. The programmer need only examine or debug autogenerated code when Tame itself has a bug. Programmers can disable line-translation and view human-readable output from the Tame preprocessor. Relative to a tamed function in the input file, a tamed function in the output file differs only in its Tame-generated preamble, at `twait` points, and at `return` statements. The rest of the code is passed through untouched.

Tame also has debugging advantages over legacy *libasync* with unmodified debugging tools. With legacy *libasync*, a developer must set a breakpoint at every stack rip point. With Tame, a logical operation once again fits inside a single function body. As a result, a programmer sets a break point at the suspect function, and can trace execution until a blocking point (i.e., `twait`). After the blocking point, control returns to the same breakpoint at the top of the same function, and then jumps to the code directly after the `twait` statement.

Future work calls for a Tame debugger and profiler. In both cases, the runtime nesting of closures is Tame's analogue of the call stack in a threaded program. Slight debugger modifications could allow walking this graph to produce a "stacktrace"-like feature, and similarly, measuring closure lifetime can yield a `gprof`-style output for understanding which parts of a program induce latency. Even under the status quo, programmers can access safe local variables in debuggers by simply examining the members of a function's closure and can walk the closure-chain manually if desired.

7.5 Locks and Synchronization

Programmers using events or cooperative threading often falsely convince themselves that they have "synchronization for free." This is not always the case. Global data on one side of a yield or block point might look different on the other side, if another part of the program manipulated that data in between. With threaded Tame programs, or threaded programs in general, any function invocation can result in a yield, hiding concurrency assumptions deep in the call stack. In practice, a programmer cannot know automatically know when to protect global data structures [1]. Event-only Tame programs make concurrency assumptions explicit, since they never yield; they just return to the main event loop (allowing other computations to run) on either side of a `twait` statement or environment.

When Tame programs require atomicity guarantees on either side of a `twait` (or yield in the case of threads), they can use a simple lock implementation based on Tame primitives. A basic lock class exposes the methods:

```
tamed lock::acquire(event<> done);
void lock::release();
```

The `acquire` method checks the lock to see if it's currently acquired; if so, it queues the given `event`, and if not, it triggers `done` immediately. The `release` method either triggers the head of the `event` queue, or marks the lock as available if no `events` were queued. An example critical section in Tame now looks like:

```
tamed global_data_accessor() {
   twait { global_lock->acquire(mkevent()); }
   ... touch global state, possibly blocking ...
   global_lock->release();
}
```

We have also built shared read locks with Tame, in which a writer's release of a lock can cause all queued readers to unblock.

8 Performance Measurements

The Tame implementation introduces potential performance costs relative to threaded code and traditional event-driven software. Unlike cooperative-threaded code, and more so than traditional event libraries (e.g. *libasync*), Tame makes heavy use of heap-allocated data structures, such as closures and one-time events. Tame also uses synchronization primitives (namely `rendezvous` and `events`) that are potentially costlier than the lower level primitives in threading packages or *libasync*. We investigate the end-to-end cost of Tame relative to a comparable

	Capriccio	Tame
Throughput (connections/sec)	28,318	28,457
Number of threads	350	1
Physical memory (kB)	6,560	2,156
Virtual memory (kB)	49,517	10,740

Figure 7: Measurements of Knot at maximum throughput. Throughput is averaged over the whole one-minute run. Memory readings are taken after the warm-up period, as reported by `ps`.

high-performance system, and conclude that Tame incurs no performance penalties and makes better use of memory.

8.1 End-to-End Performance

A logical point of comparison for the Tame system is the Capriccio thread package [38]. Like Tame, it provides automatic memory management and cooperative task management; it is also engineered for high performance. The Capriccio work focuses its measurements on the simple "Knot" web server. We compared the performance of the original Capriccio Knot server with a lightly modified, tamed version of Knot. In selecting a workload, we factored out the subtleties of disk I/O and scheduling that other work has addressed in detail [25] and focused on memory and CPU use. We ran a SpecWeb-like benchmark but used only the smallest files in the dataset, making the workload entirely cacheable and avoiding link saturation.

For all experiments, the server was a 2-CPU 2.33 GHz Xeon 5140 with 4GB memory, running Ubuntu Linux with kernel 2.6.17-10, code compiled with GCC version 4.1.2, optimization level `-O2`. Because Capriccio does not compile with more recent compilers, it was compiled GCC version 3.3.5. Glibc and NPTL were both version 2.4. Though the machine has four cores, only one was needed in our experiments (neither Tame nor the other systems tested use multiple CPUs). Tame supports Linux's `epoll`, but its event loop was configured to use `select` in our benchmarks. Capriccio uses the similar `poll` call in its loop. We used an array of six clients connected through a gigabit switch, each making 200 simultaneous requests to the server. The servers were given a thirty-second "warm-up" time in which they pulled all of the necessary files from disk into cache, and then ran for a one-minute test. The results are shown in Figure 7.

The high level outcome is that under this workload, the Capriccio and Tame versions of Knot achieve the same throughput, but Tame Knot uses one-third the physical memory, and one-fifth the virtual memory. We note that neither Knot server in this scenario ever blocks: both servers use 100% of available CPU, even when idle. A version of the Tame server that blocks when there is no work to be done achieves a surprising 4,000 fewer connections per second on our benchmark machine. Another important optimization was to avoid dropping into the select loop when outstanding connection attempts could be `accepted` [3]. Microbenchmarks in Section 8.2 show the select loop is expensive relative to other Tame primitives.

Operation	Min	Median	Mean
Simple function call	63	63	66
Simple function call with `int` allocation	182	196	196
Tame call (`nullfn()`)	399	455	463
`wrap()`	217	224	231
`gettimeofday()`	2618	2660	2781

Figure 8: Cost of system calls and *libasync* and Tame primitive operations, measured in cycles.

Optimizing Capriccio Knot's performance required manual tuning. The size of the thread pool must be sufficiently large (about 350 threads) before Capriccio Knot can achieve maximal throughput. Threads' stack sizes must also be set correctly—stacks that are too small risk overflow, while stacks that are too big waste virtual memory—but the default 128 kB per stack sufficed for these experiments. Capriccio can automate these parameter settings, but the Knot server in the Capriccio release does not use automatic stack sizing, and manual thread settings were more stable in our tests. Further work could bring Capriccio's memory usage more in line with Tame's, but we note that Tame achieves its memory usage automatically without changes to the base compiler.

Memory allocation in Tame Knot happens mainly on the heap, in the form of event and closure allocations. In our test cases, we noted 12 closure allocations and 12.6 event allocations per connection served. We experimented with "recycling" events of common types (such as `event`<>s) rather than allocating and freeing them each time. Such optimizations had little impact on performance, suggesting Linux's malloc automatically optimizes Tame's memory access pattern.

8.2 Microbenchmarks

We performed microbenchmarks to get a better sense for how Tame was spending its cycles in the web benchmark, and to provide baseline statistics for other applications. A first cost of Tame relative to thread programming is closure allocation. We measured closure costs with the most basic tamed function that uses a closure:

```
static tamed nullfn()
  { tvars { int i(0); } i++; }
```

For comparison, we also measure a trivial function, a function that performs a small heap allocation, *libasync*'s closure-approximating function (i.e., `wrap`), and a trivial system call (`gettimeofday`). In each case, an experiment consisted of executing the primitive 10,000 times, bracketed by cycle counter checks. We ran each experiment 10 times and report averaged results over the 10 experiments, and the median results over all 100,000 calls. In all cases, the standard deviation over the 10 experiments was within 5% of the mean. Figure 8 summarizes our results: entering a tamed function is about 2.2 times the cost of a simple function with heap allocation, and 1.8 times the cost of a wrap invocation.

A second function, benchfn, measures Tame overhead when managing control flow:

```
1  static tamed benchfn (int niter, event<> done) {
2     tvars { int i; }
3     for (i = 0; i < niter; i++)
4        twait { timer(0, mkevent()); }
5     done.trigger();
6  }
```

Line 4 of benchfn is performing Tame's version of a thread fork and join. A call to mkevent and later twait is required to launch a potentially blocking network operation, and to harvest its result. Unlike a *libasync* version of benchfn, the tamed version must manage closures, an implicit rendezvous, and jumping into and out of the function once per iteration. We compare three versions of benchfn: with an implicit rendezvous, with an explicit rendezvous, and with only *libasync* features.

We ran all versions with niter=100, and repeated the experiment one thousand times. The results are presented in Figure 9. All experiments spend a majority of cycles in the core select loop. The benchfn that uses an implicit rendezvous is only slightly more expensive, performing within 2% of the native *libasync* code. Tame's low-level implementation special-cases the implicit rendezvous, reducing memory allocations and virtual method calls along the critical path. Hence, the benchfn version that uses an explicit rendezvous runs about 6% slower still. We also experimented with replacing *libasync*'s native scheduler with that of the PTH thread library, as is required when running Tame with thread support.

Based on these benchmarks, we can estimate how Tame Knot's CPU time is spent. Tame Knot uses 81,877 cycles for each request. Assuming the microbenchmark results hold, and given Tame Knot's use of 12.6 events and 12 closure allocations per request, roughly 7.6% of these cycles are spent on event management and 6.8% on closure management, with the remainder going towards system calls and application-level processing.

Figure 9 also gives similar benchmarks for a version of benchfn written in pure thread abstractions,

```
static void noop() { pthread_exit(NULL); }
void benchfn_thr(int niter) {
   for (int i = 0; i < niter; i++) {
      pthread_t t;
      pthread_create(&t, NULL, noop, NULL);
      pthread_join(t, NULL);
   }
}
```

and a version using Tame's thread wrappers:

```
static void noop_tame() {}
void benchfn_tame_thr(int niter) {
   for (int i = 0; i < niter; i++)
      twait { tfork(wrap(noop_tame)); }
}
```

Function	Cycles	In Core	Outside
benchfn without Tame	5,251	4,906	344
benchfn with twait{}	5,331	4,840	491
with PTH core loop	6,010	5,476	534
benchfn with twait(r);	5,642	4,887	755
with PTH core loop	6,565	5,678	887
benchfn_thr in PTH	37,540	-	-
in NPTL	28,803	-	-
in Capriccio	7,892	-	-
benchfn_tame_thr	64,957	-	-

Figure 9: Results from running the benchfn code in 1,000 experiments, with niter=100. Costs shown are the average cycles per iteration, averaged over all experiments. These costs are broken down into cycles spent in the core event loop, and time spent outside.

A thread allocation and join is five to seven times as expensive as an event allocation and join in Tame when using standard Linux libraries like PTH and NPTL. Tame's thread wrappers added additional overhead relative to native PTH since they require locks and condition variables. Capriccio is much faster and competitive with Tame. Traditionally, threaded programs allocate threads not quite as cavalierly as benchfn_thr; they might use thread-pooling techniques to accomplish more than one operation per thread. However, examples like those in Sections 3.2 and 3.3 and those in real-world Web site programming (Section 7.2) benefit greatly from repeated thread creation and destruction. Tame primitives are certainly fast enough to support this.

In sum, Tame's primitive operations are marginally more expensive than *libasync*'s and roughly equivalent to those of a good thread package. The observed costs are cheap relative to real workloads in network applications.

9 Summary

Tame confers much of the readability advantage of threads while preserving the flexibility of events, and modern thread packages have good performance: the clichéd performance/readability distinction between events and threads no longer holds. Programmers should choose the abstraction that best meets their needs. We argue that event programming with Tame is a good fit for networked and distributed systems. The Tame system has found adoption in real event-based systems, and the results are encouraging: fewer lines of code, simplified memory management, and simplified code maintenance. Our hope is that Tame can solve the software maintenance problems that plague current event-based systems, while making events palatable to a wider audience of developers.

Acknowledgments

We thank the following people for helpful comments on the Tame system and earlier drafts of this paper: Jeff Fischer, Jon Howell, Rupak Majumdar, Todd Millstein, Michael Walfish, Russ Cox, Jeremy Stribling, the other members of the PDOS group, and the anonymous reviewers. Russ Cox and Tim Brecht helped us in benchmarking Capriccio, and we thank Vivek Pai for his “flexiclient” workload generator. Chris Coyne inspired the authors to think about simplifying events. Thanks to him and the developers at OkCupid.com who have adopted the system. We also thank David Mazières for creating the *libasync* system on which Tame is based. Eddie Kohler’s work on Tame was supported by the National Science Foundation under Grant No. 0427202.

The Tame system (with minor syntactic differences) is distributed along with the *libasync* libraries, all under a GPL version 2 license, at `http://www.okws.org/`.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proc. 2002 USENIX Annual Tech. Conference*, June 2002.

[2] A. Birkett. Parsing C++. `http://www.nobugs.org/developer/parsingcpp`.

[3] T. Brecht, D. Pariag, and L. Gammo. accept()able strategies for improving Web server performance. In *Proc. 2004 USENIX Annual Tech. Conference*, June 2004.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

[5] K. Claessen. A poor man’s concurrency monad. *Journal of Functional Programming*, 9(3), May 1999.

[6] R. Cunningham and E. Kohler. Making events less slippery with eel. In *Proc. HotOS-X*, June 2005.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th SOSP*, October 2001.

[8] G. L. Davies. Teaching concurrent programming with Pascal-FC. *SIGCSE Bulletin*, 22(2), 1990.

[9] U. Drepper. The native POSIX thread library for Linux. `http://people.redhat.com/drepper/nptl-design.pdf`.

[10] T. Duff. Duff’s device. `http://www.lysator.liu.se/c/duffs-device.html`.

[11] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proc. 2006 SenSys*, Nov. 2006.

[12] R. S. Engelschall. Portable multithreading: The signal stack trick for user-space thread creation. In *Proc. 2000 USENIX Annual Tech. Conference*, June 2000.

[13] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. 1st NSDI*, March 2004.

[14] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10), 1974.

[15] P. Hudak et al. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5), 1992.

[16] Humor Rainbow, Inc. OkCupid.com. `http://www.okcupid.com`.

[17] G. Jones. *Programming in Occam*. Prentice Hall International (UK) Ltd., 1986.

[18] M. Krohn. Building secure high-performance web services with OKWS. In *Proc. 2004 USENIX Annual Tech. Conference*, June 2004.

[19] J. Lemon. Kqueue: A generic and scalable event notification facility. In *Proc. 2001 FREENIX*, 2001.

[20] P. Li and S. Zdancewic. Combining events and threads for scalable network services. In *Proc. 2007 PLDI*, Jun 2007.

[21] D. Mazières. A toolkit for user-level file systems. In *Proc. 2001 USENIX Annual Tech. Conference*, June 2001.

[22] Microsoft. Inside I/O completion ports. `http://www.microsoft.com/technet/sysinternals/information/IoCompletionPorts.mspx`.

[23] MySQL AB. MySQL. `http://www.mysql.com`.

[24] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proc. 1999 USENIX Annual Tech. Conference*, June 1999.

[25] D. Pariag, T. Brecht, A. Harji, P. Buhr, and A. Shukla. Comparing the performance of web server architectures. In *Proc. 2007 EuroSys*, March 2007.

[26] K. Park and V. S. Pai. Connection conditioning: Architecture-independent support for simple, robust servers. In *Proc. 2006 NSDI*, May 2006.

[27] N. Provos. libevent — an event notification library. `http://www.monkey.org/~provos/libevent`.

[28] N. Provos. A virtual honeypot framework. In *Proc. 13th USENIX Security Symposium*, Aug 2004.

[29] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogeneous architectures. In *Proc. 27th International Symposium on Fault-Tolerant Computing*, June 1997.

[30] J. H. Reppy. CML: A higher concurrent language. In *Proc. 1991 PLDI*, June 1991.

[31] Silicon Graphics, Inc. State threads for Internet applications. `http://state-threads.sourceforge.net/docs/st.html`.

[32] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2), 1998.

[33] G. Steele. *Common Lisp*. Digital Press, Jun 1984.

[34] J. Stribling, J. Li, I. G. Councill, M. F. Kaashoek, and R. Morris. OverCite: A distributed, cooperative CiteSeer. In *Proc. 2006 NSDI*, May 2006.

[35] C. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proc. 16th SOSP*, October 1997.

[36] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *Proc. 2006 NSDI*, May 2006.

[37] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proc. HotOS-IX*, May 2003.

[38] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for Internet services. In *Proc. 19th SOSP*, October 2003.

[39] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS Defense by Offense. In *Proc. 2006 NSDI*, May 2006.

[40] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th SOSP*, October 2001.

[41] N. Zeldovich et al. Multiprocessor support for event-driven programs. In *Proc. 2003 USENIX Annual Tech. Conference*, June 2003.