Pseudo-Preemptive Threads: A New Multitasking Primitive for Reliable, Composable and Maintainable Software

Why Events, Threads and Coroutines are All Bad Ideas[1]

Benjamin Ylvisaker
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

This is the text of the abstract.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

It is hard to write multitasking software that is reliable, composable and maintainable. The most popular abstractions for building such software (events, threads, coroutines) all have significant weaknesses.

This paper describes a new abstraction called *pseudo-preemptive threads* that promises to make writing multitasking software easier. To save ink and breath, we also call this new abstraction *activities*. To validate the ideas proposed in this paper we implemented activities in a dialect of C called Charcoal and wrote several microbenchmarks to investigate the performance of activities.

This paper starts with brief a description and critique of the dominant multitasking abstractions.

Next we describe activities and provide abstract arguments for why activities do not suffer from the reliability, composability and maintainability weakness of other abstractions.

Next the paper describes the high points of the Charcoal implementation. All the code for the implementation and tests described in this paper is available in a public repository on GitHub.

Finally, we report on the results of our microbenchmarks that support our performance claims.

[1] This tongue-in-cheek subtitle is a reference to Ousterhout [11] and von Behren, et al. [14]

1.1 Contributions

This paper has two primary contributions:

- The definition of *pseudo-preemptive threads/activities*. The most concise version is that they are similar to cooperative threads, with the addition of automatic yield insertion and a scoped no-yield primitive.
- A new call frame allocation strategy for multithreaded programs called *hot stacking*. Language implementations that use this strategy allocate long-lived call frames individually in the heap and most short-lived frames contiguously in a single, shared memory area.

2. Established Multitasking Abstractions

The three most widely used multitasking frameworks are events, threads and coroutines. Each suffers from serious software engineering problems that we categorize into reliability, composability and/or maintainability. In this section we describe these problems; in later sections we argue that activities completely avoid or substantially mitigate all of them.

2.1 Events

Events are an extremely widely used abstraction for relatively simple multitasking patterns for tasks like reacting to mouse clicks and network message arrivals. Events were the only multitasking abstraction in some popular ecosystems browser JavaScript (until recently).

When an event dispatcher calls a handler/listener procedure, that call must return before another event can be handled. This pattern makes avoiding many kinds of concurrency bugs easy, because calls to handlers do not overlap in time. However, it comes at a high cost in maintainability and reliability for complex applications.

Potentially long-running (or blocking) tasks must be manually broken up into smaller handler procedures by application programmers. This leads to a style of programming referred to as *stack ripping*[1], or more colloquially *callback hell*. Callback hell can make it hard to decipher the logical flow of a task, because it is split among multiple callback procedures. This is bad for reliability, because it is easy to make a mistake in the lifetime of some piece of data (compared with, for example, threads) [14]. Callback hell is even worse for maintainability, because like any complex code it is typically much harder to understand and modify later than it was to write in the first place.

2.2 Threads (Preemptive)

Events are at the safest and least flexible end of the multitasking abstraction spectrum and threads occupy the opposite extreme. (In this paper *thread* means *preemptive thread*.)

The primary strength of threads is that they can wait/block indefinitely or run for an arbitrarily long time without preventing other threads from making progress. This makes it possible to write multitasking software in a natural single-task style (i.e. threads completely avoid callback hell).

The primary weakness of threads is that it is extremely hard to avoid and debug concurrency defects like data races, deadlocks, atomicity violations and livelocks. In the last decade a significant amount of research effort has been devoted to making it easier to write reliable multithreaded applications, because of the emergence of mainstream multiprocessor computers. While this body of work is quite impressive, most mainstream application programmers still view threads as too dangerous for multitasking programming (correctly in the current authors' opinion).

Applications that use threads and conventional concurrency control mechanisms also suffer from serious composability issues [6, 7]. The very brief summary of the arguments in the cited papers is that when using mutexes, semaphores and their cousins, application programmers must design and (somehow) enforce subtle global concurrency control properties. Using transactions instead of conventional concurrency control mechanisms promises to make threads safer and more composable. Unfortunately, questions about how transactions should interact with input/output and performance/scalability concerns have severely limited the adoption of general purpose transactional concurrency control.

Threads are the only multitasking abstractions discussed in this paper that naturally allow the parallel execution of tasks. The authors view this as closer to a bug than a feature. Serial multitasking abstractions like events and coroutines (and activities) can be used in combination with parallelism frameworks like processes and threads. The authors consider providing both parallelism and multitasking in a single language features (like threads) mostly a bad idea. Nevertheless, in the discussion section at the end of this paper we speculate about the feasibility of running activities in parallel.

2.3 Coroutines

Coroutines have an interesting history. According to Knuth, the term was coined by Melvin Conway in 1958, but coroutines remained on the margins of mainstream software practice until quite recently, a gap of more than 5 decades. A few examples of recent implementations: the async/await framework (coroutines by a different name) was added to C# in version 5, which was released in 2012; `function*` (coroutines by a different name) was added to the 2015 revision of ECMAScript; D4134 is a proposal to add coroutines to C++17.

Our interpretation of this history is that (1) coroutines are far from perfect as a multitasking primitive (otherwise they would have been widely adopted much sooner), and (2) mainstream applications have gotten more sophisticated in their use of multitasking, making life in callback hell ever more painful. Coroutines are being adopted as the least bad alternative to events.

Using the coroutine abstraction requires application programmers to partition procedures into normal procedures (functions, methods, subroutines, whatever) and coroutines. In most implementations the procedure calling syntax is overloaded; what appears to be a call to coroutine is actually a concurrent task spawn. Within the body of a coroutine definition, a *yield* (or *await*) primitive can be used to permit context switching to a different task (i.e. coroutine). Invoking yield in a normal procedure (a non-coroutine) is not permitted.

Relative to events, coroutines provide more flexibility, because multiple tasks can be in progress at the same time. Coroutines are much more resistant to concurrency bugs than threads, because only one coroutine can be active at a time, and the application explicitly states when switching between tasks is permitted.

The primary weakness of coroutines is a subtle but nasty tension with conventional procedural abstraction. It is common for application programmers to want to yield in a *procedure* called by a coroutine, but this is not possible. This can be quite inconvenient on its own, and it makes refactoring strategies like procedure extraction trickier to apply. Another consequence of this issue is that coroutines tend to be viral; if a programmer decides to convert a procedure to a coroutine (for example because it needs to wait for the arrival of a network message), it tends to be the case that any callers of that procedure need to be converted to coroutines as well.

These software engineering problems have not prevented the adoption of coroutines, but they do make it harder than it needs to be to develop multitasking software. Activities solve these problems!

2.4 Cooperative Threads

Cooperative threads are not as widely used as events, threads and coroutines, but we briefly describe them because they are closely related to activities. Cooperative threads can be seen as a compromise between coroutines and threads. Like coroutines, cooperative threads must explicitly invoke a yield primitive to context switch. Like threads, cooperative threads do not have a separate kind of procedure (i.e. coroutine/async) and yield can be invoked anywhere (i.e. it is not restricted to coroutine bodies).

Cooperative threads have a tension with procedural abstraction that is complementary to the tension in coroutine abstractions. Whether a particular procedure executes atomically depends on whether or not any of the procedures it calls will invoke yield. This can be annoying when initially writing code, and it is especially problematic during maintenance. If yield is added to a procedure that did not previously have it, there is the possibility that any caller of that procedure will have its atomicity properties violated. This is extra nasty when indirect calls are considered, because it is not possible in general to identify all call sites to a particular procedure.

A quick note on jargon: It is fairly common to see the term *coroutine* used where we use *cooperative thread*, and *stackless coroutine* where we use *coroutine*. Naturally, we prefer the terms as defined in this paper.

2.5 Others

There are other approaches to multitasking that we briefly mention here only to argue that they are not directly relevant to the main points of this paper.

Isolated processes can be used for multitasking. However, for things like handling GUI events, database transactions and network communication, shared memory is an extremely useful feature.

Functional reactive programming (FRP) is an entirely different approach to multitasking. It is interesting, but it is not clear whether it can be integrated with mainstream programming practice.

3. Pseudo-Preemptive Threads

As mentioned in the introduction, *pseudo-preemptive threads/ activities* can be understood as cooperative threads plus automatic yield insertion. The automatic yield insertion makes activities behave more like preemptive threads from an application programmer's perspective. However, activities are still far more resistant to concurrency bugs than threads.

Before going into any more detail on the definition and implementation of activities, we consider an example stolen from [9]. This example concurrently fetches DNS information using the standard `getaddrinfo` procedure. It has a parameter (`max_concur`) to limit the number of DNS requests that will be sent concurrently to avoid flooding the network with too many requests.

```
 1 void multi_getaddrinfo(
 2     int n, const char **nodes, const char **services,
 3     const struct addrinfo *hints,
 4     struct addrinfo **ress, int max_concur )
 5 {
 6     int requested = 0;
 7     max_concur = max_concur < 1 ? default : max_concur;
 8     activity_t acts[ max_concur ];
 9     for( int i = 0; i < max_concur; ++i ) {
10         activate[ &acts[i] ]() {
11             while( requested < n ) {
12                 int idx = requested;
13                 requested++;
14                 int rc = getaddrinfo(
15                     nodes[ idx ], services[ idx ],
16                     hints, ress[ idx ] );
17                 if( rc != 0 )
18                     ress[ idx ] = NULL;
19             }
20         }
21     }
22     for( int i = 0; i < max_concur; ++i ) {
23         activity_wait( acts[i] );
24     }
25 }
```

There are things worth noting about this code. The central syntactic addition is the *activate* statement (line 10). The body of an activate statement executes concurrently (but not in parallel) with whatever comes after the statement (i.e. its continuation). The program will only context switch between activities when yield is invoked. Any blocking or potentially long-running procedures (like `getaddrinfo`) will yield while the system is performing its action. Also every loop has an implicit (in the source code) yield at the end of every iteration.

Notice that multiple activities read and write the local variable `requested`. If these were threads, this would cause a data race, which at best leads to unpredictable results and at worst makes the whole program undefined. With activities this is not a problem at all; the system (compiler, processors, etc.) is not permitted to move memory operations across yields, which dramatically simplifies the concurrent memory model problem.

The activate statement takes as a parameter a pointer to application-allocated memory for storing necessary metadata about the new activity. In the above example the memory for storing activity information is allocated locally. Because of this the example procedure must wait for all the activities to finish. (Returning before they finish would cause the runtime system to access deallocated memory). It would be possible to change the interface so that the caller passes in the backing memory. In that case, the procedure could return while the activities were still fetching DNS information, allowing the application to go on with other work.

Note that as compact as this example is, it incorporates a slightly fancy feature of limiting the number of concurrent network connections to an application-specified number.

The parentheses after the activate keyword are for controlling whether local variables are access by-value or by-reference. The default is by-reference, which is how all the variables are used in this example. In some cases it is more convenient to capture the value of a local variable at activation time. This can be done by simply listing the variable inside the parentheses. Making a local variable by-value effectively makes two copies of it that can be updated independently.

3.1 Don't know

Additionally, like all properly implemented cooperative threads, activities cannot suffer from data races

[3]

3.2 Activate

The activate statement creates a new activity, which runs concurrently with the activity that spawned it. The runtime system is responsible for scheduling activities. Context switching is only possible when an activity executes a yield primitive. The semantics of a program with activities can be seen as batches of actions separated by yield invocations. Batches from different activities are interleaved to execute a complete program. Actions are not permitted to cross yield boundaries. In other words, yields define the minimum granularity of interleaving. In this way activities are identical to conventional cooperative threads.

3.3 Yield Insertion

These automatically-inserted yields make activities behave more like threads than events or coroutines from an application programmer's perspective. However, there is a big difference between activities and threads.

What distinguishes activities from cooperative threads is yield insertion. Informally, to support activities properly a language definition must ensure that a program cannot block or run indefinitely without executing a yield action. This is achieved by automatically inserting yields in certain places before a program is compiled or interpreted.

Part of the definition of activities is that by default yields should happen "frequently". But what does this mean precisely? The answer depends on the details of the rest of the programming language definition. However we can state two design rules that apply to any language. These rules are in strong tension with each other:

1. It should **not** be possible for an activity to run indefinitely without executing a yield.
2. Yields should happen as infrequently as possible.

One clear consequence of these rules is that anything that could cause an activity to pause indefinitely (e.g. a syscall) by default must be modified to allow other activities to run while the paused activity waits. More challenging, loops must be interrupted.

3.4 No-Yield

If there was no way to limit yielding, activities would suffer from many of the same concurrency bug challenges as threads. To combat this we add no-yield. Any statement, expression or procedure declaration can be "wrapped" with no-yield. In the dynamic scope of a no-yield block (statement, expression, procedure), the current activity cannot be interrupted. This is a simple and powerful tool for enforcing atomicity. We expect that most of the uses of locks in current multithreading practice can be directly replaced by the simpler and safer no-yield.

Because activities are like threads in that tasks can be written in a natural (not callback hell) style, activities do not suffer from the maintenance problems of events. Because the primary concurrency control mechanism (no-yield) enforces a simple global property, activities do not suffer from the composability problems of threads. Because the concurrency control is generally simple, activities do not suffer from the reliability problems of threads. To be fair, applications written with activities will surely have concurrency bugs; it seems impossible to write multitasking software with zero exposure to concurrency bugs. However, ...

With the addition of no-yield, activities can be seen as quite similar to coroutines with a complementary default. Normal procedures in an activity framework are like coroutines and normal procedures in a coroutine framework are like no-yield procedures in an activity framework. As similar as this argument makes coroutines and activities seem, we believe the differences are still significant. For example, in an activities framework, all procedures are part of the same type; normal procedures and no-yield procedures are not distinguished at the type level.

3.4.1 Recursive Procedure Calls

The current design of Charcoal does not follow rule #1 perfectly. Function calls and returns do not implicitly yield, which means that recursion can be used to make an activity run indefinitely without yielding. We consider this a bug, not a feature, in the language design, but we have not found any less bad alternatives.

The most obvious approach to avoiding yield-free recursion is to say that every call and/or return has an implicit yield. This idea is bad for two reasons. The simpler reason is that it violates rule #2; calls and returns happen frequently in most programs and introducing a yield for every call would be very costly for performance. The more important reason has to do with procedural abstraction. If yields were inserted around calls, then the function extraction refactoring pattern would change the concurrency behavior of the program. This seems totally unacceptable.

One could imagine trying to identify recursive calls specifically and saying that only recursive calls carry an implicit yield. However, with indirect calls it is impossible to precisely statically analyze which calls are recursive in general. This means that the language design would have to codify some rules about which classes of calls could be guaranteed to be analyzed as recursive, which seems like a fragile design. [Check on work in the FP community]

3.4.2 Programmer Control

It would be bad if programmers could not control activity yielding in some way. In Charcoal there are two primary tools for controlling yielding: no-yield and explicit yielding. Any procedure or statement can be marked "`no_yield`" which means that the activity will not yield during the execution of the statement or call to the procedure.

3.5 Comparison to Threads, Events and Coroutines

Activities avoid or mitigate all the software engineering problems identified in section **??**.

The contrast with events is simplest. Like threads, activities completely avoid callback hell. Multiple activities can be in progress simultaneously and yield can be invoked anywhere (except in no-yield mode) to context switch between tasks.

From an application programmer's prespective, activities are most similar to threads. The most obvious difference is that activities cannot easily be run in parallel. However, event handlers and coroutines also cannot be run in parallel, and they are widely used for their multitasking features.

Activities are more resistant to concurrency defects than threads. Like events, coroutines and (properly implemented) cooperative threads, data races do not exist with activities. Higher-level concurrency defects like atomicity violations, deadlocks and livelocks are still possible, of course. However, no-yield provides an extremely simple and powerful tool for concurrency control. It is very similar to *atomic* in transactional languages, but the implementation of no-yield is vastly simpler and there is no awkwardness around doing I/O operations in no-yield blocks.

From an implementation perspective,

4. Implementation

There are a few interesting features of our implementation of activities. The the following sections we describe the allocation of call frames, the implementation the yield primitive, and compiling a version of each procedure for yielding and no-yield mode.

4.1 Call Frame Allocation

One of the tricky issues in the implementation of any thread-like abstraction is the allocation of procedure call frames. The first few sections below describe existing strategies for frame allocation in multithreaded code. Then we describe a new approach that we call *hot stacking*.

4.1.1 Contiguous Allocation

In single-thread applications it is convenient and efficient to allocate *the* stack of call frames contiguously in a single large region of memory. This strategy cannot be directly used for multithreaded applications, because each thread needs to allocate and deallocate frames independently.

The most common strategy for multithreading is to allocate a moderately large area of memory in the heap for each thread's frames. Individual frames are allocated contiguously within this area. This strategy is fast and simple, but it has a nasty tension with memory efficiency.

If the allocated areas are too small, the application will experience stack overflows, which are tricky to handle properly and will simply crash many applications. If the allocated areas are too large, a significant amount of (virtual) memory is wasted. In practice most existing multithreaded software takes a conservative approach, allocating much larger areas for stacks than is strictly necessary.

Especially in 64-bit address spaces, the wasted virtual memory space is not an acute problem. However, because memory is typically allocated in page-sized chunks this allocation strategy wastes the physical memory between the top of the stack and the end of the page that it happens to be in. This can easily mean a couple of wasted kilobytes per thread.

These memory efficiency issues are one reason that most mainstream applications use only a few threads, and very few applications use more than a few dozen. For software architectures that might require more than this limit, thread pooling is a common solution. However, thread pooling brings its own inefficiencies and software engineering challenges.

4.1.2 Individual Heap Allocation

A completely different approach to frame allocation is individually allocating each frame in the heap. This avoids the memory concerns associated with contiguous allocation. However, heap allocation of frames comes at a significant performance cost for call and return operations.

Simple implementations of heap allocation are generally more than an order of magnitude slower than contiguous allocation (see more details in the microbenchmarking section below). More sophisticated implementations can be substantially more efficient (e.g. [13]). However, even the most efficient implementations that we are aware of are still measurably slower than contiguous allocation for call-heavy code.

4.1.3 Split/Segmented Stacks

For completeness, we briefly mention another strategy called *split* or *segmented* stacks. The idea is that stack space is allocated in small chunks (or segments). The common case call/return execution looks like traditional contiguous allocation. When a thread reaches the end of its segment it allocates a new one and links them together. This idea is appealing: the implementers of Rust and Go both used it. Unfortunately it has really unpleasant *uncommon* case behavior: the implementers of Rust [2] and Go (search "contiguous stacks in go") both abandoned it in later versions.

4.1.4 Hot Stacking

The call frame allocation strategy introduced in this paper is a hybrid of contiguous and individual heap allocation. The primary observation is that the slowness of individual allocation is only important for short-lived calls. For long-lived calls (generally calls nearer the base of the call stack), the overhead of the call and return operations can be amortized over the long running time of the call and its callees. So the main idea of hot stacking is that long-lived frames are individually allocated and (most) short-lived frames are contiguously allocated in a memory area shared by multiple activities.

For this strategy to work correctly, the frames allocated in the shared area must all be deallocated before context switching to another task. For regular (cooperative) threads it's not clear how this deallocation could be enforced easily. Activities make it practical to use hot stacking, because the implementation can use no-yield mode as an indicator of when it should use contiguous allocation. Context switches are not permitted in no-yield mode, so there is no problem with needing/wanted to context switch when frames in the shared contiguous area are still in use.

In the microbenchmarking section below we provide evidence that this strategy captures most of the benefits of heap allocation (per-activity memory overhead is very small) and contiguous allocation (fast calls and returns when it matters most). It is certainly possible for short-lived calls to happen in yielding mode, which means that the frames will be individually allocated (slow). This should be a very small performance concern for most applications.

The name *hot stacking* is a reference to a practice used in some military and business organizations called *hot racking* or *hot desking*. The idea is that some limited resource (e.g. a bunk or desk) is used in shifts by multiple people. In our case, the resource is the memory area for contiguous frame allocation, and it is shared in shifts by multiple activities. The word *hot* is also a reference to the fact that this top of stack area should remain very hot in the memory hierarchy/cache sense as long as any code is running.

4.2 Fast and Slow

There are important differences at the implementation level between how code executes in yielding versus no-yield mode, not least among them the call frame allocation strategy. To implement this dual mode concept in a reasonably simple and efficient way, the current Charcoalimplementation generates two versions of each procedure: one for each mode. The yielding mode implementation includes inserted yields and assumes its own frame was individually allocated. The no-yield mode implementation does not include yields; for most intents and purposes it is a simple translation to plain C.

Of course making a single implementation that could be run in either mode would be possible. Such an implementation would need to branch on which mode it was running, potentially quite frequently. This seemed like a performance killer for inner loop code, though we have not investigated this assumption yet.

This implementation strategy could lead to a substantial increase in code size. We address this issue in the microbenchmarks. The main intuition about why this does not seem to be a big problem is that most procedures are only ever called in yielding mode or no-yield mode. A sufficiently smart linker can identify the large amount of dead code and eliminate it.

This is kinda related to ideas from the Cilk-5 implementation [5].

4.2.1 Function Pointers

One challenge with a dual implementation strategy is how to handle function pointers. In general, when the address of a function is taken there is no way to know which mode the function will later be called in. Therefore simply taking the address of either implementation would be at best complicated and inefficient, and quite likely lead to subtle bugs. The current implementation generates a small piece of code for every function that can be called in either yielding or no-yield mode. A parameter is passed by the caller to indicate which mode it is in. The generated code calls the appropriate implementation. This makes indirect calls somewhat more expensive than in plain C. However, indirect calls are already expensive enough that there is an extensive body of research on how to convert them to direct calls (e.g. [4]), so adding a modest amount of overhead to indirect calls should not have a large performance impact on many applications.

4.3 Yield Implementation

The most important factor in the implementation of the yield primitive is that most yield invocations should not result in context switching, even if multiple activities are ready to run. In well designed activity code, the time between yield invocations should be in the range of microseconds to milliseconds. Context switching has the moderate direct cost of manipulating a handful of data structures in the runtime system, and the potentially higher indirect cost of using the memory hierarchy poorly because the activities have different working sets. Therefore in normal operation yields should lead to context switches at a relatively low frequency, perhaps every few milliseconds.

The speed of the yield primitive itself is somewhat important. As described in the previous section, code that is compiled in no-yield mode simply does not have yields at all, so yield performance is not an issue for the most performance critical loops. However, we expect that moderately frequent yielding (perhaps as frequently as many per microsecond) would be common in real-world code. Therefore, the performance of the yield primitive does matter to some degree.

The simplest implementation of yield would check a counter or clock of some sort. Reasonably efficient implementations of this strategy would certainly be non-portable (e.g. using processor-specific counter registers) and probably still be somewhat expensive. Instead the current Charcoalimplementation uses periodic system timers/alarms that deliver a signal to the program. The handler for these signals atomically modifies a global variable. The yield primitive atomically reads this global variable; as long as it has not changed the program continues execution immediately. Therefore, in the common case the cost of a yield is only an atomic read and a (highly predictable) branch, plus a fast call and return to get to the yield code itself. (Of course the yield code can be inlined, but that is *not* obviously a good idea because the branch prediction hardware works much better if there is only a single branch instruction for all yielding.)

4.4 Asynchronous System Calls

One important part of making activities work properly is having system calls work "in the background". In our implementation we used libuv for this. Using libuv seems to mostly have been a good decision. One problem is that libuv was designed with a "push" architecture for event dispatching. A "pull" architecture would have made it possible to do asynchronous calls more easily. What we did was run the libuv event loop in a different thread. Every time user code wants to call a yielding system function it has to send an asynchronous message to the event loop thread. This is clearly not ideal, but it seems like something that could be improved with some careful engineering.

4.5 Translation

We use a modified version of Cil [10] to translate Charcoal to plain C with calls to our runtime system. Cil is great.

4.6 Odds and Ends

The system described in this paper shares a lot of ideas with Continuation Passing C (CPC) [8]. In some of the CPC publications, the designers claimed that it was not possible to implement some more exotic features of C, like setjmp/longjmp and alloca. A detailed description of our implementation of these features is beyond the scope of this paper, but a working implementation is available in the project repository. The hot stacking frame allocation strategy makes the implementation of these features (and related things like exceptions) more complicated, but not impossible.

5. Benchmarking

To establish the practicality of activities, we implemented XXX microbenchmarks to compare our implementation of Charcoal against threads (C with pthreads) and events (libuv, a popular event loop library written in C).

The first couple microbenchmarks demonstrate that basic concurrency primitives, like task spawning and task switching, are far faster with activities than threads. This is not particularly surprising. Because threads can be interrupted at any time, context switching requires the operating system to copy all processor state to memory. Thread creation requires the allocation of lots of system resources.

5.1 Task Memory Overhead

The first microbenchmark gets at memory overhead by measuring how many concurrent tasks can exist. To measure this the benchmark simply spawns tasks until memory allocation fails; however many tasks existed before the failure is the limit. For activities and event listeners this overhead is small and more or less fixed. For threads this overhead is harder to put a single number on, because most multithreading APIs allow the amount of memory reserved for the call stack to be controlled by the application. For this benchmark we simply used the default stack size in the installed pthreads implementation. The results of this test are not surprising, but we believe it is worth emphasizing that it is not practical to use more than a few hundred or maybe a few thousand threads, whereas the limit on activities or event listeners is several orders of magnitude higher.

A legitimate question is whether it is ever good software architecture to have many concurrent tasks. We do not attempt to answer that question definitively here, but only suggest that when there are several orders of magnitude differences between somewhat comparable things in engineering, that usually means that the "cheaper" option opens qualitatively different possibilities.

memory limit: thread: 1,000 activity: 1,000,000 libuv: 1,000,000,000

5.2 Task Spawn Speed

The second microbenchmark measures how quickly new tasks can be spawned. The task in this test waits for the previous task to finish, then spawns the next task. With pthreads, our test system was able to spawn about 60 threads per millisecond; our activities implementation was about to spawn over 2,000 per millisecond. This difference of well over an order of magnitude is significant, because it means that it can be efficient to spawn an activity for much smaller units of work than it would be to spawn a thread. In other words, there is no reason to build *activity pools*; it is always a better idea to simply spawn an activity for the task at hand.

spawns per ms: thread: 60 activity: 2,400 libuv: 7,100

5.3 Task Switching

The third microbenchmark measures how quickly the system can switch from one task to another. For this test we spawned 20 tasks and organized them in a ring, each waiting for a signal of some kind from its neighbor. The test injects a signal at one point and then measures how quickly it can cycle around the ring. On our system we measured about 600 switches per millisecond with pthreads and 5,000 per millisecond with activities. This difference is almost an order of magnitude. Like the spawning benchmark, the main takeaway is that the overhead for activities is low enough that individual activities can do quite small units of work per context switch without paying a high overall efficiency penalty.

context switching (bucket brigade): thread: 630 activity: 4,900

5.4 Just Calling

```
int f( int d, int x ) {
    if( d > 0 )
        return f( d - 1, f( d - 1, x ) );
    else
        return small_computation( x );
```

The fourth microbenchmark measures the overhead of heap-allocating call frames. This test has a very simple recursive function that calls itself twice. At the leaves of the recursion is a very small computation, just complicated enough to prevent a smart compiler from statically computing the whole answer. For this benchmark plain C is more than an order of magnitude faster than Charcoal. This difference is painful, but there are several things to say about it. First, the Charcoal implementation is a research prototype, so it is quite likely that good engineering work would close the gap to some extent. Second, this is a microbenchmark; no application spends all of its time in calls and returns, so real application performance impact will be proportionally smaller. Third, procedure calling in no-yield mode is just as efficient as plain C, which means that the programmer has some control over this overhead.

To investigate this foo further, ...

```
int f( int d, int x ) {
    if( d > N )
        return f( d - 1, f( d - 1, x ) );
    else if( d > 0 )
        no_yield {
            return f( d - 1, f( d - 1, x ) );
        }
    else
        return small_computation( x );
```

This shows that only the N closest calls to the leaves need to be in no-yield mode for calling overhead to get below XX%.

just calling: C: 590 Charcoal: 13

5.5 Yielding

The yield primitive is implemented to be very fast when the current activity will continue executing (i.e. not switch to another activity). The key implementation trick here is described in section **??**. Even with this clever implementation, yielding is still not free. To quantify its expense, we benchmarked a simple but problematic function: `strcmp`. `strcmp` is tricky for a few reasons: First, the input strings could be very long, so it is not acceptable to make the implementation simply never yield. Second, the body of the loop is extremely simple; good implementations are just a few assembly instructions. This means that yielding every iteration would cause significant performance overhead. Third, whether each iteration executes depends on the computation of the previous iteration, so simple loop tiling/blocking tricks don't work. The best-performing implementation we have found so far looks like:

```
int strcmp( const char *s1, const char *s2 ) {
    while_no_yield( *s1 && !( *s1 - *s2 ) ) {
        ++s1; ++s2;
        if( !( s1 & 0xFF ) )
            yield;
    }
    return *s1 - *s2;
}
```

`while_no_yield` is a special variant of the regular while loop that does not have an implicit yield after every iteration. This is different from wrapping a regular while loop in `no_yield`, because the latter would prevent nested yields from happening, whereas the former does not.

In the code about `B` is a bit-mask with some number of the low bits set to one. The effect is that once every 2^N iterations there is a yield, where N is the number of bits set in the mask. This implementation gets the overhead down to 15% (???), which is neither awful nor good enough to never think about this issue again. One reasonable workaround is that when callers of `strcmp` are certain that the strings are not especially long, they can make the call in no-yield mode. In this case, it should be just as efficient as plain C.

strcmp: C: 1000 Charcoal: 100

5.6 Code Size

5.7 Microbenchmark Summary

6. Foreign Code

Foreign code (including legacy code) will never yield. This could lead to starvation pretty easily. Here are three strategies:

- Do nothing. Just run the foreign code. This is a perfectly reasonable strategy as long as the foreign code does not run for a long time.
- Run the foreign code in its own thread. If it has not returned by the end of some time slice, pause it to allow other activities to run. This runs the risk of creating atomicity violations galore. It also reintroduces the possibility of data races. However, it might be a reasonable strategy in situations where there is very little sharing between the foreign code and the rest of the application.
- Run the foreign code in its own thread, but only interrupt it at special "safe-ish" points, like system calls. This is a compromise between the previous two strategies in the sense that it opens the door to both starvation and atomicity violations, but provides some (imperfect) protection against both.

We have not thought at all about what the best default is or what syntactic sugar would be nice.

Another important implementation issue to consider is foreign code that calls back in to activity-aware code. There will definitely be some fancy footwork necessary there, no matter which strategy is used.

7. Related Work

Continuation Passing C.

Threads without the Pain

8. Summary and Discussion

In this paper we introduced pseudo-preemptive threads, a new concurrency primitive, called activities for short. Activities have advantages in reliability, composability and/or maintainability over the most common approaches to writing multitasking software (events, threads and coroutines). From an application programmer's perspective activities feel most like threads, but they do not suffer from several of the nasty concurrency problems that plague multithreaded software. Activities can be thought of as actually working the way inexperienced programmers think threads work.

One weakness that activities share with most multitasking frameworks is that activities cannot easily be run in parallel. Naively running activities simultaneously on parallel processors would immediately violate the simple sequential memory model. However, it is interesting to consider what it would take to run activities in parallel without violating their semantics. The most obvious way to accomplish that would be with a transactional memory system; every activity would always be running in a transaction. Yielding would cause one transaction to end and the next to begin.

The authors are not optimistic that such a transaction based system would work well, but similar things have been tried. (Stone OCM)

[12]

Acknowledgments

Acknowledgments, if needed.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-880446-00-6. URL `http://dl.acm.org/citation.cfm?id=647057.713851`.

[2] B. Anderson. Abandoning segmented stacks in rust. rust-dev mailing list, November 2013.

[3] H.-J. Boehm. How to miscompile programs with "benign" data races. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2001252.2001255`.

[4] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-60160-0. URL `http://dl.acm.org/citation.cfm?id=646153.679523`.

[5] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. . URL `http://doi.acm.org/10.1145/277650.277725`.

[6] D. Grossman. The transactional memory / garbage collection analogy. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 695–706, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. . URL `http://doi.acm.org/10.1145/1297027.1297080`.

[7] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9. . URL `http://doi.acm.org/10.1145/1065944.1065952`.

[8] G. Kerneis. *Continuation-Passing C: Program Transformations for Compiling Concurrency in an Imperative Language*. PhD thesis, Universit Paris Diderot Paris 7, 2013.

[9] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 7:1–7:14, Berkeley, CA, USA, 2007. USENIX Association. ISBN 999-8888-77-6. URL `http://dl.acm.org/citation.cfm?id=1364385.1364392`.

[10] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4. URL `http://dl.acm.org/citation.cfm?id=647478.727796`.

[11] J. Ousterhout. Why threads are a bad idea (for most purposes). In *USENIX Technical Conference*, 1996.

[12] M. E. ONeill and C. A. Stone. Making impractical implementations practical: Observationally cooperative multithreading using hle. In *Proceedings of the 10th ACM SIGPLAN Workshop on Transactional Computing*, June 2015.

[13] Z. Shao and A. W. Appel. Efficient and safe-for-space closure conversion. *ACM Trans. Program. Lang. Syst.*, 22(1):129–161, Jan. 2000. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/345099.345125`.

[14] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251054.1251058`.