Cooperative Task Management without Manual Stack Management

or, Event-driven Programming is Not the Opposite of Threaded Programming

Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, John R. Douceur
Microsoft Research
1 Microsoft Way, Redmond, Washington 98052
{*adya, howell, theimer, bolosky, johndo*}*@microsoft.com*

Abstract

Cooperative task management can provide program architects with ease of reasoning about concurrency issues. This property is often espoused by those who recommend "event-driven" programming over "multithreaded" programming. Those terms conflate several issues. In this paper, we clarify the issues, and show how one can get the best of both worlds: reason more simply about concurrency in the way "event-driven" advocates recommend, while preserving the readability and maintainability of code associated with "multithreaded" programming.

We identify the source of confusion about the two programming styles as a conflation of two concepts: *task management* and *stack management*. Those two concerns define a two-axis space in which "multithreaded" and "event-driven" programming are diagonally opposite; there is a third "sweet spot" in the space that combines the advantages of both programming styles. We point out pitfalls in both alternative forms of stack management, *manual* and *automatic*, and we supply techniques that mitigate the danger in the automatic case. Finally, we exhibit adaptors that enable automatic stack management code and manual stack management code to interoperate in the same code base.

1 Introduction

Our team embarked on a new project and faced the question of what programming model to use. Each team member had been burned by concurrency issues in the past, encountering bugs that were difficult to even reproduce, much less identify and remove. We chose to follow the collective wisdom of the community as we understood it, which suggests that an "event-driven" programming model can simplify concurrency issues by reducing opportunities for race conditions and deadlocks [Ous96]. However, as we gained experience, we realized that the popular term "event-driven" conflates several distinct concepts; most importantly, it suggests that a gain in reasoning about concurrency cannot be had without cumbersome manual stack management. By separating these concerns, we were able to realize the "best of both worlds."

In Section 2, we define the two distinct concepts whose conflation is problematic, and we touch on three related concepts to avoid confusing them with the central ideas. The key concept is that one can choose the reasoning benefits of cooperative task management without sacrificing the readability and maintainability of automatic stack management. Section 3 focuses on the topic of stack management, describing how software evolution exacerbates problems both for code using manual stack management as well as code using automatic stack management. We show how the most insidious problem with automatic stack management can be alleviated. Section 4 presents our hybrid stack-management model that allows code using automatic stack management to coexist and interoperate in the same program with code using manual stack management; this model helped us find peace within a group of developers that disagreed on which method to use. Section 5 discusses our experience in implementing these ideas in two different systems. Section 6 relates our observations to other work and Secton 7 summarizes our conclusions.

2 Definitions

In this section, we define and describe five distinct concepts: *task management*, *stack management*, *I/O* *response management*, *conflict management*, and *data partitioning*. These concepts are not completely orthogonal, but considering them independently helps us understand how they interact in a complete design. We tease apart these concerns and then return to look at how the concepts have been popularly conflated.

2.1 Task management

One can often divide the work a program does into conceptually separate tasks: each task encapsulates a control flow, and all of the tasks access some common, shared state. High-performance programs are often written with *preemptive* task management, wherein execution of tasks can interleave on uniprocessors or overlap on multiprocessors. The opposite approach, *serial* task management, runs each task to completion before starting the next task. Its advantage is that there is no conflict of access to the shared state; one can define inter-task invariants on the shared state and be assured that, while the present task is running, no other tasks can violate the invariants. The strategy is inappropriate, however, when one wishes to exploit multiprocessor parallelism, or when slow tasks must not defer later tasks for a long time.

A compromise approach is *cooperative* task management. In this approach, a task's code only yields control to other tasks at well-defined points in its execution; usually only when the task must wait for long-running I/O. The approach is valuable when tasks must interleave to avoid waiting on each other's I/O, but multiprocessor parallelism is not crucial for good application performance.

Cooperative task management preserves some of the advantage of serial task management in that invariants on the global state only need be restored when a task explicitly yields, and they can be assumed to be valid when the task resumes. Cooperative task management is harder than serial in that, if the task has local state that depends on the global state before yielding, that state may be invalid when the task resumes. The same problem appears in preemptive task management when releasing locks for the duration of a slow I/O operation [Bir89].

One penalty for adopting cooperative task management is that every I/O library function called must be wrapped so that instead of blocking, the function initiates the I/O and yields control to another task. The wrapper must also arrange for its task to become schedulable when the I/O completes.

2.2 Stack management

The common approach to achieving cooperative task management is to organize a program as a collection of event handlers. Say a task involves receiving a network message, reading a block from disk, and replying to the message. The receipt of the message is an event; one procedure handles that event and initiates the disk I/O. The receipt of the disk I/O result is a second event; another procedure handles that event and constructs the network reply message. The desired task management is achieved, in that other tasks may make progress while the present task is waiting on the disk I/O.

We call the approach just described *manual stack management*. As we argue in Section 3.1, the problem is that the control flow for a single conceptual task and its task-specific state are broken across several language procedures, effectively discarding language scoping features. This problem is subtle because it causes the most trouble as software evolves. It is important to observe that one can choose cooperative task management for its benefits while exploiting the *automatic* stack management afforded by a structured programming language. We describe how in Section 3.3.

Some languages have a built-in facility for transparently constructing closures; Scheme's call-with-current-continuation is an obvious example [HFW84, FHK84]. Such a facility obviates the idea of manual stack management altogether. This paper focuses on the stack management problem in conventional systems languages without elegant closures.

2.3 I/O management

While this paper focuses on the first two axes, we explicitly mention three other axes to avoid confusing them with the first two. The first concerns the question of *synchronous* versus *asynchronous I/O management*, which is orthogonal to the axis of task management. An I/O programming interface is *synchronous* if the calling task appears to block at the call site until the I/O completes, and then resume execution. An *asynchronous* interface call appears to return control to the caller immediately. The calling code may initiate several overlapping asynchronous operations, then later wait for the results to arrive, perhaps in arbitrary order. This form of concurrency is different than task management because I/O operations can be considered independently from the computation they overlap, since the I/O does not access the shared state of the computation. Code obeying any of the forms of task management can call either type of I/O interface. Furthermore, with the right primitives, one can build wrappers to make synchronous interfaces out of asynchronous ones, and vice versa; we do just that in our systems.

2.4 Conflict management

Different task management approaches offer different granularities of atomicity on shared state. *Conflict management* considers how to convert available atomicity to a meaningful mechanism for avoiding resource conflicts. In serial task management, for example, an entire task is an atomic operation on shared state, so no explicit mechanism is needed to avoid inter-task conflicts on shared resources. In the limiting case of preemptive task management, where other tasks are executing concurrently, tasks must ensure that invariants hold on the shared state all the time.

The general solution to this problem is synchronization primitives, such as locks, semaphores, and monitors. Based on small atomic operations supplied by the machine or runtime environment, synchronization primitives let us construct mechanisms that maintain complex invariants on shared state that always hold. Synchronization mechanisms may be pessimistic or optimistic. A pessimistic mechanism locks other tasks out of the resources it needs to complete a computation. An optimistic primitive computes results speculatively; if the computation turns out to conflict with a concurrent task's computation, the mechanism retries, perhaps also falling back on a pessimistic mechanism if no forward progress is being made.

Cooperative (or serial) task management effectively provides arbitrarily large atomic operations: all of the code executed between two explicit yield points is executed atomically. Therefore, it is straightforward to build many complex invariants safely. This approach is analogous to the construction of atomic sequences with interrupt masking in uniprocessor OS kernels. We discuss in Section 3.3 how to ensure that code dependent on atomicity stays atomic as software evolves.

2.5 Data partitioning

Task management and conflict management work together to address the problem of potentially-concurrent access to shared state. By partitioning that state, we can reduce the number of opportunities for conflict. For example, task-specific state needs no concurrency considerations because it has been explicitly *partitioned* from shared state. Data is transferred between partitions by value; one must be careful to handle implicit references (such as a value that actually indexes an array) thoughtfully.

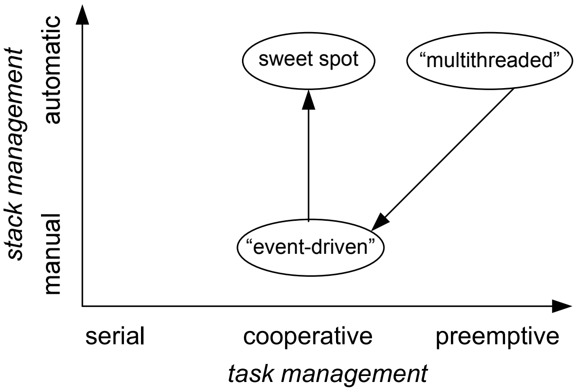

Figure 1: Two axes that are frequently conflated.

Explicitly introducing data partitions to reduce the degree of sharing of shared state can make it easier to write and reason about invariants on each partition; data partitioning is an orthogonal approach to those mentioned previously.

2.6 How the concepts relate

We have described five distinct concepts. They are not all precisely orthogonal, but it is useful to consider the effects of choices in each dimension separately. Most importantly, for the purposes of this paper, the task management and stack management axes are indeed orthogonal (see Figure 1).

The idea behind Figure 1 is that conventional concurrent programming uses preemptive task management and exploits the automatic stack management of a standard language. We often hear this point in the space referred to by the term "threaded programming." The second interesting point in the space is "event-driven programming," where cooperative tasks are organized as event handlers that yield control by *returning* control to the event scheduler, manually unrolling their stacks. This paper is organized around the observation that one can choose cooperative task management while preserving the automatic stack management that makes a programming language "structured;" in the diagram, this point is labeled the "sweet spot."

3 Stack management

Given our diagram one might ask, "what are the *pro*s and *con*s of the two forms of stack management? We address that question here. We present the principal advantages and disadvantages of each form, emphasizing how software evolution exacerbates the disadvantages of each. We also present a technique that mitigates the principal disadvantage of automatic stack management.

3.1 Automatic versus manual

Programmers can express a task employing either *automatic* stack management or *manual* stack management. With automatic stack management, the programmer expresses each complete task as a single procedure in the source language. Such a procedure may call functions that block on I/O operations such as disk or remote requests. While the task is waiting on a blocking operation, its current state is kept in data stored on the procedure's program stack. This style of control flow is one meaning often associated with the term "procedure-oriented."

In contrast, manual stack management requires a programmer to rip the code for any given task into event handlers that run to completion without blocking. Event handlers are procedures that can be invoked by an *event-handling scheduler* in response to events, such as the initiation of a task or the response from a previously-requested I/O. To initiate an I/O, an event handler "E_1" schedules a request for the operation but does not wait for the reply. Instead, E_1 registers a task-specific object called a *continuation* [FHK84] with the event-handling scheduler. The continuation bundles state indicating where E_1 left off working on the task, plus a reference to a different event-handler procedure E_2 that encodes what should be done when the requested I/O has completed. After having initiated the I/O and registering the continuation, E_1 returns control to the event-handling scheduler. When the event representing the I/O completion occurs, the event-handling scheduler calls E_2, passing E_1's bundled state as an argument. This style of control flow is often associated with the term "event-driven."

To illustrate these two stack-management styles, consider the code for a function, `GetCAInfo`, that looks in an in-memory hash table for a specified certificate-authority id and returns a pointer to the corresponding object. A certificate authority is an entity that issues certificates, for example for users of a file system.

```
CAInfo GetCAInfo(CAID caId) {
  CAInfo caInfo = LookupHashTable(caId);
  return caInfo;
}
```

Suppose that initially this function was designed to handle a few globally known certificate authorities and hence all the CA records could be stored in memory. We refer to such a function as a *compute-only* function: because it does not pause for I/O, we need not consider how its stack is managed across an I/O call, and thus the automatic stack management supplied by the compiler is always appropriate.

Now suppose the function evolves to support an abundance of CA objects. We may wish to convert the hash table into an on-disk structure, with an in-memory cache of the entries in use. `GetCAInfo` has become a function that may have to yield for I/O. How the code evolves depends on whether it uses automatic or manual stack management.

Following is code with automatic stack management that implements the revised function:

```
CAInfo GetCAInfoBlocking(CAID caId) {
  CAInfo caInfo = LookupHashTable(caId);
  if (caInfo != NULL) {
    // Found node in the hash table
    return caInfo;
  }
  caInfo = new CAInfo();
  // DiskRead blocks waiting for
  // the disk I/O to complete.
  DiskRead(caId, caInfo);
  InsertHashTable(caId, CaInfo);
  return caInfo;
}
```

To achieve the same goal using manual stack management, we rip the single conceptual function `GetCAInfoBlocking` into two source-language functions, so that the second function can be called from the event-handler scheduler to continue after the disk I/O has completed. Here is the continuation object that stores the bundled state and function pointer:

```
class Continuation {
```

```
  // The function called when this
  // continuation is scheduled to run.
  void (*function)(Continuation cont);
  // Return value set by the I/O operation.
  // To be passed to continuation.
  void *returnValue
  // Bundled up state
  void *arg1, *arg2, ...;
}
```

Here is the original function, ripped into the two parts that function as event handlers:

```
void GetCAInfoHandler1(CAID caId,
               Continuation *callerCont)
{
  // Return the result immediately if in cache
  CAInfo *caInfo = LookupHashTable(caId);
  if (caInfo != NULL) {
    // Call caller's continuation with result
    (*callerCont->function)(caInfo);
    return;
  }

  // Make buffer space for disk read
  caInfo = new CAInfo();
  // Save return address & live variables
  Continuation *cont = new
    Continuation(&GetCAInfoHandler2,
           caId, caInfo, callerCont);
  // Send request
  EventHandle eh =
      InitAsyncDiskRead(caId, caInfo);
  // Schedule event handler to run on reply
  // by registering continuation
  RegisterContinuation(eh, cont);
}


void GetCAInfoHandler2(Continuation
*cont) {
  // Recover live variables
  CAID caId = (CAID) cont->arg1;
  CAInfo *caInfo = (CAInfo*) cont->arg2;
  Continuation *callerCont =
      (Continuation*) cont->arg3;
  // Stash CAInfo object in hash
  InsertHashTable(caId, caInfo);
  // Now "return" results to original caller
  (*callerCont->function)(callerCont);
}
```

Note that the signature of `GetCAInfo` is different from that of `GetCAInfoHandler1`. Since the desired result from what used to be `GetCAInfo` will not be available until `GetCAInfoHandler2` runs sometime later, the caller of `GetCAInfoHandler1` must pass in a continuation that `GetCAInfoHandler2` can later invoke in order to return the desired result via the continuation record. That is, with manual stack management, a statement that returns control (and perhaps a value) to a caller must be simulated by a function call to a continuation procedure.

3.2 Stack Ripping

In conventional systems languages, such as C++, which have no support for closures, the programmer has to do a substantial amount of manual stack management to yield for I/O operations. Note that the function in the previous section was ripped into two parts because of one I/O call. If there are more I/O calls, there are even more rips in the code. The situation gets worse still with the presence of control structures such as *for* loops. The programmer deconstructs the language stack, reconstructs it on the heap, and reduces the readability of the code in the process.

Furthermore, debugging is impaired because when the debugger stops in `GetCAInfoHandler2`, the call stack only shows the state of the current event handler and provides no information about the sequence of events that the ripped task performed before arriving at the current event handler invocation. Theoretically, one can manually recover the call stack by tracing through the continuation objects; in practice we have observed that programmers hand-optimize away tail calls, so that much of the stack goes missing.

In summary, for each routine that is ripped, the programmer will have to manually manage procedural language features that are normally handled by a compiler:

function scoping Now two or more language functions represent a single conceptual function.

automatic variables Variables once allocated on the stack by the language must be moved into a new state structure stored on the heap to survive across yield points.

control structures The entry point to every basic block containing a function that might block must be reachable from a continuation, and hence must be a separate language-level function. That is, conceptual functions with loops must be ripped into more than two pieces.

debugging stack The call stack must be manually recovered when debugging, and manual optimization of tail calls may make it unrecoverable.

Software evolution substantially magnifies the problem of function ripping: when a function evolves from being compute-only to potentially yielding, *all* functions, along every path from the function whose concurrency semantics have changed to the root of the call graph may potentially have to be ripped in two. (More precisely, all functions up a branch of the call graph will have to be ripped until a function is encountered that already makes its call in continuation-passing form.) We call this phenomenon "stack ripping" and see it as the primary drawback to manual stack management. Note that, as with all global evolutions, functions on the call graph may be maintained by different parties, making the change difficult.

3.3 Hidden concurrency assumptions

The huge advantage of manual stack management is that every yield point is explicitly visible in the code at every level of the call graph. In contrast, the call to `DiskRead` in `GetCAInfo` hides potential concurrency. Local state extracted from shared state before the `DiskRead` call may need to be reevaluated after the call. Absent a comment, the programmer cannot tell which function calls may yield and which local state to revalidate as a consequence thereof.

As with manual stack management, software evolution makes the situation even worse. A call that did not yield yesterday may be changed tomorrow to yield for I/O. However, when a function with manual stack management evolves to yield for I/O, its signature changes to reflect the new structure, and the compiler will call attention to any callers of the function unaware of the evolution. With automatic stack management, such a change is syntactically invisible and yet it affects the semantics of *every* function that calls the evolved function, either directly or transitively.

The dangerous aspect of automatic stack management is that a semantic property (*yielding*) of a called procedure dramatically affects how the calling procedure should be written, but there is no check that the calling procedure is honoring the property. Happily, concurrency assumptions can be declared explicitly and checked statically or dynamically.

A static check would be ideal because it detects violations at compile time. Functions that yield are tagged with the *yielding* property, and each block of a calling function that assumes that it runs without yielding is marked `atomic`. The compiler or a static tool checks that functions that call *yielding* functions are themselves marked *yielding*, and that no calls to *yielding* functions appear inside `atomic` blocks. In fact, one could reasonably abuse an exception declaration mechanism to achieve this end.

A dynamic check is less desirable than a static one because violations are only found if they occur at runtime. It is still useful in that violations cause an immediate failure, rather than subtly corrupting system state in a way that is difficult to trace back to its cause. We chose a dynamic check because it was quick and easy to implement. Each block of code that depends on atomicity begins with a call to `startAtomic()` and ends with a call to `endAtomic()`. The `startAtomic()` function increments a private counter and `endAtomic()` decrements it. When any function tries to block on I/O, `yield()` asserts that the counter is zero, and dumps core otherwise.

Note that in evolving code employing automatic stack management, we may also have to modify every function extending along every path up the call graph from a function whose concurrency semantics have changed. However, whereas manual stack management implies that each affected function must be torn apart into multiple pieces, automatic-stack-management code may require no changes or far less intrusive changes. If the local state of a function does not depend on the yielding behavior of a called function, then the calling function requires no change. If the calling function's local state *is* affected, the function must be modified to revalidate its state; this surgery is usually local and does not require substantial code restructuring.

4 Hybrid approach

In our project there are passionate advocates for each of the two styles of stack management. There is a hybrid approach that enables both styles to coexist in the same code base, using adaptors to connect between them. This hybrid approach also enables a project to be written in one style but incorporate legacy code written in the other.

In the Windows operating system, "threads" are scheduled preemptively and "fibers" are scheduled cooperatively. Our implementation achieves cooperative task management by scheduling multiple fibers on a single thread; at any given time, only one fiber is active.

In our design, a scheduler runs on a special fiber called MainFiber and schedules both manual stack management code (event handlers) and automatic stack management code. Code written with automatic stack management, that expects to block for I/O, always runs on a fiber other than MainFiber; when it blocks, it always yields control back to MainFiber, where the scheduler selects the next task to schedule. Compute-only functions, of course, may run on any fiber, since they may be freely called from either context.

Both types of stack management code are scheduled by the same scheduler because the Windows fiber package only supports the notion of explicitly switching from one fiber to another specified fiber; there is no notion of a generalized yield operation that invokes a default fiber scheduler. Implementing a combined scheduler also allowed us to avoid the problem of having two, potentially conflicting, schedulers running in parallel: one for event handlers and one for fibers.

There are other ways in which the two styles of code can be made to interact. We aimed for simplicity and to preserve our existing code base that uses manual stack management. Our solution ensures that code written in either style can call a function implemented in the other style without being aware that the other stack management discipline even exists.

To illustrate the hybrid approach, we show an example that includes calls across styles in both directions. The example involves four functions: FetchCert, GetCertData, VerifyCert, and GetCAInfo. (GetCAInfo was introduced in Section 3.1). FetchCert fetches a security certificate using GetCertData and then calls VerifyCert in order to confirm its validity. VerifyCert, in turn, calls GetCAInfo in order to obtain a CA with which to verify a certificate. Here is how the code would look with serial task management:

```
bool FetchCert(User user,
               Certificate *cert) {
  // Get the certificate data from a
  // function that might do I/O
  certificate = GetCertData(user);
  if (!VerifyCert(user, cert)) {
     return false;
  }
}

bool VerifyCert(User user,
                Certificate *cert) {
    // Get the Certificate Authority (CA)
    // information and then verify cert
    ca = GetCAInfo(cert);
    if (ca == NULL) return false;
    return CACheckCert(ca, user, cert);
}

Certificate* GetCertData(User user) {
  // Look up certificate in the memory
  // cache and return the answer.
  // Else fetch from disk/network
  if (Lookup(user, cert))
     return certificate;
  certificate = DoIOAndGetCert();
  return certificate;
}
```

Of course, we want to rewrite the code to use cooperative task management, allowing other tasks to run during the I/O pauses, with different functions adhering to each form of stack management. Suppose that VerifyCert is written with automatic stack management and the remaining functions (FetchCert, GetCertData, GetCAInfo) are implemented with manual stack management (using continuations). We will define adaptor functions that route control flow between the styles.

4.1 Manual calling automatic

Figure 2 is a sequence diagram illustrating how code with manual stack management calls code with automatic stack management. In the figure, the details of a call in the opposite direction are momentarily obscured behind dashed boxes. The first event handler for FetchCert1 calls the function GetCertData1, which initiates an I/O operation, and the entire stack unrolls in accordance with manual stack management. Later, when the I/O reply arrives, the scheduler executes the GetCertData2 continuation, which "returns" (by a function call) to the second handler for FetchCert. This is pure manual stack management.

When a function written with manual stack management calls code with automatic stack management, we must reconcile the two styles. The caller code is written expecting never to block on I/O; the callee expects to block I/O always. To reconcile these styles, we create a new fiber and execute the callee code on that fiber. The caller resumes (to manually unroll its stack) as soon as the first burst of execution on the fiber completes. The fiber may

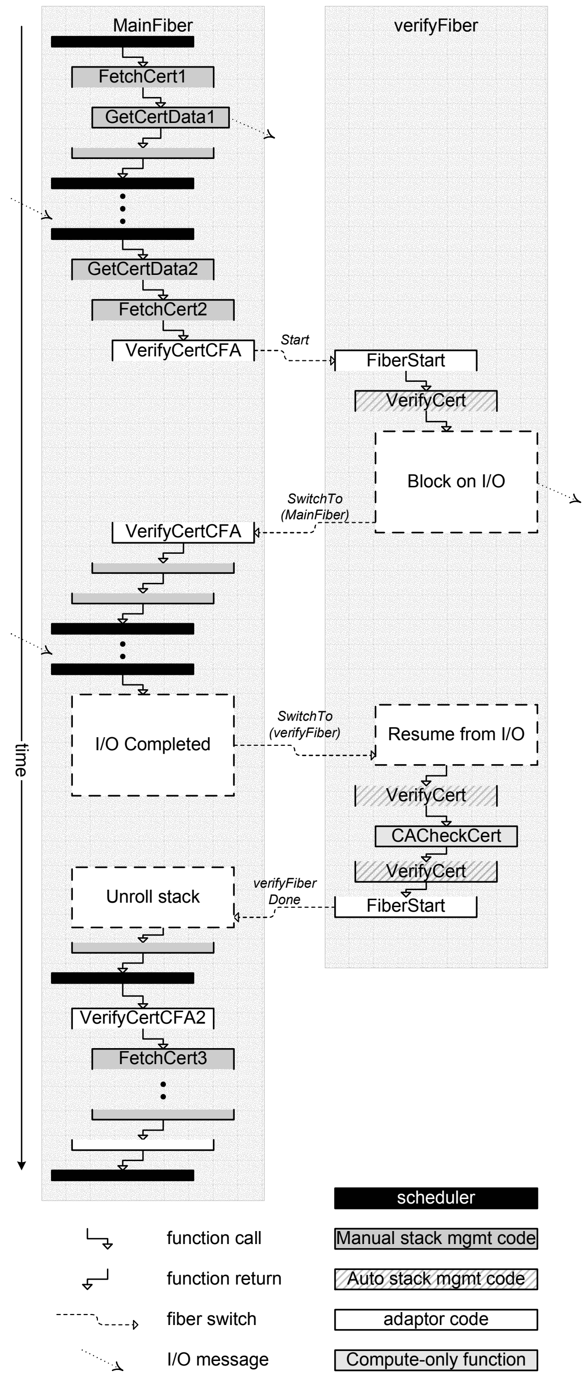

Figure 2: `GetCertData`, code with manual stack management, calls `VerifyCert`, a function written with automatic stack management.

run and block for I/O several times; when it finishes its work on behalf of the caller, it executes the caller's continuation to resume the caller's part of the task. Thus, the caller code does not block and the callee code can block if it wishes.

In our example, the manual-stack-management function `FetchCert2` calls through an adapter to the automatic-stack-management function `VerifyCert`. `FetchCert2` passes along a continuation pointing at `FetchCert3` so that it can eventually regain control and execute the final part of its implementation. The following code is for the CFA adaptor, ripped into its *call* and *return* parts; CFA stands for "Continuation-To-Fiber adaptor."

```
void VerifyCertCFA(CertData certData,
             Continuation *callerCont) {
  // Executed on MainFiber
  Continuation *vcaCont = new
           Continuation(VerifyCertCFA2,
                        callerCont);
  Fiber *verifyFiber = new
     VerifyCertFiber(certData, vcaCont);
  // On fiber verifyFiber, start executing
  // VerifyCertFiber::FiberStart
  SwitchToFiber(verifyFiber);
  // Control returns here when
  // verifyFiber blocks on I/O
}

void VerifyCertCFA2(Continuation
*vcaCont) {
  // Executed on MainFiber.
  // Scheduled after verifyFiber is done
  Continuation *callerCont =
         (Continuation*) vcaCont->arg1;
  callerCont->returnValue =
         vcaCont->returnValue;
  // "return" to original caller (FetchCert)
  (*callerCont->function)(callerCont);
}
```

The first adaptor function accepts the arguments of the adapted function and a continuation ("stack frame") for the calling task. It constructs its own continuation `vcaCont` and creates a object called `verifyFiber` that represents a new fiber (VerifyCertFiber is a subclass of the Fiber class); this object keeps track of the function arguments and `vcaCont` so that it can transfer control to `VerifyCertCFA2` when `verifyFiber`'s work is done. Finally, it performs a fiber-switch to `verifyFiber`. When `verifyFiber` begins, it executes glue routine

`VerifyCertFiber::FiberStart` to unpack the parameters and pass them to `VerifyCert`, which may block on I/O:

```
VerifyCertFiber::FiberStart() {
  // Executed on a fiber other than MainFiber
  // The following call could block on I/O.
  // Do the actual verification.
  this->vcaCont->returnValue =
        VerifyCert(this->certData);
  // The verification is complete.
  // Schedule VerifyCertCFA2
  scheduler->schedule(this->vcaCont);
  SwitchTo(MainFiber);
}
```

This start function simply calls into the function `VerifyCert`. At some point, when `VerifyCert` yields for I/O, it switches control back to the `MainFiber` using a `SwitchTo` call in the I/O function (not the call site shown in the `FiberStart()` routine above). Control resumes in `VerifyCertCFA`, which unrolls the continuation stack (i.e., `GetCertData2` and `FetchCert2`) back to the scheduler. Thus, the hybrid task has blocked for the I/O initiated by the code with automatic stack management while ensuring that event handler `FetchCert2` does not block.

Later, when the I/O completes, `verifyFiber` is resumed (for now, we defer the details on how this resumption occurs). After `VerifyCert` has performed the last of its work, control returns to `FiberStart`. `FiberStart` stuffs the return value into `VerifyCertCFA2`'s continuation, schedules it to execute, and switches back to the `MainFiber` a final time. At this point, `verifyFiber` is destroyed. When `VerifyCertCFA2` executes, it "returns" (with a function call, as code with manual stack management normally does) the return value from `VerifyCert` back to the adaptor-caller's continuation, `FetchCert3`.

4.2 Automatic calling manual

We now discuss how the code interactions occur when a function with automatic stack management calls a function that manually manages its stack. In this case, the former function needs to block for I/O, but the latter function simply schedules the I/O and returns. To reconcile these requirements, we supply an adaptor that calls the manual-stack-management code with a special continuation and relinquishes control to the `MainFiber`,

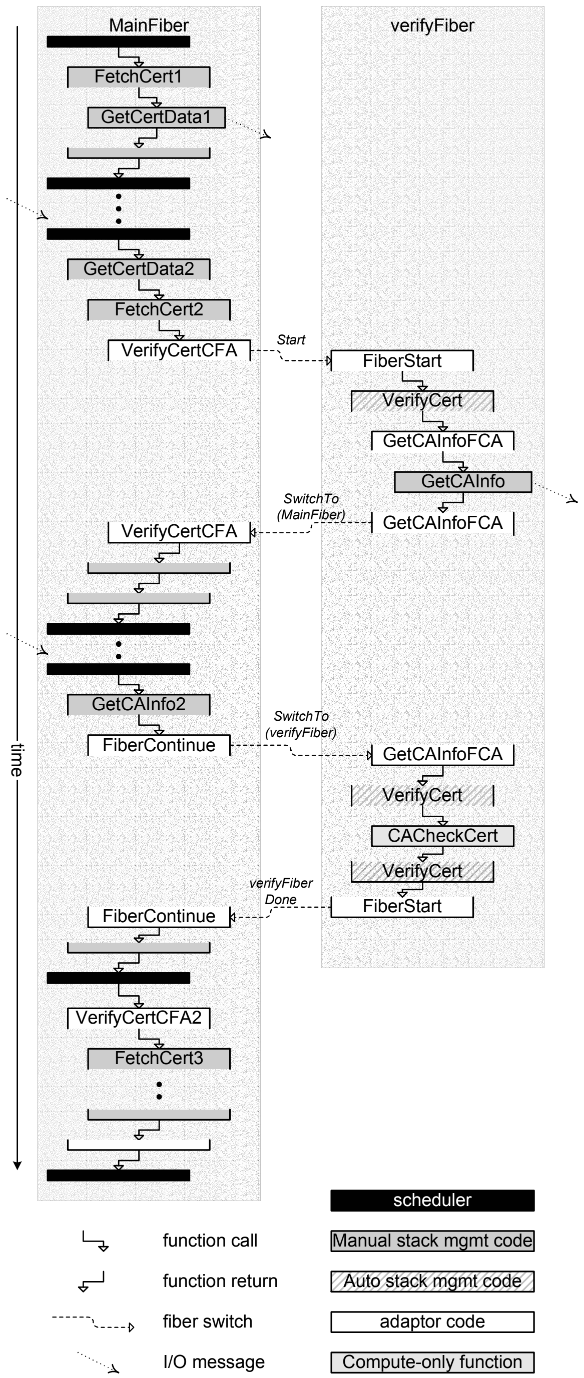

Figure 3: `VerifyCert`, code with automatic stack management, calls `GetCAInfo`, a function written with manual stack management.

thereby causing the adaptor's caller to remain blocked. When the I/O completes, the special continuation runs on the MainFiber and resumes the fiber of the blocked adaptor, which resumes the original function waiting for the I/O result.

Figure 3 fills in the missing details of Figure 2 to illustrate this interaction. In this example, VerifyCert blocks on I/O when it calls GetCAInfo, a function with manual stack management. VerifyCert calls the adaptor GetCAInfoFCA, which hides the manual-stack-management nature of GetCAInfo (FCA means Fiber-to-Continuation Adaptor):

```
Boolean GetCAInfoFCA(CAID caid) {
  // Executed on verifyFiber
  // Get a continuation that switches control
  // to this fiber when called on MainFiber
  FiberContinuation *cont = new
      FiberContinuation(FiberContinue,
                        this);
  GetCAInfo(caid, cont);
  if (!cont->shortCircuit) {
    // GetCAInfo did block.
    SwitchTo(MainFiber);
  }
  return cont->returnValue;
}

void FiberContinue(Continuation *cont) {
  if (!Fiber::OnMainFiber()) {
    // Manual stack mgmt code did not perform
    // I/O: just mark it as short-circuited
    FiberContinuation *fcont =
              (FiberContinuation) *cont;
    fcont->shortCircuit = true;
  } else {
    // Resumed after I/O: simply switch
    // control to the original fiber
    Fiber *f = (Fiber *) cont->arg1;
    f->Resume();
  }
}
```

The adaptor, GetCAInfoFCA, sets up a special continuation that will later resume verifyFiber via the code in FiberContinue. It then passes this continuation to GetCAInfo which initiates an I/O operation and returns immediately to what it believes to be the event-handling scheduler; of course, in this case, the control returns to GetCAInfoFCA. Since I/O was scheduled and short-circuiting did not occur (discussed later in this section), GetCAInfoFCA must ensure that control does not yet return to VerifyCert; to achieve this effect, it switches control to the MainFiber.

On the MainFiber, the continuation code that started this burst of fiber execution, VerifyCertCFA, returns several times to unroll its stack and the scheduler runs again. Eventually, the I/O result arrives and the scheduler executes GetCAInfo2, the remaining work of GetCAInfo. GetCAInfo2 fills the local hash table (recall its implementation from Section 3.1) and "returns" control by calling a continuation. In this case, it calls the continuation (FiberContinue) that had been passed to GetCAInfo.

FiberContinue notices that verifyFiber has indeed been blocked and switches control back to that fiber, where the bottom half of the adaptor, GetCAInfoFCA, extracts the return value and passes it up to the automatic-stack-management code that called it (VerifyCert).

The *short circuit* branch not followed in the example handles the case where GetCAInfo returns a result immediately without waiting for I/O. When it can do so, it must *not* allow control to pass to the scheduler. This is necessary so that a caller can optionally determine whether or not a routine has yielded control and hence whether or not local state must be revalidated. Without a *short circuit* path, this important optimization and an associated design pattern that we describe in Section 5 cannot be achieved. Figure 4 illustrates the short-circuit sequence: The short-circuit code detects the case where GetCAInfo runs locally, performs no I/O, and executes ("returns to") the current continuation immediately. FiberContinue detects that it was not executed directly by the scheduler, and sets the shortCircuit flag to prevent the adaptor from switching to the MainFiber.

4.3 Discussion

An important observation is that, with adaptors in place, each style of code is unaware of the other. A function written with automatic stack management sees what it expects: deep in its stack, control may transfer away, and return later with the stack intact. Likewise, the event-handler scheduler cannot tell that it is calling anything other than just a series of ordinary manual-stack-management continuations: the adaptors deftly swap the fiber stacks around while looking like any other continuation. Thus, integrating code in the two styles is straightforward: fiber execution looks like a continuation to the event-driven code, and the continuation scheduler looks

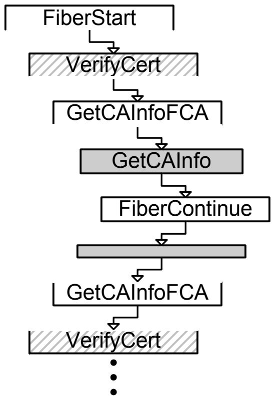

Figure 4: A variation where `GetCAInfo` does not need to perform I/O.

like any other fiber to the procedure-oriented code. This adaptability enables automatic-stack-management programmers to work with manual-stack-management programmers, and to evolve a manual-stack-management code base with automatic-stack-management functions and *vice versa*.

5 Implementation Experience

We have employed cooperative task management in two systems: Farsite [BDET00], a distributed, secure, serverless file system running over desktops, and UCoM [SBS02], a wireless phone application for handheld devices such as PDAs. The Farsite code is designed to run as a daemon process servicing file requests on the Windows NT operating system. The UCoM system, designed for the Windows CE operating system, is a client application that runs with UI and audio support.

The Farsite system code was initially written in event-driven style (cooperative task management and manual stack management) to enable simplified reasoning about the concurrency conditions of the system. As our code base grew and evolved over a period of two years, we came to appreciate the costs of employing manual stack management and devised the hybrid approach discussed in the previous section to introduce automatic stack management code into our system. The UCoM system uses automatic stack management exclusively.

Farsite uses fibers, the cooperative threading facility available in Windows NT. With Windows fibers, each task's state is represented with a stack, and control is transferred by simply swapping one stack pointer for another, as with `setjmp` and `longjmp`. Since fibers are unavailable in the Windows CE operating system, UCoM uses preemptive threads and condition variables to achieve a cooperative threading facility: each thread blocks on its condition variable and the scheduler ensures that at most one condition variable is signalled at any moment. When a thread yields, it blocks on its condition variable and signals the scheduler to continue; the scheduler selects a ready thread and signals its condition variable.

We implemented the hybrid adaptors in each direction with a series of mechanically-generated macros. There are two groups of macros, one for each direction of adaptation. Within each group, there are variations to account for varying numbers of arguments, void or non-void return type, and whether the function being called is a static function or an object method; multiple macros are necessary to generate the corresponding variations in syntax. Each macro takes as arguments the signature of the function being adapted. The macros declare and create appropriate `Fiber` and `Continuation` objects.

Our experience with both systems has been positive and our subjective impression is that we have been able to preempt many subtle concurrency problems by using cooperative task management as the basis for our work. Although the task of wrapping I/O functions (see Section 2.1) can be tedious, it can be automated, and we found that paying an up-front cost to reduce subtle race conditions was a good investment.

Both systems use extra threads for converting blocking I/O operations to non-blocking operations and for scheduling I/O operations, as is done in many other systems, such as Flash [PDZ99]. Data partitioning prevents synchronization problems between the I/O threads and the state shared by cooperatively-managed tasks.

Cooperative task management avoids the concurrency problems of locks only if tasks can complete without having to yield control to any other task. To deal with tasks that need to perform I/O, we found that we could often avoid the need for a lock by employing a particular design pattern. In this pattern, which we call the *Pinning Pattern*, I/O operations are used to *pin* resources in memory where they can be manipulated without yielding. Note that *pinning* does not connote exclusivity: a *pinned* resource is held in memory (to avoid the need to block on I/O to access it), but when other tasks run, they are free to manipulate the data structures it contains. Functions are structured in two phases: a loop that repeatedly tries to execute all potentially-yielding operations until they can all be completed without yielding, and an atomic block that computes results and writes them into the shared state.

An important detail of the design pattern is that there may be dependencies among the potentially-yielding operations. A function may need to compute on the results of a previously-pinned resource in order to decide which resource to pin next; for example, in Farsite this occurs when traversing a path in a directory tree. Thus, in the fully general version of the design pattern, a check after each potentially-yielding operation ascertains whether the operation did indeed yield, and if so, restarts the loop from the top. Once the entire loop has executed without interruption, we know that the set of resources we have pinned in memory are related in the way we expect, because the final pass through the loop executed atomically.

6 Related Work

Birrell offers a good overview of the conventional thread-ed programming model with preemptive task management [Bir89]. Of his reasons for using concurrency (p. 2), cooperative task management can help with all but exploiting multiprocessors, a shortcoming we mention in Section 2.1. Birrell advises that "you must be fastidious about associating each piece of data with one (and only one) mutex" (p. 28); consider cooperative task management as the limiting case of that advice. There is the complexity that whenever a task yields it effectively releases the global mutex, and must reestablish its invariants when it resumes. But even under preemptive task management, Birrell comments that "you might want to unlock the mutex before calling down to a lower level abstraction that will block or execute for a long time" (p. 12); hence this complexity is not introduced by the choice of cooperative task management.

Ousterhout points out the pitfalls of preemptive task management, such as subtle race conditions and deadlocks [Ous96]. We argue that his "threaded" model conflates preemptive task management with automatic stack management, and his "event-driven" model conflates cooperative task management with manual stack management. We wish to convince designers that the choices are orthogonal, that Ousterhout's arguments are really about the task management decision, and that programmers should exploit the ease-of-reasoning benefits of cooperative task management while exploiting the features of their programming language by using automatic stack management.

Other system designers have advocated non-threaded programming models because they observe that for a certain class of high-performance systems, such as file servers and web servers, substantial performance improvements can be obtained by reducing context switching and carefully implementing application-specific cache-conscious task scheduling [HS99, PDZ99, BDM98, MY98]. These factors become especially pronounced during high load situations, when the number of threads may become so large that the system starts to thrash while trying to give each thread its fair share of the system's resources. We argue that the context-switching overhead for user-level threads (fibers) is in fact quite low; we measured the cost of switching in our fiber package to be less than ten times the cost of a procedure call. Furthermore, application-specific cache-conscious task scheduling should be just as achievable with cooperative task management and automatic stack management: the scheduler is given precisely the same opportunities to schedule as in event-driven code; the only difference is whether stack state is kept on stacks or in chains of continuations on the heap.

For the classes of applications we reference here, processing is often partitioned into stages [WCB01, LP01]. The partitioning of system state into disjoint stages is a form of data partitioning, which addresses concurrency at the coarse grain. Within each stage, the designer of such a system must still choose a form of conflict management, task management, and stack management. Careful construction of stages avoids I/O calls within a stage; in that case, cooperative task management within the stage degenerates to serial task management, and no distinction arises in stack management. In practice, at the inter-stage level, a single task strings through multiple stages, and reads as in manual stack management. Typically, the stages are monotonic: once a task leaves a stage, it never returns. This at least avoids the ripping associated with looping control structures.

Lauer and Needham show two programming models to be equivalent up to syntactic substitution [LN79]. We describe their models in terms of our axes: their procedure-oriented system has preemptive task management, automatic stack management ("a process typically has only one goal or task"), monitors for conflict management, and one big data partition protected by those monitors. Their message-oriented system has manual stack management with task state passed around in messages, and no conflicts to manage due to many partitions of the state so that it is effectively never concurrently shared.

Notably, of the message-oriented system, they say "neither procedural interfaces nor global naming schemes are very useful," that is, the manual stack management undermines structural features of the language. Neither model uses cooperative task management as we regard it, since both models require identically-detailed reasoning about conflict management. Thus their comparison is decidedly not between the models we associate with *multithreaded* and *event-driven* programming.

7 Conclusions

In this paper we clarify an ongoing debate about "event-driven" versus "threaded" programming models by identifying two separable concerns: task management and stack management. Thus separated, the paper assumes cooperative task management and focuses on issues of stack management in that context. Whereas the choice of task management strategy is fundamental, the choice of stack management can be left to individual taste. Unfortunately, the term "event-driven programming" conflates both cooperative task management and manual stack management. This prevents many people from considering using a readable automatic-stack-management coding style in conjunction with cooperative task management.

Software evolution is an important factor affecting the choice of task management strategy. When concurrency assumptions evolve it may be necessary to make global, abstraction-breaking changes to an application's implementation. Evolving code with manual stack management imposes the cumbersome code restructuring burden of stack ripping; evolving either style of code involves revisiting the invariant logic due to changing concurrency assumptions and sometimes making localized changes to functions in order to revalidate local state.

Finally, a hybrid model adapts between code with automatic and with manual stack management, enabling cooperation among disparate programmers and software evolution of disparate code bases.

8 Acknowledgements

We would like to thank Jim Larus for discussing this topic with us at length. Thanks also to the anonymous reviewers for their thoughtful comments.

References

[BDET00] William Bolosky, John Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proceedings of the ACM Sigmetrics 2000 Conference*, pages 34–43, June 2000.

[BDM98] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Better operating system features for faster network servers. In *Proceedings of the Workshop on Internet Server Performance (held in conjunction with ACM SIGMETRICS '98)*, Madison, WI, 1998.

[Bir89] Andrew D. Birrell. An introduction to programming with threads. Technical Report 35, Digital Systems Research Center, January 1989.

[FHK84] Daniel P. Friedman, Christopher T. Haynes, and Eugene E. Kohlbecker. Programming with continuations. In P. Pepper, editor, *Program Transformation and Programming Environments*, pages 263–274. Springer-Verlag, 1984.

[HFW84] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *ACM Symposium on LISP and Functional Programming*, pages 293–298, Austin, TX, August 1984.

[HS99] J. Hu and D. Schmidt. JAWS: A Framework for High Performance Web Servers. In *Domain-Specific Application Frameworks: Frameworks Experience by Industry*. Wiley & Sons, 1999.

[LN79] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, January 1979.

[LP01] J. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. Technical Report MSR-TR-2001-39, Microsoft Research, March 2001.

[MY98] S. Mishra and R. Yang. Thread-based vs event-based implementation of a group communication service. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, 1998.

[Ous96] John Ousterhout. Why threads are a bad idea (for most purposes). In *USENIX Technical Conference (Invited Talk)*, Austin, TX, January 1996.

[PDZ99] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Technical Conference*, Monterey, CA, June 1999.

[SBS02] E. Shih, P. Bahl, and M. Sinclair. Wake on Wireless: An event driven power saving strategy for battery operated devices. Technical Report MSR-TR-2002-40, Microsoft Research, April 2002.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, October 2001.