# AC: Composable Asynchronous IO for Native Languages

Tim Harris[†] Martín Abadi[‡⋆] Rebecca Isaacs[‡] Ross McIlroy[†]

Microsoft Research, Cambridge[†] Microsoft Research, Silicon Valley[‡]
University of California, Santa Cruz[⋆] Collège de France[⋆]

tharris@microsoft.com abadi@microsoft.com risaacs@microsoft.com rmcilroy@microsoft.com

## Abstract

This paper introduces AC, a set of language constructs for composable asynchronous IO in native languages such as C/C++. Unlike traditional synchronous IO interfaces, AC lets a thread issue multiple IO requests so that they can be serviced concurrently, and so that long-latency operations can be overlapped with computation. Unlike traditional asynchronous IO interfaces, AC retains a sequential style of programming without requiring code to use multiple threads, and without requiring code to be "stack-ripped" into chains of callbacks. AC provides an `async` statement to identify opportunities for IO operations to be issued concurrently, a `do..finish` block that waits until any enclosed `async` work is complete, and a `cancel` statement that requests cancellation of unfinished IO within an enclosing `do..finish`. We give an operational semantics for a core language. We describe and evaluate implementations that are integrated with message passing on the Barrelfish research OS, and integrated with asynchronous file and network IO on Microsoft Windows. We show that AC offers comparable performance to existing C/C++ interfaces for asynchronous IO, while providing a simpler programming model.

***Categories and Subject Descriptors*** D.1.3 [*Programming Techniques*]: Concurrent Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Input/output; D.4.4 [*Operating Systems*]: Communications Management—Message sending

***General Terms*** Languages, Performance

## 1. Introduction

In the future, processors are likely to provide a heterogeneous mix of core types without hardware cache coherence across a whole machine. In the Barrelfish project we are investigating how to design an operating system (OS) for this kind of hardware, in which we can no longer rely on traditional shared-memory within the OS [4].

The approach we are taking is to construct the OS around separate per-core kernels, and to use message passing for communication between system processes running on different cores. Other contemporary OS research projects take a similar approach [38]. Our hypothesis is that systems built on message passing can be mapped to a wide variety of processor architectures without large-scale re-implementation. Using message passing lets us accommodate machines with heterogeneous core types, and machines without cache-coherence; we can map message passing operations onto specialized messaging instructions [18, 34], and we can map them onto shared-memory buffers on current hardware [4].

However, it is difficult to write scalable low-level software using message passing. Existing systems focus either on ease-of-programming (by providing simple synchronous send/receive operations), or on performance (typically by providing asynchronous operations that execute a callback function once a message has been sent or received). The same tension exists in IO interfaces more generally [27, 35]. For example, the Microsoft Windows APIs require software to choose between synchronous operations which allow only one concurrent IO request per thread, and complex asynchronous operations which allow multiple IO requests.

We believe that the inevitable and disruptive evolution of hardware to non-cache-coherent, heterogeneous, multi-core systems makes support for asynchronous IO in low-level languages such as C/C++ both essential and timely.

In this paper we introduce AC ("Asynchronous C"), a new approach to writing programs using asynchronous IO (AIO). AC provides a lightweight form of AIO that can be added incrementally to software, without the use of callbacks, events, or multiple threads.

Our overall approach is for the programmer to start out with simple synchronous IO operations, and to use new language constructs to identify opportunities for the language runtime system to start multiple IO operations asynchronously.

As a running example, consider a `Lookup` function that sends a message to a name-service process, and then receives back an address that the name maps to. Figure 1 shows this function written using Barrelfish's callback-based interface.



*OOPSLA'11,* October 22–27, 2011, Portland, Oregon, USA.

```
void Lookup(NSChannel_t *c, char *name) {
  OnRecvLookupResponse(c, &ResponseHandler);
  // Store state needed by send handler
  c->st = name;
  OnSend(c, &SendHandler);
}

void ResponseHandler(NSChannel_t *c, int addr) {
  printf("Got response %d\n", addr);
}

void SendHandler(NSChannel_t *c) {
  if (OnSendLookupRequest(c, (char*)(c->st)) == BUSY) {
    OnSend(c, &SendHandler);
} }
```

**Figure 1.** Querying a set of name server using Barrelfish's callback-based interface for message passing.

The `Lookup` function takes a reference to a channel (`c`). The function registers a `ResponseHandler` callback to execute when a `LookupResponse` reply is received. It then registers a `SendHandler` callback to execute when channel `c` has space for the outgoing message. (Many hardware implementations of message passing provide bounded-size message channels, and so it can be impossible to send a message immediately.) In addition, `Lookup` needs to record `name` in a temporary data structure so that it is available to `SendHandler`. The `On*` functions are generated automatically from an interface definition for the `NSChannel_t` channel.

With AC, the "lookup" example becomes a single function using synchronous `Send`/`Recv` operations: (We omit some details to do with cancellation of unfinished IO operations; we return to cancellation in Section 2.)

```
// Caution: functions ending in AC may block
void LookupAC(NSChannel_t *c, char *name) {
  int addr;
  SendLookupRequestAC(c, name);
  RecvLookupResponseAC(c, &addr);
  printf("Got response %d\n", addr);
}
```

Compared with the callback-based implementation, this `LookupAC` function is clearly much simpler: it avoids the need for "stack-ripping" [3] in which the logical flow between operations is split across a series of callbacks. AC leads to a form of composability that is lost with stack-ripping. A function can simply call into other functions using AC, and it can start multiple AC operations concurrently. For instance, to communicate with two name servers, one can write:

```
void TwinLookupAC(NSChannel_t *c1,
                  NSChannel_t *c2,
                  char *name) {
  do {
    async LookupAC(c1, name);     // S1
    async LookupAC(c2, name);     // S2
  } finish;
  printf("Got both responses\n"); // S3
}
```

The `async` at statement S1 indicates that execution can continue to statement S2 if the first lookup needs to block. The `do..finish` construct indicates that execution cannot continue to statement S3 until both S1 and S2 have been executed to completion.

Throughout AC, we keep the abstractions used for asynchrony separate from the abstractions used for parallel programming; code remains single-threaded unless the programmer explicitly introduces parallelism. The `async` and `do..finish` constructs are solely there to identify opportunities for multiple messages to be issued concurrently; unlike the `async` construct in X10 [7], our `async` does not introduce parallelism. Consequently, many of our examples can be written with no concurrency-control beyond the block-structured synchronization of `do..finish`.

We make a number of additional contributions beyond the core design of AC. We introduce a new block-structured cancellation mechanism. This approach to cancellation provides a modular way for a program to start asynchrony operations and then to cancel them if they have not yet completed; e.g., adding a timeout around a function that is called. In `TwinLookupAC`, cancellation could be used to abandon one lookup as soon as the other lookup is complete. In contrast to our approach, traditional cancellation mechanisms are directed at individual IO operations [1], or at groups of operations on the same file, or at a complete thread (e.g., alerting in Modula-2+ [5]).

We introduce AC in more detail in Section 2. In Section 3 we present a formal operational semantics for a core language modeling AC, including cancellation. We give the semantics and discuss properties that it satisfies.

Section 4 describes two implementations of AC. The first implementation uses a modified Clang/LLVM tool-chain to add the AC operations to C/C++. The second implementation operates with Clang, or with GCC, and defines the AC constructs using a combination of macros and existing C/C++ extensions provided by these compilers. The second implementation has slightly higher overheads than the first.

In Section 5 we look at the performance of implementations that are integrated with message passing on Barrelfish, and also at implementations that are integrated with asynchronous IO on Microsoft Windows. In each case, AC achieves most of the performance of manually written stack-ripped code while providing a programming model that is comparable to basic synchronous IO (and comparable to the recent C# and F#-based abstractions for performing asynchronous IO [32]). We discuss related work and conclude in Sections 6 and 7.

## 2. Composable Asynchronous IO

In this section we introduce AC informally. We continue with the example of a name-service lookup from the introduction. We use it to illustrate the behavior of AC operations in more detail, and to motivate our design choices.

Throughout this section our design choices are motivated by providing two properties. First, a "serial elision" property: if the IO operations in a piece of software com-

```
int LookupAllAC(NSChannel_t *cs[], int nc,
                char *name, int *addr) {
  bool seen_first = false;
  int first_addr;
  do {
    for (int i = 0; i < nc; i++) {
      async {
        if (LookupOneAC(cs[i], name, addr) == OK) {
          if (!seen_first) {
            seen_first = true;
            first_addr = *addr;
          } else {
            assert(*addr == first_addr);
          }
        }
      }
    }
  } finish;
  return OK;
}
```

**Figure 2.** Querying a set of name servers concurrently, checking that they all return the same result.

plete without needing to block, then the software behaves as though the AC extensions were removed. Second, a "synchronous elision" property: removing the AC constructs leaves a correct program using ordinary synchronous operations. Conversely, taking a synchronous program and adding these constructs produces a program using asynchronous IO. We believe both properties are valuable in simplifying the incremental adaptation of existing applications to use asynchrony (although, of course, care is still needed to determine exactly which IO requests can be issued at the same time).

In this section we focus on examples based on message passing on Barrelfish. In this setting, the AC send/receive operations block until an outgoing message has been buffered in a channel, or until an incoming message has been removed from a channel. Unlike the message passing operations in languages such as CSP [17], the AC send/receive operations do not synchronize with one another. Channels operate between pairs of processes in a single machine. They provide reliable, ordered delivery of messages. The sole kind of failure is that a channel is abruptly disconnected when the process on one end terminates without closing the channel; this failure is signaled out-of-band to the other process, so error handling code does not feature in the examples here.

It is straightforward to write functions such as `LookupAC` from the introduction: synchronous message passing avoids the complexity of callback-based interfaces. However, it also loses the benefits: we can no longer perform multiple send/receive operations concurrently, we can no longer perform computation while waiting for IO operations to complete, and we cannot abandon waiting for an IO operation once we have started it.

AC addresses the problems by providing the `async` and `do..finish` constructs to allow multiple IO operations to be issued (Section 2.1), and providing the `cancel` construct for block-structured cancellation of waiting (Section 2.2).

### 2.1 The `async` and `do..finish` Constructs

The `async` and `do..finish` constructs provide a mechanism to switch away from an operation if it blocks, and to resume its execution after it unblocks. Figure 2 gives an example, expanding the earlier `LookupAC` function to consult a series of servers and to report an error if the results differ. The `LookupOneAC` function performs a single lookup, returning the resulting address. The `LookupAllAC` function takes an array of channels, and makes a series of calls to `LookupOneAC` to perform each lookup. The `async` within the loop indicates that execution can continue to the next iteration if a given `LookupOneAC` call blocks, and the `do..finish` indicates that execution must block at `finish` until all of the `async` work is done. The example satisfies the synchronous elision property: if the new constructs are ignored, then it becomes a simple sequential look up on each server in turn.

There are a number of design choices:

***Starting work asynchronously.*** First, what code runs when reaching a statement `async` $S$—e.g., $S$ itself, or the continuation of the `async` $S$ statement (as in AME [19]), or are they interleaved (as in X10 [7])? In AC, execution proceeds immediately into $S$, without introducing parallel execution. This feature follows both from the serial elision property, and from our conceptual desire to keep separate the support for asynchrony and parallelism.

A consequence of this design choice is that, in Figure 2, the code inside the `async` statement can simply read `i` to obtain the channel to work on: the code inside the `async` statement runs directly at the start of each loop iteration before `i` is modified by the next execution of the loop header.

The example also exploits the fact that `async` does not introduce parallelism: when `LookupOneAC` returns, there is no need for synchronization on the accesses to the local variables `result`, `first_result`, or `seen_first`. We do not need, for instance, to introduce locking on these variables, or to use futures to communicate values from `LookupOneAC` to its caller.

If a local variable is declared within an `async` statement, then it is private to each invocation of this statement (unlike the example in Figure 2 where the variables are shared between the invocations of the `async` statements).

***Blocking within asynchronous work.*** The next design choice is what happens when code within `async` $S$ blocks. In AC, when $S$ first blocks, execution resumes at the continuation of the `async` statement. In this respect `async` can be seen as "catching" the blocking of the code within it, and providing the surrounding code with an opportunity to start additional IO operations, or to do computation instead. In Figure 2, the continuation of the `async` statement is the loop header which proceeds to the next iteration.

When calling a function that might block, the programmer needs to anticipate the possibility that other code may

```
int LookupFirstAC(NSChannel_t *cs[],
                  int nc, char *name,
                  int *addr) {
  bool seen_first = false;
queries: do {
    for (int i = 0; i < nc && !seen_first; i++) {
      async {
        if (LookupOneAC(cs[i], name, addr) == OK) {
          seen_first = true;
          cancel queries;
        }
      }
    }
  } finish;
  return (seen_first ? OK : CANCELLED);
}
```

**Figure 3.** Querying a set of name servers concurrently, and returning the first reply.

run before the callee returns. We follow a convention that all possibly-blocking functions have an `AC` suffix on their name. This convention is true for primitive send/receive operations, and for examples such as `LookupAllAC`. Following this convention ensures that a caller is aware that execution might switch to elsewhere in the thread while waiting. For example, in Figure 2, the value of local variable `i` may be updated while a call to `LookupOneAC` is blocked, so if the original value is needed then it should be saved before the call.

Our convention of highlighting AC operations corresponds to rules from "atomic by default" programming models such as AME [2, 19] and TIC [30] that operations that are not atomic should include annotations at the function's definition, and at each call-site. Our convention could be enforced by static checks, if desired. A simple, conservative, approach would be to issue a warning if an `AC` function is called from a non-`AC` function.

***Synchronization.*** The final design choice is how to synchronize with `async` operations. The `do..finish` construct provides this form of synchronization: execution does not proceed past the `do..finish` until all of the asynchronous work started inside it has completed. In Figure 2 the `do..finish` requires that all of the `LookupOneAC` calls have finished before `LookupAllAC` can return. From the point of view of `LookupAllAC`'s caller, blocking at the end of a `do..finish` is the same as blocking at an IO operation: the call can be placed within an `async`, and other work can be started if the call blocks (e.g., a different `LookupAllAC` for some other name).

***Rules for starting asynchronous work.*** An `async` statement must occur statically within `do..finish`. It is incorrect to write unbalanced code such as:

```
void StartAsync(int x) {  async f(x);  }
```

This design choice follows from our focus on implementations for systems software in C/C++. With predictable lifetimes for data used for synchronization: (*i*) a cactus-stack [16] can be used, rather than requiring a more general heap-allocated structure, and (*ii*) as usual, a callee can safely access stack-allocated data passed to it, irrespectively of whether or not any of the calls on the stack are asynchronous. (CILK has a similar rule in that a function implicitly synchronizes with any parallel work that it has spawned [13].)

***Integration with threads.*** Although `async` does not introduce parallelism, our implementations are nevertheless integrated with OS threads. This integration enables scenarios where a multi-threaded server handles connections to different clients in different threads, or where a function starts a thread explicitly to perform work after the function returns (such as a variant of `StartAsync`, above).

The runtime system provides concurrency-control primitives such as mutexes and condition variables. These primitives can be used between pieces of `async` work in the same OS thread, or between different OS threads. Blocking on concurrency-control primitives is handled in exactly the same way as blocking on IO: the OS thread's execution can switch to a different piece of work. This work can either be the continuation of an enclosing `async` statement, or it can be a piece of work that has become unblocked. Work retains affinity to the OS thread that started it. The primitive message send/receive operations are themselves thread-safe.

### 2.2 Cancellation

The `async` and `do..finish` constructs let the programmer start multiple IO operations, and they let the programmer overlap computation with communication. However, these constructs do not recover all of the expressiveness of the low-level callback-based APIs; in particular, we also wish to be able to stop waiting for an IO operation once we have started it.

AC provides a `cancel` command to allow a program to stop waiting for an IO operation. Cancellation is somewhat analogous to thread interruption in Java: it causes operations that are blocked on IO to be unblocked.

Figure 3 shows how cancellation can be used to write a `LookupFirstAC` function that queries a set of name servers, returns the first response, and then cancels the remaining queries: the loop is within a `do..finish` labeled "`queries`", and a `cancel queries` command is executed when the first response is received. Cancellation causes any blocked `LookupOneAC` calls within the `do..finish` to become unblocked, and then to return `CANCELLED`. The `do..finish` block behaves as usual: once all of the `async` work started within it has finished, execution can proceed to the end of `LookupFirstAC`.

When used with a label, `cancel` must occur statically within the `do..finish` block that the label identifies. If `cancel` is used without a label then it refers to the closest statically enclosing `do..finish` block. It is an error for `cancel` to occur without a statically enclosing `do..finish` block This requirement makes it clear exactly which block of operations is being cancelled: one can-

```
int LookupWithTimeoutAC(NSChannel_t *cs[],
                        int nc, char *name,
                        int *addr, int t) {
  int result;
timeout: do {
    async {
      result = LookupFirstAC(cs, nc, name, addr);
      cancel timeout;
    }
    async {
      SleepAC(t);
      cancel timeout;
    }
  } finish;
  return result;
}
```

**Figure 4.** Adding a timeout around an existing AC function.

not call into one function and have it "poison" the caller's function unexpectedly by using cancellation internally.

***Semantics of cancellation.*** Care is needed when cancellation involves operations with side-effects. Even for a "read-only" operation such as `LookupOneAC`, there is a question of the state of the channel after cancellation. For example, what will happen if cancellation occurs after a message has been sent, but before a response has been received? What will happen if a response subsequently arrives—can it be confused with the response to a different message?

Both on Barrelfish and on Microsoft Windows, we follow a convention that we term "exact cancellation": upon cancellation then either (*i*) a call returns `CANCELLED`, without appearing to perform the requested operation, or (*ii*) the operation is performed and the function returns `OK`. In particular, an `OK` result may be seen, even after executing `cancel`, if an IO operation was completed by a device concurrently with cancellation being requested by software.

This convention represents a division of work between the application programmer and the implementer of the IO library: the IO library guarantees exact cancellation in its own functions, but the application programmer is responsible for providing exact cancellation in the functions that they write. The programmer must shoulder this responsibility because the correct behavior depends on the semantics of the operations that they are writing—e.g., whether or not a compensating operation must be performed and, if so, exactly what.

To allow compensating code to be written without itself being cancelled, AC lets functions be marked "non-cancellable" and, on Barrelfish, we provide non-cancellable variants of all of the message passing primitives. In Section 4 we show how these non-cancellable primitives are themselves implemented over the AC runtime system.

***Composable cancellation.*** Consider the example in Figure 4 of adding a timeout to a `LookupFirstAC` call. The first `async` starts the `LookupFirstAC` request. The second `async` starts a timer. Whichever operation completes first attempts to cancel the other. This block-structured approach lets programs use cancellation in a composable way: the cancellation triggered in `LookupWithTimeoutAC` propagates into both `async` branches (and recursively into their callees, unless these are non-cancellable).

Unlike systems that target cancellation requests at individual operations, AC lets a caller cancel a set of operations without being able to name them individually.

Note that, in the `LookupWithTimeoutAC` function, the return value is always taken from `LookupFirstAC`. There are three cases to consider in checking that this return value is correct. First, if `LookupFirstAC` returns `OK`, then exact cancellation semantics mean that the lookup has been performed and the result can be passed back as usual. Second, if the `SleepAC` timeout expires and cancels `LookupFirstAC`, then the resulting `CANCELLED` return value is correct. Finally, if `LookupWithTimeoutAC` is itself cancelled by its caller, then the result of the call to `LookupFirstAC` determines the overall result of the `LookupWithTimeoutAC` operation.

## 3. Operational Semantics

In this section we define an operational semantics for a core language modeling `async`, `do..finish` and `cancel` (Figure 5). The aim is to define precisely their interaction. For instance, exactly when execution can switch between one piece of code and another, and exactly how execution proceeds at the point when a `cancel` command is executed (e.g., if new IO is subsequently issued, or if further `async` commands are run).

We start from a simple imperative language with global mutable variables. We model IO via `send`/`recv` operations on message channels. The semantics of message channels (which we define below) are based on those of the Barrelfish OS. For simplicity, we treat only the special case of `cancel` without a label. Since we focus on the behavior of the AC constructs, we keep the remainder of the language simple: we omit functions; channels carry only integer values; all names are global; and variable names are distinct from channel names.

The constructs for boolean conditionals *BExp* and numerical expressions *NExp* are conventional. The symbol $x$ ranges over variable names, $r$ ranges over message channel names, and $v$ ranges over values. A store $\sigma$ is a mapping from variable names to values, and a buffer state $\beta$ is a mapping from channel names to lists of values that represent a channel's buffered contents.

***Commands.*** $C$ and $D$ range over commands. Many are conventional: `skip`, assignment, sequencing, conditionals, and `while`.

The command `async` $C$ models the `async` construct. The command `pool` $f$ $CS$ generalizes the `do..finish` construct, representing a "pool" of commands (multi-set $CS$) that are eligible to continue within the `do..finish`. The flag $f$ indicates whether the `do..finish` is active, or whether it has been cancelled. The source construct `do` C

**Definitions**

$$
\begin{array}{rcll}
b & \in & \mathit{BExp} & = \ \ldots \\
e & \in & \mathit{NExp} & = \ \ldots \\
x & \in & \mathit{Var} & \\
r & \in & \mathit{Chan} & \\
v & \in & \mathit{Value} & = \ \ldots \\
\sigma & \in & \mathit{Store} & = \ \mathit{Var} \mapsto \mathit{Value} \\
\beta & \in & \mathit{Buffers} & = \ \mathit{Chan} \mapsto [\mathit{Value}]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E} & = & [\,] \\
 & \mid & \mathcal{E}; C \\
 & \mid & \texttt{pool}\ f\ \mathcal{E}\ ::CS
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_\mathrm{c} & = & \mathcal{E}_\mathrm{c}; C \\
 & \mid & \texttt{pool ACTV}\ \mathcal{E}_\mathrm{c} ::CS \\
 & \mid & \texttt{pool CNCL}\ \mathcal{E} ::CS
\end{array}
$$

$$
\begin{array}{rcll}
C, D & \in & \mathit{Com} & = \ F \\
 & & & \mid \ x := e \\
 & & & \mid \ C; D \\
 & & & \mid \ \texttt{if}\ b\ \texttt{then}\ C\ \texttt{else}\ D \\
 & & & \mid \ \texttt{while}\ b\ \texttt{do}\ C \\
 & & & \mid \ \texttt{async}\ C \\
 & & & \mid \ \texttt{pool}\ (\texttt{ACTV} \mid \texttt{CNCL})\ \mathit{CS} \\
 & & & \mid \ \texttt{cancel}
\end{array}
$$

$$
\begin{array}{rcll}
F & \in & \mathit{FinalCom} & = \ \texttt{skip} \\
 & & & \mid \ \texttt{try}\ \mathit{IO}\ \texttt{else}\ D \\
 & & & \mid \ \mathit{IO}
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{IO} & \in & \mathit{IOCom} & = \ \texttt{send}\ e\ \texttt{to}\ r \\
 & & & \mid \ \texttt{recv}\ x\ \texttt{from}\ r
\end{array}
$$

**Top-level transitions**

$$\boxed{\langle \sigma,\, \beta,\, C\rangle \to \langle \sigma',\, \beta',\, C'\rangle}$$

$$\frac{\langle \sigma,\, \_,\, \texttt{pool}\ f\ \mathit{CS}\rangle \Rightarrow\ \langle \sigma',\, \_,\, C'\rangle}{\langle \sigma,\, \beta,\, \texttt{pool}\ f\ \mathit{CS}\rangle \to \langle \sigma',\, \beta,\, C'\rangle}\ \text{(T-Eval)}$$

$$\frac{\beta(r) = vs \quad \mathit{pushleft}(\sigma(e), vs, vs')}{\langle \sigma,\, \beta,\, \mathcal{E}[\,\texttt{send}\ e\ \texttt{to}\ r\,]\rangle \to \langle \sigma,\, \beta[r \mapsto vs'],\, \mathcal{E}[\,\texttt{skip}\,]\rangle}\ \text{(T-Send)}$$

$$\frac{\beta(r) = vs \quad \mathit{popright}(vs, vs', v)}{\langle \sigma,\, \beta,\, \mathcal{E}[\,\texttt{recv}\ x\ \texttt{from}\ r\,]\rangle \to \langle \sigma[x \mapsto v],\, \beta[r \mapsto vs'],\, \mathcal{E}[\,\texttt{skip}\,]\rangle}\ \text{(T-Recv)}$$

$$\frac{\langle \sigma,\, \beta,\, \mathcal{E}[\,\mathit{IO}\,]\rangle \to \langle \sigma',\, \beta',\, \mathcal{E}[\,\texttt{skip}\,]\rangle}{\langle \sigma,\, \beta,\, \mathcal{E}[\,\texttt{try}\ \mathit{IO}\ \texttt{else}\ D\,]\rangle \to \langle \sigma',\, \beta',\, \mathcal{E}[\,\texttt{skip}\,]\rangle}\ \text{(T-Try-IO)}$$

$$\langle \sigma,\, \beta,\, \mathcal{E}_\mathrm{c}[\,\texttt{try}\ \mathit{IO}\ \texttt{else}\ D\,]\rangle \to \langle \sigma,\, \beta,\, \mathcal{E}_\mathrm{c}[\,D\,]\rangle\ \text{(T-Try-Cancel)}$$

**Big-step evaluation of commands**

$$\boxed{\langle \sigma,\, \texttt{pool}\ f\ \mathit{CS},\, C\rangle \Rightarrow\ \langle \sigma',\, \texttt{pool}\ f'\ \mathit{CS}',\, C'\rangle}$$

$$\frac{F \in \mathit{FinalCom}}{\langle \sigma,\, p,\, F\rangle \Rightarrow\ \langle \sigma,\, p,\, F\rangle}\ \text{(Final)} \qquad \langle \sigma,\, p,\, x := e\rangle \Rightarrow \langle \sigma[x \mapsto \sigma(e)],\, p,\, \texttt{skip}\rangle\ \text{(Assign)}$$

$$\frac{\langle \sigma,\, p,\, C\rangle \Rightarrow\ \langle \sigma',\, p',\, \texttt{skip}\rangle \quad \langle \sigma',\, p',\, D\rangle \Rightarrow\ \langle \sigma'',\, p'',\, D'\rangle}{\langle \sigma,\, p,\, C; D\rangle \Rightarrow\ \langle \sigma'',\, p'',\, D'\rangle}\ \text{(Seq-1)}$$

$$\frac{\langle \sigma,\, p,\, C\rangle \Rightarrow\ \langle \sigma',\, p',\, C'\rangle \quad C' \neq \texttt{skip}}{\langle \sigma,\, p,\, C; D\rangle \Rightarrow\ \langle \sigma',\, p',\, C'; D\rangle}\ \text{(Seq-2)}$$

$$\frac{\sigma(b) = \texttt{true} \quad \langle \sigma,\, p,\, C\rangle \Rightarrow\ \langle \sigma',\, p',\, C'\rangle}{\langle \sigma,\, p,\, \texttt{if}\ b\ \texttt{then}\ C\ \texttt{else}\ D\rangle \Rightarrow\ \langle \sigma',\, p',\, C'\rangle}\ \text{(If-1)}$$

$$\frac{\sigma(b) = \texttt{false} \quad \langle \sigma,\, p,\, D\rangle \Rightarrow\ \langle \sigma',\, p',\, D'\rangle}{\langle \sigma,\, p,\, \texttt{if}\ b\ \texttt{then}\ C\ \texttt{else}\ D\rangle \Rightarrow\ \langle \sigma',\, p',\, D'\rangle}\ \text{(If-2)}$$

$$\frac{\sigma(b) = \texttt{true} \quad \langle \sigma,\, p,\, C; \texttt{while}\ b\ \texttt{do}\ C\rangle \Rightarrow\ \langle \sigma',\, p',\, C'\rangle}{\langle \sigma,\, p,\, \texttt{while}\ b\ \texttt{do}\ C\rangle \Rightarrow\ \langle \sigma',\, p',\, C'\rangle}\ \text{(While-1)}$$

$$\frac{\sigma(b) = \texttt{false}}{\langle \sigma,\, p,\, \texttt{while}\ b\ \texttt{do}\ C\rangle \Rightarrow\ \langle \sigma,\, p,\, \texttt{skip}\rangle}\ \text{(While-2)}$$

$$\frac{\langle \sigma,\, p,\, C\rangle \Rightarrow\ \langle \sigma',\, p',\, \texttt{skip}\rangle}{\langle \sigma,\, p,\, \texttt{async}\ C\rangle \Rightarrow\ \langle \sigma',\, p',\, \texttt{skip}\rangle}\ \text{(Async-1)}$$

$$\frac{\langle \sigma,\, \texttt{pool}\ f\ \mathit{CS},\, C\rangle \Rightarrow\ \langle \sigma',\, \texttt{pool}\ f'\ \mathit{CS}',\, C'\rangle \quad C' \neq \texttt{skip}}{\langle \sigma,\, \texttt{pool}\ f\ \mathit{CS},\, \texttt{async}\ C\rangle \Rightarrow\ \langle \sigma',\, \texttt{pool}\ f'\ C' ::CS',\, \texttt{skip}\rangle}\ \text{(Async-2)}$$

$$\langle \sigma,\, p,\, \texttt{pool}\ f\ \emptyset\rangle \Rightarrow \langle \sigma,\, p,\, \texttt{skip}\rangle\ \text{(Pool-1)}$$

$$\frac{\langle \sigma,\, \texttt{pool}\ f\ \mathit{CS},\, C\rangle \Rightarrow\ \langle \sigma',\, \texttt{pool}\ f'\ \mathit{CS}',\, C'\rangle \quad C' \neq \texttt{skip}}{\langle \sigma,\, p,\, \texttt{pool}\ f\ C ::CS\rangle \Rightarrow\ \langle \sigma',\, p,\, \texttt{pool}\ f'\ C' ::CS'\rangle}\ \text{(Pool-2)}$$

$$\frac{\langle \sigma,\, \texttt{pool}\ f\ \mathit{CS},\, C\rangle \Rightarrow\ \langle \sigma',\, \texttt{pool}\ f'\ \mathit{CS}',\, \texttt{skip}\rangle \quad \langle \sigma',\, p,\, \texttt{pool}\ f'\ \mathit{CS}'\rangle \Rightarrow\ \langle \sigma'',\, p'',\, C''\rangle}{\langle \sigma,\, p,\, \texttt{pool}\ f\ C ::CS\rangle \Rightarrow\ \langle \sigma'',\, p'',\, C''\rangle}\ \text{(Pool-3)}$$

$$\langle \sigma,\, \texttt{pool}\ f\ \mathit{CS},\, \texttt{cancel}\rangle \Rightarrow \langle \sigma,\, \texttt{pool CNCL}\ \mathit{CS},\, \texttt{skip}\rangle\ \text{(Cancel)}$$

**Figure 5.** Operational semantics.

`finish` is syntactic sugar for `pool ACTV` $\{C\}$ and, intuitively, the multi-set $CS$ is used to accumulate asynchronous work that is started within a given `do..finish`. The command `cancel` triggers cancellation of the enclosing pool.

The command `send` $e$ `to` $r$ sends the value of expression $e$ on channel $r$. The command `recv` $x$ `from` $r$ receives a value from channel $r$ and stores it in variable $x$. The `send`/`recv` operations model non-cancellable IO. Cancellable IO is modeled by a `try` $\mathit{IO}$ `else` $D$ command where $\mathit{IO}$ ranges over `send`/`recv` operations. $D$ is a cancellation action which is executed if the IO operation is can-

celled. $D$ might, for instance, update a variable to indicate that cancellation has occurred.

Finally, we require that a program initially has the form `pool ACTV` $\{C\}$. This requirement ensures that the notion of "enclosing pool" is always defined within the body of the program $C$.

***Top-level transitions.*** We define a transition relation $\rightarrow$ between states $\langle \sigma,\ \beta,\ C \rangle$. We call these "top-level transitions", and each step either models the execution of a piece of $C$ until it next blocks, or it models an IO operation or the cancellation of an IO operation. Execution therefore proceeds as a series of $\rightarrow$ transitions, interspersing computation and IO.

The rule (T-Eval) models execution of a command via the big-step relation $\Rightarrow$ (defined below). Note that the rule takes the entire program as a command, rather than extracting part of it from a context. The rules ensure that the top-level command is either a pool (if execution is not yet complete), or that it has been reduced to `skip` (in which case no further applications of (T-Eval) can occur). The states for the $\Rightarrow$ transitions include a "current pool" component in place of the message buffers $\beta$. In the hypothesis of (T-Eval) we leave the current pool blank ("_") because that component of the state is not accessed when the command itself is a pool, as in the case of (T-Eval). Formally, the blank "_" can be replaced with an arbitrary pool (even two different pools on the two sides of the hypothesis).

The remaining four top-level transition rules model IO and cancellation of IO. We define these rules in terms of execution contexts $\mathcal{E}$ and cancelled-execution contexts $\mathcal{E}_{\mathrm{c}}$. Within an ordinary execution context, the hole $[\ ]$ can occur on the left hand side of sequencing, and at any command within a pool (we write $C{::}CS$ to decompose a pool by selecting an arbitrary element $C$ from it, leaving the remainder $CS$). Within a cancelled-execution context, the hole must occur within a cancelled pool (either directly, or with intermediate non-cancelled pools).

(T-Send) selects a `send` command in an execution context, and pushes the value being sent onto the left hand end of the message buffer named by $r$. We define the relation $\mathit{pushleft}(e, vs, vs')$ to be true iff the list $vs'$ is obtained by pushing $e$ on the left hand end of $vs$. (T-Recv) selects a `recv` command in an execution context, and removes the value being received from the right hand end of the message buffer named by $r$. The relation $\mathit{popright}(vs, vs', v')$ is true iff the list $vs$ can be decomposed by taking $v'$ from the right and leaving $vs'$. Hence, this rule can apply only when $vs$ is non-empty. (T-Try-IO) allows an IO command to be performed within a `try` statement in an execution context. The `try` is discarded if the IO completes. Finally, (T-Try-Cancel) allows a `try` $IO$ `else` $D$ command to be reduced to $D$ if it occurs within a cancelled-execution context.

***Evaluation of commands.*** We define a big-step structural operational semantics for commands. A transition $\langle \sigma,\ p,\ C \rangle \ \Rightarrow\ \langle \sigma',\ p',\ C' \rangle$ means that command $C$ enclosed by pool $p$ with variable state $\sigma$ evaluates to leave command $C'$, with modified pool $p'$ and state $\sigma'$. The pool may be modified, for instance, by adding commands to it if $C$ spawns asynchronous work that blocks.

The $\Rightarrow$ transition relation embodies a big-step semantics: it models the complete execution of $C$ until it is finished ($C' = \texttt{skip}$), or until $C$ next blocks. This design captures our decision to keep the AC constructs for controlling asynchrony separate from constructs for parallel execution: a small-step semantics would need a mechanism to prevent interleaving between multiple pieces of work that have been started asynchronously. (An earlier small-step version of our semantics attempted to control interleaving by marking a designated "active command"; however, it was cumbersome to ensure that the marker was moved in a way that provided the serial elision property.)

In the definition of $\Rightarrow$, the rule (Final) handles `skip`, `try`, `send`, and `recv`. No further evaluation is possible for these "final" commands: execution is complete in the case of `skip`, and has blocked in the other cases. The rule (Assign) handles assignment, by updating the state and leaving `skip`. The rule (Seq-1) handles sequencing when the first command in the sequence evaluates to `skip`. The rule (Seq-2) handles sequencing when the first command blocks; the sequence as a whole blocks. The rules (If-1), (If-2), (While-1) and (While-2) are conventional.

The rule (Async-1) handles `async` commands that run to completion: `async` $C$ runs to completion if $C$ does. This rule reflects our design decision to run the body of an `async` command before running its continuation, and helps to provide the serial elision property. The rule (Async-2) handles `async` $C$ commands where $C$ blocks: $C$ runs as far as $C'$, this remainder is then put into the pool, and `async` $C$ as a whole evaluates to `skip` instead of blocking.

There are three rules for pools. The first (Pool-1) reduces a pool that has emptied to `skip`. The second (Pool-2) takes a command $C$ from a pool and evaluates $C$ until it blocks (leaving $C' \neq \texttt{skip}$) and is put back into the pool. The third (Pool-3) takes a command $C$ from a pool, evaluates it to completion (leaving `skip`), and then continues evaluating the modified pool. Collectively, these rules allow a top-level transition under (T-Eval) to execute any of the commands from within a set of nested pools.

The final rule is (Cancel); a `cancel` command simply sets the flag on the current pool to be `CNCL`.

***Properties.*** We briefly state properties of the semantics (for brevity, we omit the proofs, which are typically routine inductions on derivations):

— If $\langle \sigma,\ p,\ \texttt{pool}\ f\ CS \rangle \ \Rightarrow\ \langle \sigma',\ p',\ C' \rangle$ then $p = p'$. That is, evaluating one pool has no effect on the surrounding pool, either in terms of cancellation or the addition or removal of commands. This property is why we omit a surrounding pool in the hypothesis of (T-Eval).

— Let $ae(C)$ be the asynchronous elision of command $C$, applied recursively to commands and leaving them unchanged except that $ae(\texttt{async}\ C) = ae(C)$. If commands $CS$ do not include `send` or `recv` operations and $\langle \sigma, \beta, \texttt{pool}\ f\ CS\rangle \rightarrow \langle \sigma', \beta', C'\rangle$ then there is a transition $\langle \sigma, \beta, ae(\texttt{pool}\ f\ CS)\rangle \rightarrow \langle \sigma', \beta', ae(C')\rangle$. That is, the use of `async` does not affect the behavior of a program that does not block; this follows our informal serial elision property from the introduction.

— Let $ce(C)$ be the cancellation elision of command $C$, applied recursively to commands and leaving them unchanged except that $ce(\texttt{cancel}) = \texttt{skip}$ and that $ce(\texttt{pool}\ f\ CS) = ce(\texttt{pool CNCL}\ CS)$. If commands $CS$ do not include `try` (i.e., they are non-cancellable) and $\langle \sigma, \beta, \texttt{pool}\ f\ CS\rangle \rightarrow \langle \sigma', \beta', C'\rangle$ then there is a transition $\langle \sigma, \beta, ce(\texttt{pool}\ f\ CS)\rangle \rightarrow \langle \sigma', \beta', ce(C')\rangle$ That is, the use of `cancel` does not affect the behavior of a program in which the IO operations are non-cancellable.

## 4. Implementation

In this section we discuss our implementations of AC. We discuss how the core language features are built via a modified compiler (Section 4.1) or via C/C++ macros (Section 4.2). Finally, we show how to integrate callback-based asynchronous IO with AC (Section 4.3).

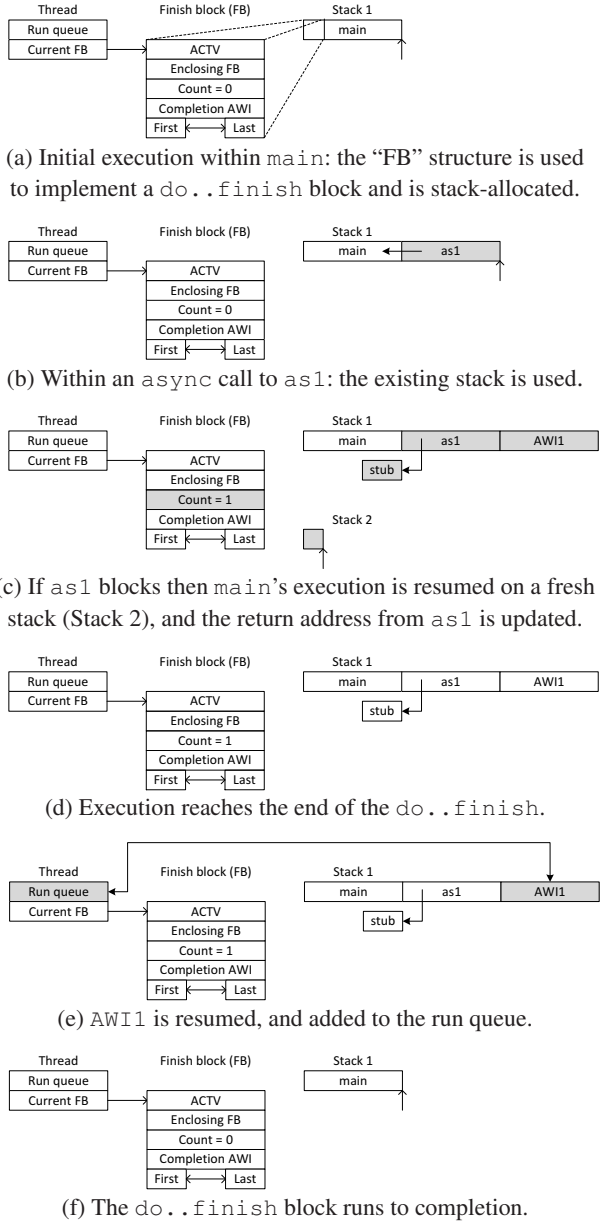
### 4.1 Clang/LLVM Implementation

Our first implementation is based on the Clang/LLVM tool-chain (v2.7). We de-sugar `async` code blocks into `async` calls to compiler-generated functions. We do this de-sugaring by translating the contents of the `async` statement into an LLVM code block (a form of closure added in LLVM as an extension to C).

At runtime, the implementation of `do..finish` and `async` is based on a cactus stack with branches of the stack representing `async` operations that have started but not yet completed. In our workloads, many `async` calls complete without blocking (e.g., because a channel already contains a message when a receive operation is executed). Therefore, we defer as much bookkeeping work as possible until an `async` call actually does block (much as with lazy task creation [26] in which the creation of a thread is deferred until an idle processor is available). In particular, we do not update any runtime system data structures when making an `async` call, and we allow the callee to execute on the same stack as the caller (rather than requiring a stack switch). To illustrate this technique, consider the following example:

```
void main(void) {
  do {
    async as1();
  } finish;
}
```

Figure 6(a) shows the initial state of the runtime system within the `do..finish` block. Associated with each thread is a run queue of "atomic work items" (AWIs), which




(a) Initial execution within `main`: the "FB" structure is used to implement a `do..finish` block and is stack-allocated.

(b) Within an `async` call to `as1`: the existing stack is used.

(c) If `as1` blocks then `main`'s execution is resumed on a fresh stack (Stack 2), and the return address from `as1` is updated.

(d) Execution reaches the end of the `do..finish`.

(e) `AWI1` is resumed, and added to the run queue.

(f) The `do..finish` block runs to completion.

**Figure 6.** Runtime system data structures. Changes are shaded at each step.

represent pieces of work that are ready to run when the thread becomes idle. Concretely, an AWI is simply a saved program counter, a stack-pointer, and a thread ID for the thread that created the AWI. To reduce locking overheads, the run queue is structured as a pair of doubly linked lists, one that is accessed without concurrency-control by the thread itself, and a second that is accessed by other threads (e.g., when an AWI increments a semaphore, then the thread running that AWI may need to the access the run queue for an AWI that was blocked on the semaphore).

In addition to the run queue, each thread has a "current FB" that identifies the `do..finish` block that it is within. Each FB (Finish Block) structure is stack-allocated in the

frame where the do..finish block is entered; the semantics of do..finish mean that this function can return only once the block is complete. The FB has a cancelled/active flag (initially ACTV), a pointer to the FB structure for the dynamically enclosing FB, a count of the number of asynchronous calls that have been started but not yet finished, a reference to a special "completion AWI" (which we describe below), and a doubly-linked-list holding (*i*) functions to execute if the FB is cancelled, and (*ii*) FBs for any do..finish blocks dynamically nested within this one. In the figure, the list is empty.

In the example, execution starts with an async call to as1 (Figure 6(b)). A new stack frame is allocated in the usual way; if as1 were to return normally then execution would simply continue after the asynchronous call.

However, if as1 blocks (Figure 6(c)), then the runtime system (*i*) increments the count of blocked items in the enclosing FB, (*ii*) allocates a new AWI representing the blocked work, placing this AWI at the end of the stack frame for as1, and (*iii*) walks the stack to find the closest enclosing call site (if any) corresponding to an async call whose continuation has not been resumed. A compiler-generated table of return addresses is used to identify async calls.

If there is an async call then the return address from the callee's stack frame is rewritten to go to a stub function (described below), and a new stack is allocated for the caller. In our implementations we reserve 1MB of virtual address space for each stack, and within this space we lazily allocate 4KB pages of physical memory using a guard page. In the figure, Stack 2 is allocated and execution resumes within main on the new stack. To allow execution to move between stacks, a new calling convention is used for asynchronous calls: (*i*) all registers are treated as caller-save at an asynchronous call site, and (*ii*) a frame pointer is used for all functions containing asynchronous calls. The first rule allows execution to resume after the call without needing to recover values for callee-save registers, and the second rule allows the resumed caller to execute on a discontiguous stack from the original call (e.g., in Figure 6(c)) by restoring the original frame pointer but using a fresh stack pointer.

If a function blocks when there are no async calls then execution continues by resuming an AWI from the current thread's run queue. If the run queue itself is empty then execution blocks for an asynchronous IO operation to complete.

In the example, execution continues in main and reaches the end of the do..finish (Figure 6(d)). At this point the runtime system checks whether (*i*) the FB has any async calls which are not yet complete (i.e., $count \neq 0$), and (*ii*) if the count is zero, whether execution is on the original stack that entered the do..finish. In this case the count on the current FB is non-zero, so execution blocks.

Figure 6(e) shows the situation when the IO performed within as1 has completed: the suspended AWI is added to the run queue, and is resumed by the thread. The function as1 then completes, and returns to the "stub" function linked to the stack in Figure 6(c). The stub function re-examines the count field to check whether as1 was the last outstanding async function for that FB: in this case the count is decremented to 0, and execution resumes outside the do..finish back on Stack 1 (Figure 6(f)).

The completion AWI field is not used in this example. Its role is to ensure that execution leaves a do..finish block on the same stack that entered it (in this case Stack 1). If the work running on the original stack finishes when the current FB's count is still non-zero then the completion AWI is initialized for the work immediately after the finish. Then, when the count reaches zero on another stack, execution is transferred to the completion AWI and hence back to the original stack.

```
// Scheduling
void Suspend(awi_t **xp);
void SuspendUnlock(awi_t **xp, spinlock_t *l);
void Schedule(awi_t *x);
void Yield();
void YieldTo(awi_t *x);

// Cancellation
bool IsCancelled();
void AddCancelItem(cancel_item_t *ci, fn_t fn, void *arg);
void RemoveCancelItem(cancel_item_t *ci);
```

**Figure 7.** Low-level API for integrating asynchronous IO operations.

### 4.2 Macro-Based Implementation

In addition to our compiler-based implementation, we have developed an implementation based on C/C++ macros that exploits existing extensions for defining nested functions. This implementation lets us use AC on platforms that are not supported by LLVM (e.g., the Beehive FPGA-based processor [34]). Comparing the two implementations also lets us assess the advantages and disadvantages of including language features to support asynchronous IO.

There are two differences from the Clang/LLVM implementation: First, syntactically, the macro-based implementation uses ASYNC(X) to express an async statement containing statements X, DO_FINISH(X) for a do..finish containing X, and DO_FINISH_(lbl,X) for a block with label lbl. The DO_FINISH macros define blocks that start and end with calls to the AC runtime system. The ASYNC macro defines a nested function that contains the contents of the async statement, and then it calls the AC runtime system to execute the nested function.

The second difference is that the macro-based implementation does not produce the compiler-generated tables to let us walk the stack, when blocking, to identify async calls. We therefore investigated two alternative approaches at async calls: (*i*) eagerly allocating a stack when making the async call; or (*ii*) pushing an explicit marker onto the stack during an async call, to enable lazy allocation of the stack only if the async call blocks. Eager allocation is relatively simple to implement, however, as we show in Sec-

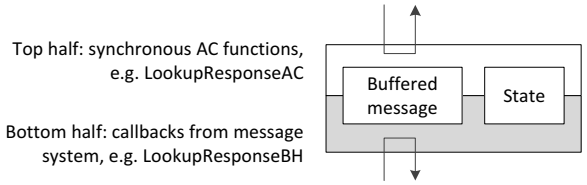



**Figure 8.** Integrating AC with callback-based messages.

tion 5, it incurs a performance cost of around 110 cycles per `async` call. Macro-based lazy allocation allows stack allocation itself to be made lazy, but still requires some eager bookkeeping to initialize an AWI for the call's continuation, and it requires an additional level of indirection on each `async` call. It adds around 30 cycles overhead per `async` call, when compared with the compiler-integrated lazy implementation.

### 4.3 Integrating Callback-Based IO with AC

The AC runtime system provides a set of functions through which asynchronous IO operations interact with the new language constructs; these provide a way to adapt existing callback-based abstractions to a form where they can be invoked from AC. Figure 7 shows the operations:

***Scheduling.*** The first set of operations is used to control scheduling of AWIs. `Suspend(xp)` ends the current AWI, and initializes `*xp` with a pointer to a new AWI for the continuation of the `Suspend` call. In practice the AWI is allocated in the stack frame of the `Suspend` call, as in Figure 6(c). `SuspendUnlock` is the same as `Suspend`, except that a given spinlock is released after blocking (we illustrate its use below). `Schedule(x)` adds the AWI pointed to by `x` to the run queue of the thread to which the AWI belongs. `Yield` ends the current AWI, and adds the continuation of the `Yield` immediately back to the run queue of the current thread. Finally, `YieldTo(x)` is a directed yield: If `x` belongs to the current thread, then the continuation of `YieldTo` is put on the run queue, and execution proceeds immediately to the AWI to which `x` refers. If `x` belongs to a different thread, then `YieldTo(x)` is equivalent to `Schedule(x)`.

***Cancellation.*** The second set of operations in Figure 7 interacts with cancellation. This basic abstraction is a "cancellation item" which is a callback function to be run when cancellation is triggered. These callbacks are registered when cancellable operations are started, and de-registered when cancellable operations complete. The cancellation items are stored on the doubly-linked-list held in the FB structure for the enclosing `do..finish` block. These callbacks are run when the `cancel` statement is executed.

***Example.*** We have developed versions of AC for message passing on Barrelfish, and for asynchronous IO more

```
enum { EMPTY, BH_WAITING,
       TH_WAITING, TH_CANCELLED } rx_state;

typedef struct {
  spinlock_t spinlock;  // Lock to protect other fields
  enum rx_state state;  // Receiver record state
  ac_lock_t bh_lock;    // Lock to order BH execution
  ac_lock_t th_lock;    // Lock to order TH execution
  awi_t *rx_awi;        // Saved TH context
  int addr;             // Message payload
} LookupResponse_t;

// Bottom-half function to execute on incoming message:
static void LookupResponseBH(NSChannel_t *c, int addr) {
  LookupResponse_t *r = ...;
  LockNAC(&r->bh_lock); // Wait for previous BH to complete

  SpinlockAcquire(&r->spinlock);
  r->addr = addr;
  if (r->state == EMPTY) { // No top-half waiting
    r->state = BH_WAITING;
    SpinlockRelease(&r->delivery_lock);
  } else {              // Pass control to top-half
    YieldTo(r->rx_awi);
} }

// AC top-half function to receive a message:
int LookupResponseAC(NSChannel_t *c, int *addr) {
  LookupResponse_t *r = ...;
  int result = OK;

  // Wait for previous TH to complete
  if (LockAC(&(r->th_lock)) == CANCELLED) return CANCELLED;

  SpinlockAcquire(&r->spinlock);
  if (r->state == EMPTY) { // No BH present: wait
    r->state = TH_WAITING;
    cancel_item_t ci;
    AddCancelItem(&ci, &CancelRecv, r);
    SuspendUnlock(&r->rx_awi, &r->spinlock);

    // Resumed here after delivery or cancellation
    RemoveCancelItem(&ci);
    if (r->state == TH_CANCELLED) {
      result = CANCELLED;  // We were cancelled
      goto done;
  } }

  *addr = r->addr;          // Extract message contents
  Unlock(&r->bh_lock);      // Allow next BH callback
done:
  r->state = EMPTY;
  SpinlockRelease(&r->spinlock);
  Unlock(&(r->th_lock));    // Allow next TH operation
  return result;
}

static void CancelRecv(LookupResponse_t *r) {
  SpinlockAcquire(&r->spinlock);
  if (r->state == TH_WAITING) {
    r->state = TH_CANCELLED;
    Schedule(r->rx_awi);
  } else {
    SpinlockRelease(&r->spinlock);
} }
```

**Figure 9.** Interfacing Barrelfish callbacks with AC.

generally on Microsoft Windows. For brevity we focus on receiving a message on Barrelfish, showing how the synchronous AC operation is built over the callback-based interface sketched in the introduction. Other functions follow a similar pattern.

Figure 8 shows the overall approach. For each message channel, we record a buffered message and a state. These are updated by a "bottom-half" function that runs as a callback, and a "top-half" function that runs as a blocking AC operation. The callback waits until the buffer is empty, and

then deposits the next message. The top-half function waits until a message is available in the buffer.

Figure 9 shows a simplified version of the implementation (the logic follows the full version, but we use shorter names, and omit some casts and function parameters). The `LookupResponse_t` structure provides the buffering. A spinlock protects access to the other fields. The `state` field records whether buffered data is waiting from a bottom-half function (`BH_WAITING`), whether a top-half function is blocked waiting for data (`TH_WAITING`), or whether a top-half function has just been cancelled (`TH_CANCELLED`). Two locks serialize executions of the top-half and bottom-half handlers, allowing only one of each kind to execute at any given time on a given buffer. The `rx_awi` field stores the saved context when a top-half function waits. The `addr` field carries the buffered message payload (in this case the address returned from the name-service lookup).

`LookupResponseBH` is the example bottom-half function. It waits on the bottom-half lock, and then updates the buffered state. It uses a directed `YieldTo` to transfer execution directly to the top-half, if one was waiting.

`LookupResponseAC` is the example top-half function. It waits on the top-half lock, consumes a buffered message if present, and otherwise marks the state as `TH_WAITING` before suspending itself. If cancellation occurs while the top-half is waiting, then the `CancelRecv` function is executed. This function tests whether or not a message has been delivered. If no message was delivered, then `CancelRecv` updates the state to `TH_CANCELLED` and then resumes the top-half code (note that the spinlock is held from within `CancelRecv` to the end of the top-half function—ensuring that the state is reset to `EMPTY` before the next message can be delivered).

## 5. Performance Evaluation

In this section we evaluate the performance of AC. We first look at microbenchmarks to show the overhead of individual AC operations (Section 5.1). We then measure the performance of AC using larger examples on Barrelfish and on Microsoft Windows. On Barrelfish, we use AC in the implementation of a low-level capability management system (Section 5.2). On Microsoft Windows, we use AC in a series of IO-intensive applications (Section 5.3).

We use an AMD64 machine with 4 quad-core processors (Sections 5.1 and 5.2), and an HP workstation with an Intel Core 2 Duo processor (Section 5.3). All results use optimized code (`-O2`). We validated that our modified compiler's performance is consistent with the baseline compiler and with `gcc` 4.3.4. Experiments are typically run with 10 000 iterations, using the first 9 000 as warm-up and reporting results from the final 1 000. In each case we confirmed that this delay avoided start-up effects. We report median values, and give 5-95%-ile ranges for any results with significant variance.

### 5.1 Microbenchmarks

We compared the performance of (*i*) a normal function call, (*ii*) an `async` call to an empty function, and (*iii*) an `async` call to a function that yields—i.e., blocking, and then immediately unblocking. We examined the Clang/LLVM-based implementation which uses lazy bookkeeping, and the two macro-based implementations. The test makes calls in a tight loop, and we measured the cycles needed per call:

| | Call | Async | Yield |
|---|---|---|---|
| Lazy, compiler-integrated | 8 | 10 | 245 |
| Lazy, macros | 8 | 44 | 269 |
| Eager, macros | 8 | 120 | 247 |

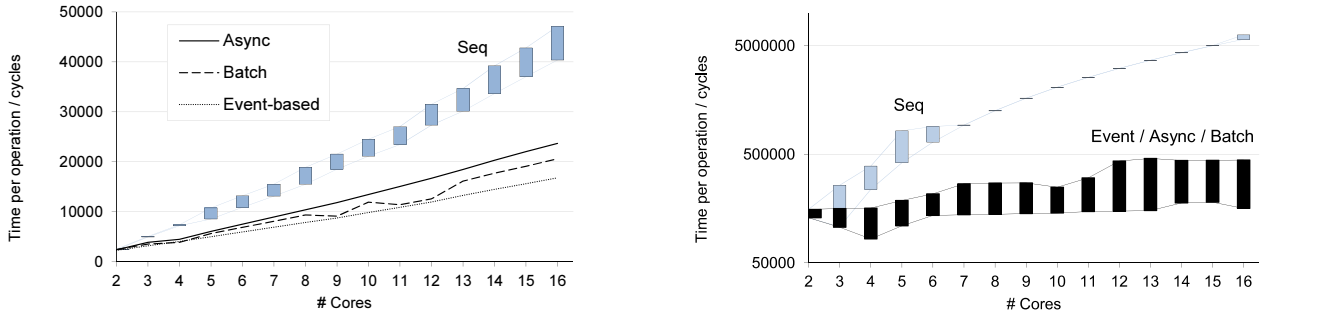
This kind of low-level timing is inevitably affected by the processor implementation and compilation details: the results confirm, however, that compiler-integrated lazy bookkeeping allows `async` to be used with very low overhead when the callee does not block, and that performing bookkeeping lazily does not harm performance if a callee does subsequently block. The lazy, macro-based implementation performs slightly less well than the compiler-integrated implementation. This result reflects an additional level of indirection that the macro-based implementation introduces on each asynchronous call.

We measured the performance of a ping-pong microbenchmark between processes on different cores. We used the compiler-integrated implementation of AC. The results show the time from sending a message from one core, until the corresponding response is received back on the same core:

| | Round trip latency / cycles |
|---|---|
| Callback-based API | 1121 |
| AC one side | 1198 |
| AC both sides | 1274 |
| AC without `YieldTo` | 1437 |
| MPI (HPC-Pack 2008 SDK) | 2780 |

We compared five implementations. "Callback-based API" uses the existing Barrelfish callback-based messaging library. This library provides a callback-based API which is built over shared memory on conventional systems, and which is implemented over hardware message passing on systems such as the Intel SCC [20],

The AC results show the cost of using AC on one or both sides of the ping-pong test. The difference is less than 15% over the callback-based API. Tracing the code, the sequence of operations performed is essentially the same in the callback-based and AC variants: the use of `YieldTo` stitches a bottom-half callback function directly to the top-half AC receive operation that was waiting. Replacing `YieldTo` with a non-directed `Schedule` leads to a 1437-cycle total cost. For comparison we also ran an analogous shared-memory MPI ping-pong test on identical hardware using Microsoft HPC-Pack 2008, and found it to be over twice as expensive as AC both sides.




(a) Low-latency configuration, unbounded spinning.

(b) Default configuration, brief spinning before pre-emption.

**Figure 10.** Capability re-typing benchmark.

### 5.2 Capability Management on Barrelfish

Our second test uses AC within Barrelfish. The OS uses capabilities to control access to physical resources. Possession of a capability on a core confers the right to perform a set of operations on the underlying resource without synchronization with other cores. The OS nodes use a 2-phase commit protocol to ensure that management operations on capabilities are performed in a consistent order across all cores. This protocol is an ideal usage scenario for AC: it is a performance-critical part of the OS, where easily understood and maintainable code is desirable to deal with the complexity.

Figure 10 shows the time to perform a capability operation. The protocol involves an initiating core sending messages to all the other cores, waiting for responses, and then sending a result back to the other cores. The first set of results, Figure 10(a), show an artificial configuration in which processes spin without bound while waiting for incoming messages. This configuration lets us focus on the best-case performance of different implementations (in practice the OS would preempt processes when they have no incoming messages).

We show four implementations. "Seq" uses synchronous communication: the initiator contacts each other core in turn, waiting for a response before moving on. The implementation is simple but performance is poor. "Event" is the existing Barrelfish implementation using manual stack-ripping: it comprises 19 callback functions which use 5 different kinds of temporary data structure. It gives the best performance, at the cost of code clarity in managing intermediate state. "Async" uses AC and follows the structure of "Seq" while adding `async` statements to interact with each core (it uses a single function, and no temporary structures). "Batch" uses AC, but restructures the communication to send a batch of messages followed by waiting for a batch of responses. Both of these AC versions perform much better than "Seq" and scale similarly to "Event".

In Figure 10(b) we configure the system to preempt processes while waiting for incoming messages. Performance of "Seq" is much worse because a receiver is often not running when a message arrives (note the log scale). The "Event", "Async" and "Batch" implementations are indistinguishable: the implementations using AC provide the same performance as the callback-based implementation, while avoiding the need for manual stack-ripping.

### 5.3 IO on Microsoft Windows

We integrated AC with the existing asynchronous IO facilities on Microsoft Windows.

***Disk workload.*** Our first test program is a synthetic disk workload, modeling the behavior of a software RAID system. The test can be configured to use Windows asynchronous IO directly, or to use AC, or to use basic synchronous IO. The test issues random reads to two Intel 25-M solid-state storage devices (SSDs), configured as RAID0, each with sustained read throughput of up to 250MB/s (we ran other workloads, but omit them for brevity because they showed similar trends in the performance of the different implementations). The AC version uses `ReadFileAC`, keeping track of the number of requests that have been issued, and blocking when a limit is reached. Tracking IOs lets us control the depth of the pipeline of concurrent requests exposed to the disk subsystem. The AIO implementation uses the callback-based IO interfaces directly; as in the example in the introduction, the AIO implementation needs to be stack-ripped manually.

Figure 11 shows the results for 4k and 64k IO block sizes respectively. The AC and AIO implementations achieve similar throughput. The implementation using synchronous IO is substantially slower.

We compared the CPU consumption of the AC and AIO implementations. This experiment forms a "sanity check" that AC's throughput does not come at the expense of vastly greater CPU consumption. For 4k transfers the overhead ranges from 5.4% to 11.8%, while for 64k transfers we measured up to 9.8% more CPU cycles (user mode + kernel mode) used by AC. As the pipeline depth increases the number of passes through the AC scheduler loop decreases

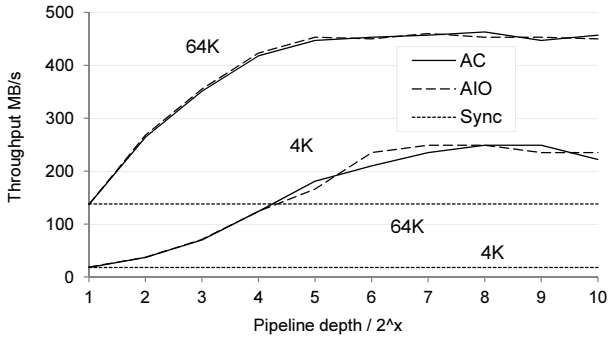



**Figure 11.** Microsoft Windows random access workload.

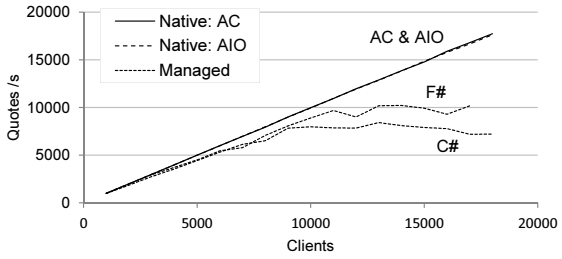
because each pass handles multiple completed requests. This mitigating factor meant that for 64k reads and 1024-request pipelines, we actually observed a *lower* CPU cost (-3.7%) for AC.

***Stock quote server.*** Our final test program is the "stock quote server" program used to illustrate the features in recent version of the .NET framework for asynchronous programming [32]. The server sends one message per second to a variable number of clients. We ran the server and clients on separate cores on the same machine. We increased the number of connections until system limits were reached. We compared the performance of four single-core server implementations: (*i*) an AC implementation, (*ii*) a C implementation using asynchronous IO, (*iii*) a C# implementation using asynchronous IO, and (*iv*) an F# implementation using asynchronous IO. The AC, C#, and F# implementations all avoid stack-ripping; the C implementation is written as stack-ripped callbacks that run in response to IO completion events.

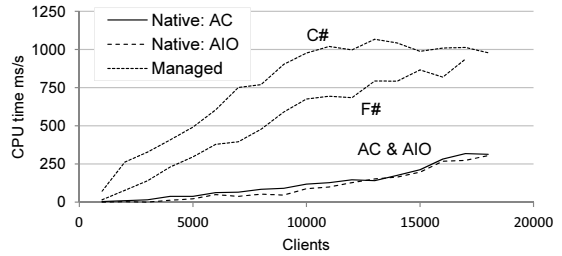
Figure 12(a) shows the throughput of the different implementations, measuring the number of clients actually serviced per second, as the number of client connections increases. The AC implementation scales as well as the stack-ripped AIO implementation: for a server running on a single core, both scale to around 18 000 clients (at which point system limits on network connections are reached). In contrast, the F# and C# clients are saturated at around 10 000 and 8 000 clients respectively.

Figure 12(b) shows the CPU consumption of the different implementations, measuring the milliseconds of CPU time consumed by the server process per second of elapsed time. The AC and AIO implementations have similar CPU consumption, less than $1/5$ of the consumption of the C# and F# implementations at 10 000 clients.

We compared the performance of these single-core server implementations with the performance of a server using multiple co-operating threads and synchronous IO. The multi-threaded server saturated at 4 500 clients per core, substantially less than the 18 000 clients handled by the AC and AIO implementations. This difference in performance




(a) Throughput.




(b) CPU consumption.

**Figure 12.** Stock-quote server benchmark.

was due to the synchronous IO operations performing more kernel-mode work than the asynchronous operations.

We examined why the F# and C# implementations differ in performance from the AC and AIO implementations. All of these implementations are ultimately built on the same asynchronous IO interfaces to the OS kernel, and the different performance comes from the user-mode parts of the implementations. There were two main factors which contributed approximately equally. First, the F# and C# implementations use heap-allocated temporary data structures to record information about pending IO operations. Allocating these adds to pressure on the GC. Second, the F# and C# implementations rely heavily on "pinning" data buffers in memory so that the buffers are not moved by the GC. Pinning introduced additional work allocating and de-allocating the `GCHandle` structures that are used to pin objects.

## 6. Related Work

Our techniques build on many areas of related work; we structure this discussion around (*i*) frameworks for performing asynchronous IO, (*ii*) implementations of abstractions for parallel programming, (*iii*) languages based on message passing, and (*iv*) techniques for cancellation of asynchronous IO.

***Frameworks for asynchronous IO.*** The merits of thread-based and callback-based programming models are frequently revisited. Lauer and Needham observed that particular forms of these models can be seen as duals: a program written in one model is essentially identical to a program written in the other [23]. Ousterhout argued that threads are a bad idea for most purposes (in terms of using them cor-

rectly, as well as performance) [27]. Adya *et al.* argued that the idea of "threads" conflates many related notions and there is value in using cooperative task management without manual stack management [3]; AC is inspired by this argument, and keeps management of asynchronous IO within a thread separate from management of parallel work. Based on related arguments, von Behren *et al.* suggested that many criticisms of threads were really due to the poor performance of early implementations [35].

Several techniques simplify asynchronous IO without removing manual stack-ripping. Dabek *et al.*'s `libasync` library helps manage event handlers' state, and provides concurrency-control between handlers [9]. Elmeleegy *et al.* propose that asynchronous IO interfaces should operate synchronously if a given call can complete immediately [11]. Cunningham and Kohler show how to restructure asynchronous IO interfaces to help link related operations [8]. Our work differs from these in avoiding stack-ripping.

Grand central dispatch (GCD) schedules tasks within an application over thread pools (`http://developer.apple.com/technologies/mac/snowleopard/gcd.html`). Typically, tasks run to completion and are defined as a form of closure in Objective C. Manual stack-ripping can be mitigated by nesting one closure within another (capturing the outer closure's variables).

Several systems allow asynchronous IO programs to be written without stack-ripping. Protothreads provides a system for programming embedded systems in a threaded style [10]. Each "thread" is a single function that is split into a series of event handlers. Fischer *et al.* use a more general version of this idea in TaskJava [12], allowing callees to wait for events. A manual annotation causes the TaskJava compiler to replace a method's body with a switch statement enabling it to be restarted at intermediate points; variable accesses are replaced with accesses to state records in the heap. Srinivasan and Mycroft use such transformations in the implementation of Kilim, an actor-based framework for Java [31]. We modify the runtime system to avoid the need for this kind of transformation.

Haller and Odersky's Scala Actors [15] combine thread-based and event-based programming models; code can block using a `receive` operation, or it can register an event handler using a `react` operation.

Tame provides a set of C++ abstractions for writing asynchronous IO in a thread-like style [22]. A thread can block within a function call; the call returns immediately, storing the contents of designated local variables into heap structures. Tame provides callback-based synchronization primitives and uses reference counting to manage temporary storage. CLARITY supports a thread-like model with nonblocking calls and a monitor-like synchronization mechanism [6]. As with an `async` call, execution proceeds to the continuation of a nonblocking call if the callee blocks. We integrate cancellation, and provide block-structured synchronization.

CPC [21] uses a compiler that transforms code to continuation passing style (CPS) to support large numbers of threads: threads comprise a dynamic chain of heap-allocated suspension records. As in AC an IO library provides integration with asynchronous IO operations exposed by the OS. AC avoids the need for CPS transformation, and adds language constructs for creating, synchronizing, and cancelling asynchronous work.

Li and Zdancewic combine callback-based and thread-based communication abstractions in GHC Haskell [25]. Thread-based operations are written in a monadic style to provide sequencing. A scheduler forces each thread's work to be evaluated up to the point of the next IO operation. Vouillon describes Lwt, an implementation of cooperative threads for OCaml [37]. Lwt provides primitive threads that perform individual AIO operations, along with a `bind` operator to chain these primitives together.

Syme *et al.* provide an "asynchronous modality" in the F# language [32]. A value of type `Async<T>` is a computation that can be run asynchronously and produce a value of type `T`. Control-flow syntax from the core F# language can be used to structure these asynchronous computations. Version 4 of the Microsoft .NET Framework supports asynchronous IO without manual stack-ripping (`http://www.microsoft.com/events/pdc/` session FT09). Functions that execute asynchronously must include an `async` modifier in their signature and have a `Task<T>` return type; their execution can then be suspended and resumed using heap-allocated temporary objects for the suspended state. In contrast, AC allows code performing asynchronous IO to be compiled as usual and uses temporary data directly on the stack for managing asynchronous work; this leads to better performance, as we illustrated in Section 5.

***Threads and parallel programming*** Although we keep the AC abstractions separate from those for expressing parallelism, our design and implementation builds on techniques for parallel programming. We gain concurrency-control simplifications by applying these techniques to computations within a single thread (e.g., we keep information about the continuations of `async` calls implicit in a thread's stack because these are accessed only by that thread itself). Mohr introduced the use of lazy task creation [26], deferring creation of a new thread to execute a computation until a spare processor is available. Our Clang/LLVM implementation defers creating a separate stack until a callee blocks. Goldstein introduced a taxonomy of techniques for lightweight threads [14]. Our handling of `async` calls is akin to "lazy disconnect" in Goldstein's terminology. CILK [13] implementations have used separate clones of functions to include or omit synchronization; we do not use cloning because our sequential model reduces the amount of synchronization involved. As in AC, CILK-4 uses a cactus-stack. CILK-5 uses heap-allocated frames.

StackThreads/MP provides a form of `async` call in which the callee can be stolen by an idle processor [33]. Stealing is co-operative (the victim is responsible for suspending the current work of the callee). StackThreads/MP multiplexes frames from multiple threads over a single stack; new frames are allocated at the top of the stack, but holes that appear are not filled. As in AC, the Capriccio system of von Behren *et al.* [36] multiplexes IO-intensive workloads on a single OS thread. It provides a traditional thread-based programming model whereas AC provides block-structured constructs for synchronizing and cancelling asynchronous IO.

The X10 language provides `async` and `finish` constructs, which partly inspired our design [7]. In X10, `async` creates work for parallel processors. X10's constructs would not provide, for example, the serial elision property of AC. Lee and Palsberg define an operational semantics for Featherweight X10 [24] using small-step transitions that can interleave work from parallel threads. AME provides a programming model based on serializable atomic actions. Executing `async` $C$ creates a new atomic action that will run $C$ after the current atomic action [19].

Sivaramakrishnan *et al.* describe a form of lightweight threading known as "parasitic threads" [29]. Parasitic threads are multiplexed over host threads, with multiple parasites using the same host stack at the same time. A combination of static analysis and dynamic checking is used to prevent collisions between frames from different parasites. We could apply these techniques to further reduce the cost of allocating stacks in AC.

***Message-based languages.*** Hoare's CSP [17] has inspired numerous language designs. Unlike CSP, our core language provides buffered send/receive operations as primitives and omits an "alt"-style operation to send/receive on exactly one of a set of alternatives. Our design reflects the primitives available in the OSes that we target. Typical OS interfaces do not provide exactly-one semantics because multiple requests could complete concurrently on different devices. A programmer using AC can build an "alt"-style operation via an additional software layer.

As in AC, the Alef [39] and Go (`golang.org`) languages both provide message passing operations within an imperative setting. Alef supports cooperative scheduling of coroutines as they block on communication operations. Go introduces a "goroutine" abstraction; these are multiplexed over OS threads (so different routines can run in parallel), but a segmented-stack implementation is used. AC focuses just on structuring asynchronous IO operations within a single thread, rather than using multiple OS threads. This choice lets us retain a sequential programming model.

Concurrent with our work, Ziarek *et al.* developed a system for composable asynchronous events [40], building on the first-class event abstraction of Concurrent ML [28]. Ziarek *et al.*'s asynchronous events encapsulate the implicit thread creation associated with an asynchronous action within an event structure, thus enabling composable construction of asynchronous protocols.

***Cancellation.*** AC's block-structured approach to cancellation is distinct from previous work. Modula-2+ [5] provides an "alert" mechanism that causes an exception to be raised in another thread if that thread is blocked at a synchronization operation. Java provides a similar mechanism. POSIX defines a notion of "cancellation points" at which one thread's work may be cancelled by another thread [1]. Typically, these are blocking system calls. In addition, POSIX asynchronous IO provides a `aio_cancel` operation to cancel either a single specified asynchronous IO operation, or to cancel all operations on a given file. Windows provides a "cancellation token" abstraction; a token can be passed to an asynchronous IO operation when it is started, and cancelling a token cancels all asynchronous requests associated with it.

## 7. Conclusion

AC provides IO with performance comparable to native callback-based asynchronous interfaces while retaining the composable programming style of synchronous operations. Our overall thesis is that asynchronous IO should be supported by using abstractions such as `async`, `do..finish` and `cancel` to describe the sources of asynchrony within an ordinary sequential program—rather than by the usual technique of developing a new, alternative set of IO interfaces based around explicit events or callbacks. Our results show that AC can match the performance of callback-based interfaces.

The Barrelfish research OS, including AC, is available from `http://barrelfish.org`.

## Acknowledgements

We would like to thank Zach Anderson, Mahesh Balakrishnan, Gérard Berry, Andrew Birrell, Miguel Castro, Austin Donnelly, Aleksandar Dragojević, Vladimir Gajinov, Steven Hand, Orion Hodson, Suresh Jagannathan, Butler Lampson, Simon Peyton Jones, Gordon Plotkin, Chandu Thekkath, and the anonymous reviewers for feedback on earlier drafts of this paper.

## References

[1] *POSIX 1003.1-2008*. The Open Group, 2008. `http://www.opengroup.org/onlinepubs/9699919799/toc.htm`.

[2] M. Abadi. Automatic mutual exclusion and atomicity checks. In *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, pages 510–526. Springer-Verlag, 2008.

[3] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *USENIX-02: Proc. 2002 Annual Technical Conference*, pages 289–302, 2002.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP '09: Proc. 22nd Symposium on Operating Systems Principles*, pages 29–44, 2009.

[5] A. Birrell, J. Guttag, J. Horning, and R. Levin. Synchronization primitives for a multiprocessor: a formal specification. In *SOSP '87: Proc. 11th Symposium on Operating Systems Principles*, pages 94–102, 1987.

[6] P. Chandrasekaran, C. L. Conway, J. M. Joy, and S.K. Rajamani. Programming asynchronous layers with CLARITY. In *ESEC-FSE '07: Proc. 6th European Software Engineering Conference & Symposium on the Foundations of Software Engineering*, pages 65–74, 2007.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proc. 20th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

[8] R. Cunningham and E. Kohler. Making events less slippery with eel. In *HotOS '05: Proc. 10th Conference on Hot Topics in Operating Systems*, 2005.

[9] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *Proc. 10th ACM SIGOPS European Workshop*, pages 186–189, 2002.

[10] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proc. 4th Conference on Embedded Networked Sensor Systems*, pages 29–42, 2006.

[11] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. Lazy asynchronous I/O for event-driven servers. In *USENIX-04: Proc. 2004 Annual Technical Conference*, pages 21–21, 2004.

[12] J. Fischer, R. Majumdar, and T. Millstein. Tasks: language support for event-driven programming. In *PEPM '07: Proc. 2007 Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 134–143, 2007.

[13] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proc. 1998 Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[14] S. C. Goldstein. *Lazy Threads Compiler and Runtime Structures for Fine-Grained Parallel Programming*. PhD thesis, University of California–Berkeley, Berkeley, 1997.

[15] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[16] E. A. Hauck and B. A. Dent. Burroughs' B6500/B7500 stack mechanism. In *AFIPS '68: Proc. of the Spring Joint Computer Conference*, pages 245–251, 1968.

[17] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[18] Intel Corporation. Single-chip cloud computer. `http://techresearch.intel.com/articles/Tera-Scale/1826.htm`, December 2009.

[19] M. Isard and A. Birrell. Automatic mutual exclusion. In *HotOS '07: Proc. 11th Workshop on Hot Topics in Operating Systems*, 2007.

[20] J. Howard *et al.* A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC '10: Proc. Solid-State Circuits Conference*, pages 108–109, 2010.

[21] G. Kerneis and J. Chroboczek. CPC: programming with a massive number of lightweight threads. In *PLACES '11: Proc. Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software*, 2011.

[22] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *USENIX-07: Proc. 2007 Annual Technical Conference*, pages 1–14, 2007.

[23] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *Proc. 2nd International Symposium on Operating Systems, IRIA*, 1978.

[24] J.K. Lee and J. Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *PPoPP '10: Proc. 15th Symposium on Principles and Practice of Parallel Programming*, pages 25–36, 2010.

[25] P. Li and S. Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *PLDI '07: Proc. 2007 Conference on Programming Language Design and Implementation*, pages 189–199, 2007.

[26] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 2:264–280, 1991.

[27] J. Ousterhout. Why threads are a bad idea (for most purposes). *Presentation given at the 1996 Usenix Annual Technical Conference*.

[28] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[29] K. C. Sivaramakrishnan, L. Ziarek, R. Prasad, and S. Jagannathan. Lightweight asynchrony using parasitic threads. In *DAMP '10: Proc. 2010 Workshop on Declarative Aspects of Multicore Programming*, 2010.

[30] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. In *OOPSLA '07: Proc. 22nd Conference on Object Oriented Programming Systems and Applications*, pages 191–210, 2007.

[31] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP '08: Proc. 22nd European Conference on Object-Oriented Programming*, pages 104–128, 2008.

[32] D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. In *PADL '11: Proc. 13th International Symposium on Practical Aspects of Declarative Languages*, 2011.

[33] K. Taura, K. Tabata, and A. Yonezawa. StackThreads/MP: integrating futures into calling standards. In *PPoPP '99:*

*Proc. 7th Symposium on Principles and Practice of Parallel Programming*, pages 60–71, 1999.

[34] C. Thacker. *Beehive: A many-core computer for FPGAs (v5)*. MSR Silicon Valley, January 2010. `http://projects.csail.mit.edu/beehive/BeehiveV5.pdf`.

[35] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS '03: Proc. 9th Conference on Hot Topics in Operating Systems*, 2003.

[36] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Proc. 19th Symposium on Operating Systems Principles*, pages 268–281, 2003.

[37] J. Vouillon. Lwt: a cooperative thread library. In *ML '08: Proc. 2008 Workshop on ML*, pages 3–12, 2008.

[38] D. Wentzlaff, C. Gruenwald III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An operating system for multicore and clouds: Mechanisms and implementation. In *SOCC '10: Proc. 2010 Symposium on Cloud Computing*, pages 3–14, June 2010.

[39] P. Winterbottom. Alef language reference manual. Technical report, Bell Labs, 1995.

[40] L. Ziarek, K. C. Sivaramakrishnan, and S. Jagannathan. Composable asynchronous events. In *PLDI '11: Proc. 2011 Conference on Programming Language Design and Implementation*, pages 628–639, June 2011.