Microsoft Research

AC: COMPOSABLE ASYNCHRONOUS IO FOR NATIVE LANGUAGES

Tim Harris, Martín Abadi, Rebecca Isaacs & Ross McIlroy

Microsoft

Synchronous IO in the Windows API

Read the contents of "h", and compute a result

```
BOOL ProcessFile(HANDLE h, int *outp
  BOOL ok = TRUE;
  DWORD size = FileSize(h);
  LPVOID buf = malloc(size);
  for (int i = 0; ok && i<(size/1024); i++) {
    ok &= ReadFile(h, buf+(i*1024), 1024, NULL, NULL);
  }
  if (ok) {
    *output = ...
  }
  free(buf);
  return ok;
}
```

It exposes reads one by one to the IO system: no pipelining, poor performance

Implem
"Read

If we want to process multiple files we must either (i) handle them serially, (ii) introduce threading

Asynchronous IO

```
static OVERLAPPED o[PD];

BOOL ProcessFile(HANDLE h, int *output) {
  DWORD size = FileSize(h);
  LPVOID buf = malloc(size);
  HANDLE cp = CreateIoCompletionPort(h, NULL, 0, 0);
  BOOL ok = TRUE;
  DWORD idx;
  LPOVERLAPPED *op;
  for (int i = 0; i<PD; i++) {
    PostQueuedCompletionStatus(cp, 0, i, &o[i]);
  }
  for (int i = 0; i<size/1024; i++) {
    DWORD idx;
    LPOVERLAPPED *op;
    GetQueuedCompletionStatus(cp, NULL, &idx, &op, INFINITE);
    op->offset = i*1024;
    ReadFile(h, buf+(i*1024), 1024, NULL, op);
  }
  for (int i = 0; i<PD; i++) {
    GetQueuedCompletionStatus(cp, NULL, &idx, &op, INFINITE)
  }
  if (ok) {
    *output = ...
  }
  CloseHandle(cp);
  free(buf);
  return ok;
}
```

Create completion port to synchronize with IOs

Main loop, issuing IO operations

Drain the completion port, waiting for all IOs to be done

...and asynchronous IO is not composable: "ProcessFile" is just a normal synchronous function

(And this is with a lot of error handling code omitted)

AC: asynchronous C

AC:
similar programming model to synchronous,
similar performance to async

Synchronous IO

Easy to use

Poor perf

Async IO

Difficult to use

1. Focus on performance and ability for use in low-level code (e.g., "monitor" process in Barrelfish research OS)

2. Small number of constructs: starting async work, synchronize with async work, stop waiting for async work

3. Enable higher level features to be built over this --- join patterns, futures, …

AC: almost the same as synchronous

```
BOOL ProcessFileAC(HANDLE h, int *output) {
  BOOL ok = TRUE;
  DWORD size = FileSize(h);
  LPVOID buf = malloc(size);
  do {
    for (int i = 0; ok && i<n; i++) {
      async {
        ok &= ReadFileAC(h, i*1024, buf+(i*1024), 1024);
    } }
  } finish;
  if (ok) {
    *output = …
  }
  free(buf);
  return ok;
}
```

1. Use the AC primitives for data access

2. Use "async" to identify work that can continue while waiting:
"{ async X } ; Y" – start "X" and, if it blocks, continue with "Y"

3. Add "finish" to identify synchronization:
"do { X } finish" – don't go past "finish" until all the async work in "X" is done

AC is composable

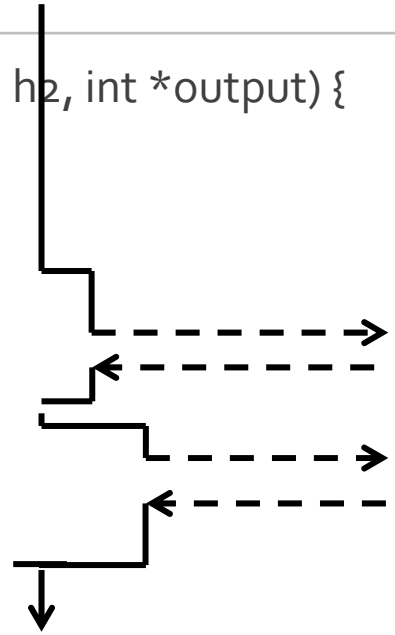
We can build layered abstractions using AC: user code can be called asynchronously, not just primitives:

```
BOOL ProcessTwoFilesAC(HANDLE h1, HANDLE h2, int *output) {
  int o1, o2; BOOL ok = TRUE;
  do {

    async { ok &= ProcessFileAC(h1, &o1); }


    async { ok &= ProcessFileAC(h2, &o2); }

  } finish;
  if (ok) *output = ...
  return ok;
}
```



Making multiple requests

```
// Counting semaphore to limit requests
SemAC_t   sem;
SemACInit(&sem, 100);

// Loop potentially starting lots of work
do {
  for (int i = 0; i < 1000000; i++) {

    P_SemAC(&sem);

    async {
      …                          // Do work

      V_SemAC(&sem);
    }
  }
} finish;
```

Cancellation

Suppose we want to add a timeout…

```
BOOL ok = TRUE;
with_timoeut: do {
  async { ok = ProcessFileAC(h, result); cancel with_timeout; }
  async { SleepAC(result); cancel with_timeout; }
} finish;
return ok;
}
```

2. It marks the named do..finish as cancelled, and "pokes" blocked operations dynamically within it

1. Suppose we execute this cancel statement

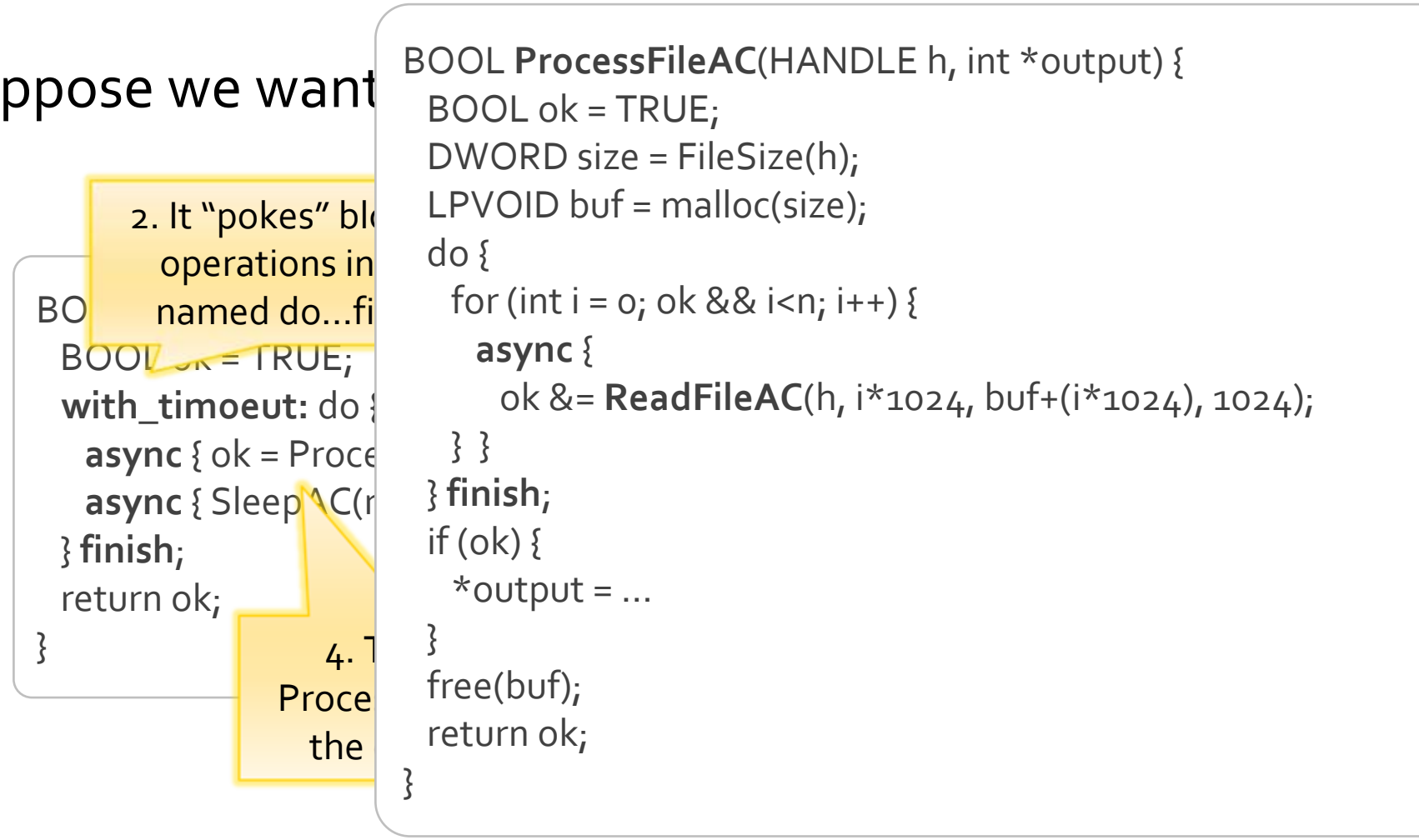
3. SleepAC is woken and terminates early with FALSE

4. This second cancellation has no effect

5. The "finish" works as before: wait until both asyncs are done

Cancellation

Suppose we want



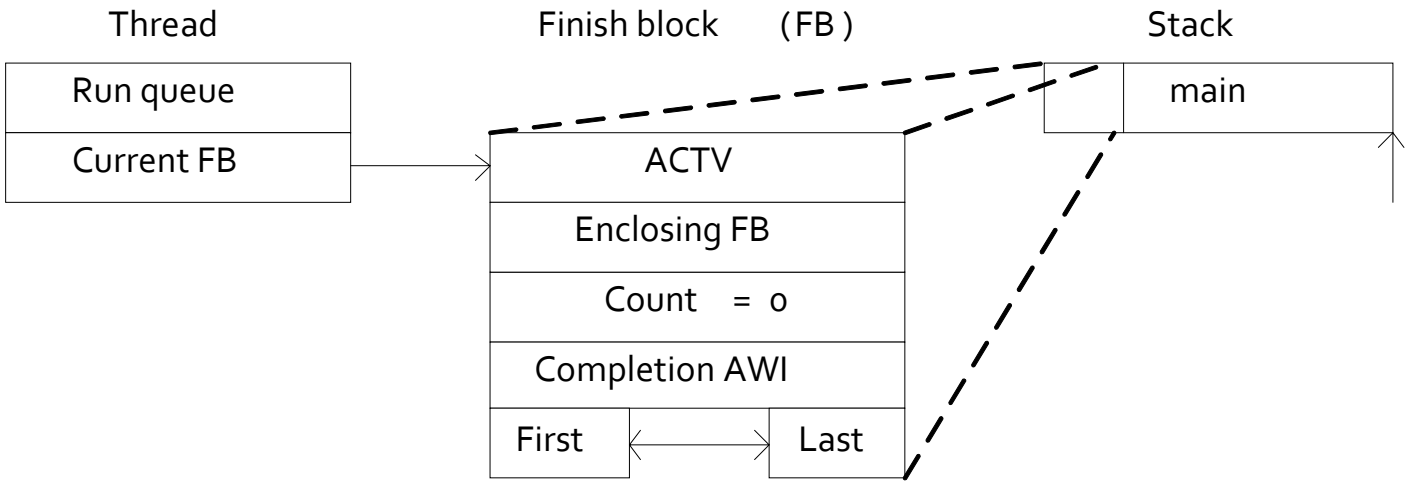
“Exact cancellation”

- A convention: provided by built-primitives, recommended for user code
- An *AC function either
  - Returns TRUE if it completes successfully without cancellation
  - Returns FALSE and undoes its partial side effects if it is cancelled

Easier said than done…

1. Remember cancellation is co-operative
2. *NAC non-cancellable primitives
3. For responsiveness, push clean-up work to a thread-pool

Implementation



Techniques:

-Per-thread run queue of atomic work items ("AWIs")

-Wait for next IO completion when AWI queue empty

-Book-keeping data all stack allocated

-Cactus stack implementation, creating a new stack lazily when blocking (rather than eagerly on entering an async)

Performance

	Function call latency (cycles)
Direct (normal function call)	8
async X() (X does not block)	12
async X() (X blocks)	1692

- "Do not fear async"
  - Think about correctness: if the callee doesn't block then perf is basically unchanged

Performance: Barrelfish ping-pong test

	Latency (cycles)
Using ring-buffer channel directly	931

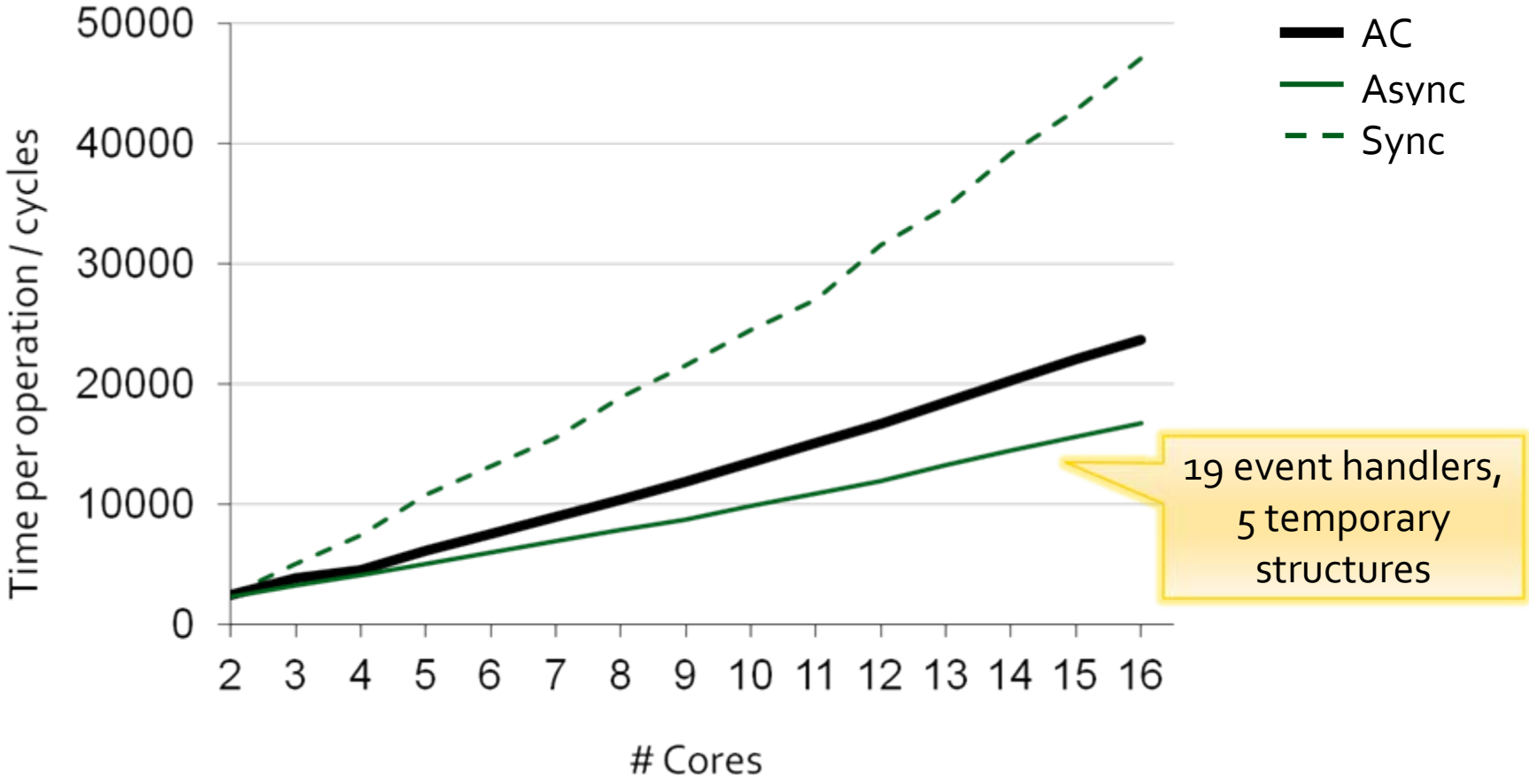
Ping-pong test

Minimum-sized messages

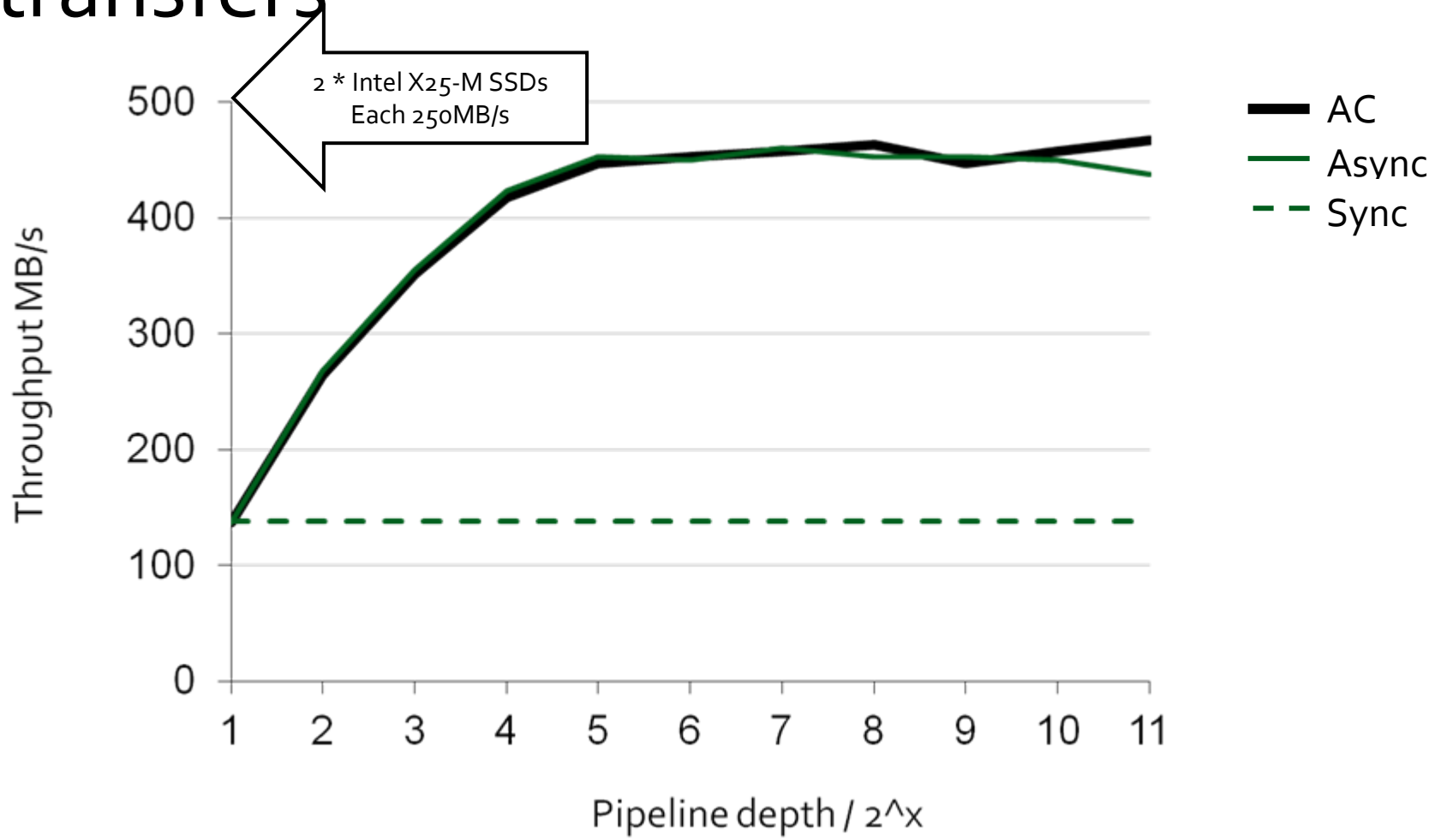
AMD 4 * 4-core machine

Using cores sharing L3 cache

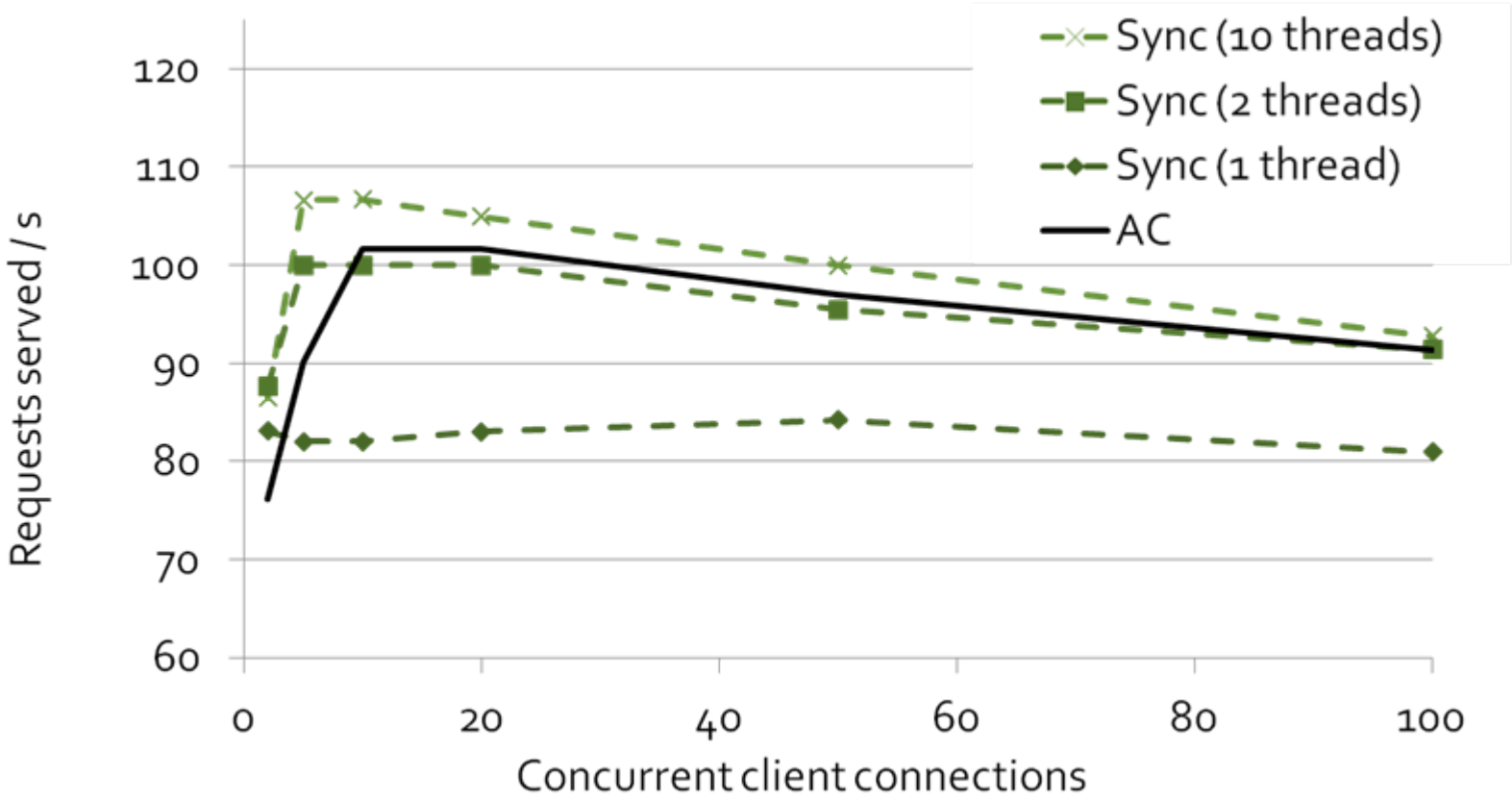
2-phase commit between cores



Software RAIDo model, 64KB transfers



Performance (embedded web server)



- Apache benchmark workload driver
- Running across WAN between CA and UK

Summary

- AC: composable asynchronous IO for native languages
  - Similar programming style to synchronous IO APIs
  - Similar performance for asynchronous IO APIs
- Also in the paper:
  - Operational semantics for core language
  - Integration of IO with runtime system
- Available as part of the Barrelfish research OS http://www.barrelfish.org

We're hiring: tharris@microsoft.com

Microsoft Research

www.research.microsoft.com