Extensions to Barrelfish Asynchronous C

Michael Quigley
michaelforrquigley@gmail.com

School of Computing, University of Utah

October 27, 2016

1 Abstract

The intent of the Microsoft Barrelfish Asynchronous C (AC) project was to provide fast and lightweight asynchronous functionality to the C language. We have added some useful extensions and bug fixes for the AC project. The goal is to apply these extensions to facilitate decomposition of a custom capability-based GNU/Linux kernel. We measure performance to ensure consistency with the original Barrelfish AC numbers as well as to compare asynchronous inter-process communication (IPC), which makes use of these extensions, with synchronous IPC. We also demonstrate some extensions which improve lightweight context switching performance.

2 Introduction

The goal of the Barrelfish AC project [1] was to add cooperative, asynchronous execution functionality to the Microsoft Barrelfish OS. The approach that the Barrelfish researchers took allowed for lightweight context switching between atomic work elements (AWEs). This paper discusses work done using the macro based implementation of AC.

Shown in Figure 1 is an example of an AC program. There are three key parts of this example:

1. DO_FINISH macro - All ASYNC macros must exist inside a DO_FINISH block.
2. ASYNC macro - Specifies that code contained within this macro may block execution.
3. THCYield - Function to be called if execution blocks within an ASYNC invocation.

```
static void do_something()
{
    THCYield();
    print("Got here too.\n");
}

static void do_something_else()
{
    print("Got here.\n");
}

...

DO_FINISH(
        {
            ASYNC(do_something(););
            ASYNC(do_something_else(););
        });
```

Figure 1: Simple Example

The output of the above example is:

```
Got here.
Got here too.
```

In this program, the function 'do_something' in the first block will yield to another block of execution before anything is printed out from the current block of execution. After this, execution will go into the second ASYNC block and execute 'do_something_else' which will invoke a print statement. After this, the end of the DO_FINISH block is reached, but since there is pending work, it will be executed and the print statement that was in the first ASYNC block will be invoked.

We have added extensions that are targeted toward an asynchronous IPC implementation. These extensions are discussed in section 5 and performance of the extensions is discussed in section 6.

3 Background

3.1 AWEs

AWEs represent an atomic block of work. They are responsible for saving the context of a particular execution path. The AWE struct stores the %esp, %ebp, and %eip needed for keeping track of context. It also stores the current finish block, the current per-thread state struct, its stack, and pointers to two other AWE structs that can be set to allow for insertion into a linked list.

3.2 DO_FINISH Macro

All invocations of ASYNC must happen directly in a DO_FINISH block or from code that is within a DO_FINISH block. At a high level, the DO_FINISH block just represents a block of code that contains asynchronous behavior. This block makes the guarantee that all asynchronous work will be finished by the end of the block. In other words, code execution will not continue past the DO_FINISH block until all pending work that was started in the DO_FINISH block is finished. The eager and lazy configuration have some minor implementation differences, but conceptually they are the same.

The DO_FINISH block starts by calling _thc_startfinishblock and passes in a reference to a new stack allocated finish block struct as a parameter. The finish block keeps track of information such as how many pending AWEs there are for a specific DO_FINISH. The execution will keep track of a linked list of finish blocks. This linked list encodes the nesting structure of invocations of DO_FINISH. An example of this is shown in Figure 2. In this example, the first DO_FINISH creates a finish block and adds it to the linked list at the bottom of the figure. Both the start_node and end_node for 'finish block 1' will have a reference to the finish block structure for 'finish block 1'. The second DO_FINISH will add two more nodes to the linked list for 'finish block 2'. These nodes are placed between the nodes for 'finish block 1' since the second DO_FINISH is nested inside the first.

Once _thc_startfinishblock has finished adding nodes to the linked list of finish blocks, the code inside the DO_FINISH is executed, and _thc_endfinishblock is called. The function _thc_endfinishblock will check to see if there are any pending AWEs for the current finish block. If there are, it will initialize the finish_awe as the continuation of _thc_endfinishblock, and call into the dispatch loop to dispatch pending work. Once execution reaches _thc_endfinishblock again via the finish_awe, it is assumed that there is no pending work for the current finish block, so the function will then remove the current finish block from the linked list.

The only difference between the eager and lazy configurations are, the lazy configuration must keep track of the stack pointer that is used upon invocation of the DO_FINISH block. This is because in the eager configuration, when _thc_endasync is called at the end of an ASYNC, execution can always free an AWE's stack; whereas in the lazy configuration, execution can only free the stack if it ensures ASYNC is executing on a different stack than the enclosing DO_FINISH is using.

3.3 ASYNC Macro

There are two configurations of the ASYNC macro. There are advantages and drawbacks to either approach. Figure 3 shows the required stack allocations for both approaches.

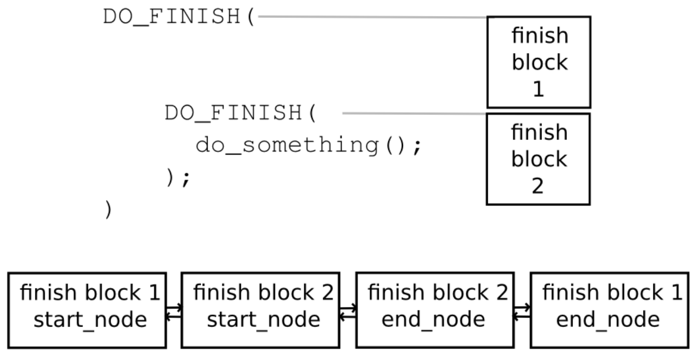

Figure 2: Finish Block Nesting Structure

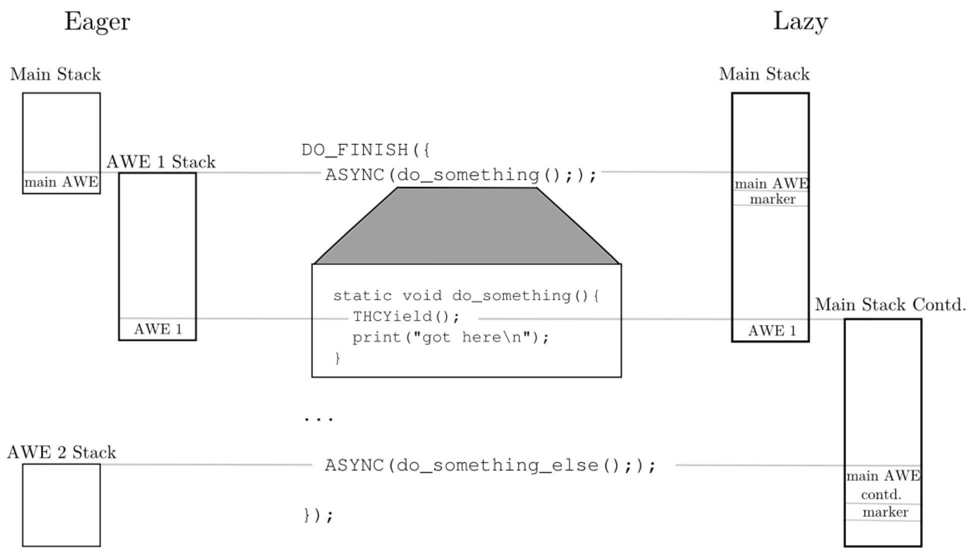

Figure 3: ASYNC Stack Allocations

3.3.1 Eager Configuration

The eager configuration involves two stack allocations for AWEs in the example in Figure 3. When execution arrives at a usage of ASYNC, a new stack is allocated for the code placed inside of the ASYNC macro. An AWE for the continuation of the code using the main stack is placed on the main stack. This continuation represents the code that would be executed after the ASYNC macro. The AWE struct ('main AWE' in this example) contains the stack pointer, frame pointer, and instruction pointer needed to continue execution after the first ASYNC. This AWE is then added to the front of the dispatch queue so that it can be executed later. This first ASYNC then executes the code placed inside of it on the newly allocated stack inside of a nested function. Since the function 'do_something' blocks (calls THCYield), another AWE is put on AWE one's stack that represents AWE one. AWE one is then added to the back of the dispatch queue and the thc_dispatch loop is entered. This loop will take 'main AWE' off of the dispatch queue and execute it. The 'main AWE' will continue to run on the main stack until the second ASYNC at which point a new stack for the second ASYNC will be created and a new 'main AWE' will be stored on the main stack and added to the front of the dispatch queue. The code for the second ASYNC will execute until the end without blocking since there is no THCYield in the function 'do_something_else'. At the end of this ASYNC, the function 'thc_endasync' is called which will free the stack for the second ASYNC, and call into the dispatch loop again. At this point, 'main AWE' will be taken off of the dispatch_queue. The AWE 'main AWE' will execute from the end of the second ASYNC to the end of the DO_FINISH. At the end of the DO_FINISH, thc_endfinishblock is called to see if there are any pending AWEs on the dispatch queue. If there are any pending AWEs, thc_endfinishblock will create another AWE (finish_awe) to save the current context. At this point, 'AWE 1' is the only AWE left in the queue, so this gets executed. Once it finishes, it calls thc_endasync which executes the finish_awe since there are no more pending AWEs.

One of the main advantages to the eager approach is that if the designer knows that the code will always block, or almost always block, the eager implementation runs more quickly. This is because in this situation, a stack allocation happens in both the eager and lazy configurations, but stack walking does not need to occur in the eager configuration. The main disadvantage of the eager configuration is that in a situation where a program using AC usually doesn't block, time would be wasted allocating unnecessary stacks.

3.3.2 Lazy Configuration

The lazy configuration involves one stack allocation for AWEs in the example in Figure 3. Some of the details about AWEs will be skipped over since much of it is the same as what is described in the eager configuration. The notable difference with the lazy configuration is that a new stack is allocated only when an AWE blocks execution.

When execution arrives at an ASYNC, a special marker is put in place of the return address to indicate that the current AWE is lazily allocated. In addition to the special marker, the status of this AWE is set to LAZY_AWE. The code placed inside the ASYNC is executed in a nested function. If an AWE blocks, the function 'check_for_lazy_awe' is called which walks up the stack looking for this marker to determine if a stack needs to be allocated. If it finds this marker in place of the return address, a stack is allocated for the continuation of the code that was originally using the main stack. In this example, the AWE representing this continuation is then scheduled after being popped off the dispatch queue. When execution arrives at the second ASYNC, the marker is once again put in place of the return address. Since this ASYNC does not block, no more stacks are allocated and the pending AWEs are executed after being taken off of the dispatch queue just like in the eager configuration.

The main advantage of the lazy configuration is, if an ASYNC block does not call THCYield, then no additional stack needs to be allocated. If all the calls to ASYNC never block, then there will only ever be one stack that is used. The disadvantage is that any time ASYNC does block, execution must walk up the stack to decide if a stack allocation is necessary. Execution walking up the stack incurs a little bit of additional overhead.

4 Yielding

At a high level, the purpose of yielding is for an AWE to save its context, suspend its execution and pass execution to another AWE. When an AWE invokes a yield function, this is referred to as blocking. There are two types of yields supported by Barrelfish. They are THCYield and THCYieldTo. The function THCYield will suspend the current AWE and execute other pending work if available. The function THCYieldTo will yield to a specific AWE. Both the eager and lazy configuration of the yield functions will put an AWE struct on the current stack that represents the continuation of the current AWE. After that, there are some minor differences with how the yield functions work for the two configurations.

4.1 Eager Configuration

4.1.1 THCYield

Inside of THCYield, the macro 'CALL_CONT' is passed a function pointer to the function 'thc_yield_with_cont'. This macro will put an AWE on the stack that represents the current continuation (what happens after the code currently executing), then call into an assembly function ('thc_callcont') to initialize this AWE. This function will set the AWE's %eip to the return address of THCYield, the AWE's %esp to the current %esp + 8, and the AWE's %ebp to the current %ebp. This function then calls 'thc_callcont_c' which will finish the initialization of this AWE by initializing its current finish block and per thread state struct. Since the currently executing context already has its own stack, this is all the initialization that needs to be done for the current AWE. This function then invokes the function pointer that was passed to it ('thc_yield_with_cont' in this case). The function 'thc_yield_with_cont' will call 'THCScheduleBack' and pass in the AWE used for the current continuation. The function 'THCScheduleBack' will add this AWE to the tail of the dispatch queue. After this, 'thc_dispatch' is called which directly executes an AWE that represents the 'thc_dispatch_loop' function. The function 'thc_dispatch_loop' will pop an AWE off of the head of the dispatch queue and execute it directly.

4.1.2 THCYieldTo

This process is very similar to THCYield. The macro 'CALL_CONT' is used here as well except the function 'thc_yieldto_with_cont' is used instead of 'thc_yield_with_cont'. The function 'thc_yieldto_with_cont' is similar to 'thc_yield_with_cont', but instead of calling 'thc_dispatch', it directly invokes the AWE that is passed in as an argument. All the initialization for the current AWE is the same between THCYield and THCYieldTo. Since this function avoids using the dispatch queue to find work, it does not call into the dispatch loop, although it does need to remove the yielded to AWE from the dispatch queue. There was a bug in the original Barrelfish AC implementation where this AWE was not being removed from the dispatch queue. This would result in page faults since after the AWE would execute and clean up its stack, it would still be in the dispatch queue. The dispatch loop would eventually pop it off the queue and try to execute it with an invalid stack.

4.2 Lazy Configuration

The lazy configuration only has a few small differences from the eager configuration for both THCYield and THCYield to. The main difference with the lazy configuration is that it must perform the walk up the stack to see if there is an AWE that needs a stack allocated for it. This check is performed inside of the function 'check_for_lazy_awe' which is called from the function 'thc_yield_with_cont' for THCYield or 'thc_yieldto_with_cont' for THCYieldTo. This function is passed in the AWE's %ebp (this was just set from 'CALL_CONT_LAZY'). At this point, it will use the current %ebp it has to get the previous %ebp value from the stack. It will loop until either an old %ebp value, or the return address that was placed on the stack is null. If it reaches this point, it is assumed that the code is not executing inside of another AWE. There is however an issue with this assumption in the Linux kernel (discussed in 5.5). While the stack walking is happening, if an old return address value equals a special marker, then execution knows there is a lazy AWE just above this location on the stack. If this function discovers a lazy AWE on the stack, it will call 'init_lazy_awe' to see if the status of the AWE is LAZY_AWE. If the status is LAZY_AWE, execution will allocate a stack for this AWE, and set its status to ALLOCATED_LAZY_STACK.

5 Extensions

Extensions to the Asynchronous C module primarily focus on making asynchronous message passing between isolated domains easier to manage.

5.1 AWE-Mapper

The AWE-mapper is responsible for associating an AWE with an id number. The mappings are stored using an awe_table structure. Each per-thread-state structure has its own awe_table for storing its own AWE mappings. The AWE-mapper is initialized with the function 'awe_mapper_init' and uninitialized with the function 'awe_mapper_uninit'. The function 'awe_mapper_create_id' is called to get an id number that is unique in the current thread. The function 'awe_mapper_remove_id' marks a particular id as available. The function 'awe_mapper_set_id' associates a particular id with an AWE pointer. The function 'awe_mapper_get_awe_ptr' returns the AWE pointer for a particular id.

Internally, the AWE-mapper is just an array. When a new id is requested, this is simply an index into the array. This index is found by starting at the index number directly after the last returned index number, and performing a linear scan until an available index number is found. This scan may wrap around if the end of the array is reached.

5.2 Updated Yields

The AC code initially came with two types of yields. The first one (THCYield) would just yield to any available work. The second one (THCYieldTo) would take as input a particular AWE to execute. Three new yield types were added to facilitate asynchronous message passing.

5.2.1 THCYieldAndSave

This function is the same as THCYield except that it associates the currently executing AWE with the provided id number. The id number is assumed to have been allocated using awe_mapper_create_id.

5.2.2 THCYieldToId

This function yields to the AWE corresponding to the provided id number.

5.2.3 THCYieldToIdAndSave

This function takes in an id number to associate with the currently executing AWE as well as yielding to an AWE that corresponds to another provided id number.

In addition to these three types of yields, code was added to remove a specific AWE from the dispatch queue. The new THCYieldTo* functions and the original THCYieldTo function make use of this code. As mentioned in section 4.1.2, without this code, resources in the AWEs would be cleaned up, but the AWE pointers would still be in the dispatch queue causing page faults later on.

5.2.4 No Dispatch Yields

Yield functions that avoid using the dispatch loop have been created that are similar to the above three yield functions. The functions are 'THCYieldAndSave_NoDispatch', 'THCYieldToId_NoDispatch', and 'THCYieldToIdAndSave_NoDispatch'. These functions work the same way as the above three functions except they assume all AWEs they deal with (both callee and caller AWE) are not and should not be in the dispatch queue. This means that these functions do not schedule AWEs back into the dispatch queue. This also means they do not have to be removed from the dispatch queue. These functions were made to improve performance in specific situations where a dispatch queue isn't needed. There is an example of the use of these functions in section 6.2.

5.3 IPC Dispatch Loop

The dispatch loop discussed in this section is not the dispatch loop that is currently in the AC code. Instead, this is a higher level loop designed to process messages and asynchronously dispatch functions. A general dispatch loop is shown below.

```
void dispatch(struct thc_channel_group* rx_group)
{
  DO_FINISH_(ipc_dispatch,{
    struct thc_channel_group_item* item;
    struct fipc_message* msg;

    while(true)
    {
     if(!thc_poll_recv_group(rx_group, &item, &cmsg))
      {
         if(item->dispatch_fn)
         {
           ASYNC_({
             item->dispatch_fn(item->channel, msg);
           },ipc_dispatch);
         }
      }
    }
  });
}
```

The dispatch loop is not expected to return and is intended to be run on a thread that is always waiting to receive messages and dispatch work based on the messages. If thc_poll_recv_group returns 0, then a message is made available in 'msg' and 'item' is set to the item that the message corresponds to. The dispatch function that corresponds to 'item' is then called in an ASYNC block so that it handles the message and performs more work that may yield.

5.4 Asynchronous IPC Functions

These functions are integrated with a message passing mechanism that uses a producer-consumer ring buffer to pass cache-aligned structures from one process to another. These cache-aligned structures contain a field to populate with arbitrary message data as well as a field for flags such as whether the message is a response or a request type message.

5.4.1 thc_ipc_recv_response

This function should be called within an ASYNC block to asynchronously receive a message response. This function polls on an IPC message channel checking for messages that are marked as response messages. If a message response is available and it is for the currently executing AWE, then the message is returned. Execution knows if a message is for the current AWE by comparing the id field in the message with the id that was passed into this function. If a message response is available and has a pending AWE waiting for it that is not the current AWE, execution associates the current AWE with an id, and yields to the AWE corresponding to the message by calling THCYieldToIdAndSave. Otherwise execution yields and associates the current execution context with an id number by calling THCYieldAndSave.

5.4.2 thc_ipc_poll_recv

This function is similar to thc_ipc_recv_response except that it will return both request type and response type messages and if there is no message present, it just returns -EWOULDBLOCK instead of yielding.

5.4.3 thc_ipc_poll_recv_group

This function calls thc_poll_recv on a list of channels. If a channel has a message available that does not correspond to a pending AWE, the message and the channel are returned in out parameters. If there is no message available in any of the channels, this function also returns -EWOULDBLOCK.

5.4.4 thc_ipc_send_request

This function takes in a message and a channel variable and marks the message as a request message, then sends it on the channel. In addition, a new id is obtained using the AWE mapper. This is is stored in the message as well as returned in the form of an out parameter.

5.4.5 thc_ipc_reply

This function takes a channel, an id, and a response message. It marks the message as a response type, sets the message's id to the id provided, and sends the message on the channel.

5.4.6 thc_ipc_call

This function takes a request message and a channel as parameters. It calls thc_ipc_send_request and passes in the request message and the channel. The id that is returned by thc_ipc_send_request is passed into thc_ipc_recv_response. When a response is obtained, the response message is returned as an out parameter. This function abstracts the logic of managing ids and yielding to specific AWEs away from the user.

5.5 Modified Return Address in KThreads

In the lazy configuration of AC when an AWE yields, it walks up the stack to check for a special marker in place of the return address to see if the AWE was lazily created. If it finds this marker, it can create a new stack for the current AWE. The stack-walking terminates when either a special marker is found in place of the return address, or when both the saved return address and frame pointer that are on the stack are NULL. In the first case, when executing in user space, this would indicate the bottom of the stack. We had an issue with this when using this with kernel threads. The issue was that, at the bottom of the kernel thread stack, the old saved frame pointer and return address are not guaranteed to be NULL. This usually results in a kernel oops since if the frame pointer is not NULL, it is dereferenced.

To mitigate this issue, we created a macro (LCD_MAIN) that sets the return address of a function to NULL for the duration of AC code execution, and restores the return address once this code has finished.

6 Experiments

Experiments have been performed to check for both performance consistency with the Barrelfish AC paper [1], and to test the extensions. All the experiments were performed on a four core Intel Xeon 5530 processor at 2.4GHz. The timing was performed using the processor's time stamp counter.

6.1 Dispatch Loop

The dispatch loop experiment was designed to test the process of efficiently sending and receiving messages from multiple IPC channels on the same core. The setup is shown in Figure 4. In this figure, there are two types of messages that 'core 1' sends. The first type of message contains a number and is intended to reach 'core 2', at which point the number is incremented by one, and sent back to 'core 1'. The second type of message also contains a number that is sent to 'core 2'. When it is received by 'core 2', it is passed along to 'core 3'. This core will increment the number in the message by 10, then sleep for 10 milliseconds before sending the message to 'core 2'. Once 'core 2' receives the message, it will send it back to 'core 1'. Both 'core 2' and 'core 3' make use of the IPC dispatch loop described in section 5.3.

The experiment executes 60,000 iterations of a loop on 'core 1'. This loop sends two messages of the first type per iteration, and every ten iterations, a message of the second type is also sent. A thc_ipc_recv_response is placed after each send, and each transaction (send/recv) is placed inside a separate ASYNC. The expected result is that since the message of type two will cause 'core 3' to sleep before it responds, both 'core 2' and 'core 1' which are waiting for a response for this transaction should still be able to dispatch more work. Specific timing results for specfic messages were not the intended take away from this example. Instead, we simply wanted to show that when asynchronous IPC is used with the IPC dispatch loop, more messages can be sent and

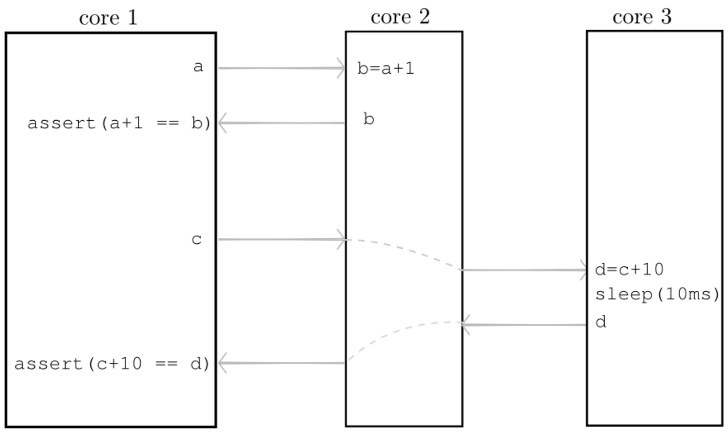

Figure 4: Dispatch Loop Experiment Setup

received using multiple IPC channels even when a response to a message blocks. The results, as expected were that hundreds of messages of type one are received on ‘core 1’ before any messages of type 2 are received.

6.2 Context Switching

This experiment was designed to test the time required to switch from executing in one AWE to executing in another. The experiment first creates an AWE and assigns an id to it. This AWE then yields to another AWE that creates an id for itself and yields back to the first AWE. Throughout this process, each AWE yields to the other 50,000 times, creating 100,000 context switches which are timed. Results were collected for the standard eager and lazy configuration (which saves all AWEs to the dispatch queue and removes them when necessary), as well as a test where the AWEs being yielded to were not added to the dispatch queue. The motivation for not adding these to the dispatch queue is that if we know they will be yielded to (which is true in the case of asynchronous IPC where an AWE is waiting for a specific message), then we don’t need them in the dispatch queue since they are still accessible via their id. The results of these experiments are shown in table 1.

We then performed another experiment to get bounds on the context switch time. This experiment involved taking 100,000 timing measurements (one timing measurement per context switch instead of one measurement over 100,000 context switches). The time stamp counter incurs 42 to 60 cycles of overhead, and 51 cycles on average. To account for this overhead, we approximate the maximum timing bound as:

$$max = maxMeasurement - maxTimerOverhead$$

and the minimum as:

$$min = minMeasurement - minTimerOverhead$$

As a sanity check, we also computed the average number of cycles per context switch in this experiment as well and made sure that after subtracting the timer overhead from the average, it wasn’t farther than ten cycles away from the average of the previous exper-

Table 1: Context Switch Times (in cycles)

Configuration	Average
Lazy	95
Lazy (no dispatch)	80
Eager	83
Eager (no dispatch)	71

iment. It should be noted that because of the limitations of measuring individual context switches accurately, the results in table 2 are just good approximations.

The max result for the lazy configuration is much larger than the max result for the eager configuration. This is because a new stack is allocated the first time an AWE calls THCYield in the lazy configuration, but in the eager configuration, the stack was already allocated upon creation of the AWE. When a new stack is requested, the function _thc_allocstack will see if there are any stacks that have previously been allocated and freed that can be re-used. This is done by keeping a linked list of freed stacks. If there are no stacks on this list, then kmalloc is invoked to create a stack. This max value for the lazy configuration represents the call to kmalloc. Upon consecutive iterations, the AWE will be able to reuse the stack that was allocated and freed in previous iterations.

Table 2: Context Switch Bounds (in cycles)

Configuration	Min	Max
Lazy	90	1712
Lazy (no dispatch)	79	2814
Eager	74	404
Eager (no dispatch)	54	316

6.3 AWE Creation

This experiment was designed to measure the time required to create an AWE. The time interval measured here is started just before a call to ASYNC, and ended just after the call. This is performed over 100,000 iterations. These results were obtained in a similar way as the results for the last context switching experiment. As a result, we again assumed that the minimum time was:

$$min = minMeasurement - minTimerOverhead$$

the maximum time was:

$$max = maxMeasurement - maxTimerOverhead$$

and the average time was:

$$avg = avgMeasurement - avgTimerOverhead$$

As a result, the true value of these numbers can vary by around 10 cycles.

The maximum number of cycles measured for the eager configuration is much larger than the maximum for the lazy configuration. This is because for the lazy configuration, when ASYNC is invoked, a new stack is not allocated unless the ASYNC block calls THCYield later on. In the eager configuration, a new stack is always allocated at the point of AWE creation.

Table 3: AWE Creation Times (in cycles)

Configuration	Min	Max	Median	Average
Lazy	4	96	8	7
Eager	20	1920	26	24

6.4 Message Passing

This experiment was designed to compare the throughput of synchronous versus asynchronous message passing between cores. There are two cores involved with this experiment. The first core creates a message to send to the second core. The second core takes a number that is inside the message it received and increments the number by one, then sends it back to the first core. Each timing measurement is taken starting from before the first core sends a message, and ends after this core receives a response for the message. This is performed over 60,000 iterations. There are three sets of throughput results gathered for this experiment as shown in Table 4. Both the lazy and eager configurations are compared with a synchronous implementation. The synchronous results indicate that about four more messages can be ping-ponged in the same amount of time, but again the advantage of async is that is these messages block, async can continue to dispatch more work while a synchronous implementation could not.

6.5 AWE Creation and Tear Down

This experiment was designed to measure the time it takes for an AWE to be created and torn down.

Table 4: Message Throughput

Configuration	Throughput (msgs / 10,000 cycles)
Lazy	15
Eager	17
Sync	20

The timing results were collected over 100,000 iterations for an empty function invocation, a function invocation inside of an ASYNC block, and a function invocation inside of an ASYNC block that yields and immediately gets yielded back to. In the last case, results were measured using both THCYield and then THCYieldTo. These results are shown in table 5.

Table 5: AWE Creation and Tear Down Time (cycles)

Experiment	Lazy	Eager
Empty invocation	6	6
Empty invocation in ASYNC	26	74
Blocking function (THCYield)	335	330
Blocking function (THCYieldTo)	326	260

7 Conclusion

We have demonstrated that AC can be extended for efficient asynchronous IPC and lightweight context switching. We have also shown that performance can be improved through use of the extensions we have added. We have started to integrate these extensions into a custom capability-based GNU/Linux kernel, but full integration is a topic of future work. The research presented in this thesis was supported by the National Science Foundation under Grants No. 1319076 and No. 1527526.

References

[1] Harris, T., Abadi, M., Isaacs, R., and McIlroy, R. Ac: composable asynchronous io for native languages. *ACM SIGPLAN Notices 46*, 10 (2011), 903–920.