Compiler Support for Lightweight Context Switching

STEPHEN DOLAN and SERVESH MURALIDHARAN, Trinity College Dublin
DAVID GREGG, Lero, Trinity College Dublin

We propose a new language-neutral primitive for the LLVM compiler, which provides efficient context switching and message passing between lightweight threads of control. The primitive, called SWAPSTACK, can be used by any language implementation based on LLVM to build higher-level language structures such as continuations, coroutines, and lightweight threads. As part of adding the primitives to LLVM, we have also added compiler support for passing parameters across context switches. Our modified LLVM compiler produces highly efficient code through a combination of exposing the context switching code to existing compiler optimizations, and adding novel compiler optimizations to further reduce the cost of context switches. To demonstrate the generality and efficiency of our primitives, we add one-shot continuations to C++, and provide a simple fiber library that allows millions of fibers to run on multiple cores, with a work-stealing scheduler and fast inter-fiber sychronization. We argue that compiler-supported lightweight context switching can be significantly faster than using a library to switch between contexts, and provide experimental evidence to support the position.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Coroutines*; D.4.1 [**Operating Systems**]: Process Management—*Concurrency*

General Terms: Languages, Performance, Experimentation

Additional Key Words and Phrases: Compiler, continuation, fiber, synchronization

ACM Reference Format:
Dolan, S., Muralidharan, S., and Gregg, D. 2013. Compiler support for lightweight context switching. ACM Trans. Architec. Code Optim. 9, 4, Article 36 (January 2013), 25 pages.
DOI = 10.1145/2400682.2400695 http://doi.acm.org/10.1145/2400682.2400695

1. MOTIVATION

Many popular programming models provide only a single thread of control in the program. This thread of control, which we refer to as a *context*, consists of the current execution point, as well as the state of local variables, return addresses, and other information stored on the execution stack. When parallel programs are written using these languages, a separate such context is created for each parallel thread, which can run independently.

One very useful feature of some programming systems is the ability to create multiple contexts in a program, and to switch between them within a single operating system thread. This feature can be used to implement important language abstractions such as coroutines [Conway 1963], closures [Sussman and Steele 1975], continuations

This research was supported, in part, by an IBM Faculty Award, in part by the School of Computer Science and Statistics, Trinity College Dublin, in part by Science Foundation Ireland grant 10/CE/I1855 and in part by IRCSET Enterprise Partnership Scheme in collaboration with IBM.
Authors' addresses: S. Dolan, S. Muralidharan (corresponding author), and D. Gregg, Computer Science Department, Trinity College Dublin, Ireland; email: muralis@tcd.ie.

DOI 10.1145/2400682.2400695 http://doi.acm.org/10.1145/2400682.2400695

[Abelson et al. 1998], Stackless Python tasklets, and Python generators [Schemenauer and Hetland 2001].

It can also be used to implement user-level threads, or fibers, which run within a single OS thread and are normally scheduled cooperatively. User-level threads normally explicitly yield control, rather than relying on OS calls and interrupts for scheduling and synchronization. As a result, fibers are significantly more lightweight than OS threads: they typically require less space, and allow much faster fiber switching, synchronization, and scheduling, making it feasible to run thousands or millions of fibers on just a few cores.

Implementing lightweight context switching efficiently and portably is not easy. Therefore, there are significant advantages to providing a small set of efficient primitives for creating and switching contexts that are generally useful. In this article we present such primitives for LLVM, a compiler and intermediate represenation that can be used to compile many different source languages, and can target several machine architectures. These primitives can be used by any language implementation based on LLVM to build higher-level language structures such as continuations and coroutines. We make the following contributions.

—We extend LLVM with lightweight language-neutral primitives for creating and switching contexts, and demonstrate their portability by implementing them for the `x86-64` and `PowerPC` architectures.
—We exploit existing and new compiler optimizations to cheapen context switching, and to reduce the amount of state stored in suspended contexts.
—We provide a mechanism that allows parameters to be passed in registers to suspended contexts as they are resumed, allowing contexts to efficiently message pass or yield values as they switch.
—To demonstrate the effectiveness of our primitives we present an extension to C++ that adds one-shot continuations to the language, and use this to implement generators, lightweight fibers, and an efficient parallelizing fiber library that can execute applications with millions on fibers across multiple cores.
—We show how to use our stack swapping primitives to create extremely efficient fiber implementations of communication and synchronization mechanisms such as mutexes, message passing channels, condition variables, and a work-stealing scheduler.

In the next section we outline some background and describe contexts and continuations. We present a form of continuation for LLVM in Section 3, which we then use to implement generators (Section 4) and extremely lightweight independent threads (Section 5). In Section 6, we see how the ability to easily pause and resume a thread of control makes it possible to distribute units of work across the processors of a multicore system. Results of our evaluation are presented in Section 7.

2. BACKGROUND

Language-level coroutines have been found useful at least as far back as Simula 67, and a form called *generators* has gained popularity in Python, C#, and other languages. These constructs ease the programming of cooperating algorithms, like iteration over and processing of a complex data structure's contents, or a lexer feeding tokens to a parser.

Similarly, lightweight threads are found in Erlang [Hedqvist 1998], Haskell [Li et al. 2007], and Go, where they are used to write more straightforward solutions to

concurrent problems like network servers or discrete event simulations, and to take advantage of multicore systems.

These structures require multiple contexts: separate storage for local variables, different program counters, and so on. A single stack is insufficient to represent these multiple contexts. Instead, we must allocate a separate stack for each new context. When performing a context switch, the current context is saved and a *continuation* is created, which represents the paused context. The continuation may later be *invoked*, resuming the paused context.

Context switching to a given continuation involves creating a new continuation to represent the current context, and restoring the target context from the given continuation.

Standard continuation interfaces present a problem for efficient implementations, so we use a restricted form known as the *one-shot* continuation [Bruggeman et al. 1996], which requires that each paused context is only resumed once. This is a powerful enough control structure to implement generators, coroutines, and lightweight threads, but it can be implemented extremely efficiently.

3. EXTENDING LLVM WITH CONTEXT SWITCHING

We extended LLVM with two new primitives and related compiler support. First, we added a new type of function call named a SWAPSTACK call, which in addition to transferring control also swaps the current stack for another. SWAPSTACK calls invoke continuations rather than functions, implementing a form of context switch. During a SWAPSTACK call, a continuation is created to represent the current pausing context, and this new continuation is passed to the context being resumed.

Secondly, we added an intrinsic function NEWSTACK. It takes a region of memory and a function pointer, and sets up a continuation for that function using the provided memory as stack, and can therefore be used to spawn new contexts.

3.1. SWAPSTACK Calling Convention

We implemented SWAPSTACK as a new calling convention. When a SWAPSTACK-type call is executed, any live registers are spilled and a continuation is created containing the stack and frame pointers and the *resume address*: the point in the code where execution will continue when the context is resumed. The new stack and frame pointers are restored from the continuation being invoked, and execution proceeds from that continuation's resume address.

Implementing SWAPSTACK as a new type of function call with its own calling convention has several advantages. First, we can use the existing LLVM mechanism for saving registers at call sites. The calling convention for SWAPSTACK has no callee-save registers, so all live registers are saved by the LLVM compiler. In comparison to other context switching mechanisms which save all registers regardless of whether they are live, this results in a significant saving of both time and space when context switching.

Secondly, we can use the existing call/return mechanisms in LLVM to easily implement context switching. SWAPSTACK is implemented as a function call which suspends the current context and transfers control to a different one. When the original context is next invoked, control returns to the point just after the SWAPSTACK call. Therefore, it can be implemented in the compiler using essentially the same mechanism as function return. To the optimizer, SWAPSTACK looks like any other function call (except that all registers are caller-save) and LLVM's existing optimizations work without modification.

Thirdly, implementing SWAPSTACK using the function calling mechanism allows an arbitrary number of arguments to be passed from the invoking context to the invoked context. These parameters are passed in registers, making communication at the time of the context switch extremely efficient. To the invoking context, these arguments are

parameters to the SWAPSTACK call. To the invoked context, the arguments appear to be return values from the previous call to SWAPSTACK. Recall that a context suspends itself and switches to another by calling SWAPSTACK, and when control eventually resumes, the call to SWAPSTACK appears to return.

3.2. Implementing SWAPSTACK

The data structure to store a suspended context is identified by its stack pointer. On top of this stack, a resume address is stored. Thus, in order to perform a SWAPSTACK call the following sequence of operations must happen.

```
    push address(ResumePoint)
    R2 <- SP
    SP <- Target_Stack
    pop R1
    jmp R1
ResumePoint:
```

The thread performing the context switch pushes a return address and moves the stack pointer to a particular register (R2). It then updates the stack pointer to point at the target context's stack, pops a return address, and continues from there. The target will see a "return" from its SWAPSTACK call, and a pointer to the stack of the suspended context will be stored in R2.

The context switch operation does not need registers other than one to store the suspended context (R2), and one scratch register (R1). Since the others are not modified, they may be used to pass values across the context switch. So, the calling convention states that arguments to the SWAPSTACK call are passed in registers, and return values are returned in those same registers, so that no additional work need be done to pass values.

There is usually zero additional cost to pass arguments across a SWAPSTACK call: since SWAPSTACK is implemented in the compiler before register allocation it can ensure that the result of a computation which is to be passed can simply be computed in the correct register for it to be passed as a parameter, and often no instruction need be added to copy the value to the right location.

The stack frame layout for a SWAPSTACK call is somewhat different to a standard call. For a normal call on most architectures, the arguments are passed on the stack or in registers, and the return value is returned in a register. If the return value is too large to fit in the available register space, then it is the caller's responsibility to allocate sufficient extra stack space to hold the value. A pointer to this buffer (marked `retarea` in Figure 1) is passed in as a hidden extra argument, which the callee uses to store the return values before passing control back.

The registers used to pass call and return values are often different. However, for SWAPSTACK calls the conventions for calls and returns must be symmetrical, since a SWAPSTACK call passes control directly to a SWAPSTACK return. Hence, we design the calling convention to specify exactly the same sequence of registers for arguments passed and values returned.

The situation is more complex when the argument does not fit in registers. In this situation, we allocate space for the extra returns on the stack (similar to most platform's C ABI). We then store a pointer to this space just below the return address when switching off the stack. When this thread is resumed, the resuming thread locates this pointer and initializes the return value space before passing control.

Allocating this space before the context pauses requires that its size is known, which precludes SWAPSTACK calls with a variable number of arguments. We do not consider this a significant limitation.

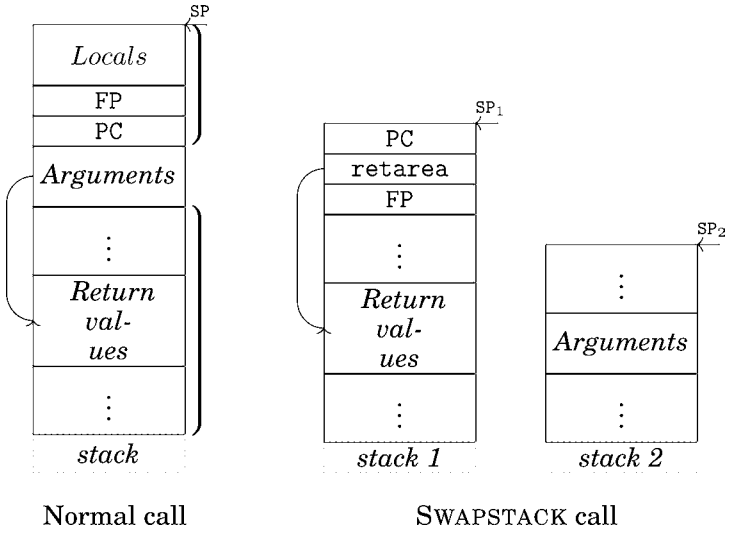

Fig. 1. Stack layout for normal and SWAPSTACK calls, just after the call instruction has passed control to the callee or the context switch has happened.

Some architectures require that the called function save a "frame pointer" (FP in Figure 1) to the stack, which must be restored when the function returns. For instance, on x86 this is the EBP register (RBP on 64-bit). When we do a context switch using SWAPSTACK, we cannot rely on the invoked context to correctly restore this pointer, because the invoked context will be using a completely different stack. So, we must explicitly save this register on the current stack. We save it below the program counter and `retarea` pointer, so that code compiled with and without the frame pointer is binary compatible.

Note that, unlike other forms of context switching, we add no code to explicitly save any data registers (i.e., other than stack pointer and program counter). Instead, we specify as part of the calling convention that no registers are preserved across a SWAPSTACK call, and ensure that the register allocator spills and restores any live registers. Again, we benefit from doing this as a compiler extension rather than a library, since the compiler is then able to make smarter register allocation decisions (see Section 3.4).

For instance, if a thread does a lot of synchronization or communication with other threads inside a tight loop, the compiler will generally be able to hoist the saves and restores of the callee-saved registers out of this loop. Also, if a piece of complex code precedes a SWAPSTACK, the compiler will realize that the callee-saved registers are about to be clobbered by the SWAPSTACK call and are fair game for allocation in the complex code, without needing restoration afterwards.

Another advantage of being internal to the compiler is that we can expose the return-address mechanism to the resume-address logic to LLVM's branch folding optimizer. Suppose we have a piece of code like the following.

```
if (condition){
        code;
        context switch to ctx1;
}
else{
        code;
        context switch to ctx2; }
```

Normally, if the `ctx1` path is taken, the first instruction executed after this thread is resumed is a branch over the "else" part. However, after the optimizer runs, the resume address actually points to the first instruction following the if statement, eliminating the branch.

3.3. Creating New Contexts

To create a new context, we provide a new LLVM intrinsic NEWSTACK. This intrinsic is passed a region of memory and a function pointer, and it creates a partial stackframe at the top of the region. This partial stackframe is formatted as a paused context, with its resume address pointing at the start of the function's code. A SWAPSTACK operation may then be performed using this new stack as a target. Control will be passed to the new context's function. The called function receives arguments passed across the context switch (including the parent's continuation) as formal parameters.

Spawning a new context and immediately switching into it is quite common, and as such it would be nice to further reduce the cost of the switch. In particular, if a context has just been created by passing NEWSTACK a constant function pointer, then the address to which control will be passed is known and the usual indirect branch performed by a SWAPSTACK is suboptimal.

Rather than adding new logic to detect such cases, we again use existing optimizations in LLVM. We lower the SWAPSTACK operation to a load of the return address followed by the actual context switch and defer later processing after various optimization passes such as alias analysis and store-load forwarding have run. Then, when emitting code for the context switch, we check whether the address being jumped to is, by this stage, a compile-time constant. If so, we use a direct rather than indirect branch.

This catches more cases than a simple pattern match would, including some which are not even obvious from the source-code. Suppose the program contains a function which takes a function pointer as an argument, creates a context to run this function, and switches into it. The target address of this context switch is not, in general, known at compile time. However, if this function is inlined into a callsite where it was passed a constant parameter, then the target address becomes known for the inlined copy and a direct jump may be emitted.

This optimization will not work with dynamically linked function calls. If a function does a context switch in a callback from a different shared object, we would have to save and restore all of the callee-saves registers, the reason being, we will not be able to prove whether the function in the other shared object was using these registers or not.

3.4. Optimizing Context Size

Heavyweight approaches to context switching, such as those found in operating systems, and some implementations of C's `setjmp` and `longjmp` save and restore all registers when switching context. This can be quite a lot of data (for instance, 1024 bytes on 64-bit PowerPC with AltiVec), which has both a time cost and a space cost. The space cost can make it difficult to spawn very large numbers of contexts, because each suspended context must store a full copy of all registers. For example, in certain types of server applications, we may want to support many connections with a separate context for each connection. In this situation, a minimum stack size of even a few kilobytes may be a barrier to maintaining very large numbers of threads.

Because our context switches are known to the compiler before register allocation happens, we can reduce the space requirement dramatically. The compiler knows which values are live across a context switch, so not all of the registers need be saved. In fact, our context switch operation only preserves the stack pointer and program counter,

and leaves it up to the register allocator to ensure that no other registers are live at the time of the switch.

It does this by the standard mechanisms of inserting spill code. This means that we can benefit from all of the standard optimizations that have been applied to LLVM's register allocator: for instance, if a constant is loaded and the value is live in the current function across a SWAPSTACK invocation, the register allocator can repeat the load instead of spilling and restoring the value.

As well as reducing space usage, this also reduces time usage: the register allocator will optimize the placement of spill code by, say, hoisting it out of loops.

Saving only the live variables during a context switch is not a new idea. Some existing approaches [Grunwald and Neves 1996; Zhou and Petrov 2006; Jääskelainen et al. 2008] analyze the assembly code to identify live registers which must be spilled. Our approach differs in that we expose the code to save and restore registers to the full range of compiler optimizations, which allows it to reduce the amount of context to be saved. For instance, consider the following function.

```
void foo(int x){
  while (somecondition){
    int local = large_const;
    global_var = local;
    yield; // performs a context switch
    global_var = local;
  }
  other_function(x);
}
```

The compiler normally allocates `x` and `local` in registers. Thus, library implementations of context switching need to save and restore all registers, in case they contain a live value.

The register allocator in our compiler, using standard rematerialization techniques, notices that the value of `local` is always equal to that of `large_const`, and the cost of spilling and restoring `local` is less than the cost of reloading `large_const`. The variable `x` must be spilled, but the spill code placement logic in our compiler hoists the spill and restore out of the loop. Thus, this function only performs a single spill and a single restore, no matter how many times the loop iterates.

3.5. Preserving Callee-Save Registers

Using only function-local information, we soon hit a lower bound on the amount of context to be spilled: those machine registers which are defined by the ABI to be "callee-save" must be preserved across calls. In the case of a SWAPSTACK, we cannot guarantee that the target context will preserve the values in the registers, so all callee-save registers must be saved. Even if some of them are never live (on machines with large register files, the callee-save registers are used by few functions), we still spend time and space saving and restoring them.

If a supposedly callee-save register is known to be dead, its value need not be preserved during the context switch. Grunwald and Neves [1996] use a whole-program live register analysis to save and restore only the necessary registers. We use a somewhat different approach, due in part to limitations of LLVM's internal architecture which does not support inter-procedural post-register allocation optimization passes.

We extend the LLVM intermediate representation to allow a function to be marked `nocalleesave`, which indicates that it may not preserve the values of the standard callee-save registers. Our compiler marks all functions containing a context switch with this annotation. And all functions which call a `nocalleesave` function on all

possible execution paths are also marked `nocalleesave` by the compiler. This requires an interprocedural data-flow analysis phase, which can run cross-module at link time.

In many cases, this means that the `nocalleesave` attribute percolates up from the low-level functions which may context switch up to a high-level function containing a fiber's main loop. If this happens, the callee-save registers are never saved, so the cost of preserving them across a context switch is never paid.

If it is necessary to preserve a value across a call to a function marked `nocalleesave`, the register allocator instead resorts to other methods (moving the use across the call if possible, rematerialising the value, or simply spilling and restoring). Our compiler automatically applies the `nocalleesave` attribute to those functions which would otherwise have to spill and restore all the callee-save registers.

4. C++ GENERATORS VIA CONTINUATIONS

As a simple example of the context switching features we have added to LLVM, we demonstrate the implementation of generators in C++ in terms of our new primitives. Generators (or iterators, in some languages) are functions which may "yield" results back to their caller. After yielding, the generator may be resumed by the caller and will run until it produces another value via "yield".

Generators are a standard part of several modern OO languages such as C# and Python, and allow iteration over complex data structures to be implemented without inversion of control and complex state-machine logic.

We extended the Clang compiler front-end to allow us to apply the SWAPSTACK annotation to C and C++ functions. We can then perform context switching and continuation invocation by calling a function pointer with this annotation.

Suppose we are to implement a generator which yields 10 successive even numbers starting with 42. For each number, it yields control to its caller via a SWAPSTACK call. The object being called is a continuation, passed to the generator by its caller. The SWAPSTACK call invokes the continuation, passing it the number being yielded and transferring control back to the caller. When the generator is resumed, the SWAPSTACK call returns a new continuation which can be used for the next iteration of the loop.

The type of this yield function pointer is as follows.

```
typedef swapstack void* (*yieldfn)(int);
```

It takes an integer to be yielded to the caller, and returns a continuation represented as a `void*`, which must be cast to type `yieldfn` before being invoked.

Thus, our even-number generator function is as following.

```
swapstack void evens(yieldfn yield){
  for (int i = 0; i < 10; i++){
    yield = (yieldfn)yield(42 + i*2);
  }
}
```

The `evens` function is also annotated with the `swapstack` keyword to indicate that it is not directly called, but is invoked on a new, separate stack.

The other side of the operation, that is, the function which invokes the generator and iterates over the yielded values, must allocate and initialize some stack space for the generator. The initialization is done via a call to NEWSTACK, which is exported to C++ as the new compiler-intrinsic `__builtin_newstack`. The actual SWAPSTACK call into the generator results in two values: the continuation of the generator, to be invoked when the next value is required, and the actual value itself. As a C++ function may not return multiple values, we pack the values into a structure. The function is as following.

```
typedef struct {
        void* cont;
        int val;
} ret;
typedef swapstack ret (*genfn)();
void uses_generator(){
  char stk[4096];
  genfn generator = (genfn)
    __builtin_newstack(stk,sizeof(stk),evens);
  for (int i = 0; i < 10; i++){
    ret r = gen();
    gen = (genfn)r.cont;
    cout << r.val << '\n';
  }
}
```

We have developed a C++ template library which abstracts away some of these details, and provides static typechecking of the values being yielded. Using it, the previous example becomes as following.

```
void evens(generator<int>* g){
  for (int i = 0; i < 10; i++)
    g->yield(42 + i*2);
}
void uses_generator(){
  generator<int> g(even_generator2);
  for (int i = 0; i < 10; i++)
    cout << g.next() << '\n';
}
```

5. A FAST LIBRARY FOR FIBERS

We extended our C++ template library to provide *fibers*, lightweight threads of control which can be paused and resumed. A fiber is a context which runs until it blocks, by terminating, trying to acquire a held lock, reading from an empty channel with no current writer, explicitly yielded, or another blocking operation.

We wrote a simple scheduler that maintains the set of currently runnable fibers as a queue of continuations (the *run-queue*). When a fiber blocks, a new fiber is removed from this queue and resumed, allowing it to run for some time. When the fiber pauses, its continuation is placed back on the run-queue so that it will be scheduled again.

Synchronization objects like mutexes and channels are implemented with queues of continuations to keep track of blocked fibers. For instance, a contented mutex contains a queue of continuations of fibers which are trying to acquire the mutex. One of these will be selected for resumption when the mutex is unlocked.

A new fiber can be spawned by allocating some memory for a stack and using NEWSTACK to initialize it as a continuation. This continuation can then be enqueued on the run-queue, from where it will be scheduled when CPU time becomes available.

5.1. Mutexes and Other Synchronization Objects

A mutex consists of a flag `is_locked`, and an initially empty queue of continuations `waitq`. To lock or unlock a mutex, a fiber attempts to set or clear the `is_locked` flag. If it is found to be true when attempting to lock, the mutex is contended. The current fiber

pauses and places its continuation on `waitq`. If `waitq` is nonempty during an unlock operation, a fiber is removed from that queue and woken.

There are two ways to wake a fiber. The simplest is to add it to the run-queue, so that it is eligible for CPU time. Alternatively, the running fiber could directly context switch to the fiber to be woken and add itself to the run-queue. There is a trade-off: the former avoids the context switch and profits from a hot instruction cache by continuing the same path through the code. The latter makes starvation is less common, and profits from a hot data-cache containing the data protected by the mutex.

Having control of synchronization, context switching, and scheduling in userspace allows the programmer or compiler to decide such policy per application or even per mutex, unlike when the logic must be embedded into an operating system kernel.

Using a standard thread library, a blocking operation requires two context switches: one from the application into the kernel, which decides which thread to schedule next, and one from the kernel into the elected thread. Our library requires only one: the blocking fiber selects the next fiber to run via a simple subroutine call, and then context switches directly into it.

Many other standard synchronization objects can be implemented using this scheme of a queue of paused fibers and a small number of states. For instance, an Occam-style [Daniel 1995] blocking channel is implemented using three states: there may be no outstanding requests, readers waiting for writers, or writers waiting for readers. `read` and `write` can then be implemented by checking the state and either waking a paused fiber to finish its request, or pausing and adding a continuation to the queue. Using a SWAPSTACK call for this means that a message may be passed from fiber to fiber while remaining in registers.

Section 6 describes how this scheme may be modified to work in a multicore environment, with fibers executing in parallel on distinct hardware threads.

5.2. Fairness of Wait-Queues

Wait-queues like the queue of fibers waiting for a mutex can be implemented as a simple FIFO queue. This provides a strong fairness guarantee: if several fibers try to lock the same mutex, they will receive the lock in the same sequence that the lock attempts occurred. While improving the latency of blocking operations, this often reduces throughput. The last fiber to pause is the one most likely to have its working set still in cache, so placing it on the end of the queue will increase the cache miss rate. The cache miss rate decreases if we adopt a less fair policy and instead wake up the fiber that blocked most recently.

This trade-off is particularly important for the run-queue. Some common patterns involve two fibers continually context switching between each other, for instance, in a producer-consumer relation like a lexer feeding tokens to a parser. With a FIFO run-queue, the lexer will produce a token, switch to the parser which will consume the token, and then run every runnable fiber in the system once before returning to the lexer. This leads close to worst-case cache usage.

A strictly LIFO policy is even worse: the lexer and parser would run efficiently, but starve all other fibers in the system. Instead we use a hybrid approach: the first few tasks added to the run-queue will be added to its head in a LIFO fashion. After this, future add operations will add to the tail, until the run-queue has cycled completely (every fiber initially on the queue has run). Then, the LIFO count will be reset and the queue will switch back to LIFO mode for a while. This change yielded improvements of 20% on some benchmarks, while affecting others very little.

Again, being able to decide these policies in userspace makes optimization significantly easier. Applications can use a scheduling policy according to their throughput

versus latency requirements, rather than an OS kernel requiring a single algorithm which performs adequately under all possible workloads.

6. PARALLEL FIBERS

So far, we have described the fiber library in a single-processor context, with all of the fibers executing on a single hardware thread of control, switching between fibers using SWAPSTACK calls.

This system can be extended to support parallel execution of fibers on multicore systems. Instead of a single hardware thread, we use a set of *worker* threads (generally one per core), and load balance the set of fibers across them. Each worker is given its own run-queue, each containing a set of fibers competing for CPU time on that worker's core.

Our synchronization constructs can be extended to support concurrent execution, and we can perform the operations of the fiber library (spawning, awaiting, and synchronizing with fibers) with little to no actual hardware synchronization required.

Migrating a fiber from one worker to another is quite straightforward. A continuation created by a SWAPSTACK call may be invoked on any core, not just the one that created it. All of the information about the paused context is stored in shared memory, so if the continuation is passed to another hardware thread, it may be invoked there. After the SWAPSTACK call returns, the fiber may be running in the context of a different hardware thread.

This allows load balancing of fibers: if a worker's run-queue is short of work or entirely empty, it can "steal" more work by dequeueing waiters from other worker's run-queues and enqueuing them onto its own. Fibers thus moved will be resumed on a different hardware thread to the one on which they were paused, but this does not affect the fiber itself: its thread of control continues past the migration transparently.

6.1. Parallelizing Wait-Queues

In order to support work-stealing, our run-queues must use algorithms which are safe when accessed concurrently by multiple workers. For the wait-queues used in mutexes, channels, and the like, we must additionally avoid race conditions between enqueue/dequeue operations and updating the state variables.

For instance, if a mutex is found to be already locked during an acquire operation, the current fiber will pause and add itself to the mutex's wait-queue. Suppose the fiber owning the mutex concurrently unlocks it on a different worker. We must ensure that it is impossible for the unlocking fiber to perform its unlock operation after the locking fiber has noticed the mutex is locked, but before it has added itself to the wait-queue.

We solve these problems using the concurrent compare-and-swap-based algorithm described in the next section. To prevent races between the state updates and the queue/dequeue operation, we maintain the state as the low bits of the tail pointer of the queue. This means that the locking fiber can perform the state transition and the enqueue operation in a single atomic instruction, without the possibility of a race.

6.2. Queue Algorithm

High-performance concurrent queue data structures come in two flavors: those organized as linked lists [Michael and Scott 1996; Fober et al. 2002; Ladan-Mozes and Shavit 2004; Herlihy et al. 2003] and those organized as ring buffers [Tsigas and Zhang 2001].

Most efficient algorithms for managing concurrent queues as buffers use a fixed-size buffer, and algorithms for resizing the buffer while in use are often complicated. Furthermore, most of the queues in our code are used for synchronization objects like mutexes and condition variables, which only rarely contain any waiting fibers. Having

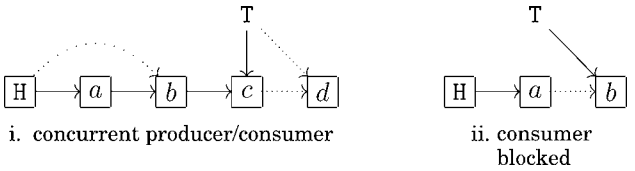

Fig. 2. Queue data structure examples.

to preallocate an amount of memory sufficient for the worst case would cause a huge memory burden.

In order to enqueue or dequeue from a buffer-style queue, three cache lines must be touched in addition to the data itself:

—the data itself;
—the list header, holding the current head and tail pointers;
—some part of the buffer, where a pointer to the data is inserted or removed.

Additionally, the fixed-size linked list nodes can easily be allocated intrusively on the stack of a fiber about to suspend, keeping the number of distinct cache lines touched to a minimum.

For a linked list where new nodes are allocated for each insertion, the number would be the same (the data, the list header, and the newly allocated node). However, for an "intrusive" linked list where the nodes are allocated as part of the data, we can reduce this number to only two cache lines: the list header and the data.

The data in question is the stack of a paused fiber. We can intrusively allocate a list node as part of this stack very simply: we declare it as a local variable in the function that suspends. So, the list node is located on the same cache line as the top of the suspended stack. As the fiber is suspending, this line is already in cache. When a fiber resumes, bringing this line into cache is necessary anyway, as the fiber is about to use it to access locals.

Thus, the number of distinct cache lines that must be accessed in order to suspend or resume a fiber is kept to the absolute minimum. One consequence of this is that a list node is deallocated as soon as it is removed from the list (as the fiber uses that stack memory for other purposes). This precludes using standard lock-free queues like M&S [Michael and Scott 1996] where deallocation lags the data by one node.

Instead, we implement our own low-lock queue algorithms, which provide weaker progress guarantees than M&S queues, but tend to be faster in practice as long as the number of concurrent accesses does not exceed the number of hardware threads (that is, they perform well in the face of hardware concurrency but badly in the face of multiprogramming).

The queue, shown in Figure 2, is a singly linked list of nodes. The head of the queue `H` is a sentinel node, and the queue's tail pointer `T` points to the most recently inserted node. When the queue is empty, `T` points to `H`. The enqueue and dequeue algorithms are shown in Figure 3. Both functions return 0 on failure, and it is assumed they are rerun until success.

When a node is to be inserted, the tail pointer is first swung to point to the new node and then the previous node's `next` pointer is updated. If a producer pauses during the short window between these operations (i.e. on line 5), a consumer trying to dequeue that node may block (in the loop on lines 10–11, 16, or 22) until the write to the `next` pointer has completed (see Figure 2, part ii). This mechanism sacrifices strict lock-freedom, but in practice increases performance (see experiments to follow), and avoids the risk of writes to nodes removed from the queue.

```
1  int enqueue(queue* Q, node* N){
2    N->next = 0;
3    node* tail = Q->tail;
4    if (!cas(&Q->tail, tail, N)) return 0;
5    tail->next = N;
6    return 1;
7  }
8  node* q_dequeue(){
9    node *first, *next;
10   do { first = Q->head.next }
11     while(!first && Q->tail != &Q->head);
12   if (Q->tail == &Q->head) return 0;
13   else if (Q->tail == first){
14     if (CAS(&Q->head.next, first, 0)){
15       if (!CAS(&Q->tail, first, &Q->head)){
16         do { next = first->next; } while(!next);
17         *head = next;
18       }
19       return first;
20     }
21   }else{
22     do { next = first->next; }
23       while (!next && Q->tail != first &&
24                       Q->head.next == first);
25     if (*tail != first && CAS(head, first, next))
26       return (node*)first;
27   }
28   return 0;
29 }
```

Fig. 3. Enqueue and dequeue algorithms.

Despite this blocking case, there is still a high degree of concurrency in this algorithm: producers may run concurrently, consumers may run concurrently, and producers and consumers may run concurrently if the queue is long enough.

When nodes are deallocated as soon as they are removed from the queue, hazards arise: how do we prevent access to deallocated memory? We prevent writes to deallocated memory by sacrificing strict progress guarantees and introducing blocking, as described above.

We do not need to prevent reads, as they have no side-effects. In fact, the only ill effect of a read to possibly deallocated memory is that the memory may no longer be mapped and cause a fault. To handle this case, we register a signal handler to detect and recover from such faults, which may potentially occur from the reads on lines 16 and 22. The performance overhead is zero if the fault does not happen, and up to 20,000 cycles if it does.[1] The penalty is unlikely: memory allocators prefer to reuse recently deallocated memory rather than return it immediately to the OS. In fact, we did not observe the penalty even once during all of the testing and benchmarking except when explicitly testing it.

The potential fault is an artifact of our optimized low-lock queue algorithm. We found our algorithm outperformed others, but runs the theoretical risk of reading from (but never writing to) memory that's already been freed by a different thread. Reading from deallocated memory would cause a spurious fault if the memory has been returned to the OS and unmapped. It is completely possible to use our system with a different queue algorithm, such as Michael and Scott's (against which we compared ours in Figure 4) or any other concurrent queue algorithm. In fact, when we did a proof-of-concept port

[1]Measured on Linux 2.6.32 on a Core 2 Quad, with cold caches.

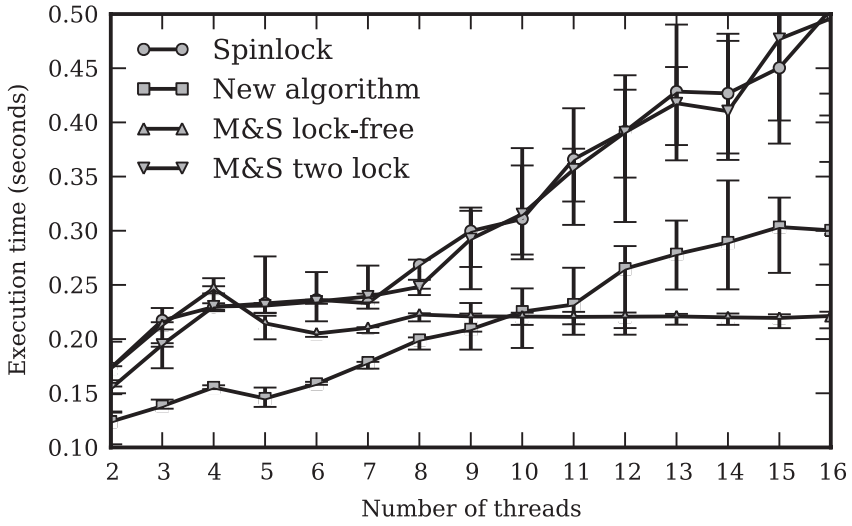

Fig. 4. Performance results from simple benchmarks of various queue algorithms.

of our system to the PowerPC architecture, we replaced our optimized queue with a simple spin-lock protected queue.

In some circumstances, the risk of a spurious fault may be eliminated entirely. For example, many high-performance memory allocators (e.g., TCMalloc [Google 2012]) never return memory to the OS, and so the case of deallocated memory becoming unmapped does not arise. If the system designer can guarantee that such a memory allocator is used, then the signal handler becomes unnecessary.

Finally, there are a number of instances in which production software have successfully relied upon signal handlers for optimizations. For example, the HotSpot implementation of the Java Virtual machine uses signal handlers to detect null pointer exceptions. It does unconditional loads of the address and relies on careful handling of the fault to determine if the address is actually pointing to a valid memory location. This is because in most scenarios the chance of the address being null is quite rare and by manipulating the signal handler it is possible to avoid any need for redundant check in cases where the address is in fact valid.

The system runs on small number of OS-level threads, known as workers. The workers are launched when the system starts and their number is a startup parameter and never changes. Typically, there would be the same number of workers as physical CPU cores.

A larger number of fibers are run on these worker threads. The number of fibers can grow and shrink dynamically as user-level threads are created and destroyed. Up to millions of fibers can be created, with the primary limiting factor being the amount of available RAM to hold their stack frames.

Figure 4 gives performance results from a simple benchmark of various queueing algorithms, on a machine with eight hardware threads. A number of threads concurrently attempt to enqueue and dequeue nodes from a shared queue. We measure the total time taken to pass a million nodes through the queue. In addition to having

[2]The "M&S lock-free" result in Figure 4 refers to the concurrent, nonblocking version of Michael and Scott's queue algorithm [1996]. The "M&S two lock" is the version of Michael and Scott's queue algorithm that uses two locks to control concurrent access to the queue.

```
if lock is owned by current worker:
  use biased fastpath
else if lock was never acquired:
  mark lock owned by current worker
  use biased fastpath
else if lock is shared:
  use shared slowpath
else if lock is biased towards another worker:
  migrate to other worker
  mark lock as shared
  use shared slowpath
```

Fig. 5. Biasing algorithm.

simpler memory management, our new algorithm outperforms M&S queues when the number of threads is not more than the number of hardware threads.

6.3. Migration and Biased Locking

As we have seen in the description of work-stealing, migrating a fiber from one worker to another is relatively inexpensive and easy. It turns out there is another use of this feature: the implementation of biased locks.

A mutex or other synchronization primitive may be "biased" towards one particular thread. This thread can then acquire, release, or otherwise manipulate the object without atomic synchronization instructions or memory fences. In cases where most of the traffic on a lock originates in one thread (which is quite common [Kawachiya et al. 2002]), this can result in a large speedup.

Biased locks must correctly handle the case where some other thread attempts to acquire the lock, even though the thread towards which the lock is biased may acquire it at any moment without synchronization.

Two solutions proposed for this are to have the non-owner thread interrupt and pause the owner thread [Kawachiya et al. 2002], or to have the owner thread poll for lock requests [Vasudevan et al. 2010]. The former requires an expensive kernel call and fixup mechanism to pause the owner and take the lock, while the latter can exhibit starvation if the owner does not poll regularly.

In our library, a biased lock is associated with a hardware worker thread rather than a single fiber. This provides the same synchronization guarantees, as each worker may only execute a single fiber at a time. More complex patterns of locality may be accelerated by biased locks: for instance, if there are two fibers in a producer-consumer relationship which happen to run on the same hardware worker, then a lock taken only by those threads can be biased towards that worker and accessed without atomics.

When a fiber on a non-owning worker tries to acquire the lock we simply migrate the offending fiber onto the owning worker. Rather than migrate the biased lock to the acquiring fiber, we migrate the fiber to the lock.

Fiber migration, while relatively inexpensive, is still more expensive than taking a shared lock. So, we unbias a lock if there is a conflict while trying to acquire it, as in Kawachiya et al. [2002].

Biased locks are implemented with a single extra word per lock, which indicates the state of the lock: it has never been taken (and will be biased towards the first worker that tries to acquire it), it is already biased towards a worker, or it is unbiased and requires normal synchronization. When a lock is to be taken or released, the biasing algorithm (Figure 5) runs to determine whether it is safe to use the synchronization-free fastpath.

Unfortunately, biasing interacts badly with our current simple algorithm for balancing the load between workers. The load balancing operates by stealing work from

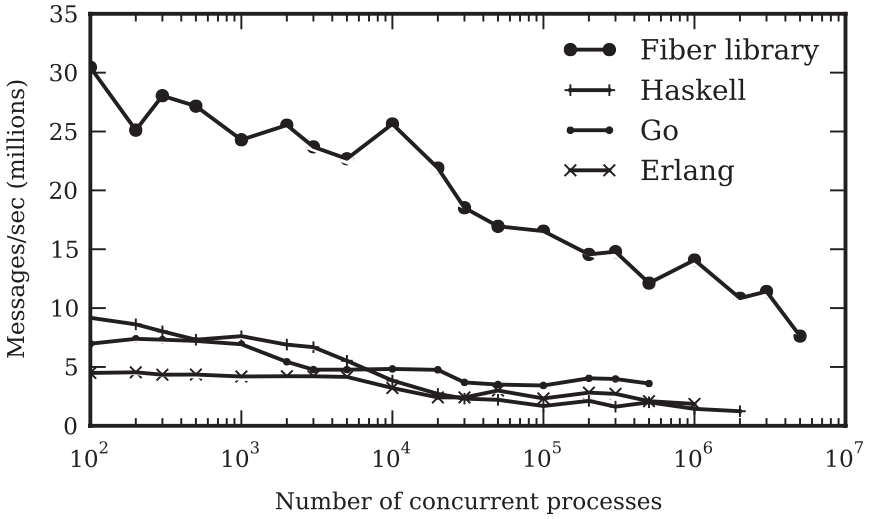

Fig. 6. Message passing performance of our fiber library compared to others.

a different worker (selected at random) whenever the current one runs low, making it too aggressive and causing lot more motion of work than desirable. It essentially works against biasing: the work-stealing algorithm aims to profit from load balancing by spreading work around, while the biasing algorithm aims to profit from a form of locality by keeping fibers that operate on the same data in the same hardware thread.

So, due to the relatively primitive nature of our load balancing and the interplay between these two algorithms, we do not see significant speedups in normal operation with biasing turned on.

In order to study the effect of biasing in isolation, we modified one of the benchmark programs (condvar) by adding explicit affinity hints rather than relying on our load balancing algorithm. With these affinity hints, biasing gave a significant performance improvement, bringing the benchmark's runtime down from 2.74 seconds to 1.96 (an improvement of nearly 30%).

7. EXPERIMENTAL EVALUATION

To evaluate our contributions, we benchmarked our extensions to LLVM and our fiber library by comparing them to existing systems.

7.1. Test Environment

The benchmarks were run on 64-bit `x86` architecture. The `x86` test machine was an Intel Sandy Bridge processor at 2.10 Ghz with 6MB of cache, 2GB of RAM, and 8 hardware threads.

Due to its RAM requirements, the message passing benchmark in Figure 6 was run on a different machine, with an Intel Xeon E5520 with 8MB of cache and 24GB of RAM. We were not able to secure exclusive access to this machine, and hence the results in Figure 6 are a little noisier than we would like.

7.2. Other Libraries

We compared our fiber library with these alternatives:

pthread. Standard POSIX Threads implementation on Linux (NPTL, part of `glibc`).

Entirely kernel space: creation, sleeping, and context switching require kernel calls.

Table I. Context Switch Performance on x86-64

	Cycles	Speedup over Pthreads
Raw SWAPSTACK	5	345.0
Fiber library	35	33.8
ST	105	11.5
LWP	191	6.3
Pthreads	1207	1.0
Pth	6111	0.2

Pth. GNU Portable Threads [Engelschall 2000], version 2.0.7. Designed for portability over performance, Pth calls into the kernel on every context switch even though threads are managed in userspace.

LWP. CMU Lightweight Processes, distributed as part of the Coda File System, version 2.4. Uses C's setjmp and longjmp to perform userspace context switches (a technically invalid usage, but works reliably on most platforms).

ST. State Threads (a derivative of the Netscape Portable Runtime library), version 1.9. Contains custom per-platform assembly code to perform context switching entirely in userspace. We were not able to successfully run benchmarks with this library on the PowerEN processor, so it is omitted from the results for that platform.

Of these, only Pthreads supports any hardware parallelism: Pth, LWP, and State Threads all perform their context switching within a single OS thread.

We benchmarked two versions of our fiber library, the single-threadeded fiberS and the parallel fiberP. fiberS is compiled without support for multiple workers and is therefore comparable to Pth, LWP, and ST. fiberP can load balance fibers over a number of worker threads, and so is more comparable to pthreads. In some synchronization-heavy benchmarks, fiberS outperforms fiberP as it can avoid the overhead of various atomic instructions necessary to ensure correctness in the presence of multiple workers.

7.3. Evaluation of Primitives

To measure context switch performance, we wrote a simple benchmark that passes control back and forth between a pair of threads, with results summarized in Table I. Those libraries which didn't require kernel crossings were naturally significantly faster. The "raw SWAPSTACK" line measures a direct SWAPSTACK, as would occur when using generators or directly message passing. The "fiber library" line measures the fiber library's implementation, which involves overhead such as manipulating the run-queue.

We investigated the performance of our SWAPSTACK primitive on x86 and found that our context switch is extremely fast for a number of reasons. First, the number of spilled registers is typically very low, particularly where the spill code can be hoisted out of loops. Secondly, the compiler inlines the code to implement the lightweight context switch. Thirdly, when a context suspends itself, it stores the continuation on the top of its own stack; this is extremely unlikely to cause a cache miss. In fact, in our testing we found that the main cost of SWAPSTACK is the result of branch mispredictions. First, the context switch is implemented with an indirect branch, which may be poorly predictable. Second, the processor's hardware return address stack (RAS) no longer correctly predicts the targets of return instructions, although our compiler's inlining of the context switch code means we have one fewer such return address misprediction than other mechanisms.

By way of comparison, we also benchmarked the time to communicate between different hardware threads on our x86-64 test machine. The communication was via simple loads and stores to shared cached memory, no atomic operations were used. It took either 20–35 cycles or 120–150 cycles to communicate between the threads,

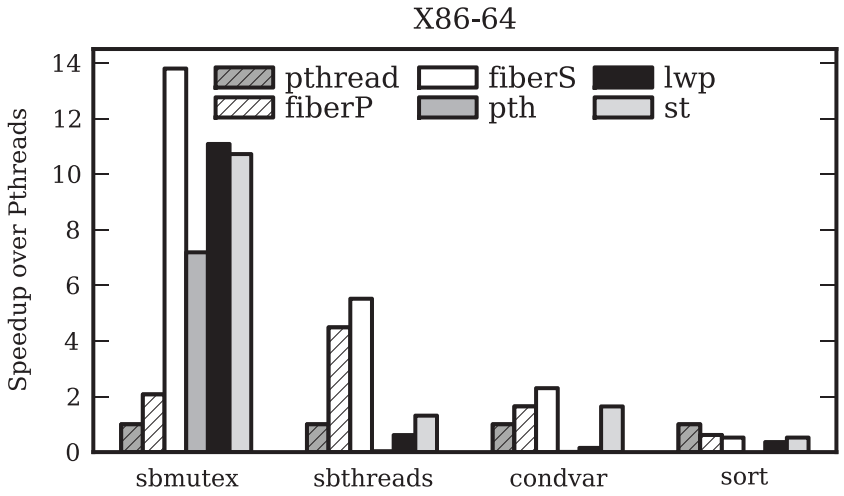

Fig. 7. Benchmark results comparing our fiber library with others.

depending on whether the two threads shared caches or not. Thus, SWAPSTACK between fibers is usually faster than passing a message between two hardware threads running in parallel.

The `threadring` benchmark is a simple microbenchmark used to measure context switch and message passing performance, in which the time to pass a single token N steps around a ring of M processes is measured. This benchmark has been implemented as part of the Computer Language Benchmarks Game [CLBG 2011]. We wrote a simple library for actor-model message passing concurrency on top of our fiber library, and compared its performance at this benchmark with the top three languages for this benchmark on CLBG, which were Haskell, Google Go, and Erlang (Haskell was around 21 times faster than C on this benchmark due to its lightweight threading support).

We compared the number of tokens passed per second using each of these three languages and our library. The results, shown in Figure 6, show our system outperforming the others by a significant margin. There is a fall-off in performance in the second half of the graph, which we believe is due to an increasing cache miss rate. The benchmark was run on a machine with 8MB of L2 cache, which is enough for around 10^4 fibers.

We limited the benchmark to 4GB of RAM. The lines on Figure 6 extend to the point where the respective systems reached this limit.

7.4. Evaluation of Fiber Library

To benchmark general thread and mutex performance, we used the `threads` and `mutex` benchmarks of the open-source `sysbench` suite of benchmarks. We also wrote a simple benchmark to test performance of thread suspension and resumption: a bounded producer-consumer queue using condition variables to handle underflow and overflow (`condvar`), and a parallel recursive quicksort to test thread creation performance (`sort`).

We compared the performance of these benchmarks when compiled against our fiber library in single-threaded and parallel modes, and against the various threading libraries mentioned in the previous section.

The results are displayed in Figure 7. The two hatched bars at the left of each benchmark are the parallel systems (`pthreads` and `fiberP`, our fiber library in parallel mode), while the rest are single-threaded. On most of the benchmarks, `fiberP` outperformed `pthreads` and `fiberS` outperformed the single-threaded libraries (`pth`, `lwp` and `st`).

In the case `sort` benchmark the performance of pthreads is actually better than our library. This is because in the `sort` benchmark almost all the time is spent actually doing the sorting computation and there is only a very tiny amount of context swithing. The advantage of our fiber library is the very fast context switching and very small amount of memory to store contexts. Other aspects of our system, such as effective load balancing apportioning work to cores to maintain locality, are less well developed. The much more mature Linux and pthreads systems do a better job of balancing the work across cores, and keeping memory accesses local to particular cores, and so do better on the `sort` benchmark where there is little context switching.

On the synchronization-heavy benchmarks, the single-threaded libraries outperformed the parallel ones as the overhead of hardware synchronization outweighed the advantages of parallelism.

In addition we have added results from benchmark programs Fibonacci calculator and threadrings. Fibonacci is a simple recursive Fibonacci calculator which spawns a new thread for each recursive call, and thus heavily stresses thread creation and joining. Threadring is a copy of the C version of the benchmark used in the Computer Language Benchmarks Game [CLBG 2011] (with minor changes for compatibility), and tests message passing performance between threads. We had used threadring earlier to evaluate the number of tokens passed per second using various languages and our library in the back-end.

Since the system used to run the earlier benchmarks was not available anymore, the additional benchmarks and the previous ones were executed in a system consisting of quad core Intel Xeon E5620 in a dual socket configuration with 24GB DDR3 RAM as memory.

The results for the additional benchmarks are shown in Figure 8. We find that our fiber library is 85 times faster than pthreads in the case of Fibonacci calculator and over 400 times faster in the case of threadring when compared to pthreads. The rest of the benchmarks show similar speedups seen in the earlier results.

7.5. Amount of Data to Save/Restore across Context Switches

One of the main benefits of our technique is the reduced amount of data saved and restored around context switches. When we look at the generated assembly code we usually see that only a small number of registers need to be saved. However, it is surprisingly difficult to measure the number of registers that must be saved/restored at each point where SWAPSTACK is used. The reason is that rather than simply spilling all the live registers at the point in the program where the SWAPSTACK is invoked, we instead implement SWAPSTACK as a calling convention that does not preserve the values in any registers. The compiler must therefore preserve the live values, and in many cases it simply saves the registers at the site of the context switch. However, the compiler may equally save the value at an earlier place in the code, perhaps where it is last used, perhaps outside a loop, and perhaps in a function that is further up the call chain. This makes it difficult to identify how much state must be saved for a particular context switch.

Nonetheless, it is possible to say something concrete about the range of data sizes that must be saved and restored on a context switch. The following discussion is with reference to the x86-64 architecture using the System V application binary interface (ABI) [Michael Matz and Mitchell 2012], which is the one commonly used in Linux systems. Specific numbers for other architectures and ABI standards will differ, but the general ideas are the same.

When a preemptive threading system like pthreads decides to perform a context switch, it knows nothing about what state the program may be in. Without any information indicating which registers are live or dead, it must save and restore the

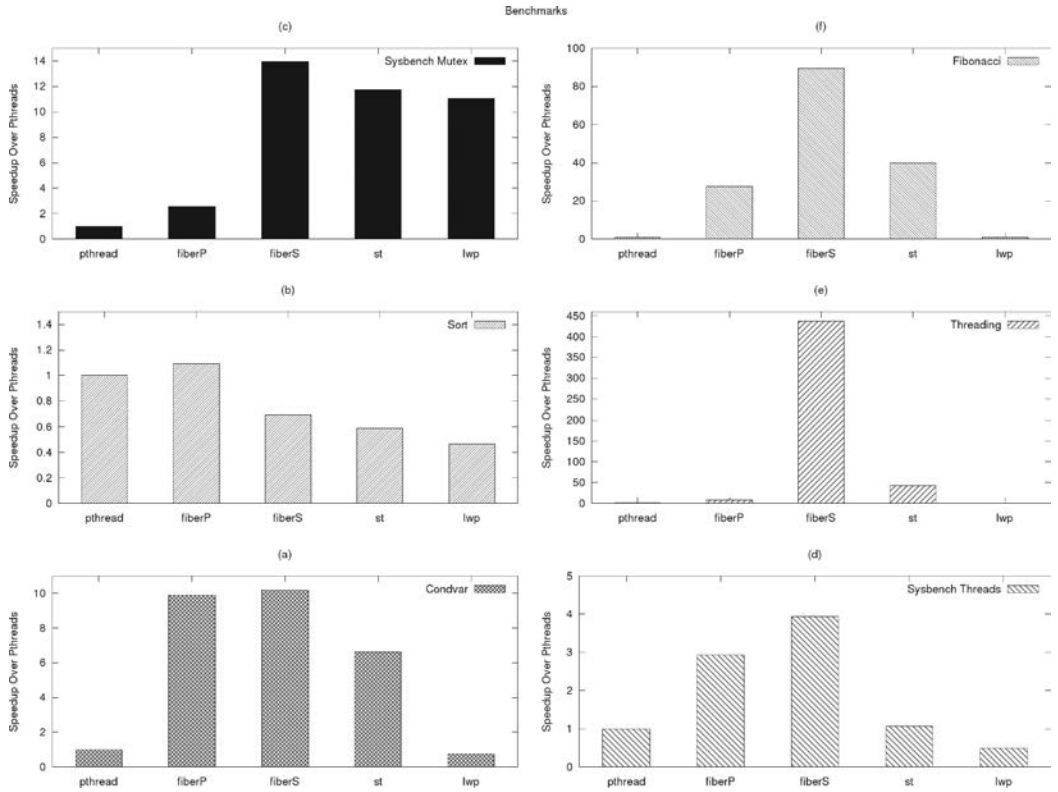

Fig. 8. Additional benchmark results comparing our fiber library with other thread libraries.

entire processor register file. On x86-64, there are 16 general-purpose integer registers of 8 bytes each, giving 128 bytes of general-purpose registers. The x86-64 architecture provides specific instructions (FXSAVE and FXRSTOR) to save and restore the floating-point and SSE vector registers, and the total stored state is guaranteed to fit inside 512 bytes. The exact amount of state to be stored is complex, but we can expect somewhat less than 512 bytes to be actually stored. The instruction pointer and flag pointer must also be saved, each of which requires eight bytes. So the total number of bytes required to save the full register of set of an x86-64 processor can be as much as 656 bytes.

Cooperative threading systems which use the POSIX functions getcontext, setcontext, and swapcontext store essentially the same data: a ucontext_t on x86-64 Linux is 936 bytes long (which is larger than our estimate above because of some padding and reserved fields in ucontext_t, and since it must also store the signal mask).

Most of this state is ABI-defined as caller-save, and not guaranteed to be preserved across function calls. Some cooperative threading libraries exploit this fact by placing the context switch code in a function which the compiler is prevented from inlining. This means that the compiler is forced to ensure that all the other registers are dead before the context switch code is called, and so it needs to only save and restore the callee-saved registers. So these kinds of cooperative threading libraries must allocate space for six integer callee-saved registers, the stack pointer, the program counter, and stack frame overhead because these libraries are called as an out-of-line function. Therefore these libraries require an absolute minimum of 80 bytes. In addition these libraries

also need to have spill space allocated in the caller's stack frame by the compiler in order to preserve values stored in caller-saved registers. This space can be as small as zero for a simple function but can grow large for a function with many variables that are live across the context switch.

With our SWAPSTACK mechanism we also normally need to save a minimum of the six callee-save x86-64 registers, just like libraries that do cooperative context switching using a library function that cannot be inlined. However, we have had SWAPSTACK a compiler intrinsic, not a function. Therefore, we need not suffer the overhead of a new stack frame, and our optimization described in Section 3.5 will try to mark functions that perform a context switch as not preserving callee-save registers. This attribute will bubble up the call chain of functions until it reaches either the top-level thread function, or an optimization barrier like a call through a function pointer. If the optimization makes its way to the top-level function, then the callee-save registers will never be saved, unless they actually contain live value. The top-level function need not preserve them as it does not return normally.

In such a case, where the optimization has been successful, the only state that we need store is the program counter and stack pointer (and possibly frame pointer, depending on optimization flags), as well as the previously mentioned spill space for live values. We do not use explicit separate buffers or structures to store this state. Instead all saved registers and other values are stored on the stack of the paused thread, and refer to the paused thread by its stack pointer. Thus, the stack pointer becomes the "handle" of the thread, and we do not allocate space to store it. So, with all optimizations turned on, the space overhead of a SWAPSTACK operation can be as low as eight bytes: a single machine word to hold the saved program counter.

In order to better understand this mechanism we manually examined the generated assembly code for the threadring benchmark using SWAPSTACK, and did our best to identify the amount of saved state at each point. There are 10 instances of SWAPSTACK in the generated code for this benchmark. Three of the SWAPSTACKs are used during initialization, and are called from entry points (main and C++ static intitializers). All three must save the callee-save registers, and therefore have context sizes of around 70 bytes. Three more SWAPSTACKS are found in the special fiber that runs the scheduler in the threadring benchmark. The scheduler maintains quite a lot of state on its stack, and a total of around 160 bytes of state are maintained. The remaining four SWAPSTACKS happen in the actual code of the benchmark. One of these saves 36 bytes, and the other three 48 bytes each. It is worth noting that in all cases in this small benchmark the amount of space that must be saved at a context switch using SWAPSTACK is much less than the maximum 656 bytes saved by mechanisms that save all registers. In addition, in some cases our optimizations allow SWAPSTACK to save less space than the minimum of 80 bytes that must be stored by cooperative threading libraries which must always save the callee-saved registers.

8. RELATED WORK

Other authors have proposed and demonstrated benefits of extending a compiler with knowledge of context switching, in order to reduce the time spent in a context switch and/or the space requirements of a paused thread.

Grunwald and Neves [1996] propose a whole-program live register analysis pass on an executable's compiled code. They then use this information to patch those points in the executable which contain context switches, avoid saving and restoring dead registers.

Other work also seeks to reduce the number of registers saved and restored during a context switch, and does so via architectural support for partitioning the register

file into smaller contexts so that more threads can remain resident on the CPU. Waldspurger and Weihl [1993] and Nuth and Dally [1995] are examples of this.

Zhou and Petrov [2006] tackle the somewhat different problem of optimizing context size for preemptive context switching, where the exact location of the context switch is not known a priori. Their technique involves live register analysis on the generated code in order to identify those points at which a context switch is least expensived. Context switching is then deferred until the execution reaches such a "cheap context switch" point. Jääskeläinen et al. [2008] apply roughly the same technique to cooperative threading, where the scheduler is polled and a context switch possibly performed at each of the inexpensive context switch sites.

All of these techniques integrate into the compiler after the register allocation phase, and take the set of live registers as a given. Our work, by contrast, integrates before register allocation and makes context saving and restoration into a special case of spill code insertion. This enables a host of standard optimizations to reduce rather than just determine the number of live registers, as described in Section 3.4.

A different approach to optimizing threads via compiler support, particularly for embedded systems, is a *serializing compiler* which lowers a threaded program into an equivalent serial program. Nácul and Givargis describe such a compiler, Phantom, in Nacul and Givargis [2005]. Another embedded-systems focused approach is described by Barthelmann [2002], where global register allocation is performed across the application and the embedded OS with a view to reducing context size.

The relationship between continuations and threads is well-known. In Bruggeman et al. [1996] it is observed that threads can be implemented using a restricted form of continuation which is only invoked once (the "one-shot continuation"). One-shot continuations are the form of continuation used by SWAPSTACK calls, and are significantly less expensive to implement than full continuations (which in many implementations involve saving and restoring not just the register file, but part of the stack). The relationship between continuations and threads exists in the opposite direction as well, as shown by Kumar et al. [1998], where threads are used to implement a form of continuation.

Designing a concurrent data structure which avoids the overhead of full locking, even one as trivial as a queue, has proven to be a surprisingly difficult problem and a fertile research ground. An array-based algorithm was described by Tsigas and Zhang [2001], while linked-list-based algorithms are given by Michael and Scott [1996], Fober et al. [2002], and Ladan-Mozes and Shavit [2004]. The idea of relaxing guarantees of forward progress, and treating progress as an engineering problem to be optimized rather than a condition to be proved, is due to Herlihy et al. [2003].

Biased locking algorithms were proposed by Vasudevan et al. [2010]. Applications for the Java programming language, where syntax-level support for locks makes them a frequent construct even when not strictly necessary, were described by Russel and Detlefs [2006], Kawachiya et al. [2002], and Onodera et al. [2004].

The usual biased locks concept biases a lock towards a thread, which has synchronization-free access. However, as this work biases a lock towards a worker rather than an individual fiber, our biased locks may have more applications. For instance, a lock only acquired by two threads in a producer-consumer relationship wouldn't be accelerated by a standard biased lock, but may be in our system when the producer and consumer reside on the same worker.

There have been many implementations of threading as a userspace library, not integrated with the compiler. Notable ones include State Threads [Gustafsson 2005] and GNU Pth [Engelschall 2000] (emphasizing portability over performance). Capriccio [Von Behren et al. 2003b] can schedule threads to dynamically optimize resource usage, and uses a compiler analysis to support efficient allocation of stacks.

None of these supports distributing tasks over multiple cores. Some operating systems (notably earlier versions of Solaris [Garcia and Fernandez 2000]) used partially userspace threading implementations: the process was divided into several kernel-level workers which each switched, in userspace, between several threads. However, such implementations had load-balancing issues (threads could not move between kernel workers) and performance issues (system calls by one thread could block an entire worker) and have been superseded.

The high-level language Erlang supports large numbers of lightweight threads running in parallel on a smaller number of worker threads [Hedqvist 1998], as does Haskell [Li et al. 2007]. Pall's LuaJIT [Pall 2011], a JIT-compiled implementation of Lua, includes an interface between C and Lua coroutines containing a primitive similar to SWAPSTACK, implemented as inline assembly for GCC on the x86 architecture.

Whether to use threads at all instead of writing single-threaded event/callback-based systems is a perennial debate among those writing high-concurrency server software. The pro-threads side is argued by Ven Behren et al. [2003a], the pro-events side by Ousterhout [1996], and a duality between the two sides was shown by Lauer and Needham [1979].

With this article, we hope to support the threads side of this argument by showing that the performance and scalability limitations of threads can be overcome.

Work-stealing with per-worker run-queues is a standard means of distributing tasks over many processors. Analyses and enhancements were presented by Dinan et al. [2009] and Acar et al. [2000]. A specific work-stealing algorithm was proven optimal for a class of multithreaded program by Blumofe and Leiserson [1999]. This algorithm was employed in the Cilk-5 multithreaded language [Frigo et al. 1998].

9. CONCLUSION

We have shown that by tightly integrating context creation and switching into the compiler, it is possible to exceed the performance of library-based techniques. This is due to the superior optimization opportunities (in particular register allocation) afforded to the compiler. Our results show that this is a promising technique for accelerating highly concurrent, frequently synchronizing programs, and that context switching need not be a bottleneck.

The ability to create very lightweight contexts and switch between them inexpensively is valuable for implementing a variety of features in programming languages and computer systems, such as coroutines and user-level threads. In this article we argue that tightly integrating context creation and switching into the compiler allows extremely efficient context management code to be created.

We test this hypothesis by adding primitives to the LLVM compiler system for creating and switching contexts. By expressing context switching as a special-purpose calling convention, the compiler is able to take context switching during other stages of compiler optimization and code generation, and in particular during register allocation.

Our results show that our techniques greatly reduce the amount of context that must be stored during a task switch. This allows contexts to be stored with much less memory that would be needed if all the registers had to be spilled at each context switch. The result of this is that contexts can be created that require only a very small amount of memory, allowing very large numbers of contexts to be created. In addition, spilling significantly registers at context switches also greatly reduces the cost of context switching. Our experiments show that this is a promising technique for accelerating highly concurrent, frequently synchronizing programs, and that context switching need not be a bottleneck.

REFERENCES

ABELSON, H., DYBVIG, R. K., HAYNES, C. T., ROZAS, G. J., ADAMS IV, N. I., FRIEDMAN, D. P., KOHLBECKER, E., STEELE, JR., G. L., BARTLEY, D. H. HALSTEAD, R., OXLEY, D., SUSSMAN, G. J., BROOKS, G., HANSON, C., PITMAN, K. M., AND WAND, M. 1998. Revised report on the algorithmic language scheme. *Higher Order Symbol. Comput. 11*, 1, 7–105.

ACAR, U., BLELLOCH, G., AND BLUMOFE, R. 2000. The data locality of work stealing. In *Proceedings of the the Symposium on Parallel Algorithms and Architectures.* ACM, 1–12.

BARTHELMANN, V. 2002. Inter-Task register-register-allocation for static operating systems. In *Proceedings of the Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems (LCTES/SCOPES'02).* ACM Press, New York, 149–154.

BLUMOFE, R. D. AND LEISERSON, C. E. 1999. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 720–748.

BRUGGEMAN, C., WADDELL, O., AND DYBVIG, R. 1996. Representing control in the presence of one-shot continuations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM Press, New York, 99–107.

CLBG. 2011. The computer language benchmarks game. http://shootout.alioth.debian.org/.

CONWAY, M. E. 1963. Design of a separable transition-diagram compiler. *Comm. ACM 6,* 7, 396–408.

DANIEL, C. H. 1995. Introduction to the programming language occam. http://www.eg.bucknell.edu/~cs366/occam.pdf.

DINAN, J., LARKINS, D., SADAYAPPAN, P., KRISHNAMOORTHY, S., AND NIEPLOCHA, J. 2009. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.* ACM Press, New York, 53.

ENGELSCHALL, R. 2000. Portable multithreading: The signal stack trick for user-space thread creation. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 20.

FOBER, D., LETZ, S., AND ORLAREY, Y. 2002. Lock-Free techniques for concurrent access to shared objects. *Actes des Journes d'Informatique Musicale, Marseille*, 143–150.

FRIGO, M., LEISERSON, C., AND RANDALL, K. 1998. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not. 33*, 5, 212–223.

GARCIA, F. AND FERNANDEZ, J. 2000. Posix thread libraries. *Linux J. 2000*, 70es, 36.

GOOGLE. 2012. TCMalloc: Thread-Caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

GRUNWALD, D. AND NEVES, R. 1996. Whole-Program optimization for time and space efficient threads. *SIGOPS Oper. Syst. Rev. 30*, 5, 50–59.

GUSTAFSSON, A. 2005. Threads without the pain. *Queue 3*, 9, 34–41.

HEDQVIST, P. 1998. A parallel and multithreaded ERLANG implementation. Masters dissertation, Uppsala University, Uppsala, Sweden.

HERLIHY, M., LUCHANGCO, V., AND MOIR, M. 2003. Obstruction-Free synchronization: Double-Ended queues as an example. In *Distrib. Comput. Syst*. IEEE, 522–529.

JÄÄSKELAINEN, P., KELLOMAKI, P., TAKALA, J., KULTALA, H., AND LEPISTO, M. 2008. Reducing context switch overhead with compiler-assisted threading. In *Embedded and Ubiquitous Computing*, Vol. 2, IEEE, 461–466.

KAWACHIYA, K., KOSEKI, A., AND ONODERA, T. 2002. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.* 130–141.

KUMAR, S., BRUGGEMAN, C., AND DYBVIG, R. K. 1998. Threads yield continuations. *LISP Symbol. Comput. 10*, 223–236.

LADAN-MOZES, E. AND SHAVIT, N. 2004. An optimistic approach to lock-free fifo queues. *Distrib. Comput*. 117–131.

LAUER, H. AND NEEDHAM, R. 1979. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev. 13*, 2, 3–19.

LI, P., MARLOW, S., PEYTON JONES, S., AND TOLMACH, A. 2007. Lightweight concurrency primitives for GHC. In *Proceedings of the ACM SIGPLAN Workshop on Haskell.* ACM, 107–118.

MICHAEL, M. AND SCOTT, M. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 267–275.

MATZ, M., HUBICKA, J., JAEGER, A., AND MITCHELL, M. 2012. System V application binary interface, AMD64 architecture processor supplement. http://www.x86-64.org/documentation/abi.pdf.

NACUL, A. C. AND GIVARGIS, T. 2005. Lightweight multitasking support for embedded systems using the phantom serializing compiler. In *Proceedings of the Conference on Design, Automation and Test in Europe.* Vol. 2, IEEE, 742–747.

NUTH, P. AND DALLY, W. 1995. The named-state register file: Implementation and performance. In *Proceedings of the International Symposium on High Performance Computer Architecture*. IEEE, 742–747.

ONODERA, T., KAWACHIYA, K., AND KOSEKI, A. 2004. Lock reservation for java reconsidered. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP'04).* 1–22.

OUSTERHOUT, J. 1996. Why threads are a bad idea(for most purposes). In *Usenix Annual Technical Conference.*

PALL, M. 2011. The luajit project. http://luajit.org.

RUSSELL, K. AND DETLEFS, D. 2006. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *SIGPLAN Not. 41*, 10, 263–272.

SCHEMENAUER, N., PETERS, T., AND HETLAND, M. 2001. Pep 255:Simple generators. http://www.python.org/dev/peps/pep-0255/.

SUSSMAN, G. J. AND STEELE, G. L. JR. 1975. Scheme: An interpreter for extended lambda calculus. http://18.7.29.232/bitstream/handle/1721.1/5794/AIM-349.pdf?sequence=2.

TSIGAS, P. AND ZHANG, Y. 2001. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architecture*. ACM, 134–143.

VASUDEVAN, N., NAMJOSHI, K., AND EDWARDS, S. 2010. Simple and fast biased locks. In *Proceedings of the International Conference on Parallel Architectures and Compliation Techniques*. 65–74.

VON BEHREN, R., CONDIT, J., AND BREWER, E. 2003a. Why events are a bad idea (for high-concurrency servers). In *Conference on Hot Topics in Operating Systems*. USENIX Association.

VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G., AND BREWER, E. 2003b. Capriccio: Scalable threads for internet services. *SIGOPS Oper. Syst. Rev. 37,* 5, 268–281.

WALDSPURGER, C. A. AND WEIHL, W. E. 1993. Register relocation: flexible contexts for multithreading. In *International Symposium on Computer Architecture*. ACM, 120–130.

ZHOU, X. AND PETROV, P. 2006. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications. In *Proceedings of the Design Automation Conference (DAC'06).* ACM Press, New York, 352–357.

Received June 2012; revised October 2012; accepted October 2012