Compiler-Based Timing For Extremely Fine-Grain Preemptive Parallelism

Souradip Ghosh Michael Cuevas Simone Campanoni Peter Dinda
Department of Computer Science
Northwestern University
{sgh, cuevas}@u.northwestern.edu
simonec@eecs.northwestern.edu, pdinda@northwestern.edu

***Abstract*—In current operating system kernels and run-time systems, timing is based on hardware timer interrupts, introducing inherent overheads that limit granularity. For example, the scheduling quantum of preemptive threads is limited, resulting in this abstraction being restricted to coarse-grain parallelism. Compiler-based timing replaces interrupts from the hardware timer with callbacks from compiler-injected code. We describe a system that achieves low-overhead timing using whole-program compiler transformations and optimizations combined with kernel and run-time support. A key novelty is new static analyses that achieve predictable, periodic run-time behavior from the transformed code, regardless of control-flow path. We transform the code of a kernel and run-time system to use compiler-based timing and leverage the resulting fine-grain timing to extend an implementation of fibers (cooperatively scheduled threads), attaining what is effectively preemptive scheduling. The result combines the fine granularity of the cooperative fiber model with the ease of programming of the preemptive thread model.**

***Index Terms*—timing, preemptive scheduling, fine-granularity parallelism**

I. Introduction

Fully exploiting a modern machine of any kind depends on extracting and leveraging parallelism across a wide range of granularities ranging from below the level of individual instructions to beyond the level of independent long-running jobs (e.g. [38], [14], [1]). Future machines are likely to further expand this requirement (e.g. [35], [10], [17]). The goal of this paper is to shrink the granularity that can be supported dynamically at the software level. How small can the granularity be at which the machine can be easily programmed?

Why should the HPC community care about fine granularity parallelism? Because the finer the granularity that can be achieved, the greater the flexibility in mapping algorithmic parallelism via a programming model to disparate and dynamic hardware resources, and thus the greater the ability to fully exploit those resources. We can see this in programming models as varied as software data flow [3] to nested data parallelism [7], [24], [40], the adoption of task graphs into OpenMP [4], the search for primitives for parallel runtime systems that unify many abstractions [43], [5], [50], and the development of mechanisms to dynamically control the degree of algorithmic parallelism used at runtime [2].

A natural programming model, or certainly execution model, for a shared memory machine, is based on classic preemptively scheduled *threads*. Most programmers are familiar with threads in their widely available POSIX incarnation. Assuming an operating system kernel-based implementation, the programming model is greatly simplified compared to other alternatives because the programmer can ignore many issues, such as blocking. Furthermore, threads can be scheduled across time and space with no programmer input, or, at the other extreme, can have specific, even real-time, constraints placed on them. The scheduler can straightforwardly dynamically load balance threads using explicit moves or work-stealing. There is much to like about threads.

However, current thread implementations cannot support extremely fine granularity operation. As a consequence, parallel language extensions such as OpenMP [9], [41], [4] and Cilk [8], [18], data flow execution models such as SWARM [32], Parallelex [26], Charm++ [27] and Legion [5], [54] and run-time models such as ARGObots [50], introduce or apply far more primitive mechanisms that we refer to here as *tasks*. A task is effectively a callback function that is typically invoked on a free CPU via an indirect `call` instruction when the task's dependencies are met. Tasks are extremely challenging to program with and are often relegated to the execution model, with heavy lifting done by the compiler.

Why are threads limited in granularity? Conceptually, a thread creation and launch is a lightweight affair. After all, thread state is effectively just register content and a cache footprint. Previous work has demonstrated that thread creations and launches can be extremely fast [21], [56]. The limitation is due to the need to build around preemption, which is driven by the kernel's timer system, which is in turn driven by hardware timer interrupts. Interrupts are what fundamentally limit the granularity of preemptive threads.

Interrupts are avoided in the design of cooperative threads (which we call *fibers* here), and consequently these can achieve much finer granularity [55]. With fibers, the locus of each context switch is determined by the programmer and is known at compile-time. As consequence, the actual code for the context switch simply involves a register content swap. However, while

This project is made possible by support from the United States National Science Foundation through grants CCF-1533560, CNS-1763743, CCF-1908488, and CCF-2028851, and by equipment support from Intel Corporation. Some measurements were made possible via use of the Illinois Institute of Technology's MYSTIC Testbed, which is supported by the NSF via grant CNS-1730689.

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

cooperative threads/fibers can hide the blocking problem from the programmer, they rely on the programmer to explicitly invoke the scheduler, either by invoking a blocking call or by calling a `yield()` function. Long historical experience both in academia [42] and in practice in the pre-2000 versions of MacOS and Windows [45], suggests that programmers are likely to get this wrong. In a cooperatively scheduled system, if one programmer gets it wrong in one place, the whole system suffers. A classic scenario is the programmer assuming that some loop will be quickly finished and thus not yielding during its execution. If this assumption turns out to be wrong, the yield-free loop now blocks all other fibers from executing. As the target system scales, this problem only gets worse. Consequently, current cooperative threads/fibers have limited application, despite their performance benefits.

In this paper, we focus on the underlying problem, *timing*. In current systems, most timing-related events, including for thread scheduling, are based on *hardware-based timing*. A timing-based event, such as an invocation of the thread scheduler, is initiated by an interrupt from a hardware timing device, and processing the event races with the rest of the application and kernel code. This form of timing is powerful, precise, and accurate, but comes at the cost of having to handle interrupts. We propose *compiler-based timing* as an alternative to hardware-based timing.

In compiler-based timing, the entire codebase of the system, including the operating system kernel itself, is transformed using modern compiler analyses and transformations to introduce `calls` into the timer framework that replace hardware timer interrupts. A major challenge is that the calls are introduced *statically*, but they need to occur *dynamically* at some desired rate regardless of the code path taken through the kernel+application ensemble as it runs.

We demonstrate that compiler-based timing is feasible, and then use it to augment an implementation of fibers with automatic invocations of `yield()`. These invocations serve the same purpose as timer interrupts in preemptive threads. In other words, we make the fibers implementation "preemptive", and thus much more straightforward to program and use, while still maintaining its low overhead and support for much finer granularity parallelism.

Our contributions are as follows:

- We motivate, propose, and describe the concept of compiler-based timing.
- We describe the code analyses and transformations necessary to enable compiler-based timing to have similar precision to hardware-based timing.
- We design, implement, and evaluate a prototype of compiler-based timing within the LLVM framework [31] and the Nautilus kernel [21].
- We design, engineer, and evaluate an implementation of fibers within the Nautilus kernel that is designed with compiler-based timing in mind.
- We combine our compiler-based timing and fibers implementations, and evaluate the composite using microbenchmarks and an application benchmark.

Our results show that compiler-based timing has considerable potential. It can achieve a timer resolution that is 6.2x better than what is possible with hardware-based timing. Fibers that are made preemptive using compiler-based timing can achieve stable scheduling granularities that are up to 4x better than what is possible with preemptive, hardware-based timing-driven threads.

We motivate our work on compiler-based timing through bridging the fine granularity of cooperative threads and the ease of programmability of preemptive threads. However, it is important to understand that timing events are endemic to the operation of a modern operating system kernel, and thus compiler-based timing has potential well beyond this motivation. Additionally, compiler-based timing could also be applied at user-level, where avoiding hardware-based mechanisms brings the additional performance advantage of avoiding user/kernel transitions.

Our implementation of compiler-based timing is publicly available within the open-source Nautilus kernel codebase.

II. Software infrastructure and testbed

We leverage the LLVM compiler framework, WLLVM, and the Nautilus research kernel in this work, and do evaluations on several x64 machines. We now describe these in more detail.

LLVM: LLVM [31] is a widely-used compilation framework in academia and industry that enables sophisticated code analyses and transformations. In this work, we use the framework to implement compiler-based timing at the level of the LLVM intermediate representation (LLVM-IR), within the "middle-end" of LLVM. The middle-end provides the API to develop highly-accurate, program-wide code analyses and transformations. Alias analysis and inter-procedural passes are examples of middle-end-only support. While Clang, which targets C and C++ (including OpenMP), is our front-end, another benefit of working in the middle-end is that our tools are language-independent and can thus be used with other language front-ends to LLVM (e.g. Fortran via flang).

WLLVM: WLLVM [47] extends LLVM compilation to aggregate all the LLVM bitcode in a project that uses separate compilation. This aggregation gives us two key capabilities. First, it allows for whole program integration and optimization, in our case across the entire kernel and application. Second, it enables our compiler analyses and transformations to see the entire static control flow graph of the kernel and application.

Nautilus kernel framework: Nautilus [21] is a publicly available open-source kernel codebase that currently runs directly on x64 NUMA hardware, including Xeon Phi. It comprises over 331K lines of code as measured by sloccount. Nautilus was designed with the goal of supporting hybrid run-times (HRTs). An HRT is a mashup of an extremely lightweight OS kernel framework, such as Nautilus, and a parallel run-time system [23], [20]. Nautilus can help a parallel run-time ported to an HRT achieve very high performance by providing streamlined kernel primitives such as synchronization and threading facilities. It provides the minimal set of features needed to support a *tailored* parallel run-time

environment, avoiding features of general purpose kernels that inhibit scalability. More recent work based on Nautilus has included extending the concept to include architectural and compiler changes. This work is an example of the latter.

Nautilus has a range of features that help make the execution of an HRT more predictable. Identity-mapped paging with the largest possible size pages is used. All addresses are mapped at boot, and there is no swapping or page movement of any kind. As a consequence, TLB misses are extremely rare, and, indeed, if the TLB entries can cover the physical address space of the machine, do not occur at all after startup. There are no page faults. All memory management, including for NUMA, is explicit and allocations are done with buddy system allocators that are selected based on the target zone. For threads that are bound to specific CPUs, essential thread (e.g., context, stack) and scheduler state is guaranteed to always be in the most desirable zone. The core set of I/O drivers developed for Nautilus have interrupt handler logic with deterministic path length. As used in this work, there are no DPCs, softIRQs, etc, to reason about: only interrupt handlers and threads. Finally, interrupts are fully steerable, and thus can largely be avoided on most hardware threads. Application benchmark speedups from 20–40% over user-level execution on Linux have been demonstrated, while benchmarks show that primitives such as thread management and event signaling are orders of magnitude faster [21], [22].

Test platforms: Testing is done on two machines. KNL is a Colfax Ninja Xeon Phi server, which includes a 1.3 GHz Intel Xeon Phi 7210 (64 cores, 256 hardware threads) mated to 16 GB of MCDRAM and 96 GB of DRAM. R415 is a more traditional platform, a Dell R415, which includes two 2.2 GHz AMD 4122s (8 cores total) mated to 16 GB of DRAM. Nautilus is booted directly on these platforms.

Benchmarks: To evaluate our work, we use a range of microbenchmarks, some of which are inspired by the mibench suite [19], including Rijndael (AES encryption), MD5, SHA1, cycle detection, Dijkstra's shortest path, and custom kernels such as dot product, linked list traversal, matrix multiply, binary search tree traversal, randomized matrix multiply, level ordered tree traversal, randomized floating point operations, randomized Fibonacci, k-nearest neighbor, unweighted minimum spanning tree, quicksort, and radix sort.

III. Hardware-based vs. Compiler-based timing

Figure 1 diagrams and compares the traditional hardware-based timing model, and the proposed compiler-based timing model. The compiler-based timing model involves substantially more complexity at compile time, but then avoids the cost of interrupts and MMIO at run-time, resulting in much reduced overhead on timing events.

A. Traditional hardware-based timing

Figure 1(a) illustrates how timing operates and integrates with timing-dependent services within a kernel such as Linux

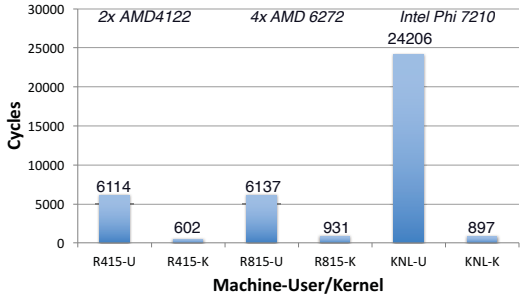


Fig. 2. Overheads of interrupt dispatch to kernel-level (Nautilus interrupt handler) and to user-level (Linux interrupt handler through signal delivery to user process).

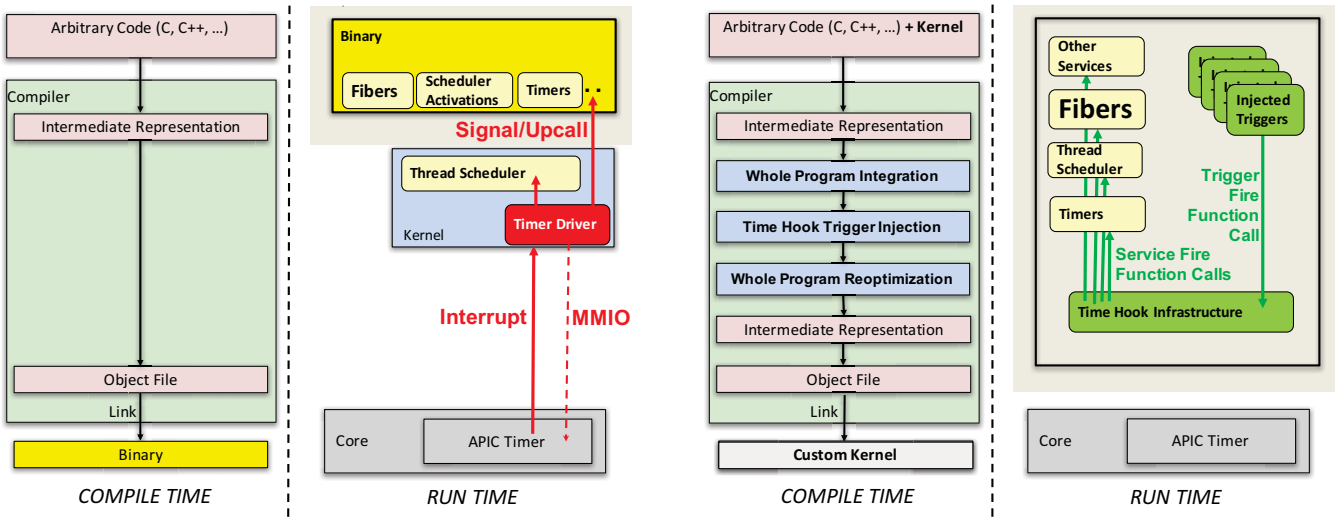
when running on x64. Compilation (left hand side) is straightforward with respect to timing for both the kernel and application code. The compiler toolchain is uninvolved with timing.

The action happens at run time (right hand side). The diagram shows the operation of a single hardware thread, here labeled a core. The other hardware threads are identical. On KNL this is repeated 256 times, one per hardware thread. The kernel's timer driver accesses the core's Advanced Programmable Interrupt Controller (APIC) via memory-mapped I/O (MMIO) to configure its timer component to fire an interrupt after a certain period of time. Most kernels, including Linux and Nautilus, use this hardware timer in one-shot mode, meaning that it is reconfigured during the interrupt handler. Modern APICs can usually manage time down to 10 ns granularity, limited by the APIC clock (typically 100 MHz), while some also support resolution to the processor cycle granularity using Intel's TSC deadline mode.

When the time is up, the APIC timer fires its interrupt. This results, eventually, in executing the interrupt handler within the timer driver. It is important to understand that interrupt dispatch is not free. In a modern Linux kernel, it will even involve an address space context switch due to the Spectre/Meltdown [34], [29] mitigation mechanisms [33] that place almost all of the kernel in a separate address space from user code, resulting in times on the order of $\sim$2000 cycles. Even in a model in which everything runs within a single address space at kernel level, such as in Nautilus, the interrupt dispatch is a $\sim$1000 cycle operation.

The timer driver will upcall relevant components of the kernel, such as the thread scheduler. Additionally, if user-level mechanisms are used, such as scheduler activations, blocking system calls based on time (e.g. `select`, `poll`, and `sleep`), user-level timers (e.g. `SIGALRM`, `SIGVTALRM`, etc), the dispatch also involves a kernel-to-user upcall, usually at a substantial additional cost.

Figure 2 illustrates these overheads at both user-level (-U) and kernel-level (-K). Timing using the cycle counter begins immediately before an instruction that intentionally causes the event. User-level reflects the time, in cycles, from the event to the first instruction of a signal handler in user code. Kernel-level reflects the time from the event to the first



(a) Traditional Hardware-based Timing

(b) Compiler-based Timing

Fig. 1. Comparison of traditional hardware-based timing and compiler-based timing models. Fibers are highlighted as they are our proof-of-concept service. In an all-kernel system with the traditional hardware-based timing model, the signal/upcall cost is avoided, but interrupt and MMIO costs remain.

instruction of the interrupt handler within the Nautilus kernel. Three machines are considered. KNL and R415 were described previously. R815 is a Dell R815 with 4 2.1 GHz AMD 6272 processors. The measurement does not include the unwinding of the interrupt dispatch, which adds several hundred cycles.

Even in a model, such as with Nautilus, which has no kernel/user distinction, the overhead of interrupt dispatch and reconfiguring the hardware time (primarily the former) limits the timing resolution that is practically possible, regardless of the resolution of the hardware timer itself. This limitation on timing resolution cascades into limitations on the services that depend on timing. Consider the 897 cycles for KNL-K shown in Figure 2. This places a hard limit on the resolution of hardware-based timing of at least this number of cycles. As described elsewhere [13], the total interrupt cost on this platform is about 1100 cycles, which is the practical timing limit. Even for the lowest overhead platform (R415-K), the practical timing limit is well over 800 cycles.

For scheduling fine granularity work, the practical timing limit places a limit on the granularity that can be handled in a system that provides preemption. As previously reported [21], preemptive thread context switches in Nautilus can be done in 1760 cycles (KNL-K) and 1391 cycles (R415-K), ignoring floating point context switching and using a constant time scheduler. These are among the fastest preemptive scheduling results reported. The current context switch cost on our testbed is about 2000 cycles, in part because of a real-time scheduler.

To avoid these hardware timing-limited high overheads for preemption, fiber or task implementations that are intended to support fine granularity workloads provide no preemption, and, indeed, often do not even support blocking. This places the onus on the developer, compiler, and/or run-time system to produce ensembles of fibers and/or tasks that provably do not require preemption in order to achieve correctness.

Although Figure 1(a) illustrates a Linux-like kernel, Nautilus, prior to compiler-based timing, is only slight different. The signal/upcall component is a simple function call in Nautilus, but the interrupt and MMIO overheads remain. More details of Nautilus's timing infrastructure and real-time thread and task framework are given in Section IV.

B. Proposed compiler-based timing

Figure 1(b) illustrates how compiler-based timing would operate. In contrast to the traditional model, there is little role for the hardware timing mechanisms. In fact, the APIC timer can be simply disabled. This avoids the overhead both of its interrupts and of configuring it via MMIO. Even if hardware timer interrupts would be needed for other reasons (e.g., a watch dog), the rate of timer interrupts would be much lower than in the traditional model.

The cost is now borne by the compiler side. All source code (or at least the compiler's intermediate representation (IR) of it) for the kernel, run-time system, and application, is supplied to the compilation process. A new compilation step, *whole program integration*, brings the totality of IR-level code, into a single IR-level representation. In addition, we are supplied with a *target timer resolution*.

In the critical next step, a new code transformation, *time hook trigger injection*, is applied. This transformation considers all statically discoverable code paths through the entire codebase and adds instrumentation to each one. The instrumentation consists of calls to a *trigger fire function*, which is exported by component of the kernel. The transformation places these calls such that, at run-time, the calls will occur at the target timer resolution (or higher), regardless of the dynamic control flow path being taken by each CPU. This is a non-trivial undertaking for several reasons. First, while the timing requirement is placed on the object code that will be

executed, the transformation is based on the IR, which is at a higher abstraction level. A second reason is that the discovery of all the possible dynamic control flow paths is complicated by the existence of things like interrupt handlers and function pointers. Third, we must guarantee that small code segments that are frequently iterated (e.g., small, but long-running loops and recursions) are instrumented. Finally, we must be able to provide a mechanism analogous to interrupt masking to be able to avoid instrumentation at times. However, this masking needs to be implemented at compile-time, not run-time.

The next compile-time step is *whole program reoptimization*, in which we revisit optimization in light of the injected instrumentation. Note that since all IR code is available to the compiler, this pass allows for optimization to cross the boundaries of the injected code and the original code, as well as module boundaries within the original code. The goal of this pass is to fuse the injected code and the original code thoroughly, to attempt to minimize the run-time overhead of the injected code, which now exists along every possible control flow path in the kernel.

Finally, back-end optimizations, object code generation, and linking are done. Note that register allocation and other resource allocation/code scheduling problems are solved with the fused code, without regard to boundaries between the injected code and the rest, which is is quite unlike traditional code generation for an interrupt handler (for hardware-based timing). A new kernel component, the *time hook infrastructure*, is linked into the final kernel.

As the kernel boots, components of the kernel and run-time register callback functions with the time hook infrastructure. These can be scoped to groups of CPUs, and each registration request contains a periodic or sporadic deadline that is a multiple of the target granularity configured at compile-time. These functions are guaranteed to be called at these times. At run-time, hardware timer interrupts are avoided. Instead, the injected calls to the trigger fire function invoke the time hook infrastructure. Unlike timer interrupts, however, these are simple calls. On each one, the infrastructure finds expired registered callbacks and invokes them, much like an timer interrupt handler might. It is the job of the callback function to determine if the context is safe for it to do its work.

IV. Fibers and threads in Nautilus

Nautilus implements a range of abstractions for concurrency, including tasks, preemptive threads, and thread groups. In this work, we have added cooperatively scheduled threads, also known as fibers. Later, we will apply compiler-based timing to add what amounts to preemption to fibers, resulting in a replacement for threads with lower overhead.

A. Threads

Our purpose in describing the technical details of Nautilus threads is to make clear they are already heavily optimized for performance. Thread scheduling in Nautilus is based on the classic periodic/sporadic/aperiodic hard real-time scheduling model [37] applied to threads [13], The thread implementation itself is preemptive, and every interrupt dispatch can invoke the scheduler to change the current thread on the CPU. Controlling time due to interrupts and scheduling overheads is critical to achieving real-time behavior, and a range of mechanisms are employed to do this, despite the unpredictability of system management interrupts (SMIs) on x64 platforms. General purpose interrupts are steered to the first CPU, and can even be isolated (to some extent) within a time block. Other CPUs see only scheduling interprocessor interrupts (IPIs) and one-shot hardware timer interrupts, which are configured by the per-CPU earliest deadline first (EDF) scheduling core.

Per-CPU schedulers cooperate globally to provide time-synchronized real-time scheduling (i.e., gang scheduling, achieved via time) to groups of threads. The per-CPU schedulers can also be configured to steal threads from each other, using a power-of-two-choices selection model. In addition to threads, the schedules manage tasks, which are stateful callback functions. These are consumed both by task threads (one per CPU) and by the per-CPU schedulers themselves when it can be proven they will not cause any admitted real-time thread to miss a deadline.

In this work, we use the scheduling infrastructure in a straightforward manner. We consider only non-real-time (non-RT) threads (aperiodic threads), which we schedule using round-robin. Work-stealing is disabled, tasks are not used, and no thread groups, real-time or otherwise, are employed. This configuration makes the thread scheduler overhead as low as it can be for comparison to fibers. In this mode of operation a thread context switch initiated by a timer interrupt takes on the order of 2200 cycles ($\sim$1.7 μs) to complete on KNL. This is among the fastest reported context switches of any thread implementation. For comparison, on the same platform, a Linux thread context switch requires $\sim$4900 cycles [21].

B. Fibers

Fibers in Nautilus are lightweight, cooperatively scheduled threads of execution that are implemented directly in the kernel. The key difference between threads and fibers is that a fiber-to-fiber context switch can only happen on an explicit `yield()`, while thread-to-thread context switches can also occur at any time, triggered by an interrupt. Except for accounting for this difference, the fiber interface has been designed to be in line with the thread interface to facilitate porting between the two.

Fiber thread: At boot time, a special "fiber thread" is created for each CPU, and it exists throughout the kernel's lifetime. The fiber thread is a full-blown Nautilus thread that can be scheduled using all the mechanisms described in Section IV-A. For example, it can be scheduled as a periodic real-time thread. When scheduled with an ordinary aperiodic model, as in this paper, if no competing threads exist, the fiber thread manages all CPU time (except for interrupts, although those can also be filtered).

Idle fiber: Unlike an ordinary thread, the fiber thread uses stack-switching to multiplex itself among the fibers assigned to the CPU. One of these fibers is the CPU's idle fiber, which

only runs when there are no other fibers available. The idle fiber exists to optionally cooperatively yield to the thread scheduler when no fiber is runnable. This can be done in three ways: spinning (not yielding at all), sleeping (temporarily blocking on a configurable timer), and waiting (blocking on a wait queue that is kicked on any fiber scheduling-related event for the CPU). These provide a tradeoff between efficiency and the latency of reacting to a fiber scheduling-related event.

Scheduling: Fiber scheduling decisions are made independently on each CPU, using per-CPU state. Fibers can be created on any CPU, for any CPU, with the choice of target CPU being either under programmer control or by random assignment. Similar to Nautilus threads, fibers also provide a fork primitive to allow arbitrary splitting control flow at any point, not just at function boundaries. No work stealing is currently done. Fibers run concurrently on different CPUs. The per-CPU scheduling policy is nonpreemptive FIFO, which is constant time, and implemented using per-CPU spinlock-mediated queues. While the maximum number of threads in Nautilus is determined at compile-time, there is no static limit to the number of fibers.

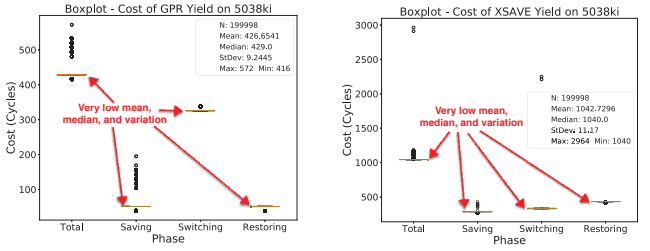
Contexts: A fiber is represented with a minimal structure whose key element is a stack pointer into the fiber's stack. Creating a fiber involves two allocations and initial stack setup to allow launching the fiber via a trampoline that is invoked via a context switch to the fiber. This model allows us to avoid differentiating between fibers that have been started and those that have not: we can always yield to any non-running fiber. When a fiber is not running, its context is fully captured on its stack. This includes the floating point state, which we capture using the x64 `xsave`/`xrstor` instructions. This requires some special run-time handling due to the alignment and initialization requirements on the targets of these instructions.

Context switching: A context switch occurs only in response to an explicit yield. It is important to understand that this means it occurs as part of a function call, and this greatly simplifies and speeds up its handling in comparison to threads.

A context switch has three phases. The first, *saving*, occurs when any yield function is called. These functions are wrapped by an assembly stub that minimally writes all general purpose registers, excluding flags, onto the stack. If floating point is enabled, the stack is further configured for the `xsave` instruction. This involves making room for a specially aligned region of appropriate size, zeroing where the header will go, and then executing the `xsave` instruction with an appropriate mask to select which register state to include.[1] The assembly then stashes the stack pointer in the fiber's structure and calls into the scheduler.

The second phase, *switching*, asks the scheduler for the next fiber to run. Currently, the scheduler is simple FIFO—we enqueue the current fiber on a per-CPU queue and dequeue the next fiber. The idle fiber is skipped if any non-idle fiber exists in the queue. We then invoke an assembly-level restore routine to switch to the next fiber. Note that next fiber's stack contains the register state that was saved when that fiber was switched away from.

[1]We point out the details of floating point save/restore because these steps can be expensive and both threads and fibers must bear those costs.



(a) Yield (no FPRs) (b) Yield (FPRs w. XSAVE)

Fig. 3. Yield cost breakdown and measurements on KNL.

In the last phase, *restoring*, we do the actual stack switch, loading `%rsp` with the next fiber's stored stack pointer. From this point, we can simply use `xrstor` to load the floating point registers, and then pop all the general purpose registers from the stack. Finally, we can do an ordinary `retq` to return from the yield that the fiber originally ran.

Performance: Figure 3 shows the measured yield cost breakdowns on KNL with and without floating point context. Note that these figures are box plots—the extremely low variance results in each box and whiskers are collapsing into to a horizontal line. The outliers shown are very rare. On R415 there is similar behavior, although the mean costs are lower.

V. Implementing compiler-based timing

We now describe our implementation of compiler-based timing, expanding on the big picture given in Section III-B. Figure 4 illustrates our compilation process. Our compiler extensions transform the entire Nautilus kernel codebase (>331K LOC) and span the kernel code and application code to provide periodic timing events. Our approach replaces hardware timer interrupts and operates with lower overhead, and thus finer granularity can be obtained.

The largest component of our implementation is an LLVM middle-end analysis and transformation pass. The pass begins by estimating the run-time clock cycle latency of each LLVM IR instruction. It then uses a new data flow analysis to integrate single-instruction latencies along all possible control flows. We call this the *accumulated latency*.

The result is estimates of the run-time latencies at the level of basic blocks, loops, and functions across the entire codebase.

Armed with these latency estimates we inject calls to the kernel's trigger fire function `nk_time_hook_fire()` throughout the entire codebase at points where the accumulated latency could reach the target time (e.g., every 1000 cycles). Hence, calls to this function will occur at run-time at the target timer resolution.

These calls take the place of hardware timer interrupts and in turn trigger run-time registered callbacks for any rate that is a multiple of the target timer resolution, providing a soft real-time guarantee for these callbacks.

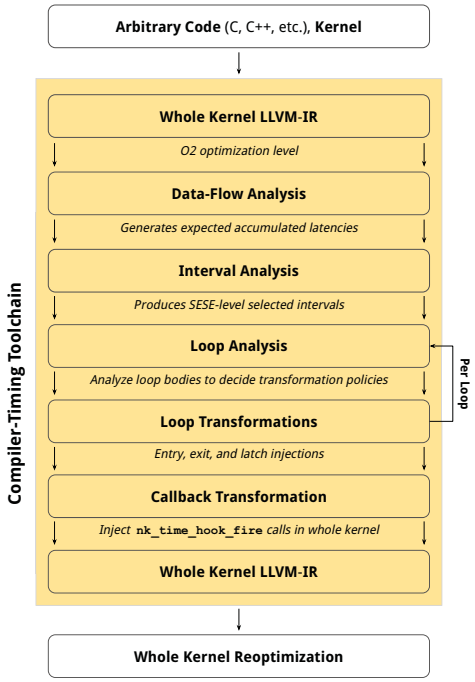


Fig. 4. Compiler-based timing transformation process.

Figure 5 shows an example of the transformation our compiler automatically performs. It is challenging to understand a transformation example at the IR level, particularly if the reader is unfamiliar with LLVM-IR. In Figure 5 we have mapped the salient result of transforming the binary search tree traversal benchmark to its approximate representation at the source code level. The injected calls to `nk_time_hook_fire()` divide the codebase into intervals. All colored code has been injected by the compiler transform. We note intervals, and the loop entry, exit, and latches.

A. *Compiler-side: analyses and transformations*

Instruction latency estimates: Our code analyses start with estimates of the latencies of individual LLVM IR instructions. These estimates are made at design time. To this end, we considered possible mappings that LLVM x64 back-end can use to translate an IR instruction into a sequence of x64 machine code instructions. Some LLVM IR instructions map to individual x64 machine instructions (e.g., `add`) [57]. Others require several x64 instructions. The latency of an IR instruction is computed by summing up the average latencies of the mapped x64 instructions. These x64 average latencies are taken from a previous detailed analysis of our target architecture [16].

We embrace the fact that this IR→x64 mapping is architecture dependent and includes inaccuracies. Typical compiler back-ends perform such translations using a context-aware approach. For example, the common "maximal munch" algorithm for instruction selection considers a sequence of IR instructions (rather than a single IR instruction) at a time. However, this paper empirically demonstrates that the precision of using a simple 1-to-many mapping is sufficient

```
/* Function structure */
double bst_traversal_and_computation (bst *tree) {
    /* Straight line code */                                   Interval 0
    uint64_t rand_int = lrand48();
    double rand_fp_cast = (double) rand_int,
           rand_fp_first = rand_fp_cast * MULTIPLIER,
           rand_fp_second = rand_fp_first / DIVIDEND;

    nk_time_hook_fire(); // INJECTION ← Exceed Policy

    double rand_fp_final = rand_fp_second + ADDITIVE;          Interval 1
    double *searches = _kernel_build_random_array(
                        rand_fp_final, NUM_RAND);
    double total_sum = 0;

    /* Loop, depth = 1 */
    // INJECTION ← Loop Latch Policy (represents counters)
    int iter_count_depth_1 = 0, iter_count_depth_2 = 0;
    nk_time_hook_fire(); // INJECTION ← Loop Entry Policy
    for (int i = 0; i < NUM_RAND; i++) // 50 searches {        Interval 2
        double local_sum = 0;
        bst *iterator = tree;
        /* Loop, depth = 2 */
        while (iterator != NULL){                              Interval 3
            double curr_value = iterator->value;
            local_sum += curr_value;

            if (searches[i] == curr_value) { break; }
            else if (searches[i] > curr_value) {
                iterator = iterator->right;
            } else { iterator = iterator->left; }

            // INJECTION ← Loop Latch Policy
            if (iter_count_depth_2 > 20) {
                nk_time_hook_fire();
                iter_count_depth_2 = 0;
            }
            iter_count_depth_2++;
        }
        // INJECTION ← Loop Exit Policy
        if (iter_count_depth_2 > (γ * 20) {
            nk_time_hook_fire();
        }
        total_sum += local_sum;
        // INJECTION ← Loop Latch Policy
        if (iter_count_depth_1 > 100) {
            nk_time_hook_fire();
            iter_count_depth_1 = 0;
        }
        iter_count_depth_1++;
    }
    // INJECTION ← Loop Exit Policy
    if (iter_count_depth_1 > (γ * 100) {                       Interval 4
        nk_time_hook_fire();
    }
    return total_sum;
}
```

Fig. 5. Result of transformation of binary search tree traversal benchmark.

for compiler-based timing. This is because our analyses care about the *sum* of the latencies of a *sequence* of LLVM IR instructions. Inaccurate estimates introduce random errors, resulting in a cancelling effect when summing them. In other words, the central limit theorem works in our favor.

The latency of a call instruction does not include the callee's latency. This allows us to handle instructions whose callees are unknown at compile-time (e.g., indirect calls). Excessive estimation errors are not incurred because *all* functions are transformed to include calls to `nk_time_hook_fire()` and, therefore, the callee will also have calls to `nk_time_hook_fire()`. While estimation

errors on par with the granularity target are still possible, this is not problematic in practice based on our empirical results.

Data-flow analysis: Our data-flow analysis is an intra-procedural forward analysis designed for our goal: estimating the latency of a sequence of IR instructions. We define the data-flow values (i.e., the accumulated latency) and the data-flow equations (i.e., how individual IR instruction latencies accumulate) to measure and calculate accumulated latency just before and after every IR instruction of the compiled code. In particular, data-flow values are integers that represent the clock cycle latencies. These values are used in data-flow equations to compute the accumulated latencies. To this end, we define the GEN set [30] of an individual instruction I (GEN[I]) as its standalone latency determined as described above. Hence, GEN[I] represents the contribution of the instruction I to the accumulated latency. As in typical data-flow analysis [30], our data-flow equations define the IN and OUT sets of an instruction I to represent, in our case, the accumulated latency just before and just after I respectively. In a straight line of code where an instruction I has only one predecessor P, the IN set is IN[I] = OUT[P]. In other words, the accumulated latency just before I is the same one as just after P. The OUT set of an instruction I is OUT[I] = IN[I] + GEN[I]. In other words, the accumulated latency just after I is that just prior to I plus the latency of I. This is sufficient for straightline control flow such as in Intervals 0 and 1 of Figure 5.

Handling more complex control flow only affects IN[I] as there can be multiple predecessors. For example, Interval 3 can be reached after two updates of `iterator` in the loop body or just after the `iterator` definition before the `while` loop). In this case, we compute the *expected accumulated latency* by calculating the mean of the accumulated latencies between all predecessors, or, more generally, the unweighted expectation. Here, the IN set is redefined as IN[I] = $\sum_k$ OUT[P_k] $\cdot$ Pr[P_k] ($\forall k \in P$, the set of predecessor instructions to I), where the probability distribution over the set of predecessors, Pr[P_k], is uniform. Notice that this equation simplifies to the previous definition of IN[I] when there is only a single predecessor.

The choice of a uniform distribution over predecessors is simple, and leaves room for improvement. For example, branch weight analysis could be used to produce a more accurate distribution over predecessors, but would demand profile-based analysis. We employ the simple design because profile-based analysis is quite challenging here. In fact, we are unaware of *any* infrastructure that would allow LLVM's (or other compilers') profilers to handle kernel code.

The propagation policy could vary by use case. For example, when stricter guarantees are needed, using a maximum instead of an weighted or unweighted average would result in a more conservative timing and therefore might be a better fit.

Interval analysis: Interval analysis uses the expected accumulated latencies (described above) to decide where timing events (`nk_time_hook_fire()`) should occur. This effectively divides the codebase in intervals.

We use the term single entry single exit (SESE) [25] to precisely define the concept of intervals. It is important to understand that SESEs are a general concept that subsumes familiar control flow structures, including loops with their exit blocks, sequences of basic blocks, if-then-elses (including their join basic blocks), and others. The definition of a SESE is a control structure where the entry point dominates, the exit point post-dominates everything else in the SESE, and every cycle containing the former also contains the latter and vice versa. We divide the control flow graph (CFG) of a function into a hierarchy of SESEs. Finally, the intervals are the selected SESEs that will end with a call to `nk_time_hook_fire()`.

To determine the intervals, our analysis starts with the entry point of the function (the entry of the outermost SESE). During the traversal we maintain a set of *interval points*, which are instructions (and their latencies) that mark the ends of the selected intervals. At the start of the traversal, the interval point set contains just the entry-point instruction of an SESE. As the traversal proceeds, we measure the accumulated distance between the current instruction and the instructions in the interval point set. When this exceeds the timer resolution (as it happens at the definition of `rand_fp_final` of Figure 5), we promote the current instruction to the interval point set, thus selecting an interval.

Interval analysis handles instructions with multiple predecessors similarly to our data flow analysis. We use the same propagation strategy to do this: the interval point value computed for an instruction with multiple predecessors is the mean of the interval point values of its predecessors.

Loop analysis and transformations: An SESE that contains at least one loop requires additional attention because we want all loops to be preemptable. It is often impossible at compile-time to determine the number of iterations that a given loop will have at run-time. For example, it is not possible to know at compile time how many iterations the loop of Interval 3 of Figure 5 will have at run-time as it depends on the tree being traversed, which is input dependent. To make all loops preemptable, our loop analysis considers injecting calls to `nk_time_hook_fire()` at three additional points: the entry, the exit, and in the *latch* of the outermost loop of such an SESE. The latch is the basic block that jumps to the loop condition from within the loop (i.e., the source of the backedge of the loop).

We analyze an SESE that contains loops in isolation from the innermost to the outermost. An inner SESE with loops may have injected calls to `nk_time_hook_fire()` at its boundary. Consequently, we must zero the accumulated latency from the perspective of the enclosing SESE.

Latch: We inject code to invoke `nk_time_hook_fire()` periodically throughout the iterations of a loop that will be executed at run-time. To decide the instruction periodicity as well as the best code injection technique to use for such periodicity, we employ the following strategy.

We use the outcome of our data-flow analysis to determine the latency of each loop iteration. Such latency is computed as the latency of the outermost SESE included in the target

loop (this SESE is often informally called "the loop body"). The ratio between the target timer resolution and the loop iteration latency ($r = \frac{granularity}{loop\ latency}$), is used to choose an appropriate transformation from among the following to handle the backedge: (a) direct injection, (b) loop unrolling, and (c) branch injection.

Direct injection is selected if $r < 1$, indicating that the loop latency is actually greater than the granularity. In this scenario, the callback function needs to execute for every iteration of the loop. Hence, we simply use the previously described interval analysis to determine the appropriate instructions to mark for injection inside the loop.

Loop unrolling is selected if $1 \leq r \leq \beta$. In this scenario, the loop is unrolled r times and the last instruction is marked for injection, effectively forcing the loop to execute the callback after every r loop iterations.

Branch injection is selected if $r > \beta$. In this scenario, the transform changes the code to execute `nk_time_hook_fire()` only every r iterations. This is accomplished by injecting a conditional branch at the end of a loop iteration that checks the current iteration number (as shown at the end of the loop body of Interval 3 in Figure 5). This approach requires us to know at run-time how many iterations have been executed since the last time the above branch was taken. We add an iteration-counting variable C to the loop, which we zero each time the branch is taken.

The threshold β is heuristically determined to avoid several potential issues: thrashing of the instruction cache, thrashing of the branch predictor, code bloat, and increasing compilation time. Specifically, for a loop with small r, the loop body can be unrolled without generating code bloat, increasing compilation time, negatively affecting the branch predictor, or placing undue strain on the instruction cache. And, as a definite plus, no additional branch is introduced. On the other hand, for a loop with a large r, any or all of these drags on performance could occur if the loop were to be unrolled. In such a case, even though a branch is introduced, branch injection is preferable because the transform can be done much faster than unrolling, the code size will not increase significantly, and the instruction cache behavior will be minimally affected. Note further that as β increases, the injected branch becomes increasingly biased, meaning that the branch predictor is more likely to predict it correctly. This ameliorates the negative effects of branch injection. In this paper, we use $\beta = 12$.

Entry: Consider a loop that begins just before the last interval of the parent SESE ends. In this scenario, the next callback will occur only after some of the loop is executed, potentially creating a gap larger than the granularity. To reduce the likelihood of this occurrence we inject a call to the entry of SESEs that contain loops (i.e., in their pre-headers).

These entry calls can, however, result in unintended additional overhead. This can be avoided when the timer granularity is large enough and small loops are the target. Hence, we do not inject entry calls for such loops in that scenario.

Exit: Timing inaccuracies can emerge for loops that exit in the middle of their body (this is the result of `break` at the source code level) or when the number of iterations a loop executes is not a multiple of the parameter r described above. To avoid such inaccuracies, we inject a call to `nk_time_hook_fire()` at the exit of SESEs that contain loops. We inject such a call within a new conditional branch we add to execute `nk_time_hook_fire()` only when $\frac{C}{r} > \gamma$. In this paper, we use $\gamma = 0.8$.

B. Kernel-side: time hooking

Within the kernel, compiler-based timing is visible via an interface, the *time hook* interface, that is designed to be similar to that of a traditional hardware timer's device driver interface. Time hooking is available shortly after boot. Any kernel component can use the interface through the following two steps. First, it requests the compiler-based timing resolution, which is a compile-time parameter that the code transformations targeted. All times and resolutions are in cycles. The kernel component's next step is to register a callback function (or object). As part of registration, the kernel component also supplies the time at which the callback function will first be invoked, and whether it will continue to be invoked periodically. The invocation time/period is limited by the timer resolution. The registration process can also scope the callback to a group of CPUs. The result is analogous to a per-CPU one-shot or periodic timer that is now active.

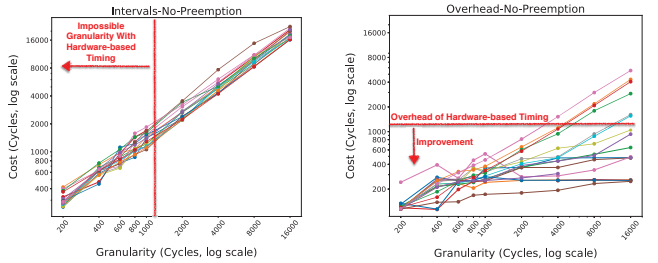
From this point on, the callback function will be invoked whenever its invocation time/period is reached. These callbacks are ultimately due to the compiler-injected invocations of the time hook interface's trigger function, `nk_time_hook_fire()`, which serves essentially as the analog of a per-CPU timer interrupt handler. The trigger function invokes all expired callbacks on the CPU. Because the trigger function maintains its own notion of time (via the cycle counter), it can filter out early callbacks that can occur when the compiler-side transformations were overly conservative.

Just as with an interrupt handler, a callback function can be invoked in almost any context. It is the responsibility of the implementor of the callback function to determine the context, if necessary. It is also the responsibility of the implementor to be aware that the callback function can be preempted. The callback function can selectively mask invocations of `nk_time_hook_fire()`, similar to interrupt masking, although here it is a compile-time option.

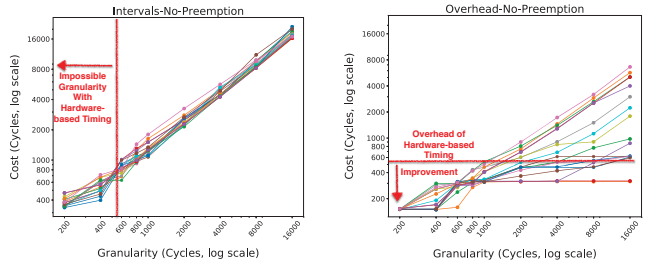
C. Performance

A key difference between hardware-based timing and compiler-based timing is that the latter involves *only* function calls. No expensive control flow via interrupt dispatch is used. Consequently, the overhead of compiler-based timing is lower.

`nk_time_hook_fire()` has been carefully crafted to avoid all cross CPU synchronization (it only needs to guard against reentrancy on the same CPU), and thus has very low overhead that is independent of scale. At 64 cores on KNL, the overhead is 148 cycles, with very little variance. At 8 cores on R415, the overhead is 224 cycles, again with minimal variance. Recall that an interrupt dispatch latency on these machines is

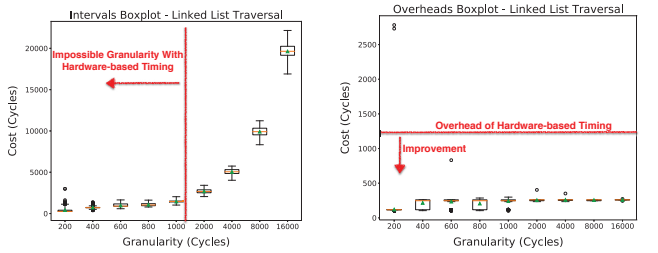


(a) Timing Accuracy (KNL) (b) Overhead (KNL)



(c) Timing Accuracy (R415) (d) Overhead (R415)

Fig. 6. Average timing accuracy and overhead for all benchmarks on both platforms. Each series is a different benchmark.

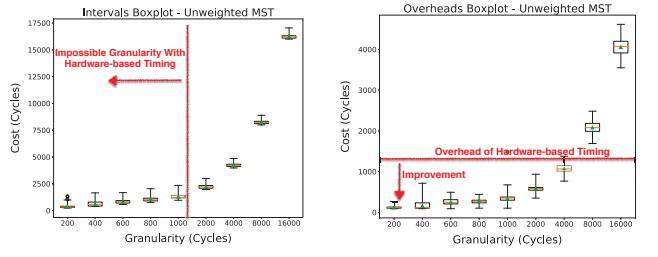


(a) Timing Accuracy (b) Overhead

Fig. 7. Timing accuracy and overhead for linked list traversal (KNL).

on the order of 1100 cycles. The latencies here are about 10x lower because no interrupts are involved.

How accurate is the timing produced by the compiler transforms and the runtime, and what is its overhead? We evaluated this on KNL and R415 for every benchmark listed in Section II, for target timer resolutions of 200, 400, 600, 800, 1000, 2000, 4000, 8000, and 16000 cycles. Here, the registered callback function simply records the time at which it was invoked. Figure 6 shows the overall average results across the benchmarks and platforms. It is generally the case



(a) Timing Accuracy (b) Overhead

Fig. 8. Timing accuracy and overhead for minimum spanning tree (KNL).

that we meet the timer resolution, but the overhead can vary.

We now draw illustrative examples from the KNL results. Figure 7 shows an example for linked list traversal. The target resolution is met, with low variance, down to the 200 cycle limit. Furthermore, the overhead has low variance and is independent of the target resolution. In some situations, however, the compiler transforms result in early callbacks, which must repeat, and this then increases overhead. Figure 8 shows an example of this situation.

Generalizing, the system can achieve timing accuracy down to resolutions of about 200 cycles on KNL. This is limited by the cost of the `nk_time_hook_fire()` runtime. The traditional timing model would have a similar function, driven by interrupts. The compiler-based timing model can achieve a timer resolution that is $(1100 + 148)/200 = 6.2$x better than the traditional model. On R415 it is 2.8x better.

VI. Compiler Timing-Based Preemptive Fibers

We applied compiler-based timing (Section V) to our cooperative fibers implementation (Section IV). In this implementation, the programmer never needs to explicitly invoke `nk_fiber_yield()` to cause a context switch, although he can still elect to do so.

A. Implementation

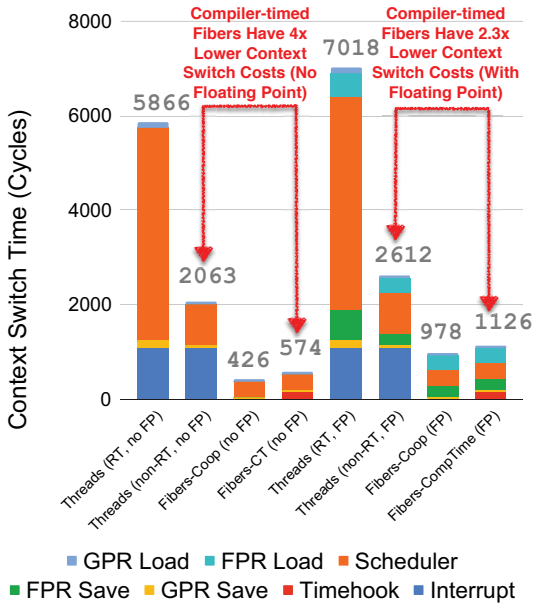
At boot time, the fibers implementation uses the time hook interface to register a single callback function that is scoped to run on all CPUs with a boot-time selected periodicity. This function will then be invoked periodically on each CPU, playing the role that a timer interrupt with the same periodicity would traditionally play.

The callback function proceeds only if the following conditions are true: (1) the CPU is not in interrupt context, (2) the CPU is currently running its fiber thread, and (3) the currently running fiber on the CPU is not the idle fiber. If these are true, then the callback function has been invoked from some active fiber. It is important to understand that this means that we must be in the middle of a simple `call` from that fiber. Therefore, it is perfectly valid to invoke `nk_fiber_yield()`, the cooperative scheduler, on that fiber's behalf. This is precisely what the callback function does. The consequence is a context switch to a different active fiber, if one exists. The idle fiber has `nk_time_hook_fire()` masked, but it is engineered to yield in a controlled manner.

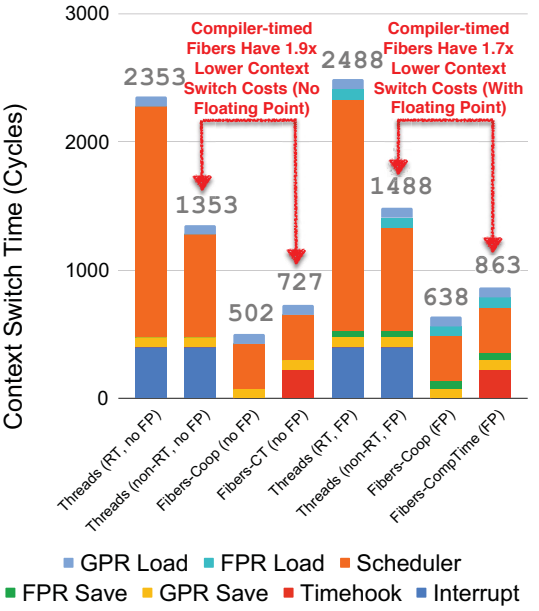
This functionality adds what is, in effect, preemption to the fibers implementation. The programmer no longer needs to explicitly/cooperatively give up the CPU by himself calling `nk_fiber_yield()`. Implementing this pseudopreemption functionality required fewer than 100 lines of C code because the heavy lifting is done by compiler-based timing.

B. Performance

Figure 9 illustrates the costs of thread and fiber context switches within Nautilus, and breaks these costs down to their constituant parts: the interrupt or time hook invocation costs, the costs of saving and loading the general purpose



(a) KNL

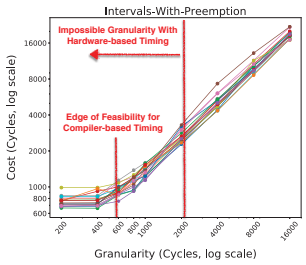


(b) R415

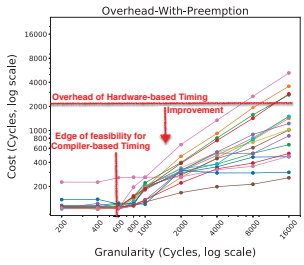
Fig. 9. Cost of context switches for real-time and non-real-time threads, fibers, and compiler-timed fibers on both platforms.

registers (GPRs) and floating point registers (FPRs), and the cost of the scheduler, which decides on the next thread or fiber. For threads, we consider both real-time scheduling and non-real-time scheduling, which have substantially different context switch costs due to the added complexity of the real-time scheduler. The non-real-time scheduler for threads and fibers is essentially identical—it is a constant time round-robin scheduler. A substantial cost that both threads and fibers may bear is FPR save/load. We show results with and without FPRs.

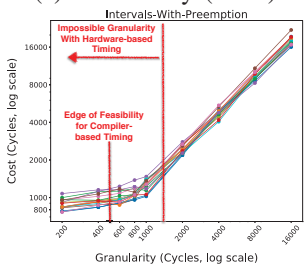
On KNL, the cost of a non-real-time preemptive thread context switch is about 2063 cycles without FPRs, and 2612



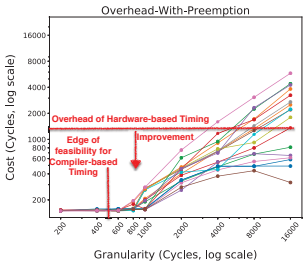
(a) Granularity (KNL)



(b) Overhead (KNL)

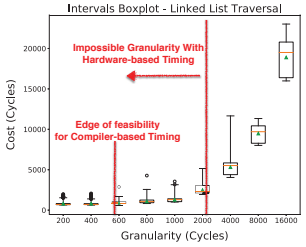


(c) Granularity (R415)

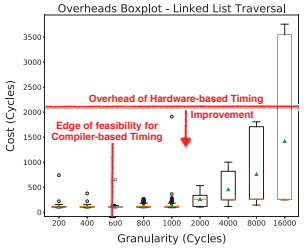


(d) Overhead (R415)

Fig. 10. Average preemptive fiber granularity and overhead for all benchmarks on both platforms. Each series is a different benchmark.



(a) Granularity



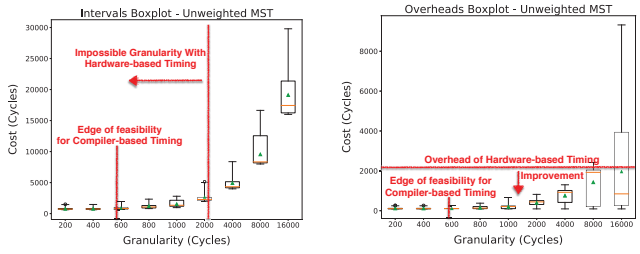
(b) Overhead

Fig. 11. Preemptive fiber granularity and overhead for linked list traversal. No floating point. KNL.

cycles with FPRs. For comparison, a cooperative fiber context switch is 426 cycles without FPRs and 978 cycles with FPRs. That is, the cost of a cooperative fiber context switch is between 2.6x (with FPRs) and 4.8x (without FPRs) lower than that of the preemptive thread context switch. On R415, the factors are 2.3x (with FPRs) and 2.6x (without FPRs).

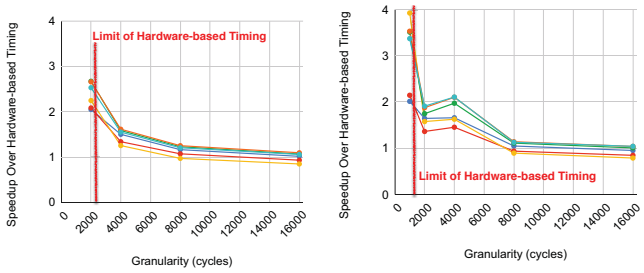
Adding preemption to fibers via compiler-based timing introduces the benefits of preemption, but increases overhead because of the cost of the time hook infrastructure processing. Without FPRs, a pseudopreemptive fiber context switch on KNL requires 574 cycles, and with FPRs, 1126 cycles. That is, the cost of a preemptive fiber context switch, enabled by compiler-based timing, is between 2.3x (with FPRs) and 4x (without FPRs) lower than that of a preemptive thread context switch initiated by hardware-based timing. On R415, the factors are 1.7x (with FPRs) and 1.9x (without FPRs).

Can fiber preemption via compiler-based timing achieve a reliable and stable scheduling granularity/quantum? Paralleling the methodology of Section V-C, we evaluated this on KNL and R415. For each benchmark, two copies were run on each CPU and the callback function context switched between



(a) Granularity (b) Overhead

Fig. 12. Preemptive fiber granularity and overhead for minimum spanning tree. No floating point. KNL.



(a) KNL (b) R415

Fig. 13. Speedup of compiler-based timing over hardware-based timing for slicing the PARSEC Streamcluster benchmark to different granularities on both platforms. Each series is a different problem size.

them. Figure 10 illustrates the average granularity targeted and achieved, as well as the average overhead, across all benchmarks and both platforms.

We focus here on KNL. Figures 11 and 12 show the achievable granularities and corresponding overheads when scheduling fibers for the same two example benchmarks used in Figures 7 and 8. Note that in both cases stable scheduling granularities down to the limit implied by the overhead (574 cycles) are possible. For finer granularities than the "edge of feasibility", the system is gracefully degrading (the achieved quantum does not get smaller). Generalizing, the system can preemptively schedule fibers with a quantum that approaches 600 cycles (no floating point) or 1200 cycles (with floating point). On R415, the limits approach 730 cycles (no floating point) and 870 cycles (with floating point)

Application benchmark: We sliced a sequential version of the PARSEC Streamcluster benchmark [6] to different granularities using both hardware-based and compiler-based timing and then compared the wall-clock time of the benchmark between the two methods. Figure 13 shows the results. The smaller the granularity, the greater the effect on performance, up to 3-4x as we approach the finest granularities possible with compiler-based timing.

VII. Related work

Concurrency abstractions geared to achieving fine granularity have been an active topic of research for quite some time. Examples include QThreads [56], MassiveThreads [39], StateThreads (http://state-threads.sourceforge.net), Tiny Theads[12], the generalized primitives of Lithe [43], the run-time system of Intel's Thread Building Blocks [48], the Converse run-time underlying Charm++ [28]), MPC [44], the Realm event run-time system underlying Legion [54], Light-Weight Contexts [36], and ARGObots [50], Our work includes fibers, similar to some of these systems, but we add preemption to fibers via compiler-based timing.

Some implementations of concurrency abstractions either are a target for the compiler, such as Maestro [46] and Nanos++ (https://github.com/bsc-pm/nanox), or extend the concurrency abstraction into the compiler to allow optimization, such as Tapir [49] and OpenMPIR [51]. In contrast, we use a compiler transform as the mechanism for creating timing events that drive a concurrency or scheduling abstraction, such as fibers. Heartbeat scheduling [2] is an abstraction that demands stable timing events that could be driven by compiler-based scheduling.

Perhaps closest to our work is the classic work by Feeley on balanced polling [15], StackThreads/MP [53], [52], and TAM [11]. Balancing polling uses compilation to introduce parallel scheduling calls at function boundaries. However, the transformation is limited, and can result in high overhead. StackThreads/MP uses minimal compiler support to allow its run-time to extend the C stack abstraction for fine-grain threads. This can include what is effectively a context switch when a fine-grain thread blocks. Finally, TAM includes a compiler to decide the schedule of threads within a CPU with the goal of maximizing hardware utilization and/or data locality. None of these systems attempts to produce timing behavior that can substitute for a hardware timer, as is the goal with compiler-based timing.

VIII. Conclusions and future work

We proposed a new approach to timing, compiler-based timing, that has the potential to replace traditional timing based on hardware timer interrupts. Compiler-based timing, which is enabled by modern compiler analysis and transformation, can offer far lower overheads and finer granularity than traditional timing, and this can in turn lead to finer granularity tasks within parallel systems. Our prototype implementation, which is based on LLVM-hosted transformations of the 331K+ line Nautilus kernel, can provide timer resolutions down to 200 cycles on the Intel Xeon Phi, which is about 6.2x better than is possible with hardware-based timing. We applied compiler-based timing to add preemption to Nautilus's fast cooperative threading system, resulting in context switch costs (and thus task granularities) that are up to 4x better than is possible with hardware-timing-based preemption. A more traditional machine produces similar results.

The concept of compiler-based timing could also be applied at user-level within a general purpose OS. In this approach, instead of avoiding the overhead of timer interrupts within the kernel, the overhead of timer alarm signals, triggered by timer interrupts, are to be avoided. We are developing a variant of compiler-based timing for user-level execution, and we are in the process of applying both the kernel-level and user-level variants to heartbeat scheduling (described in Section VII).

REFERENCES

[1] ABRAHAM, M. J., MURTOLA, T., SCHULZ, R., PÁLL, S., SMITH, J. C., HESS, B., AND LINDAHL, E. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX 1* (2015), 19–25.

[2] ACAR, U. A., CHARGUÉRAUD, A., , GUATTO, A., RAINEY, M., AND SIECZKOWSKI, F. Heartbeat scheduling: Provable efficiency for nested parallelism. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2018).

[3] ARTEAGA, J., ZUCKERMAN, S., AND GAO, G. R. Multigrain parallelism: Bridging coarse-grain parallel programs and fine-grain event-driven multithreading. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2017).

[4] AYGUADE, E., COPTY, N., DURAN, A., HOEFLINGER, J., LIN, Y., MASSAIOLI, F., TERUEL, X., UNNIKRISHNAN, P., AND ZHANG, G. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems 20*, 3 (2009), 404–418.

[5] BAUER, M., TREICHLER, S., SLAUGHTER, E., AND AIKEN, A. Legion: Expressing locality and independence with logical regions. In *Proceedings of Supercomputing (SC 2012)* (Nov. 2012).

[6] BIENIA, C. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[7] BLELLOCH, G. E., AND GREINER, J. A provable time and space efficient implementation of NESL. In *Proceedings of the International Conference on Function Programming (ICFP)* (May 1996).

[8] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing 37*, 1 (1996), 55–69.

[9] CHAPMAN, B., JOST, G., VAN DER PASS, R., AND KUCK, D. *Using OpenMP: Portable Shared Memroy Parallel Programming*. MIT Press, 2007.

[10] CHEN, J., JUANG, P., KO, K., CONTRERAS, G., PENRY, D., RANGAN, R., STOLER, A., PEH, L.-S., AND MARTONOSI, M. Hardware-modulated parallelism in chip multiprocessors. *SIGARCH Comput. Archit. News 33*, 4 (Nov. 2005), 5463.

[11] CULLER, D. E., SAH, A., SCHAUSER, K. E., VON EICKEN, T., AND WAWRZYNEK, J. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1991).

[12] DEL CUVILLO, J., ZHU, W., HU, Z., AND GAO, G. R. Tiny threads: a thread virtual machine for the cyclops64 cellular architecture. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2005).

[13] DINDA, P., WANG, X., WANG, J., BEAUCHENE, C., AND HETLAND, C. Hard real-time scheduling for parallel run-time systems. In *Proceedings of the 27th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC)* (June 2018).

[14] DURAN, A., CORBALAN, J., AND AYGUADE, E. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC 2008)* (2008).

[15] FEELEY, M. A message passing implementation of lazy task creation. In *Parallel Symbolic Computing* (1992), pp. 94–107.

[16] FOG, A. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. Tech. rep., Copenhagen University College of Engineering, 2019.

[17] FORBES, E., AND ROTENBERG, E. Fast register consolidation and migration for heterogeneous multi-core processors. In *Proceedings of the 34th IEEE International Conference on Computer Design (ICCD 2016)* (2016), pp. 1–8.

[18] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI 98)* (1998), pp. 212–223.

[19] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization (WWC-4)* (2001), pp. 3–14.

[20] HALE, K. *Hybrid Runtime Systems*. PhD thesis, Northwestern University, August 2016. Available as Technical Report NWU-EECS-16-12, Department of Electrical Engineering and Computer Science, Northwestern University.

[21] HALE, K., AND DINDA, P. Enabling hybrid parallel runtimes through kernel and virtualization support. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2016)* (April 2016).

[22] HALE, K., AND DINDA, P. An evaluation of asynchronous software events on modern hardware. In *Proceedings of the 26th IEEE International Symposium on the Modeling, Analysis, and Simulaton of Computer and Telecommunication Systems (MASCOTS 2018)* (September 2018).

[23] HALE, K. C., AND DINDA, P. A. A case for transforming parallel runtime systems into operating system kernels (short paper). In *Proceedings of the 24th International ACM Symposium on High Performance Parallel and Distributed Computing, (HPDC 2015)* (June 2015).

[24] HENRIKSEN, T., SERUP, N., ELSMAN, M., HENGLEIN, F., AND OANCEA, C. Futhark: Purely functional gpu programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2017).

[25] JOHNSON, R., PEARSON, D., AND PINGALI, K. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation* (1994), pp. 171–185.

[26] KAISER, H., BRODOWICZ, M., AND STERLING, T. ParalleX: An advanced parallel execution model for scaling-impaired applications. In *Proceedings of the 38^{th} International Conference on Parallel Processing Workshops (ICPPW 2009)* (Sept. 2009), pp. 394–401.

[27] KALÉ, L. V., RAMKUMAR, B., SINHA, A., AND GURSOY, A. The Charm parallel programming language and system: Part II–the runtime system. Tech. Rep. 95-03, Parallel Programming Laboratory, University of Illinois at Urbana-Champaign, 1994.

[28] KALE, L. V., YELON, J., AND KNAUFF, T. Threads for interoperable parallel programming. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC '97)* (1996), pp. 534–552.

[29] KOCHER, P., HORN, J., FOGH, A., , GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)* (2019).

[30] LAM, M., SETHI, R., ULLMAN, J., AND AHO, A. Compilers: Principles, techniques, and tools. *Pearson Education* (2006).

[31] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (2004), IEEE, pp. 75–86.

[32] LAUDERDALE, C., AND KHAN, R. Towards a codelet-based runtime for exascale computing. In *Proceedings of the 2^{nd} International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT 2012)* (Mar. 2012), pp. 21–26.

[33] LEYDEN, J., AND WILLIAMS, C. Kernel-memory-leaking intel processor design flaw forces linux, windows redesign. *The Register* (January 2018).

[34] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)* (2018).

[35] LIS, M., KEUN SUP SHIM, CHO, B., LEBEDEV, I., AND DEVADAS, S. Hardware-level thread migration in a 110-core shared-memory multiprocessor. In *2013 IEEE Hot Chips 25 Symposium (HCS)* (2013), pp. 1–27.

[36] LITTON, J., VAHLDIEK-OBERWAGNER, A., ELNIKETY, E., GARG, D., BHATTACHARJEE, B., AND DRUSCHEL, P. Light-weight contexts: An os abstraction for safety and performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI 2016)* (November 2016).

[37] LIU, J. W. S. *Real-Time Systems*. Prentice Hall, April 2000.

[38] MARTORELL, X., AYGUADÉ, E., NAVARRO, N., CORBALÁN, J., GONZÁLEZ, M., AND LABARTA, J. Thread fork/join techniques for multi-level parallelism exploitation in numa multiprocessors. In *Proceedings of the 13th International Conference on Supercomputing (ICS)* (1999), pp. 294–301.

[39] NAKASHIMA, J., AND TAURA, K. *MassiveThreads: A Thread Library for High Productivity Languages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 222–238.

[40] NVIDIA CORPORATION. Dynamic parallelism in CUDA, Dec. 2012.

[41] OPENMP ARCHITECTURE REVIEW BOARD. Openmp application program interface 3.0. Tech. rep., OpenMP Architecture Review Board, May 2008.

[42] OUSTERHOUT, J. Why threads are a bad idea (for most purposes). Invited presentation at USENIX ATC 1996, 1996. (widely cited).

[43] PAN, H., HINDMAN, B., AND ASANOVIĆ, K. Composing parallel software efficiently with lithe. In *Proceedings of the 31st ACM Conference on Programming Language Design and Implementation (PLDI)* (June 2010).

[44] PÉRACHE, M., JOURDREN, H., AND NAMYST, R. Mpc: A unified parallel runtime for clusters of numa machines. In *Proceedings of the 2008 European Conference on Parallel Processing (EuroPar)* (2008), pp. 78–88.

[45] PIETREK, M. Happy 10th anniversary, windows 3.0. *MSDN Magazine* (July 2000).

[46] PORTERFIELD, A., NASSAR, N., AND FOWLER, R. Multi-threaded library for many-core systems. In *2009 IEEE International Symposium on Parallel & Distributed Processing* (2009), pp. 1–8.

[47] RAVITCH, T. https://github.com/travitch/whole-program-llvm, 2016.

[48] REINDERS, J. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[49] SCHARDL, T., MOSES, W., AND LIESERSON, C. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2017)* (January 2017).

[50] SEO, S., AMER, A., BALAJI, P., BORDAGE, C., BOSILCA, G., BROOKS, A., CARNS, P., CASTELL, A., GENET, D., HERAULT, T., IWASAKI, S., JINDAL, P., KAL, L. V., KRISHNAMOORTHY, S., LIFFLANDER, J., LU, H., MENESES, E., SNIR, M., SUN, Y., TAURA, K., AND BECKMAN, P. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems 29*, 3 (2018), 512–526.

[51] STELLE, G., MOSES, W. S., OLIVIER, S. L., AND MCCORMICK, P. Openmpir: Implementing openmp tasks with tapir. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC* (2017).

[52] TAURA, K., TABATA, K., AND YONEZAWA, A. Stackthreads/mp: integrating futures into calling standards. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '99)* (1999), pp. 60–71.

[53] TAURA, K., AND YONEZAWA, A. Fine-grain multithreading with minimal compiler support-a cost effective approach to implementing efficient multithreading languages. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '97)* (1997), pp. 320–333.

[54] TREICHLER, S., BAUER, M., AND AIKEN, A. Realm: An event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT 2014)* (2014), p. 263276.

[55] VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: Scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)* (October 2003).

[56] WHEELER, K. B., MURPHY, R. C., AND THAIN, D. Qthreads: An api for programming with millions of lightweight threads. In *Proceedings of the 2nd Workshop on Multithreaded Architectures and Applications (MTAAP 2008, colocated with IPDPS 2008)* (2008).

[57] ZHAO, F., AND HAHNENBERG, M. Decoupling software pipelining in LLVM. Project Report, CMU 15-745, and Code Repo, June 2011. https://code.google.com/archive/p/15745-project-dswp/.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

The prototype system described in the paper is implemented as an extension of the LLVM 9 compiler framework and the Nautilus kernel framework.

LLVM 9 is widely available (https://github.com/llvm/llvm-project), and has a UIUC/BSD-style license or Apache 2.0 license.

WLLVM is widely available (https://github.com/travitch/whole-program-llvm), and has an MIT license. We make no changes.

Nautilus is publicly available (https://github.com/HExSA-Lab/nautilus), and has an MIT license. Changes for compiler-based timing are in the compiler-timing branch of https://github.com/PeterDinda/nautilus

Microbenchmarks are influenced/partially incorporate Michigan's mibench (http://vhosts.eecs.umich.edu/mibench//), which is in the public domain. Our offshoots are MIT licensed.

Most experiments were run on a Colfax Ninja Xeon Phi server, which includes a 1.3 GHz Intel Xeon Phi 7210 (64 cores, 256 hardware threads) mated to 16 GB of MCDRAM and 96 GB of DRAM. Dell R415s and R815s were also used for some minor elements of the paper.

Experiments consisted of:

1. Timing and overhead measurements of Nautilus's thread and new fiber implementations
2. Timing and overhead measurements of interrupt-driven thread preemption
3. Timing and overhead measurements of the compiler-based timing approach as a replacement for hardware-based timer interrupts
4. Timing and overhead measurements of the compiler-based timing approach to add preemption to the fibers implementation

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID: https://github.com/PeterDinda/nautilus
Artifact name: Nautilus Kernel (compiler-timing
↪ branch on this fork)

Persistent ID: https://github.com/llvm/llvm-project
Artifact name: LLVM compiler framework (specifically
↪ version 9)

Persistent ID: https://github.com/embecosm/mibench
Artifact name: mibench suite (influential, several
↪ used)

Persistent ID:
↪ https://github.com/travitch/whole-program-llvm
Artifact name: WLLVM

Persistent ID: 10.5281/zenodo.3879742   /
↪ https://zenodo.org/record/3879742
Artifact name: Compiler-Based Timing System - MAIN
↪ ARTIFACT
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Colfax Ninja Xeon Phi server, which includes a 1.3 GHz Intel Xeon Phi 7210 (64 cores, 256 hardware threads) mated to 16 GB of MCDRAM and 96 GB of DRAM. Dell R415s and R815s were also used for some minor elements of the paper.

Operating systems and versions: Nautilus Kernel, compiler-timing branch

Compilers and versions: LLVM 9 and associated front-ends

Applications and versions: Mibench (some), mostly an influence

Libraries and versions: none

Key algorithms: dot product, linked list traversal, matrix multiply, bst, randomized matrix multiply, level order tree traversal, randomized fp operations, Rijndael, md5, sha-1, randomized fibonacci computation, knn, cycle detection, unweighted mst, dijkstra, qsort, radix sort

Input datasets and versions: generated by benchmark algorithms

URL to output from scripts that gathers execution environment information.

`http://pdinda.org/Stuff/sc20-envs/`