Frequent Background Polling on a Shared Thread, using Light-Weight Compiler Interrupts

Nilanjana Basu
University of Illinois at Chicago
USA

Claudio Montanari
University of Illinois at Chicago
USA

Jakob Eriksson
University of Illinois at Chicago
USA

Abstract

Recent work in networking, storage and multi-threading has demonstrated improved performance and scalability by replacing kernel-mode interrupts with high-rate user-space polling. Typically, such polling is performed by a dedicated core. *Compiler Interrupts* (CIs) instead enable efficient, automatic high-rate polling on a shared thread, which performs other work between polls.

CIs are instrumentation-based and light-weight, allowing frequent interrupts with little performance impact. For example, when targeting a 5,000 cycle interval, the median overhead of our fastest CI design is 4% vs. 800% for hardware interrupts, across programs in the SPLASH-2, Phoenix and Parsec benchmark suites running with 32 threads.

We evaluate CIs on three systems-level applications: (a) kernel bypass networking with *mTCP*, (b) joint kernel bypass networking and CPU scheduling with *Shenango*, and (c) delegation, a message-passing alternative to locking, with *FFWD*. For each application, we find that CIs offer compelling qualitative and quantitative improvements over the current state of the art. For example, CI-based mTCP achieves ≈2× stock mTCP throughput on a sample HTTP application.

CCS Concepts: • **General and reference** → General conference proceedings; Empirical studies; • **Computing methodologies** → *Concurrent programming languages*; *Parallel programming languages.*

Keywords: compiler interrupts, fine-grained thread sharing, control flow graph analysis and transformation, code instrumentation, efficient polling

ACM Reference Format:
Nilanjana Basu, Claudio Montanari, and Jakob Eriksson. 2021. Frequent Background Polling on a Shared Thread, using Light-Weight Compiler Interrupts. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454107

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8391-2/21/06...$15.00
https://doi.org/10.1145/3453483.3454107

1 Introduction

A recent trend in computer systems is the use of high frequency polling to create user-level implementations of a range of tasks traditionally left to the kernel and its interrupt handlers. These include kernel-bypass networking [8, 26, 35, 44] and disk I/O [16, 30, 53], message passing [33, 46, 48], and CPU allocation [28, 41]. A common design choice is to dedicate one or more cores to polling. This allows for high frequency continuous polling, but eliminates the use of these dedicated cores for application purposes, and incurs high CPU utilization even when no productive work is being performed.

We propose *Compiler Interrupts* (CIs), a compile-time instrumentation approach, as an alternative to dedicating a CPU core to polling. With performance in mind, CIs adds a minimum number of "probes" to the target program, which measure the elapsed time and periodically call a user-provided interrupt handler at (or near) the target interval.

The use of instrumentation to periodically perform work outside the normal flow of a program has been explored before. In garbage collected languages [20, 24], collection may be triggered at specific safe points in the program to ensure useful invariants. Profilers [6, 7, 19, 21, 38, 39, 54] and Fuzzers [15, 22, 27] instrument programs to collect and analyze program statistics. In coredet [9], a deterministic instruction count is maintained using compiler instrumentation, to periodically trigger synchronization events.

The Compiler Interrupt is a general-purpose system primitive that provides an easy-to-use and highly efficient approach to fine-grained processor sharing. At runtime, the program registers an interrupt handler to be called at a regular interval throughout the execution of the program. While the instrumentation itself introduces a fixed overhead, the cost per interrupt is very small, as it is implemented as a standard function call. As a result, interrupts may be served as frequently as (approximately) every 1,000 cycles, with minor performance impact, given a short handler.

It should be noted that CI intervals are approximate: large deviations can occur as a result of executing uninstrumented code, including system calls: CIs cannot be raised when the

Table 1. Sketch of a simple program using Compiler Interrupts. The function *handler* is called periodically throughout the execution of the program.

```
volatile int increments = 0;
void handler(uint64_t insns) {
 printf("%llu insns, %d incs\n",
          insns, increments);
}
void main() {
 register_ci(100000,&handler);
 for(;;) increments++;
}
```

program is not executing, or when it is executing uninstrumented code. That said, when accuracy is important, compiling any libraries with CIs, and limiting use of system calls on the subject thread may be a feasible trade-off.

The main contributions of this paper are as follows.

- The introduction of Compiler Interrupts (CIs) as a general purpose primitive in computer systems.
- An efficient CI implementation, based on static analysis and instrumentation of arbitrary programs.
- Evaluation on three recent system use cases, as well as standard benchmark suites.

The remainder of the paper is structured as follows. In §2 we describe the high-level design and API of compiler interrupts. §3 describes the details of our analysis phase, followed by the instrumentation phase in §4. §5 evaluates CIs, both in the context of three example use-cases, and in terms of overhead and interval accuracy on benchmark programs. We then review the related work (§6), and conclude (§7).

2 Compiler Interrupts – Model and API

Conceptually, a *Compiler Interrupt* (CI) is a call to an interrupt handler function that, from the perspective of the compiled executable, is part of the normal program flow. Based on compile-time instrumentation, the call to the interrupt handler is entirely synchronous. However, from the programmer's perspective, the interrupt handler function may be called at any time, unless CIs have been explicitly disabled: the call location is not visible to the programmer, and may vary with program inputs, as well as seemingly unrelated program changes. Below, we discuss the design of Compiler Interrupts in more detail, followed by a description of our analysis and instrumentation phases in §3–§4.

2.1 Usage and Operation

From a user mode programmer's perspective, compiler interrupts are in some ways similar to signals. Table 1 sketches simple example program using the Compiler Interrupt API. The program will print instruction and increment counts, approximately every 100,000 cycles.

Table 2 lists key parts of the API used to interface with CIs. A program may register one or more interrupt handler functions with a target interval in cycles. Calls to the handler functions are interleaved with normal program execution until the handler is de-registered, except where instrumentation is explicitly disabled. The handler function accepts one integer parameter: an approximation of the number of LLVM IR instructions since the last interrupt.

To enable Compiler Interrupts, the program is compiled with our new LLVM CI optimization phase. This instruments the binary with brief interrupt probes (see an example in Table 3) at carefully selected locations. Probes increment the instruction count and, if the count has advanced sufficiently, invoke interrupt handlers. The count is incremented by an approximate (but deterministic) amount, based on the code preceding the probe, computed through static analysis. The details of where probes are inserted, and how the increment is calculated, are provided in §3–§4.

In order to meet a target interrupt period, probes are inserted at a maximum *probe interval*, smaller than the target *interrupt interval*; the *probe interval* is a compile-time configurable parameter whereas *interrupt interval* is specified at runtime using the CI API §2. Generally speaking, the shorter the probe interval, the finer the granularity at which interrupts may be issued. That said, program logic often dictates that probes be inserted relatively frequently. As a result, a long probe interval setting does not necessarily result in greatly reduced overhead. In the probe shown in Table 3, the function **call_handlers()**[1] decides, for each registered and enabled (see §2.2) CI handler, whether it is time to invoke the handler. The function **update_nextint()** computes a suitable interval for the next call to **call_handlers()**.

It is worth noting that compiler instrumentation can only be added to the parts of the program that are compiled with the CI optimization phase. Thus, when a program is executing uninstrumented library functions or system calls, or while it is not executing at all, the count cannot be incremented and CIs cannot be raised, resulting in interval inaccuracy. In §4 we describe more advanced probes which partially address this inaccuracy. We quantify the interval accuracy of CIs on multiple benchmark programs in §5.

2.2 Disabling / Enabling Interrupts

Individual Compiler Interrupts handlers are disabled/enabled using the **ci_disable, ci_enable** functions. To allow nesting of such calls, disable/enable will increment/decrement a thread-local, per-handler count. If this value is greater than zero, the interrupt handler function is not called by the probe. As a result, *"n"* enable calls are needed to enable an interrupt disabled by *"n"* disable calls.

[1]A fast-path is provided for the case of a single registered CI handler.

Table 2. An Application Programming Interface (API) for Compiler Interrupts. In addition to these library functions, compile-time configurations are needed to instrument the executable with compiler interrupt support.

API	Description
typedef void (***ci_handler**)(uint64_t)	*Interrupt handlers receive an approximation of the number of IR executed as argument.*
int register_ci(int interval, **ci_handler** func)	*Register Compiler Interrupt handler func, with a target interval (in cycles). Returns a unique identifier* **ciid**.
void **deregister**(int ciid)	*Deregister the interrupt handler identified by the* **ciid** *provided.*
void **ci_disable**(int ciid)	*Temporarily disable compiler interrupts by the* **ciid** *provided.*
void **ci_enable**(int ciid)	*Compiler interrupts are enabled by default.* **ci_enable** *re-enables interrupts following calls to* **ci_disable** *for the same* **ciid**.
#pragma **ci_probe disable**	*Disables probe instrumentation in the following function.*

Table 3. Sketch of a basic Compiler Interrupt probe, added to strategically selected locations throughout the program. Function **call_handlers()** determines, for each registered handler, whether it is time to call it.

```
__thread uint64_t inscount;
__thread uint64_t nextint;
inline void periodic_probe(uint64_t inc) {
 inscount+=inc;
 if(inscount>nextint) {
  call_handlers(inscount);
  update_nextint();
 }
}
```

The main use of this functionality is in disabling interrupt handler H during execution of H, to avoid indefinitely growing the stack when executing long interrupt handlers. Another important use for disabling interrupts is in locking. Similar to interrupt handlers (kernel), and signal handlers (userland), there is the potential for deadlock between regular program flow, and any CI handlers which acquire locks.

Depending on the application, it may not be practical to avoid acquiring locks in CI handlers. An alternative is to modify the lock implementation to disable all CI handlers during critical sections. This may be done by passing 0 as the *ciid* to **ci_disable/ci_enable**. Here, the trade-off is instead that long critical sections may arbitrarily delay interrupt delivery, which may impact performance or correctness.

The full implications of disabling compiler interrupts, and its impact on software modularity, may require further study.

2.3 Disabling / Enabling Probes

As opposed to disabling/enabling interrupts, the **#pragma ci_probe disable** directive prohibits instrumentation of probes in the immediately following function. The scope is limited to the function it is applied to, and does not get carried over to functions that are called inside the disabled function.

2.4 Compiler Interrupts vs. Signals

In UNIX systems, a process may register signal handlers, to be invoked in response to specific system events, including timers. Using the Linux Performance API (PAPI), one can register a callback function to be called at regular intervals of a hardware performance counter, such as the retired instruction counter. Conceptually, these provide a similar service to Compiler Interrupts. In practice, the differences are stark.

First, Compiler Interrupts rely on instrumentation, which essentially carries fixed overhead per instruction, and single-digit cycle overhead per interrupt delivery, since it is a simple function call. By contrast, performance-counter based signal delivery carries little or no fixed overhead, but high per-interrupt overhead as each signal delivery requires at minimum a hardware interrupt, a return to user space to execute the handler, then a signal return system call. Figure 12 illustrates the high cost of frequent signal delivery, increasing program runtime by almost 10× for interrupt intervals in the 5,000 cycle range. Compiler interrupts, meanwhile, incur extremely low additional overhead per interrupt.

Second, Compiler Interrupts are platform independent, being introduced at the LLVM intermediate representation (IR) level. Thus, they can be portable, and deterministic[2]. By contrast, performance counters are notoriously variable across architectures, often non-deterministic, or even unavailable.

2.5 Compiler Interrupts vs. Yield points

Yield points, described in [1, 2, 4, 31], are well defined points in a program, where execution may be interrupted to facilitate garbage collection, thread preemption, profiling and other virtual machine behaviors. Though similar in concept, yield points were designed with the objective of allowing an outside party to interrupt execution only at safe points in the code, while incurring minimal instrumentation overhead.

[2]In §4, we propose a probe type that uses the cycle counter (**CI-cycles**). This type trades performance, portability, and determinism for added accuracy.

Unlike the microsecond-scale interrupt intervals of CIs, yield points target much less frequent interruption, at millisecond-scale intervals. On the implementation side, yield points rely on external action to "enable" an yield point, informing the thread that it needs to yield, whether by protecting a page, setting a flag variable, or patching code. With CI, threads internally track their progress and decide, without external input, when to call the interrupt handler.

In terms of usage, yield points are very light-weight by design, and liberally added at function preambles, postambles and back-edges. According to [31], only 1 in 20,000 yield points are typically "taken". By comparison, CI probes are somewhat heavy-weight, thus care must be taken to keep the number of probes encountered, low. In our experiments, 1 in 100 CI probes would typically result in an interrupt.

2.6 Modularity

To support modular compilation and CI-instrumented libraries, the CI pass exports required metadata from each build unit to a file, which is read while building any dependent units. For the best accuracy and performance, all builds units should be instrumented using the same probe interval, though this is not necessary for correctness. Code compiled for compiler interrupts must link with the **libci** support library, which provides the CI API.

Below, we describe how our static analysis phase determines the placement and increment of each probe (§3), followed by alternative probe designs (§4) and evaluation (§5).

3 Analysis Phase

The analysis phase takes the pre-optimized intermediate representation (IR) of the program, applies code transformations, and identifies the locations where instrumentation must be added in order to achieve a periodic compiler interrupt interval. This is followed by the instrumentation phase (§4), where probe code is added.

A naïve version of CIs may simply instrument a probe on every basic block. Assuming sufficiently short basic blocks, this would be effective, but not efficient due to the large number of probes invoked at runtime. Therefore, the purpose of our static analysis phase is to minimize the number of probes executed while striving to meet a target interrupt interval. Discussed in greater detail in (§3.3), the placement of probes is dictated by the compile-time configurable parameter of *probe interval* (see §2.1) that specifies the approximate distance between probes in terms of instruction counts. Probes are also placed after uninstrumented external library calls due to their unforeseen instruction count.

During analysis, IR is first pre-processed to canonicalize the control flow graph (CFG), making analysis easier (§3.1). The CFG is then abstracted into a hierarchical graph of self-contained subgraphs (§3.2). The cost (in IR) of each subgraph is then computed (§3.3), and each subgraph is marked for instrumentation as needed given a probe interval accuracy. For long loops, or loops of unknown length, that had been identified in the earlier steps, loop cloning (§3.5) and loop transformation (§3.4) are then applied to reduce the necessary number of probes at runtime. Finally, a post-processing step (§3.6) further reduces the number of instrumentations.

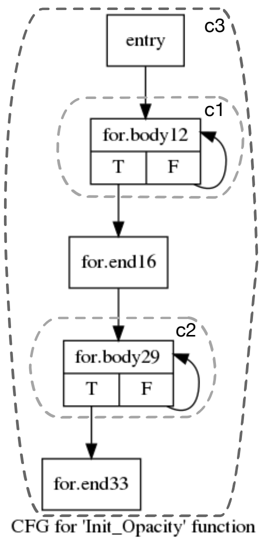

Figure 1. Hierarchical containers created from control flow graph of the `Init_Opacity()` function of Volrend, from the SPLASH-2 benchmark suite

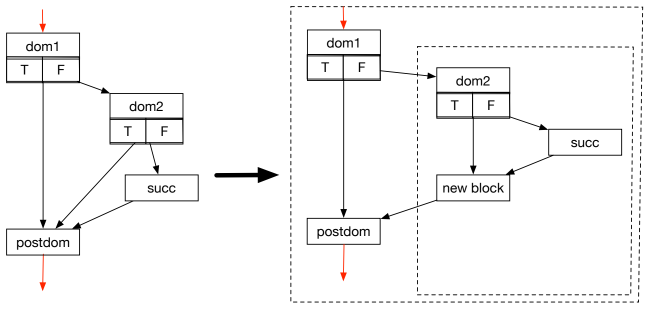

Figure 2. Transforming a CFG into a canonical form.

3.1 Pre-processing - CFG Transform

In order to facilitate the pattern matching rules used in our CFG analysis, we first transform the CFG to a canonical, equivalent form. For example, the CFG on the left of Figure 2 shows a branch pattern in its original form, and the canonical form is shown in the right hand side CFG in Figure 2. This type of transformation is done at a preprocessing stage using well known concepts of graph dominator & post-dominator to find patterns that need transformation.

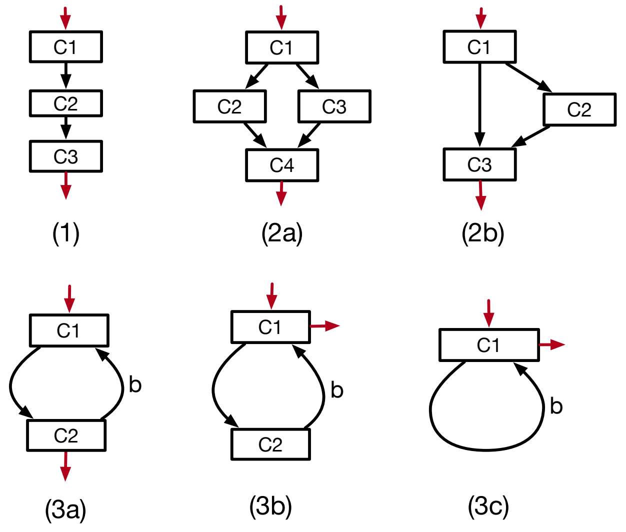

Figure 3. CFG patterns for reduction ('C' denotes container, 'b' denotes backedge count of a loop)

3.2 Abstracting the Control Flow Graph

We use a basic forward-chaining production rule system [47] to detect subgraphs in the CFG of the program, that are amenable to cost estimation. In essence, the system consists of a set of graph matching patterns and associated actions, which are applied to the CFG until progress halts.

Figure 1 represents the control flow graph of `Init_Opacity` function from the *volrend* program of the SPLASH-2 benchmark suite [49, 52]. The CFG corresponds to the *-O3* optimized C code, which originally contained several assignment statements and five unnested loops. Below, we walk through the different phases using this CFG as an example.

We start by finding simple subgraphs in the CFG of individual functions in a module, which we process in call-graph order. Figure 3 lists the graph matching patterns used for this purpose. When a matching subgraph is found, we create an in-memory abstract container C out of the subgraph, thereby reducing the original graph into a interconnection graph of containers and basic blocks. We repeatedly apply the matching rules, until the graph can be reduced no further.

The patterns in Figure 3 all have singular entry and exit points. Such subgraphs have the advantage that their cost can be estimated by a simple mathematical expression. If the accuracy of such an estimate is within the allowable error bound, instrumenting the subgraph may not be necessary. Instead, it may be abstracted as a container C with a known cost $f(C)$, which can simply be added to the total cost of its enclosing container. *rule* 1 in Figure 3 can match any number of sequential containers, *rules* 2*a*, *b* match branch structures, and *rules* 3*a*, *b*, *c* match different kinds of loops.

For example, in Figure 1, both the loops *for.body12* and *for.body29* match *rule* 3c (Figure 3), and are abstracted out into containers $c1$ and $c2$. These new containers will now

Table 4. A loop running an unknown number of times

```
void run_loop_ntimes(int n) {
  for( int i = 0; i < n; i++ ) {
    /* code */
  }
}
```

Table 5. Transformed form of loop in Table 4 with *periodic_probe* inserted

```
void run_loop_ntimes(int n) {
  for( int i = 0; i < n; ) {
    int k = i;
    int j = min(n, i + computed_iterations);
    for( ; i < j; i++ ) { // inner loop
      /* code */
    }
    periodic_probe( (i-k) * inner_loop_cost )
  }
}
```

be treated as nodes of the transformed (in-memory) graph, along with basic blocks *entry*, *for.end16* and *for.end33*. The transformed graph matches the chain *rule* 1 with 5 container nodes, and is made into a new hierarchical container encapsulating the whole function CFG.

While `rules 1 & 2` have unidirectional control flow, `rules 3a,b,c` have backedges to form a loop. Though it is not always true, in the case of `Init_Opacity()`, backedge counts were known, even after optimization. If the loop body cost is fixed, and the aggregate loop cost is below the allowable error threshold, instrumentation is not necessary. For heavily used functions, eliminating such instrumentations can significantly reduce runtime overhead. When the aggregate loop cost is large, or unknown, a different approach is required, explained in §3.4.

3.3 Cost Evaluation

Once the hierarchical container graph is generated, a cost evaluation pass is applied to compute the cost (in terms of instruction count) of each container. In the cost-evaluation phase, we recursively consider each container of the final reduced graph, starting from the outermost container. Using the cost estimation functions listed in Table 6, we attempt to compute the outermost container's *cost*, that is the total number of instructions that will be executed by this container, as a function of the cost of its sub-containers. Intuitively,

Table 6. Cost functions for the matching rules in Figure 3

$Rule\ 1 : f(C) = f(C1) + f(C2) + f(C3)$
$Rule\ 2a : f(C) = f(C1) + g(C2, C3) + f(C4)$
$Rule\ 2b : f(C) = f(C1) + g(C2, 0) + f(C3)$
$Rule\ 3a : f(C) = (f(C1) + f(C2)) * (b + 1)$
$Rule\ 3b : f(C) = (f(C1) + f(C2)) * b + f(C1)$
$Rule\ 3c : f(C) = f(C1) * (b + 1)$

the cost functions combine the cost of similarly expensive child nodes of a branch structure, or find the cost of a loop construct by multiplying the cost of the body of the loop by the number of iterations. Function g in Table 6 computes the combined cost for similarly expensive child nodes: for the purpose of this paper, g computes the *mean* of its inputs.

The cost of a container may be constant, parametric, or not computable. Depending on the type and extent of this cost, we choose to store its cost in the container's meta-data or mark it for instrumentation in the next phase. If the cost is constant, the *probe interval* is used to determine whether to use the cost of this container for cost evaluation of a parent container in the hierarchy, or whether to mark this container for instrumentation. In the case of `rule 2`, an *allowable error* is used to check if the two branches of the CFG are similar enough to be summarized as one cost, that is less than the *probe interval.* Otherwise, each of the branch containers will be marked for instrumentation in the next phase. Logically, the performance overhead is expected to improve and interval accuracy is expected to degrade, with increasing *allowable error.* However, a thorough evaluation showed that the impact of *allowable error* on the interval accuracy and performance overhead is negligible beyond 500 IR instructions, and therefore, for minimizing configurable parameters at an acceptable accuracy error and performance overhead, we have heuristically chosen *allowable error* that is same as the *probe interval* for all of our evaluation.

Function Cost Optimization. When the entire function body can be represented by a single container (for example, `Init_Opacity()` in Fig 1), a function's cost is computed as the cost of this container. Otherwise, the cost of the entry block container is saved as the function cost and the rest is instrumented. For computing the cost of a call instruction, the computed cost for the called function is added to the call-site instruction cost. Function costs are evaluated in call graph order, and function cost optimization is not applied for recursive functions. For libraries compiled with CIs, function costs are communicated through exported cost files. We use LLVM's *scalar-evolution* pass [32] for parametric function cost computation, where the cost may depend on runtime input parameters to the function.

3.4 Loop Transform

Loops with large or unknown aggregate cost often contribute a large share of the performance overhead. Of special concern are loops with a small loop body and large number of iterations that is unknown at compile time. To mitigate the overhead of instrumenting every iteration, we decompose such loops into an outer loop and an inner loop.

Table 4 shows an example loop's pseudo-code, and Table 5 shows how it will be transformed. In the transformed version, the outer loop has an unknown number of iterations at compile time, while the inner has an upper-bounded number of iterations, based on the probe interval. This is controlled by the compile-time variable $computed\_iterations$, that represents the number of times a loop with a fixed cost can run without exceeding the probe interval. This design allows us to instrument the outer loop with the known cost of the inner loop, while keeping the inner loop uninstrumented. The cost added to the local counter in *periodic_probe* is dynamically computed based on the loop induction variable value. We use LLVM's *loop-simplify* pass to canonicalize natural loops [32], so as to find the induction variable required for this transformation, where possible.

For containers of loops that do not have an induction variable, do not have a known body cost, or for those whose body cost exceeds an allowable error threshold, we conservatively instrument every container in it.

3.5 Cloning Single Block Loops

Empirically, adding instrumentation to deeply nested, tight loops (comprising a single basic block), can incur significant overhead. The transform above addresses this in long loops, but actually increases overhead for very short loops where the number of iterations is only known at runtime. To address this, we clone any simple loop into two versions, and select between these based on the runtime iteration count. If the iteration count leads to a loop cost smaller than the probe interval, we use an uninstrumented version of the loop and apply the cost increment outside of the loop. If not, we use the transformed loop as described above (see §3.4).

3.6 Post-processing - Unmatched Patterns

Even with the preprocessing steps, there may remain sections of CFGs that cannot be abstracted into single-entry single-exit regions, and remain unmatched by any of the rules. To reduce the cost of instrumenting these remaining basic blocks, we use a technique inspired from CoreDet [9]. Here, each unprocessed single basic block container is processed greedily and individually. If the cost of containers behind incoming edges are known, differ within the allowable error threshold, and do not contain back-edges, we approximate the cost of all these containers as their mean, and add it to the current container. This avoids the need to instrument the containers behind these incoming edges. For any given block, if none of the children have multiple incoming edges, the block's cost is disseminated to its children, avoiding the need to instrument the block itself.

4 Instrumentation Phase and Probe Types

The instrumentation phase recurses through the container structure, adding probes to containers marked for instrumentation. Probes add the cost of the container to the instruction counter, and conditionally call the interrupt handler, based on the target interrupt interval specified at runtime.

The analysis phase, however, provides container cost in terms of IR instruction count, while users specify a target interval in cycles. Unfortunately, the translation from IR instruction count to cycles varies between applications and architectures, and even over time as applications perform different tasks, and CPUs dynamically adjust their clock frequency. Moreover, most programs include calls to external library functions and system calls, which may not be practical to instrument with compiler interrupts, or may even suspend the execution of the program.

This raises the question of how to translate the user's desired cycle interval into a concrete condition in our instrumentation. We present two options that have worked well in our experiments: a Pure IR solution (labeled **CI**), and an IR-gated cycle counter solution (**CI-Cycles**). Both make use of a heuristic estimate of the IR-to-cycle count ratio[3], but use it in different ways.

CI: our pure IR option carries no additional cost. For this option, we empirically use a cost of 100 IR instructions to represent the time spent in any uninstrumented function. While inaccurate in modeling uninstrumented functions, and vulnerable to variability in IR execution time, pure IR instrumentation is deterministic, portable and fast.

CI-Cycles: Our IR-gated cycle counter instrumentation uses the IR count to read the cycle counter (CC) periodically. If the cycle counter indicates that enough cycles have passed since the last interval, we call the interrupt handler. If not, check the cycle counter again after a number of IR instructions proportional to the remaining time. CI-Cycles assumes the availability of a hardware cycle counter (CC), through the `llvm.readcyclecounter` intrinsic.

Below, we first demonstrate the power and utility of **CI** through three example applications, then evaluate the overhead and accuracy of our instrumentation approach vs. prior work in §5.4.

5 Evaluation

All evaluation experiments below were performed on Ubuntu Linux 18.04, with Linux Kernel version 4.15.0-74-generic. All programs were compiled with LLVM version 9, and with

[3]This can be either our default value of 4 LLVM IR per cycle, or tuned for the specific application based on an example execution.

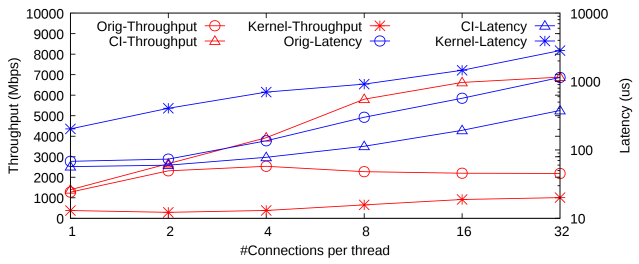

Figure 4. Throughput (download) and response latency of epserver/epwget running original mTCP, and our CI-based mTCP over a 10 Gbps link.

our compiler interrupts optimization pass, when applicable. We used the glibc malloc(), after evaluations showed it generally matching or outperforming other mallocs, such as Hoard [10] and jemalloc [25] on these programs. The CPU scaling governor was set to "performance". Except where noted otherwise, these results are from 2 socket machines with the Intel Skylake architecture, a total of 56 hardware threads, with Hyper-threading enabled, and 128 GB of RAM. However, we confirmed that similar results are achievable on the older Broadwell and Sandy Bridge x86 architectures. Reported results are from at least 10 runs, except where otherwise noted.

5.1 App: mTCP: Kernel Bypass Networking

Kernel bypass, or user level networking, has the user space application talk directly to the NIC, without kernel involvement in typical communication. This is desirable when the overhead of a general kernel network stack is deemed too high. The Data Plane Development Kit (DPDK) [23] offers a low-level API to support this type of operation. Built on top of DPDK, mTCP [26] implements TCP in user space.

One of the bigger challenges in designing a user space TCP implementation is the need to seamlessly and rapidly retransmit packets and acknowledge receipt of packets, even when the application thread is busy doing other work. At the kernel level, this is managed with NIC interrupts and timer interrupts. The mTCP approach is to pair each application thread with a helper thread, pinned to the same hardware thread, which runs the network stack. The use of a helper thread gives mTCP the ability to, when necessary, interrupt a CPU-heavy application thread to poll the NIC for incoming packets, send acknowledgments, make retransmissions etc. The authors of mTCP demonstrated large gains over Linux networking performance and prior research systems.

Using CIs, mTCP can be implemented without the helper thread. We compile the target program with CIs, and use this to run (in this case, roughly every 2500 cycles) the body of the mTCP network stack loop that the mTCP helper thread would usually run. We also made a small handful of other changes to mTCP to accommodate the new single-threaded

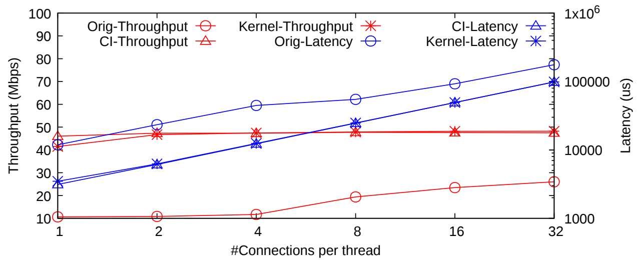

Figure 5. Throughput (download) and response latency of epserver/epwget modified to perform computational work before responding to a request, running original mTCP, and our CI-based mTCP over a 10 Gbps link.

model, notably replacing application/helper thread coordination with function calls. All in all, approximately 25 lines of code were changed. Figure 4 shows the performance of kernel networking, mTCP and CI-mTCP on the `epserver/epwget` basic HTTP server/client software included with the mTCP package. The application performs HTTP requests, repeatedly fetching a 1kB file. Server and client each run with 16 application threads, and multiple concurrent sessions per thread. We run server and client on two machines connected by 10Gbps Ethernet. We vary the total number of concurrent connections per client thread on the x axis, and plot download throughput (left, red) and latency (right, blue): the time from request to receiving a complete 1kB response.

We show results for CI-mTCP (CI), original mTCP (orig), as well as standard Linux sockets (kernel) for reference. As expected, mTCP substantially outperforms Linux sockets: Linux performance is dominated by processing incoming packet interrupts. On this NIC, 8 different IRQs can be configured with affinity to a single core each. At high load, contention between these 8 IRQ-processing cores and the cores that eventually receive the packets, is resulting in a congestion collapse that kernel bypass networking with mTCP avoids. That said, kernel socket performance can likely be improved by tuning the kernel's interrupt handling settings for this application.

Notably, the CI version consistently offers higher throughput and lower latency than the original mTCP. We discovered two reasons for this improvement. First, CIs avoid the need for context switching, as well as locking, condition variables and their associated `futex` system calls to coordinate between the application and helper thread. Second, packet processing is more efficient in larger batches. When the application is waiting for the network, the original mTCP polls the NIC in a tight loop, returning the moment packets have been received. The CI version polls the NIC periodically, based on the configured 2500 cycle CI interval, resulting in larger batches of packets arriving to the application together. Longer CI intervals further improves efficiency.

Figure 5 offers a different perspective on the advantage of CI over threads for mTCP. Here, we modified `epserver` so that for each request the server performs a fixed amount of work before responding, as a web-based application server might. In this case, however, the "work" is a 1M iteration loop without a body. Under these conditions, CI based mTCP offers 3–4× higher throughput, and 30% lower latency than thread-based mTCP.

When sharing its core with a compute-heavy application thread, the mTCP helper thread will be scheduled infrequently. As a result, acknowledging or retransmitting a packet may be delayed by an entire CPU quantum, severely impacting TCP performance. By contrast, CIs are scheduled at regular intervals independent of application behavior, avoiding this problem. Naturally, interrupt-driven kernel networking performs well under these conditions, closely tracking CI-mTCP, as neither system call overhead nor contention between threads is a concern when the application is spending most of its time doing local "work". To conclude, we find that CI-mTCP offers both the improved scalability of kernel-bypass networking with mTCP, and the robustness of kernel networking.

5.2 App: Shenango, Kernel-bypass Networking with Fine-grained Processor Allocation

In contrast with mTCP, our second application, *Shenango* [41], uses a more conventional approach to kernel bypass networking, dedicating one core to communicating with the NIC, the *IOKernel*. Application cores rely on the IOKernel for network I/O and CPU scheduling, exchanging messages via LRPC [11] queues in shared memory. The TCP/IP stack is implemented in the Shenango runtime, which controls the application threads. Aiming to improve CPU efficiency over pure kernel-bypass networking, Shenango actively manages the allocation of application cores between latency-sensitive primary applications, and batch processing applications.

While Shenango improves CPU efficiency on application cores, one significant weakness remains: it dedicates an entire core to IOKernel duties, even when there is little or no network traffic, resulting in a 0% CPU efficiency on the IOKernel core. To address this, we modified the Shenango IOKernel to run as a CI handler instead of busy-polling on a dedicated core: this was primarily a question of moving the body of the IOKernel poll loop to the interrupt handler.

In principle, the CI version of the Shenango IOKernel can be added to any (trusted) CPU-bound application which does not make heavy use of system calls. For evaluation purposes, we use a Bitcoin miner (CPUminer [18]) as an example workload, to characterize both the CPU efficiency on the IOKernel core, as well as any effect on overall Shenango performance. To prepare CPUminer to host the CI IOKernel, we compiled CPUminer with CIs, and modified the `main()` function to add IOKernel initialization and to install the IOKernel signal handler. Beyond that, both Shenango and CPUminer install handlers for the same Unix signals, which required some minimal integration. In all, changes were made to a small handful of lines of code.

Other than running the Shenango IOKernel as a CI handler within CPUminer, our setup mirrors that of the original Shenango paper [41], wherein Shenango manages core allocation and I/O for two applications: *memcached* (latency-sensitive) and *swaptions* (batch). In these experiments, a memcached server runs on a four socket Intel E5-4620, with an Intel X540-AT2 10 GBps NIC. Memcached is limited to one socket as in the original paper. We were limited to 2 client machines rather than the 6 used in the original paper, explaining the lower maximum offered load. Clients are based on unmodified Shenango. [4]

Figure 6 illustrates the performance of standard kernel networking with PThreads, stock Shenango, and Shenango with a CI-based IOKernel, for a single long-duration run. Overall, replacing the dedicated core with high-frequency CIs does not significantly impact memcached latency under Shenango.[5] Naturally, the greater the interrupt period, the greater the latency increase. However, for larger loads, this effect is less significant: as more time is spent processing packets, the interval between interrupts becomes a less important performance factor.

CPUminer, sharing a thread with the Shenango IOKernel, is able to make effective use of the CPU when IOKernel workload is low: at 200k memcached requests/s, with a 8,000 cycle CI interval, we achieve a CPUminer hash rate of over 55% of that of an unmodified CPUminer on a dedicated core, while incurring a memcached median and 99.9th percentile latency increase of approximately 10% over stock Shenango. At close to zero load, and with a large, 64,000 cycle interval, CPUminer achieved approximately 90% of its standalone performance on the shared thread, representing 90% CPU efficiency, albeit at the cost of more than doubling memcached latency at this extremely low load. For high loads, memcached performance is essentially unaffected by sharing the CPU, except with very large intervals: here, CPUminer is able to recover very little CPU, as time spent in the interrupt handler dominates the execution.

Finally, in terms of batch (swaptions) throughput on application cores managed by Shenango, CI IOKernel achieved the same throughput as the dedicated core IOKernel; we omit the plot of those results due to space constraints.

[4]We observed an intermittent problem with Shenango, where occasional runs showed many dropped requests and extremely high latencies. We tracked this down to a fixed 300 ms TCP retransmission timeout and lack of congestion control in *Shenango*, together with very frequent retransmissions on problem runs, but did not investigate further; instead, we discarded any results with dropped requests, and re-ran those experiments. This problem occurred with both stock and CI-based Shenango on the server side.

[5]We report latencies from one of our two clients. The other client machine consistently reported $20\mu s$ higher 99.9% latencies, with or without CIs, likely due to hardware differences. The two clients achieved similar throughput.

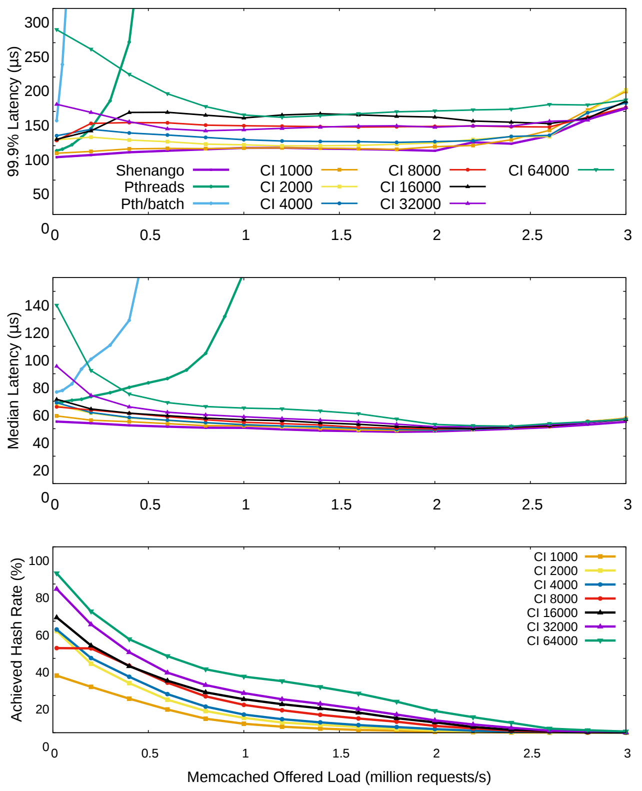

Figure 6. 99.9th percentile and median memcached latency with varying load, showing unmodified Shenango, Pthreads on a dedicated core, Pthreads sharing cores with a swaptions batch job, and Shenango+CI with varying polling intervals. Bottom plot shows achieved CPUminer hash rate on the IOKernel core: higher load, and shorter polling intervals leave less spare capacity for batch work.

5.3 App: "Designated" Server Delegation

For our third application, we consider delegation in the style of FFWD [46]. Delegation is an efficient message-passing based alternative to synchronized shared memory access. Here, only one core (a delegation server) may access a given object or data structure directly, while others access it indirectly by delegating function calls to the server. Delegation has been shown to have a substantial performance and scalability advantage over locking in highly contended data structures. However, a significant drawback of delegation is the need to dedicate at least one hardware thread as a delegation server.

We propose to use CIs to replace the infinite delegation loop of a dedicated delegation server, with periodic calls to a delegation server poll function from a thread that is otherwise running application code. For this experiment, we modify the FFWD [46] delegation system to use such a periodic function call instead of a dedicated server. We then use the CI handler to call the poll function with a small

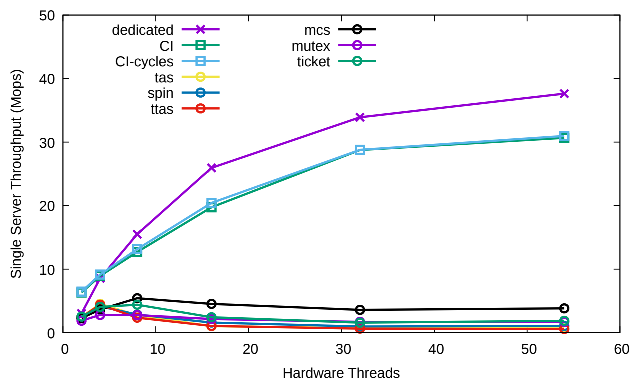

Figure 7. Performance of fetch-and-add benchmark for varying number of threads, using delegation and several lock types.

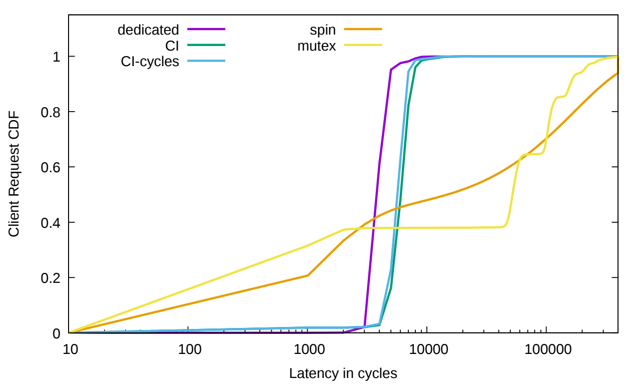

Figure 8. Distribution of client request latency in cycles.

period (250–1,000 LLVM IR instructions appears to be a good range), from one of the application threads, our *designated* server. Interestingly, after disabling Compiler Interrupts, it is perfectly safe for the client code to directly access the shared data structure, bypassing the server, and increasing overall throughput. This is implemented as a direct function call in the FFWD API.

Figure 7 compares the performance of CI-based designated server delegation both to delegation with a dedicated server, and to several lock types, on a classic fetch-and-add microbenchmark program, as we vary the number of threads. In this program, threads repeatedly increment a single shared variable, up to a fixed number of iterations per thread. With locking, the variable is incremented after entering a critical section. With delegation, an increment function is delegated to a single delegation server.

Surprisingly delegation is competitive with locking for small thread counts (in particular when using CIs), while it quickly outperforms locking as the thread count increases. Interestingly, CI-based delegation is outperforming also the dedicated server, up to 8 threads; after that the server workload is too high to be worth sharing the core with a client thread, in this application. Nevertheless, it adds much needed flexibility to an otherwise quite rigid delegation design.

By contrast, most locking approaches quickly succumb to congestion collapse after more than 8 threads. Only the queue-based MCS lock remains somewhat stable in the high contention regime, at a total lock throughput of 4–5 Mops.

Figure 8 explores the latency of delegation operations vs. locking, under both dedicated and CI-enabled *designated* delegation (CI), with 56 threads. While locking shows a wide range of latencies from less than 10 cycles to well over 100,000 cycles, both delegation approaches exhibit essentially constant latency. Designated delegation increases latency modestly, in return for not requiring a dedicated core.

5.4 Overhead and Interval Accuracy

Below, we analyze the overhead and accuracy of several CI designs, including related prior work: **CnB**) Instrumenting all calls and back-edges (similar to yield points) (§2.5). **Naïve**) Instrumenting every basic block. **CD**) Naïve plus CoreDet-style [9] optimizations. **CI**) Our proposed static analysis pass. **CI-Cycles**) Our proposed pass, plus periodic cycle counter readings. **Naïve-cycles**) Instrumenting every basic block, plus periodic cycle counter readings. **CnB-Cycles**) Instrumenting all calls and back-edges with cycle counter readings.

Here, **CI** and **CI-Cycles** are our proposed designs, where **CI-Cycles** reads the cycle counter at a set IR period. **CnB** counts the number of calls and back-edges, while **CnB-Cycles** reads the cycle counter on every back-edge and function call. **Naïve-cycles** instruments every basic block, and reads the cycle counter at a set IR period. In order to have fair comparison, we tune the interrupt interval for each method to approximate a target interval in cycles. Thus, for **CnB**, interrupts are raised after a constant number of function calls and back-edges, while for **CD** we select an IR-to-cycle ratio that achieves a median interval close to the target interval.

The instrumentation required to enable CI carries a performance penalty, especially for single-threaded applications. We measured this overhead on programs from the SPLASH-2 [49, 52], Phoenix [43, 45] and Parsec[6] [12, 13, 42] benchmark suites. The results below are the average of at least 10 runs. When possible, we chose input sizes to yield a runtime of at least 1 sec.

Figure 9 shows the overhead during single-threaded execution of several variations of CIs, when tuned for producing interrupts at a 5,000 cycle interval. Figure 11 shows the overhead using 32 threads instead of one.

[6]A number of programs were left out in some or all plots, due to difficulties modifying the build process (bodytrack, facesim, vips, x264), large external libraries (ferret), or unsupported features (freqmine - LLVM does not offer OMP). That said, no program was left out due to poor actual or expected evaluation results.

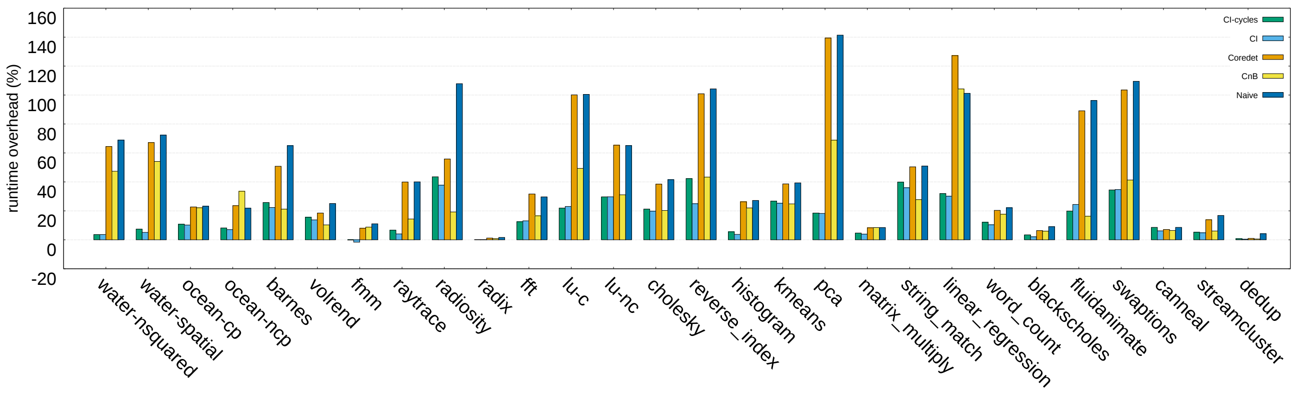

Figure 9. Overhead of different types of CIs tuned for an interval of 5000 cycles, when running with minimal threads (1, where supported).

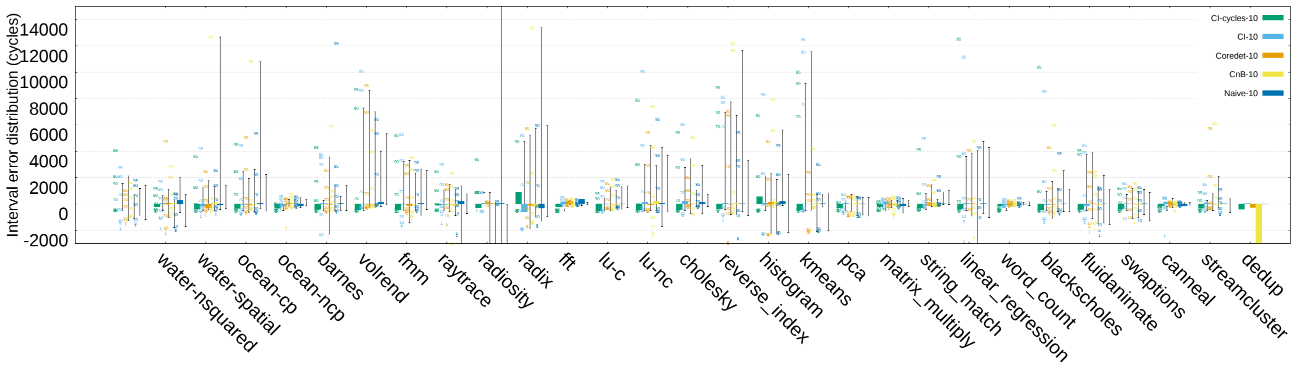

Figure 10. The interval accuracy of a single run, for a configured interval of 5,000 cycles after profiling, for SPLASH-2, Phoenix programs and Parsec running with minimum threads (ideally 1). The intervals are measured by reading hardware performance counter values for cycles executed between compiler interrupts.

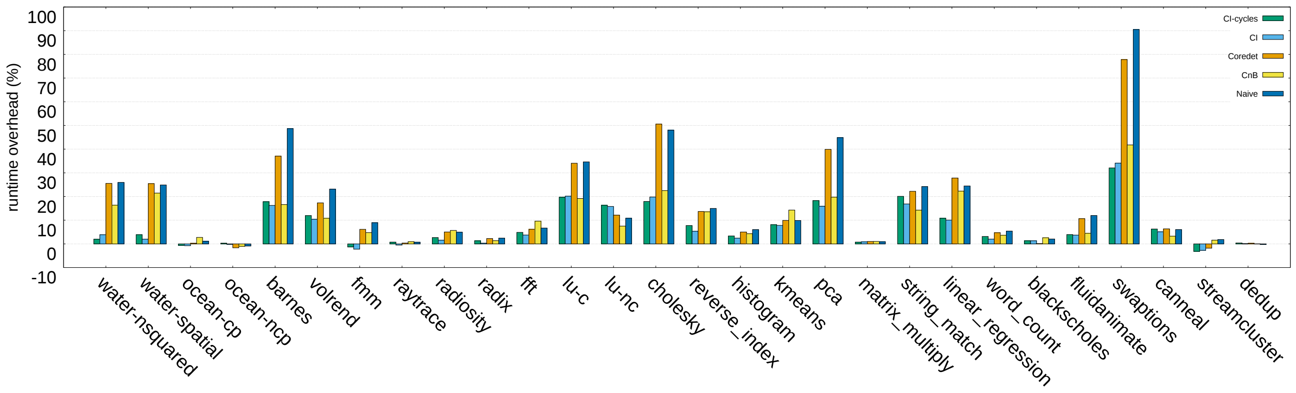

Figure 11. Overhead of different types of CIs tuned for an interval of 5000 cycles, running with 32 threads. Hardware interrupts at a 5,000 cycle interval imposes a mean overhead over 800%.

Overall, sequential overheads can be large, but diminish in a multi-threaded setting. **CI**/**CI-cycles** have the lowest overhead (median 11.7%/12.3% for 1 thread, 3.7%/3.9% for 32 threads), with **CnB** (20.7%, 6.6% respectively) close behind. **CD** (38.5%, 8.1%) and **Naïve** (39.7%, 9.4%) are much slower. These results correspond well with detailed measurements counting the number of probes executed, for each design. For example, in the vast majority of applications, **CI** reduced probe executions by over 50% vs. **Naïve**.

The variations that read the cycle counter are naturally slower than those that do not. We omit **CnB-cycles** and **Naïve-cycles** to conserve room in the plots, however **CnB-cycles** (179%, 75%) stands out as having impractically high overhead: reading the cycle counter is not a very expensive operation, but this design is very aggressive. **Naïve-cycles** (46%,12%) is also very expensive, mostly due to the frequent instrumentation. Meanwhile, **CI-cycles** (12.3%, 3.9%), strikes a much better balance. For reference, Table 7 shows CI and Naïve (N) runtimes of these benchmarks normalized to pthreads, for 1 and 32 threads.

Figure 10 shows the interval accuracy of these variations, for the single-threaded case. Naturally, any successful design needs to balance performance vs. accuracy. The colored bar shows the median error, in cycles, from the 5,000 cycle target. The error bars show the 10-90 percentile spread, and other percentiles are labeled in tiny font. To view these annotations, a digital viewer with zoom capability is recommended. On a printout, the median and 10th/90th percentiles will suffice.

Here, **CI** achieves accuracy comparable to the much more expensive **CD** and **Näive**. However, all three suffer from large errors in several benchmark programs, with 10% of interrupts occurring more than 5,000 cycles late. **CnB** shows the greatest variance in error, likely due to its coarser metric. **CI-cycles**, trades a small amount of performance to eliminate too-short intervals, but is otherwise similar to CI.

Compared to hardware interrupts, compiler interrupts do suffer from significant interval outliers. This variance is explained by several independent phenomena. First, one IR instruction does not necessarily translate into one x86 instruction, which in turn may take somewhere between 0–1,000 cycles to execute due to instruction level parallelism and cache line contention effects. Moreover, CI cannot raise interrupts during system calls, or in any uninstrumented code, but is forced to wait until control returns to the program. If both high accuracy and high performance are required, a hybrid CI/hardware-interrupt solution may offer the best of both worlds, but we did not explore this in depth.

While compiler interrupts impose significant overhead, it pales in comparison to using conventional interrupts for short interrupt intervals. Figure 12 shows the slowdown factor of **CI** (blue line) and conventional hardware interrupts (red X) for all the benchmark programs overlaid to provide an overall picture, as we vary the interrupt interval between 500–500,000 cycles. The interrupt handler we used collects these statistics using the `RDTSCP` instruction, and nothing else. We used Linux PAPI, to configure short hardware interrupt intervals.

Table 7. SPLASH-2, Phoenix, Parsec benchmark applications' mean absolute runtime in milliseconds with Pthreads (PT), and their normalized runtime overhead for CI (CI) and Naïve (N) for 1 & 32 threads.

	PT	CI	N	PT	CI	N
#threads	1	1	1	32	32	32
water-nsquared	55	1.03	1.69	11	1.09	1.27
water-spatial	45	1.04	1.71	9	1.00	1.22
ocean-cp	2345	1.10	1.23	1305	.99	1.01
ocean-ncp	137	1.06	1.21	909	.99	.99
barnes	3371	1.22	1.65	259	1.16	1.49
volrend	506	1.13	1.25	269	1.10	1.23
fmm	1212	.98	1.10	155	.98	1.09
raytrace	893	1.04	1.39	104	1.00	1.00
radiosity	1309	1.37	2.07	660	1.01	1.05
radix	17540	1.00	1.01	791	1.00	1.02
fft	2358	1.13	1.29	924	1.03	1.06
lu-c	15174	1.23	2.00	2403	1.20	1.34
lu-nc	2660	1.29	1.65	423	1.15	1.10
cholesky	130	1.20	1.41	70	1.20	1.48
reverse_index	523	1.25	2.04	272	1.05	1.15
histogram	2156	1.03	1.27	232	1.02	1.06
kmeans	2422	1.25	1.39	881	1.07	1.09
pca	1121	1.18	2.41	267	1.15	1.44
matrix_multiply	2246	1.03	1.08	1604	1.00	1.00
string_match	672	1.36	1.51	87	1.17	1.24
linear_regression	309	1.30	2.01	69	1.10	1.24
word_count	2615	1.10	1.22	514	1.01	1.05
blackscholes	443	1.02	1.09	149	1.01	1.02
fluidanimate	1442	1.24	1.96	729	1.03	1.11
swaptions	12294	1.34	2.09	759	1.34	1.90
canneal	65575	1.06	1.08	5820	1.05	1.06
streamcluster	4733	1.04	1.16	2164	.97	1.01
dedup	452	1.00	1.04	413	1.00	.99
geo-mean		1.14	1.45		1.06	1.15

The results show that each hardware interrupt incurs a considerably higher cost than a compiler interrupt, resulting in a performance collapse for short intervals between interrupts. Based on these results, we find that compiler interrupts as frequent as every 2,000 cycles are certainly feasible for short interrupt handlers. At this interrupt frequency, the cost of CI is approximately 10–100× lower than hardware interrupts. That said, due to the higher fixed cost of instrumentation, hardware interrupts retain an advantage at long interrupt periods, and for programs where instrumentation is particularly inefficient.

6 Related Work

Profilers [5–7] analyze and instrument code to collect program statistics. In comparison to [5, 21] that instrument every backedge and function entry point for profiling, CIs show significant improvement in better interval accuracy and performance by using static analysis and control flow

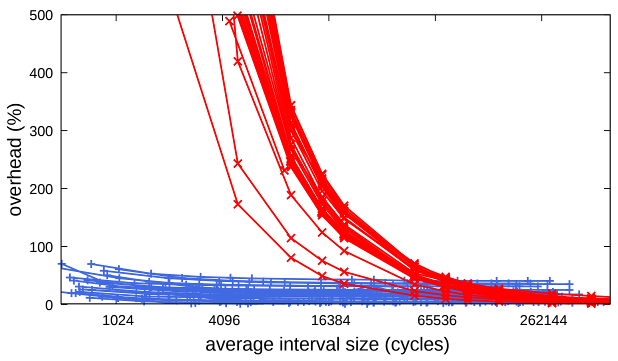

Figure 12. SPLASH-2, Phoenix and Parsec benchmark programs instrumented with compiler interrupts (blue line), or using performance counter interrupts (red X), when running with minimal threads (1, where supported). (Performance counter results are from 3 runs only, due to the very long runtime.) For intervals below 100,000 cycles, CI tends to outperform performance counter-based interrupts. (log scale X)

graph transformation. Some profilers [6, 7] use static analysis to instrument counters to find frequency of events, optimally in code, but their instrumentation is targeted towards finding frequent paths, edges, code regions, many of which can be avoided for CIs. Worst-case execution time (WCET) uses similar analysis where the set of control paths in the control flow graph are analyzed to obtain a WCET estimate of the program [51]. In contrast, our approach combines the information from static analysis to instrument the program, thus, influencing its dynamic behavior.

CIs incorporate the *Heuristic Balance* analysis used in Coredet [9] on top of its own static analysis techniques. While CoreDet focused more on determinism, CIs improve on CoreDet performance by exploiting more control flow graph structures, and by using code transformation. Instruction counting also has a long history, including [36]. Here, instead of modifying the compiler, this work directly instrumented an existing binary. By comparison, Compiler Interrupts are designed to be integrated into the program logic from the start. Other tools like pintool [34], also support instrumentation of the binary, but does not provide the flexibility of program analysis and transformation. CIs use LLVM framework's [29] intermediate representation (IR)'s immense flexibility for analyzing and transforming the behavior of a program in a source language and hardware independent way.

Languages like Java, Golang, use compiler inserted yield points (see §2.5) for garbage collection, cooperative preemption and more. Golang instruments function prologues and system calls for cooperative scheduling, and target a switching granularity of 10ms [14]. By comparison, we target microsecond intervals between interrupts.

Applications like deterministic execution [3, 9, 37], deterministic record and replay [17] use the concept of a deterministic logical clock based ordering of events. The pure IR based deterministic variant of CI can be used to implement the logical clock, by using the instruction count parameter passed in the handler function as the logical clock. Hardware performance counters have also been used for the same purpose [40] but apart from having high interrupt overhead, they are not guaranteed to be deterministic [50], making them unsuitable for enforcing determinism.

7 Conclusion

We have made the case for a new form of interrupt in computer systems: the compiler interrupt. Much like a timer-induced hardware interrupt, compiler interrupts periodically halt normal execution to execute a pre-defined interrupt handler. Unlike hardware interrupts, compiler interrupts can be both light-weight and portable, as they avoid kernel mediation. In three applications which require frequent polling, we have demonstrated the utility of Compiler Interrupts, both in qualitative and quantitative terms. Moreover, our evaluation on three common benchmark suites shows that we have substantially improved on prior work in compile-time instrumentation for the purpose of executing background tasks. We anticipate seeing many more applications of Compiler Interrupts in the future.

8 Availability

The source code for this project will be publicly available through our github repository https://github.com/bitslab/CompilerInterrupts.git.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. 1816714 and 1703425.

References

[1] Bowen Alpern, C Richard Attanasio, John J Barton, Michael G Burke, Perry Cheng, J-D Choi, Anthony Cocchi, Stephen J Fink, David Grove, Michael Hind, et al. 2000. The Jalapeno virtual machine. *IBM Systems Journal* 39, 1 (2000), 211–238.

[2] Bowen Alpern, Steve Augart, Stephen M Blackburn, Maria Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, et al. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44, 2 (2005), 399–417.

[3] Bowen Alpern, Ton Ngo, Jong-Deok Choi, and Manu Sridharan. 2000. DejaVu: deterministic Java replay debugger for Jalapeño Java virtual machine.. In *OOPSLA Addendum*. 165–166.

[4] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F Sweeney. 2004. *Architecture and policy for adaptive optimization in virtual machines*. Technical Report. Technical Report 23429, IBM Research.

[5] Matthew Arnold and Barbara G Ryder. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN*

2001 conference on Programming language design and implementation. 168–179.

[6] Thomas Ball and James R Larus. 1994. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 4 (1994), 1319–1360.

[7] Thomas Ball and James R Larus. 1996. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29.* IEEE, 46–57.

[8] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. {IX}: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14).* 49–65.

[9] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 53–64.

[10] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. 2000. Hoard: A scalable memory allocator for multithreaded applications. In *ACM SIGARCH Computer Architecture News*, Vol. 28. ACM, 117–128.

[11] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. 1989. Lightweight remote procedure call. *ACM SIGOPS Operating Systems Review* 23, 5 (1989), 102–113.

[12] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques.* 72–81.

[14] Vincent Blanchon. [n.d.]. *Go: Goroutine and Preemption.* https://medium.com/a-journey-with-go/go-goroutine-and-preemption-d6bc2aa2f4b7.

[15] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* 2329–2344.

[16] Adrian M Caulfield, Arup De, Joel Coburn, Todor I Mollow, Rajesh K Gupta, and Steven Swanson. 2010. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE, 385–395.

[17] Jong-Deok Choi and Harini Srinivasan. 1998. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools.* ACM, 48–59.

[18] cpuminer multi. [n.d.]. cpuminer-multi codebase. https://github.com/tpruvot/cpuminer-multi.

[19] Joseph A Fisher and Stefan M Freudenberger. 1992. Predicting conditional branch directions from previous runs of a program. *ACM SIGPLAN Notices* 27, 9 (1992), 85–95.

[20] Golang. [n.d.]. *Golang.* https://golang.org/.

[21] Martin Hirzel and Trishul Chilimbi. 2001. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4).* 117–126.

[22] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. 2018. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research.*

[23] DPDK Intel. 2014. "Data plane development kit.(2014)". "https://www.dpdk.org/".

[24] Java. [n.d.]. *Java.* https://www.java.com/en/.

[25] Jemalloc. [n.d.]. *Jemalloc.* https://github.com/jemalloc/jemalloc.

[26] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mtcp: a highly scalable user-level {TCP} stack for multicore systems. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14).* 489–502.

[27] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* 2511–2513.

[28] Charles Krasic, Mayukh Saubhasik, Anirban Sinha, and Ashvin Goel. 2009. Fair and timely scheduling via cooperative polling. In *Proceedings of the 4th ACM European conference on Computer systems.* 103–116.

[29] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization.* IEEE Computer Society, 75.

[30] Damien Le Moal. 2017. I/o latency optimization with polling. In *Vault Linux Storage and Filesystems Conference.*

[31] Yi Lin, Kunshan Wang, Stephen M Blackburn, Antony L Hosking, and Michael Norrish. 2015. Stop and go: Understanding yieldpoint behavior. In *Proceedings of the 2015 International Symposium on Memory Management.* 70–80.

[32] LLVM. [n.d.]. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html.

[33] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2012. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12).* 65–76.

[34] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[35] Ilias Marinos, Robert NM Watson, and Mark Handley. 2014. Network stack specialization for performance. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 175–186.

[36] John M Mellor-Crummey and Thomas J LeBlanc. 1989. A software instruction counter. In *ACM SIGARCH Computer Architecture News*, Vol. 17. ACM, 78–86.

[37] Timothy Merrifield, Joseph Devietti, and Jakob Eriksson. 2015. High-performance determinism with total store order consistency. In *Proceedings of the Tenth European Conference on Computer Systems.* ACM, 31.

[38] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. 2007. Shadow profiling: Hiding instrumentation costs with parallelism. In *International Symposium on Code Generation and Optimization (CGO'07).* IEEE, 198–208.

[39] Sagnik Nandy, Xiaofeng Gao, and Jeanne Ferrante. 2003. TFP: Time-sensitive, flow-specific profiling at runtime. In *International Workshop on Languages and Compilers for Parallel Computing.* Springer, 32–47.

[40] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices* 44, 3 (2009), 97–108.

[41] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High {CPU} Efficiency for Latency-sensitive Datacenter Workloads. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19).* 361–378.

[42] Parsec. [n.d.]. Parsec codebase. https://github.com/cirosantilli/parsec-benchmark.

[43] Phoenix. [n.d.]. Phoenix codebase. https://github.com/kozyraki/phoenix.

[44] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In

Proceedings of the 26th Symposium on Operating Systems Principles. 325–341.

[45] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Ieee, 13–24.

[46] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. ffwd: delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 342–358.

[47] Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.

[48] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. 2001. EMP: zero-copy OS-bypass NIC-driven gigabit ethernet message passing. In *SC'01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. IEEE, 49–49.

[49] Splash2. [n.d.]. Splash2 codebase. https://github.com/staceyson/splash2.

[50] Vincent M Weaver and Sally A McKee. 2008. Can hardware performance counters be trusted?. In *2008 IEEE International Symposium on Workload Characterization*. IEEE, 141–150.

[51] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. 2008. The worst-case execution-time problemâĂŤoverview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 36.

[52] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news* 23, 2 (1995), 24–36.

[53] Jisoo Yang, Dave B Minturn, and Frank Hady. 2012. When poll is better than interrupt.. In *FAST*, Vol. 12. 3–3.

[54] Qin Zhao, Ioana Cutcutache, and Weng-Fai Wong. 2010. PiPA: Pipelined profiling and analysis on multicore systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 7, 3 (2010), 1–29.