NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2025-36

September, 2025

Eliminating Hardware Interrupts with Dispersed Interrupt Polling

Kirill Nagaitsev, Kevin McAfee, Kevin Hayes, Justin Dong,
Nadharm Dhiantravan, Peter Dinda

Abstract

Each CPU on a modern architecture can receive thousands of hardware interrupts/second due to networking, I/O, and other events. In operating systems like Linux, interrupts cause expensive, hardware-driven context switches to the kernel and unexpected disruptions to caches and other hardware state. In HPC and database applications, for example, this results in significant performance impacts and unnecessary nondeterminism. Is it time to reconsider the alternative to interrupts, namely polling?

Traditional polling is inefficient as polls are concentrated in specific locations, blocking other code from execution. Our proposed alternative, Dispersed Interrupt Polling (DIP), deterministically distributes polls throughout the kernel and application via compiler transformation. These polls replace interrupts and allow for the removal of much of the interrupt hardware. We prototype this idea in Linux for RISC-V and x64, demonstrating the application of DIP to the entirety of the Linux kernel, as well as unmodified user space code. We evaluate our prototypes using Postgres (1.5M LOC) and various benchmarks (NAS, GAP, Embench, and SPEC2017). Despite running on unmodified, non-ideal hardware, the prototypes often achieve <10% overhead, without interrupt load, relative to nondeterministic hardware interrupts. When interrupt load is applied, our prototypes have virtually no overhead, and can achieve speedups of up to 5% via software coalescing. With the hardware simplifications we propose, these numbers could be improved even further. Our results make a strong case for DIP as a viable alternative to hardware interrupts.

This effort was partially supported by the United States National Science Foundation (NSF) under awards CNS-2211315, CCF-2119069, and CNS-2211508. Nagaitsev is a Department of Energy (DOE) Computational Science Graduate Fellow.

Keywords

interrupts, operating systems, compilers, RISC-V, Linux

Eliminating Hardware Interrupts with Dispersed Interrupt Polling

Kirill Nagaitsev Kevin McAfee Kevin Hayes Justin Dong
Nadharm Dhiantravan Peter Dinda

Northwestern University

Abstract

Each CPU on a modern architecture can receive thousands of hardware interrupts/second due to networking, I/O, and other events. In operating systems like Linux, interrupts cause expensive, hardware-driven context switches to the kernel and unexpected disruptions to caches and other hardware state. In HPC and database applications, for example, this results in significant performance impacts and unnecessary nondeterminism. Is it time to reconsider the alternative to interrupts, namely polling?

Traditional polling is inefficient as polls are concentrated in specific locations, blocking other code from execution. Our proposed alternative, *Dispersed Interrupt Polling* (DIP), deterministically distributes polls throughout the kernel and application via compiler transformation. These polls replace interrupts and allow for the removal of much of the interrupt hardware. We prototype this idea in Linux for RISC-V and x64, demonstrating the application of DIP to the entirety of the Linux kernel, as well as unmodified user space code. We evaluate our prototypes using Postgres (1.5M LOC) and various benchmarks (NAS, GAP, Embench, and SPEC2017). Despite running on unmodified, non-ideal hardware, the prototypes often achieve <10% overhead, without interrupt load, relative to nondeterministic hardware interrupts. When interrupt load is applied, our prototypes have virtually no overhead, and can achieve speedups of up to 5% via software coalescing. With the hardware simplifications we propose, these numbers could be improved even further. Our results make a strong case for DIP as a viable alternative to hardware interrupts.

Keywords: interrupts, operating systems, compilers, RISC-V, Linux

1 Introduction

Hardware interrupts and operating systems have been joined at the hip since the days of the UNIVAC in the early 1950s. There are good reasons that the hardware interrupt-driven approach to handling external asynchronous events was adopted at such an early point and continues to be used to interact with hardware devices, including high performance I/O devices such as NICs and storage controllers. The advent of multiprocessor systems in the 1980s introduced the use of (hardware) inter-processor interrupts (IPIs) for kernel→kernel signaling between CPUs, and this remains widely used to this day. Are the design decisions that were appropriate at these points in time still appropriate? Have other alternatives become viable?

Interrupts are not without fault. One issue is that interrupt dispatch is a fundamentally complex undertaking at the hardware level, often quite at odds with the normal processing of a superscalar out-of-order core, address translation, and the memory hierarchy. Consequently, hardware interrupt entry results in significant overhead, on the order of 1,000s of cycles per interrupt on architectures including x64 ([20, Figure 2],[21]) and RISC-V (measured in §2.3), due to the cost of doing a hardware-driven context switch. At least on x64, the semantics of interrupt delivery are so complex [15, Chapter 8] that it is almost certain that interrupt delivery is microprogrammed. All this time results in delaying the reaction to other external events and is time spent away from executing useful application instructions. An interrupt is a show-stopper for a CPU. What if processor architecture design was freed from the tyranny of the interrupt?

A second issue with hardware interrupts is that their costs show through the software of a modern kernel. Our own measurements indicate that typical interrupt handler overheads in Linux on x64 are on the order of 1,000s of cycles, which is similar to the cost of a hardware-driven context switch. Figure 1 illustrates the results of an ftrace-based study of the costs of handling the most common interrupts encountered on our test hardware and the Linux kernel. Not only does this time further delay the reaction to other external events by the application, but it is also time taken away from running application code. The combined hardware and kernel costs are so high that "user interrupts" have been introduced by Intel [22, Chapter 7], shipped in Sapphire Rapids, and a device-level extension has been proposed [3]. An analogous "N" extension was proposed for RISC-V [39]. These alternatives dispatch some hardware interrupts via a simplified interrupt dispatch process that avoids a complex context switch (the kernel is not invoked), and thus reduces the overhead. For example, Intel's implementation allows signalling between user threads with 16x lower latency than

This effort was partially supported by the United States National Science Foundation (NSF) under awards CNS-2211315, CCF-2119069, and CNS-2211508. Nagaitsev is a Department of Energy (DOE) Computational Science Graduate Fellow.

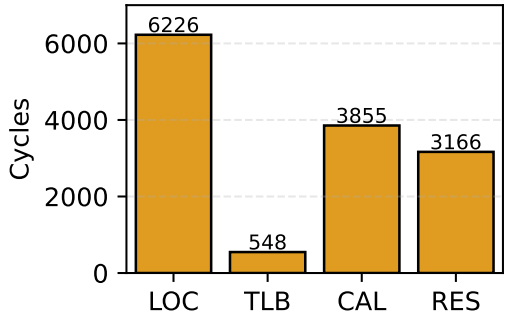


Figure 1. Mean execution time of Linux handlers of common interrupts on the hardware of §5. These times are on par with the cost of a hardware interrupt dispatch, ~1000 cycles (§2.3)

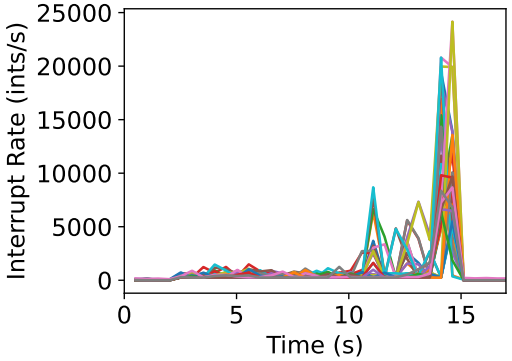


Figure 2. Interrupt rate for each CPU with Postgres running an OLTP benchmark of 100,000 SELECTs from 100 clients.

Linux signals [28]. However, they are either not implemented (RISC-V) or limited to IPIs (Intel). Even when generalized and available, they will not fundamentally change the nature of hardware interrupts. Another alternative is specific devices, such as NICs, that provide "OS-bypass" mechanisms in which "data plane" operation of the device is possible directly from user-space without interrupts. This is a long-standing approach in high-end HPC systems (e.g. [10]). User-level networking stacks (e.g., DPDK [42] and Shenango [31]) can also avoid network interrupts via polling, e.g. "Poll-mode drivers" in DPDK directly access RX and TX descriptors.

A third issue is that interrupts are becoming more and more commonplace. Certain applications in databases and HPC can be seen to reach up to 20,000 interrupts/second on a single CPU. Postgres [36] is a widely used SQL database which relies on disk I/O, network interrupts, and IPIs to operate. Figure 2 shows our own measurements of the interrupt behavior of a Linux server handling a Postgres workload—interrupt rates exceeding 20,000/s can be seen. Others have reported that databases that use *mmap* internally to manage their buffer pools are prone to high TLB shootdown interrupt rates, on the order of 1,000s or more per second depending on the workload [16]. Distributed memory parallel HPC applications see 100s of network- and timer-related interrupts per second [29], despite some of the specialized optimizations noted above.

A final, foundational issue with hardware interrupts is nondeterminism. Because the hardware can force a control flow transfer at any time, the resulting system is very difficult to reason about. Additionally, more complex coordinated operation, such as gang scheduling [14, 19], real-time behavior [18], and collective communication becomes much more challenging. For example, in the HPC space, the nondeterminism of node-local "OS Noise" is well known to cut into scalability of distributed memory applications [12, 17, 29, 40]. A typical HPC application has global coordination, and thus its execution rate is determined, at best, by the most affected node/CPU, if it is fortunate, and, if unfortunate, by a collapse of execution rate due to compounding effects with scale. Beyond raw performance, deterministic communication [11] and replayability [13, 32, 34] are of increasing interest. As the scale of applications for HPC (and for broader contexts such as databases and clouds) increases, so too does the need for performant, scalable I/O [7, 30], resulting in, of course, ever more interrupts and ever more nondeterminism.

We reconsider the diametric opposite to hardware interrupts for external events: polling. Here, the idea is to deactivate hardware interrupt dispatch entirely, and instead repeatedly check for the events themselves.[1] Consequently, hardware will never interrupt the currently running task on a CPU, removing all nondeterminism caused by interrupts. Polling has traditionally been done by having *dedicated* polling regions in the code where an application aggressively checks for *specific* events. This can be an effective technique in certain I/O bound applications [41]. However, this technique is often inefficient because polls are concentrated in specific locations. This blocks other code from executing when polling for one event is occurring, and additionally prevents events from being processed when outside of these polling regions.

Traditional polling is also a nightmare to program and does not compose across multiple events, unlike interrupts. There is, however, some reason to believe that generating polling code can be automated, and the result can operate in a performant manner. For example, background polling with Compiler Interrupts (CIs) [5] has demonstrated the ability to automate the creation of user-level polling for OS-bypass networking and other systems. The result is also more performant than user-written polling. Compiler-based timing (CT) [20] demonstrates replacing the kernel's use of hardware timer interrupts with whole kernel compiler analysis and transformation that injects timer polls. The resulting system can support much finer granularity thread scheduling. Concord [23] shows that compiler-based scheduling can be exploited to improve throughputs on microsecond-scale services, while maintaining tight tail latencies.

[1] In our conception, a *limited* interrupt dispatch capability would remain to handle instruction exceptions and special events such as NMIs.

We propose a new model, *Dispersed Interrupt Polling* (DIP), that addresses the limitations of traditional polling and of this prior work. DIP (a) is able to poll for *all* hardware events traditionally delivered by hardware interrupts, not just specific events, (b) allows composition of polling for different events, and (c) functions automatically, requiring only a recompilation. DIP operates by deterministically distributing polls throughout the kernel and user code, rather than concentrating the polls in particular locations in the kernel. Via these injected polls, each CPU queries its private interrupt controller at our injected poll points in the kernel and user code without having to do a context switch, resulting in lower overhead than what a nondeterministic hardware interrupt would incur.

To evaluate the prospects for DIP, we have developed two prototypes that share a common compiler framework, Beandip, (**B**EANDIP **E**liminates **A**voidable **N**ondeterminism with **D**ispersed **I**nterrupt **P**olling). Beandip implements DIP using a compile-time transformation that injects polls throughout the kernel and application code such that they will occur at approximately fixed-cycle intervals at runtime. This fixed-cycle interval is tunable, allowing a tradeoff between performance and event handling latency. Unfortunately, a *single* hardware and kernel combination suitable for evaluating the DIP model on large, complex benchmarks does not currently exist, leading to two prototypes in our evaluation. Beandip UX (User-space x64) targets the combination of the x64 architecture and Linux. Here, we can readily evaluate DIP on a wider range of benchmarks (NAS, GAP, Embench, SPEC2017), we can consider larger scales (more software and hardware concurrency), and we can run complex applications (Postgres). However, x64 does not allow us to *completely* avoid interrupts, making this architecture undesireable for the application of DIP to the Linux kernel. In contrast to x64, on RISC-V, interrupts can be avoided. Beandip KR (Kernel and user-space RISC-V) demonstrates the application of DIP to both the Linux kernel *and* unmodified user space applications. [2]

Our contributions are as follows:

- We provide detailed background on hardware interrupts, traditional polling, and compiler-based polling.
- We describe the DIP model in detail, explain hardware and software requirements of the model, and lay out intrinsic limitations.
- We give the details of the design and implementation of our prototype compilation framework, Beandip, and of our two prototypes that build on it, Beandip UX and Beandip KR.
- We present a performance evaluation of the prototypes using benchmarks which shows that: (a) polling intervals can be accurately achieved, (b) DIP overhead relative to hardware interrupts under minimal interrupt load is quite low (often <10%), (c) interrupt handling latency and overhead can be readily traded off, and (d) DIP overhead decreases as interrupt load increases, becoming negligible in high load scenarios and even turning into speedup via software coalescing.
- We evaluate Beandip UX with Postgres, achieving ∼90% the throughput of normal Postgres with hardware interrupts on OLTP workloads.
- We describe the limitations of our implementation, as well as potential future work.

In this work, we demonstrate that the DIP model for polling is a viable alternative to hardware interrupts. To the best of our knowledge, our work is the first to demonstrate a polling model that replaces *all* interrupts on a modern architecture like RISC-V or x64 with individual polls that are deterministically distributed throughout kernel and user code.

2 Background and Opportunity

We now provide salient information on interrupts, polling, compiler-based polling, and codebases we leverage in this work, leading to the opportunity that DIP seeks to exploit.

2.1 Hardware Interrupts vs. Polling

A hardware interrupt is any event that requires processing by the CPU, with examples including keystrokes, timer expirations, network packet receptions, and disk read completions. On modern architectures like x64, RISC-V, and ARM interrupts are identified by vectors—unsigned integers. Interrupts can be explicitly turned on or off on these architectures, meaning interrupts will only trigger a context switch on a CPU if interrupts are enabled on that CPU. Figure 4(a) illustrates the process of traditional hardware interrupt delivery.

Hardware Interrupts The easiest way to handle interrupts from a kernel developer's perspective is to register the interrupt vector they are interested in with the hardware, along with the address of its handler code. When the interrupt is asserted, the CPU will context switch and begin executing at this address. This interrupt dispatch will only occur if interrupts are enabled on the CPU. The CPU must typically spend some time saving context information before handler execution can begin, and then restore kernel-relevant context. When handler execution is complete, this process is then reversed to return to the interrupted code. The key is that it is the *hardware* which invokes the handler.

Traditional Polling Traditional polling is a method of explicitly checking if an event has occurred, either at a frequent or an infrequent interval. The event being polled does not need to be an interrupt, but it can be. In the realm of traditional polling, developers often avoid interrupts entirely and

[2]The reader may wonder why we do not evaluate only on RISC-V. Finding a RISC-V server-class machine that allows the necessary userspace access proved very challenging. Consequently, our target platform for RISC-V is a 4-core SBC, which is not fast, nor does it have much memory. Therefore, we must use Beandip UX to evaluate complex benchmarks and applications that require server-class hardware.

poll a relevant device directly. For example, an application that needs to respond to incoming network packets quickly may poll the NIC aggressively, with no need for an interrupt middleman. The need for polling is highly dependent on the application, but is typically most beneficial in I/O bound applications [41], safety-critical applications where determinism is essential [37], and applications where handling latency is critical [27]. Note that aggressive polling comes at the expense of energy, given that aggressive polling is the equivalent of constantly spinning. The key is that *software* checks for the event and *software* invokes the handler.

Because traditional polling typically means going straight to the device, it may be confusing why we discuss *polling for interrupts* in this work. The reason is that we are not just proposing an application-level solution, but rather a system-wide replacement for hardware interrupts that uses polling, which means we need to poll on *all* interrupts, not just particular devices.

2.2 Compiler-based Polling

In this work, we leverage the publicly available compile-time transformation that was designed for compiler-based timing (CT) [20]. The goal of this transformation fits our polling needs perfectly: to deterministically inject call instructions into the kernel and application code at points such that calls are made at approximately fixed-cycle intervals at run-time. CT uses this capability to deactivate the hardware timer and replace its interrupts with simple, fast callbacks.

We use LLVM [25], a widely-used compilation framework in which the CT transformation is implemented in the middle end. LLVM allows us to inject poll call instructions into the code which are ultimately emitted in the backend. We use WLLVM [33] to do whole-application analysis and transformation with large application codebases such as Postgres. We harness Clang LTO to extract LLVM IR from the Linux kernel and apply our transformation.

The CT transformation works by assigning LLVM-IR instructions expected latencies, then analyzing the total latencies along execution paths via a data-flow analysis. The focus of this transformation is the bodies of loops, since the goal of the transformation is to guarantee that the loops of the code being transformed are able to make calls to the poll function at regular intervals. We discuss further how we use the CT transformation in §4.

2.3 Polling the Interrupt Controller

Hardware interrupt dispatch is not free. It typically entails a context switch that requires a transition to kernel mode. The hardware must save and restore considerable register state (which needs to be unwound on the interrupt return). This context switch also impacts cache, pipeline, and branch predictor state. Due to Spectre/Meltdown [24, 26], modern kernels like Linux may also switch address spaces, which has a performance impact by affecting TLB and PWC state.

Interrupt Poll Costs			Interrupt
Method	Hit Cost	Miss Cost	Disp. Cost
RISC-V SiFive MMIO	72	64	2,858
RISC-V StarFive MMIO	28	27	?
x64 APIC MMIO	262	261	~1,000
x64 APIC MSR	390	361	~1,000
ARM MMIO	?	$35 \cdot N$	?
ARM Sysreg	2	2	?

Figure 3. Costs (in cycles) for polling interrupt pending registers versus interrupt dispatch cost on various machines and interfaces. RISC-V hardware is a SiFive Freedom FU740, along with a StarFive JH-7110, described further in (§6). x64 hardware described in (§5), MMIO cost is for all 8 registers. ARM is a Gigabyte server with a single Neoverse N1 128-core Ampere processor. N refers to there being a variable number of pending vector registers on ARM, and 35 is the cost to poll one. x64 interrupt dispatch cost from [20, 21].

Even ignoring the knock-on effects, the measured cost of a hardware interrupt dispatch (from interrupt assertion to first instruction of the interrupt handler) on a typical x64 is ~1,000 cycles ([20, Figure 2],[21]). On RISC-V hardware, we measured the dispatch cost to be 2,858 cycles. Such costs are quite high when considering that 1,000s of hardware interrupts can occur each second on a single CPU.

Interestingly, on x64, ARM, and RISC-V it is possible to poll the hardware interrupt controller. Even when interrupts are disabled, the controller still aggregates asserted interrupts. Figure 3 compares the cost of polling the interrupt controller to determine if any interrupt has been asserted with the cost of interrupt dispatch for, RISC-V, x64, and ARM machines. Where possible, we measure polling cost both with MMIO and with the architecture's specialized access path to the interrupt controller's control/status registers (e.g., MSRs on x64 and Sysregs on ARM). ARM Sysreg and RISC-V PLIC MMIO are the best of these methods for DIP, since they indicate if any interrupt is pending with just a single register read. On x64, we read the APIC's *interrupt pending vectors*, which form a status bitmap (if bit k is set, interrupt vector k has been asserted). Since there are 256 vectors on x64 and the bitmap registers are 32 bits wide, we must read eight registers to check if *any* of the possible interrupts is asserted.[3]

Figure 3's comparison lays bare the opportunity: the cost of a single poll is generally much lower than the cost of a hardware interrupt dispatch, often by 1-2 orders of magnitude. This helps to create the possibility that polling could be a viable alternative to hardware interrupts.

2.4 Interrupt Coalescing & Software Coalescing

In addition to the low cost relative to hardware interrupts, polling offers another opportunity: software-based interrupt coalescing. Traditional interrupt coalescing is a technique in which events that would normally trigger an interrupt are held back temporarily, until a sufficient amount of work

[3] On a typical Linux configuration, however, much of the vector space is unused, so it is often possible to avoid reading all eight.

builds up, or a timeout occurs, at which point the interrupt is allowed to dispatch. Modern NICs typically perform interrupt coalescing to ensure that an interrupt does not trigger for every single packet that arrives. When interrupts are turned off and the interrupt controller is polled, software-based interrupt coalescing can occur, given that there is a delay between subsequent polls which allows for multiple interrupts of the same type to arrive at the interrupt controller. The opportunity here is that the coalescing policy for all interrupts can be controlled entirely in software, removing the need for traditional hardware-based coalescing, and making the coalescing parameters much more configurable for individual workloads. Functionally, software-based coalescing does not break correctness, as interrupts can be missed even in a normal system (e.g. when interrupts are turned off), assuming the kernel eventually processes pending events.

3 Dispersed Interrupt Polling (DIP) Model

Figure 4 diagrams and compares asynchronous event delivery via (a) traditional hardware interrupts and via (b) the proposed DIP model. The DIP model involves substantially more complexity at compile time, and special cooperation between the kernel and userspace, but obviates the need for interrupts at run-time, resulting in a deterministic system.

3.1 Transcending traditional polling

Traditional polling has limited utility for several reasons:

- The programmer must *explicitly* write polling loops for the *individual* device events (for example, examining a specific register on a network card repeatedly.)
- Polling loops may block for considerable time (for example, waiting for a packet to arrive.) More generally, polling burns cycles that could be used for making application or kernel progress. It also burns energy.
- The polling loops for different devices or activities (IPIs) either do not compose, or the composition is left to some programmer who must read the minds of other developers to attempt such composition. This makes polling for multiple events (or all events in the system) extremely challenging to achieve.

On the other hand, handling device events via traditional polling has interesting advantages in comparison to handling them via interrupts:

- The latency of delivery of the event to the code that can handle it is low. It involves a call instead of an interrupt delivery.
- The overhead of polling *if the poll test occurs shortly after the event* can be low.
- Polling avoids nondeterminism—a device event can only take CPU time when the device is being polled.

The goal of the dispersed interrupt polling (DIP) model is to make polling transcend the disadvantages of traditional polling, while maintaining traditional polling's advantages to the greatest extent possible. If this is feasible, DIP can serve as an alternative to traditional interrupts.

3.2 Core concepts

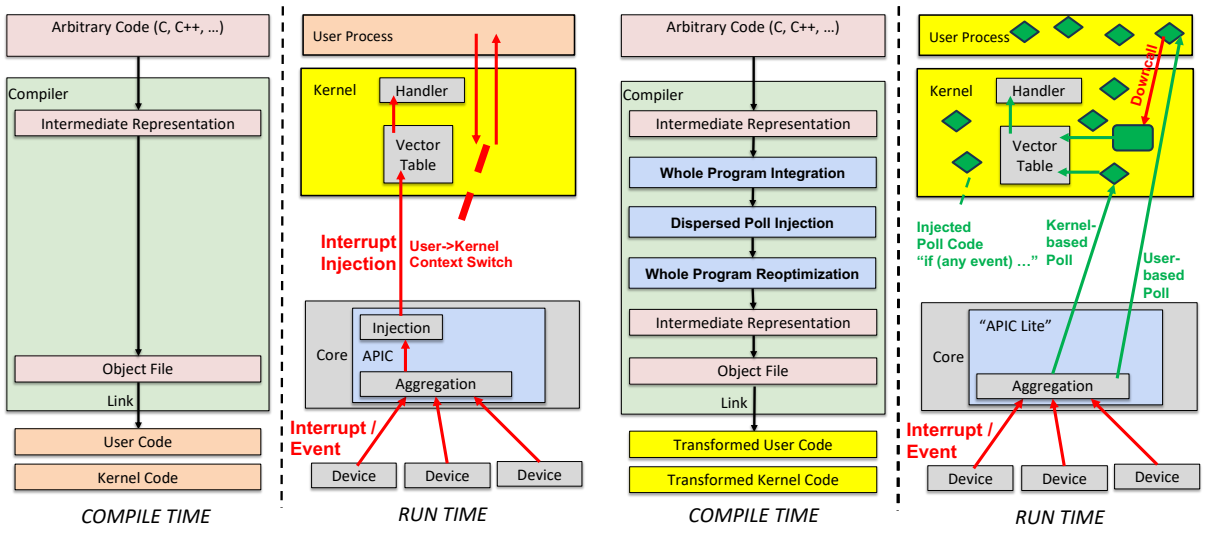
Polling loop diffusion Instead of a polling loop placed at a particular location, the contents of the loop are distributed throughout the kernel and application binaries, and the "loop" iterates due to normal code execution paths. These dispersed polls are illustrated as green diamonds in the "Run-time" side of Figure 4(b).

Compiler-based poll injection The programmer does not write a polling loop at all, much less diffuses it throughout the codebase. Instead, a compile-time transformation of the entire kernel and application codebase ("Compile-time" side of Figure 4(b)) adds and diffuses the polling loops. This is a non-obvious transformation because it must guarantee that the iterations of the polling loop execute regardless of the dynamic execution path taken at runtime. Additionally, it must guarantee that the iterations of the polling loop execute at an effective rate that is fast enough to detect device events quickly, yet not so fast that the overhead becomes significant. Finally, the injected polls are jointly optimized with the code they are injected into.

Interrupt aggregation In DIP, device and other interrupts are *generated* as normal, but delivered via the injected polls, instead of interrupts. Furthermore, DIP builds on a hardware aggregation of these interrupts. This addresses the problem of composition of polling loops noted above. The hardware aggregates the interrupts into a single place ("APIC Lite" in Figure 4(b)), and this makes it possible for there to be a *single* polling loop, which polls the hardware interrupt aggregation device instead of the actual interrupt-generating devices. While "hardware interrupt aggregation device" may seem to be an esoteric concept, all modern processors already have such a device, indeed typically one per hardware thread (CPU), in the form of the interrupt controller. DIP can interrogate the interrupt controller to determine all interrupts that are currently asserted.

Compiler/kernel cooperation DIP's injected poll checks must occur regardless of whether the kernel or a user program is running. The kernel must therefore allow the user process to read from the hardware interrupt aggregation device, at least to determine if any interrupt has been aggregated. Conversely, it must be able to trust that these polls happen as expected. To do this requires a trust relationship between the compiler and the kernel, specifically to determine if an application binary was produced by a compiler that did poll injection.

Interrupt stealing Here, the interrupt is delivered by simply invoking the originally registered interrupt handler via a call instruction instead of hardware-based interrupt dispatch. The interrupt is then deasserted by the DIP software by updating the interrupt controller state, instead of allowing the interrupt controller to proceed with the interrupt.



(a) Traditional Hardware Interrupts

(b) Dispersed Interrupt Polling (DIP)

Figure 4. Comparison of external asynchronous event delivery via traditional hardware interrupts and via dispersed interrupt polling, DIP. DIP compilation injects poll checks throughout the user and kernel code, and attests to this injection for user code. A poll check detects if any event has occurred by inspecting the interrupt aggregation state of the APIC or similar hardware device. If it has, the poll check dispatches the interrupt as a function call (or a syscall if the detection happened in user code.)

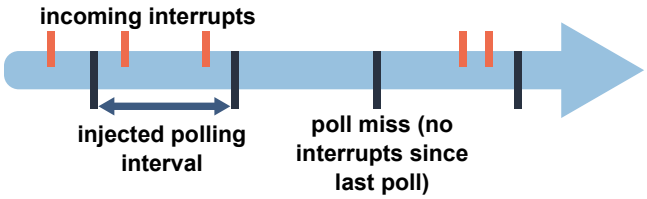


Figure 5. Timeline of DIP with fixed interval/rate polling. Compiler-injected polling calls occur periodically, and each call processes interrupts that have been aggregated since the previous call.

Putting it all together In DIP, all code is subject to a compilation process that includes injection of poll calls throughout. This injection process establishes two run-time invariants. The first is that there is no code path along which poll calls do not happen. The second is that poll calls happen, dynamically, at a target polling rate. A polling runtime is linked into the kernel and each user program, which the poll calls invoke. When the kernel boots, it turns off interrupts, relying on the injected poll calls instead. Each poll call interrogates the interrupt controller to determine if any interrupt is asserted. If the poll call is part of the user process, detecting an interrupt assertion results in a downcall to the kernel. If the poll call is within the kernel, or on a downcall, it "delivers" each asserted interrupt, in priority order, by calling the registered interrupt handler for its vector. The EOI invocation by the handler is implemented by forcing the interrupt controller to deassert the interrupt. No interrupt is actually ever delivered by the interrupt controller except in this manner.

Figure 5 illustrates what happens across time on a single CPU. Incoming interrupts are aggregated by the interrupt controller. Periodic polling calls, injected by the compiler, process these aggregated interrupts. Note that the interval between polls, combined with the cost of the poll, defines the overhead and latency of DIP. The shorter the interval is (or the higher the cost of a poll is) the higher the overhead is, and the lower the latency is. The figure also illustrates that DIP pushes the cost of interrupt handling to deterministic points in time defined by the polling interval. This is unlike traditional interrupt handling in which the interrupt costs are borne at the time the interrupt arrives.

3.3 Hardware & Software Requirements

The DIP model places requirements on the hardware and software of the system that are more strict than traditional interrupt delivery or polling. The hardware requirement is that the interrupt controller must be able to (a) aggregate interrupts, (b) make aggregated interrupts visible to software, and (c) allow software to clear (EOI) an interrupt without actually delivering it in the traditional way (i.e, the hardware must allow interrupt stealing). In our experience (a) and (b) are commonplace (verified on x64, RISC-V, and ARM). (c) is less common (RISC-V supports it in a limited capacity). Our prototype systems use x64 (§5.1) and RISC-V (§6.1). ARM is suboptimal for the purposes of DIP, given the variable, and potentially large, number of MMIO registers that would need to be polled from user-space code.

The software requirements are essentially that all code in the system be provably subject to the compiler transformation. This includes separate compilation and attestation of user-level code—the poll-injecting compiler must be able to assert that the binary was produced by it (via code-signing

for example), and the kernel must be able to verify this signature and determine whether the compiler is trusted. This is a common requirement and has been addressed in other systems that also must trust the compiler (such as in [35]).

A joint hardware/software requirement exists: protection. The user-level code should be able to read the aggregated interrupt state ((a)+(b) above) without a system call. There are several possible ways to expose such state, primarily developed for virtualizing interrupt controllers, such as AVIC ([15, Chapter 15, Section 15.29]), and techniques for leveraging such virtualization in an ordinary process context, such as DUNE [8] or virtines [38]. Very fine grain access control is possible. For example, the Intel X2APIC model provides access to APIC registers via MSRs, and the Intel hardware virtualization support would allow a DUNE-like tool to provide read-only access on a per-register basis. When a poll in attested user code detects that interrupts are available to be stolen, it will invoke the kernel to do so, and the cost of this invocation will be masked by the interrupt handling costs. Alternatively, the poll could queue interrupts it detects on a shared page with the kernel, and handling these interrupts could then be piggybacked on the next system call.

3.4 Intrinsic Limitations

Fixed-Cycle Interval Injection Compiler-based fixed-cycle interval injection is not perfect, it is an approximation. This is because the latencies used in the compile-time transformation to choose points of injection are estimates of how long LLVM IR instructions will actually take during runtime after being emitted from the compiler backend. These timing methods will have some level of inaccuracy, and we address this in our discussion of the results of Beandip UX (§5.3).

4 Beandip Compilation & Runtime

As explained in §2.2, we make use of the CT [20] compile-time transformation to instrument kernel and application code with polling calls in our prototypes. The transformation injects calls to a poll function at approximately fixed-cycle intervals by estimating LLVM instruction latencies at compile-time, then tracking the accumulatation of these latencies at runtime. Tracking loop latencies on tight inner loops can harm performance, but we counteract this with optimization techniques which can lift polls out of loops with known iteration counts, as well as select from modified versions of a loop at runtime to minimize performance impact.

The poll function is fundamentally the same on each architecture, with minor variations to account for differences in each aggregating interrupt controller. The function first reads a register, often memory mapped, which indicates if any interrupt is pending. Then, if there is an interrupt pending, the interrupt must be handled. If the poll was done in the kernel, we simply call the corresponding interrupt handler. If the poll happened in a user space program, the program yields or makes a system call to enter the kernel, where the interrupt can then be handled. Carefully written C code and assembly, alongside the cheap cost of polling (§2.3) on modern architectures, allows us to bring the cost of these poll functions to a minimum in our prototype implementations for x64 (§5.1) and RISC-V (§6.1).

5 Beandip UX Prototype

The purpose of this prototype is to evaluate DIP on Linux user space programs, using server-grade hardware (x64), along with substantial, real-world benchmarks and applications. This portion of the work explores why the DIP model is *viable*, before diving into how it could be *beneficial* in our RISC-V prototype (§6).

5.1 Implementation

Beandip UX applies DIP to user space programs, but not to the kernel. In this prototype for x64, DIP user space programs must register themselves with a custom kernel module. The kernel module serves to disable the interrupt flag (`%rflags.IF`) of the user space program, preventing hardware interrupts from being delivered while the program runs. The kernel module also maps the x64 MMIO interrupt pending registers (IRRs) as read-only into the user address space via the Linux *mmap* interface, making these registers readable from the user space program.

Returning to the DIP requirements (§3.3), we find that most, but not all, requirements are met on x64. The x64 APIC aggregates interrupts going into the core, and makes those interrupts visible to software, either via MSRs, when the X2APIC is enabled, or via MMIO registers, when the X2APIC is disabled. This fulfills two hardware requirements for DIP. However, we find the interrupt stealing requirement is not met on x64, as the architecture requires that the interrupt be dispatched in order for the interrupt's pending bit to be cleared from the IRRs. In this prototype, we overlook this unmet requirement of x64, and allow the interrupt to fire eventually so that the pending bit can be cleared.

As mentioned above, this prototype does not guarantee that all code is subject to the Beandip compilation process, as we do not apply the transformation to the Linux kernel. Given that x64 does not allow interrupt stealing, we opted to restrict this prototype to user space programs, and instead demonstrate full system (kernel and user space) support for DIP on RISC-V via Beandip KR. It is critical to evaluate the large scale benchmarks and applications targeted by this prototype nonetheless, given that our RISC-V evaluation is done on a 4-core SBC (§6.2). The benchmarks evaluated here spend very little time in the kernel, as detailed in our results (§5.3), making the impact of this design choice on our Beandip UX results negligible. Furthermore, we do not actually implement attestation to confirm user-level code has been compiled with our transformation, as prior work has already addressed this very problem (§3.3).

The joint hardware/software protection requirement of

DIP is met by mapping the IRRs as read-only into user space. These are 8 memory-mapped 32-bit registers which indicate interrupt vectors that are pending in the APIC, so the user space program can poll these after they are mapped. Ideal hardware for DIP would only need to expose one register to user space, indicating whether or not any interrupt is pending. Therefore, in our evaluation of this prototype, while we do have the ability to poll all 8 registers, we only poll 1 register, one which contains the timer interrupt and all IPIs, as polling one location would be sufficient on an ideal architecture for DIP. The 7 other registers see interrupts very infrequently, except when network interrupts are occurring, in which case we poll one additional register.

In this prototype, when a poll hit occurs, the user space program simply downcalls to the kernel with a yield. Interrupts become enabled again on entry to the kernel, and this is where the hardware interrupt is allowed to fire, clearing the pending bit for that interrupt. On an ideal DIP architecture, the downcall would go directly to the interrupt handler, and then the pending interrupt would be cleared in software via interrupt stealing. Unfortunately, as x64 does not support this, we must allow the hardware interrupt to dispatch. The fundamental concept of DIP is still present in our transformed user space programs, as interrupts cannot be dispatched while user programs are running - the user programs *must* poll the IRRs to allow CPUs to handle interrupts.

It is important to understand that because interrupt dispatch still happens within this prototype, measurements of the prototype will consistently overestimate the overhead of DIP. In a DIP implementation that allowed true interrupt stealing, the time spent in interrupt dispatches would entirely disappear, as we see in Beandip KR on RISC-V.

5.2 Testbed and Setup

Our testbed setup for Beandip UX consists of a midrange single socket server containing an AMD EPYC 7443P 24-Core (48 thread) 2.85 GHz processor and 64 GB of memory. Ubuntu 22.04 with Linux kernel 5.15 is used. The machine is configured with X2APIC disabled, which enables MMIO access to APIC IRRs (pending interrupt registers).

We run our experiments on a range of benchmarks (Embench [9], GAP [6], NAS 3.0 [4], SPEC2017) and applications (Postgres [36]). GAP and NAS are compiled with OpenMP to use 48 threads, with GAP using Kronecker graphs and 2^{24} vertices (unless noted), and NAS using class B. Embench and SPEC2017 are run single threaded.

For our Postgres experiments, we test throughput with a few OLTP workloads. Specifically, we use pgbench (packaged with Postgres) to send a variety of transactions (pgb), as well as select-only transactions (pgb_s), sending from 100 clients. We also run a TPCC-like [1] workload (a more complex OLTP workload) using 48 clients.

5.3 Results

In the following section, we ask and answer a number of questions to evaluate the viability of DIP as an alternative to hardware interrupts.

Can DIP achieve a reasonable polling interval? We begin by choosing a reasonable polling interval of 5,000 cycles. This implies an average poll latency of ~2,500 cycles, which falls close to previously measured hardware interrupt latency on an x64 machine of ~1,000 cycles (§2.3). Figure 6 shows the mean achieved poll intervals for this target, along with standard deviations. While most benchmarks achieve the target, some SPEC2017 benchmarks stray because they have significantly greater numbers of system calls and library calls. This code is not compiled with Beandip (though a full DIP system would compile it), making our measured poll intervals larger than desired and increasing standard deviations for these benchmarks. The omnetpp benchmark has a particularly high achieved rate because of its high usage of memset and memcpy. We chose to compile these functions as part of the application, but our compiler pass still struggles to make latency estimations for them, indicating that improved instruction latency tuning techniques may be needed for certain cases.

What is the runtime overhead of DIP at a reasonable polling interval? Figure 7 shows overheads of DIP relative to hardware interrupts at a target interval of 5,000 cycles with no additional load being applied to the system. No external load is the worst case for DIP in terms of overhead. Still, we achieve a low geomean of just 11% overhead. Parallel benchmarks (NAS & GAP) perform the best, while sequential benchmarks with tight inner loops perform the worst (Embench). In particular, huffbench performs poorly because it contains linked list traversals, making some loop optimizations impossible. These benchmarks spend <1% of their time in the kernel on average, justifying our claim that not transforming the kernel in this prototype has negligible impact. Additionally, we experimented with swapping the poll call in these benchmarks for a nop instruction to simulate what would happen if the poll was virtually free (1 cycle). This nop experiment achieves a geomean of just 7% overhead.

Can DIP's target polling interval be varied effectively? Yes. Beandip UX uses the cycle counter at runtime to improve timing accuracy, avoiding polls when they are made too close in sequence, but this still comes at the cost of some performance. Figure 8a shows the average measured polling interval as a function of the desired polling interval. The linear relationships are precisely what we desire. GAP benchmarks tend to overshoot the target interval, which is in part due to our measurements including time spent in the kernel and library calls. Also, as NAS and GAP benchmarks run in parallel with 48 threads (on a 48 thread machine),

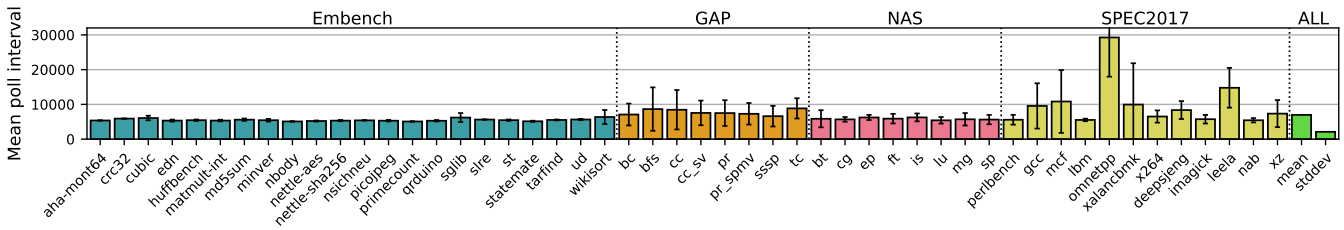


Figure 6. Beandip UX achieves target interval of 5,000 cycles relatively well. Bars display standard deviation, not standard error. Standard errors <65 cycles across all benchmarks.

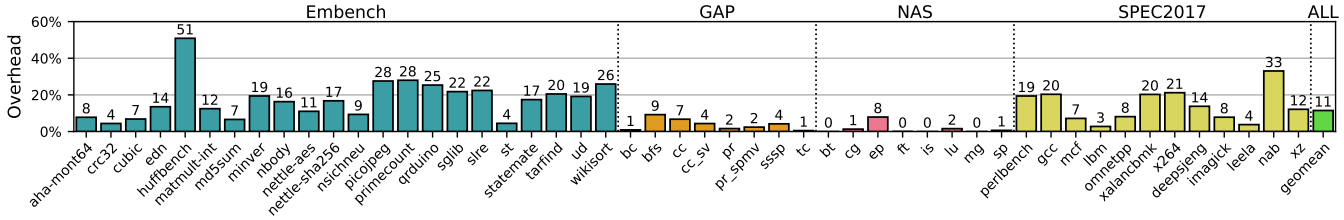


Figure 7. Relatively low overhead of Beandip UX without interrupt load (worst case for DIP), target interval 5,000 cycles

threads will occasionally get context switched or moved between CPUs, impacting our measurements. Timing scheme parameters could be further tuned to improve the accuracy of benchmarks that overshoot the target.

Does the polling interval create a tradeoff between latency and overhead? Yes. Figure 8b shows that as the target poll interval increases (leading to greater interrupt handling latency), overhead decreases. Overhead is high at high polling rates, but it quickly drops off and starts to flatten. This figure further justifies 5,000 cycles as a reasonable target interval, as this target interval falls around the point in the graph where the steep dropoff in overhead ends. However, if low interrupt handling latency is critical, one could choose to pay the price, taking a high overhead penalty.

How does the overhead change with interrupt load? To answer this question, we apply an interrupt load to our testbed machine in the form of network interrupts. We send a constant stream of the tiniest UDP packets from a neighboring machine (connected by a switch), to an unbound port on the testbed. The testbed machine's NIC receives the packets and generates an interrupt when there are enough packets in the buffer (coalescing at the hardware level), then the kernel processes the packets. Their lifecycle ends there, so this is an effective way to control the load we generate on the test machine. We configured the test to deliver ~100,000 interrupts/second. In the default Linux configuration, these interrupts are all sent to a single CPU, creating an imbalanced load for one CPU when we run multithreaded benchmarks. Like with hardware interrupts, our prototype does not attempt to balance this interrupt load.

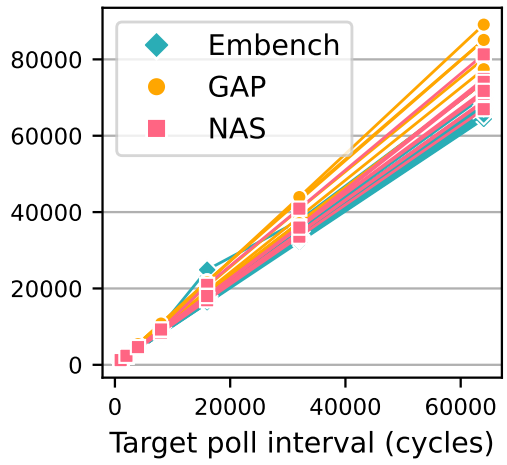
Figure 9 shows the overheads of hardware interrupts (baseline) as well as DIP when this load is applied, all relative to the runtime of these benchmarks with hardware interrupts on a load-free system. We tested a range of target intervals (4,000-16,000), and the target interval of 4,000 still achieves just 8% geomean overhead. The target of 16,000 is shown in the figure to showcase the best case for DIP. Some of the parallel benchmarks shown here are impacted significantly by the load imbalance (~30%), while others are not impacted at all. This comes down to the parallel algorithms being employed in each benchmark, and their ability to handle load imbalance. The takeaway here is that DIP performs better under load, achieving nearly identical overhead to hardware interrupts (baseline) under load.

Here, DIP effectively acts as a method of software coalescing (on top of hardware coalescing provided by the NIC, in this case) to reduce the noise caused by the interrupts. Beandip UX processes the same number of packets as the hardware interrupt version, but it does so by handling fewer interrupts, as multiple hardware interrupts would occur over the course of a single poll interval, which DIP ends up batching together. This does not impact correctness, as packets can simply build up on the network card buffer while waiting for a poll point to be reached (although it does run the risk of packets being dropped if the poll period is too long).

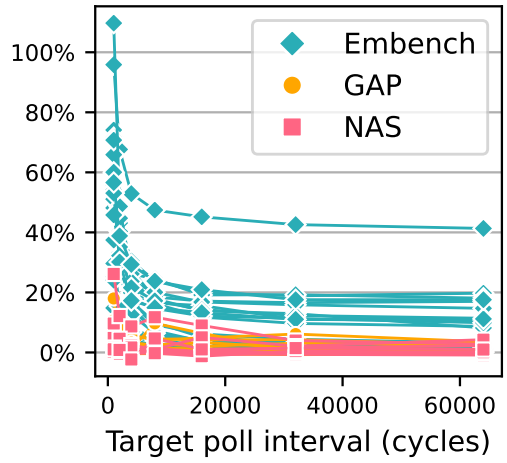
An important note is that while DIP is able to achieve nearly identical overhead to hardware interrupts under load here, it is doing unnecessary extra work due to the limitations of x64. Specifically, since x64 does not support interrupt stealing, DIP must waste additional time in these benchmarks allowing the interrupt dispatch so that interrupt pending bits can clear and polling can continue. This begs the question of whether DIP can achieve speedup when this wasteful work is eliminated, which we will address in the results of our RISC-V prototype (§6.3).

Do particular workloads benefit from DIP when under interrupt load?

Yes. Figure 10 displays two outliers not shown in Figure 9, where hardware interrupts and DIP have interrupt load applied to them via network interrupts. These outliers show a



(a) Poll interval accuracy (cycles)



(b) Percent Overhead

Figure 8. (a) Poll interval is relatively accurate accurate across a range of targets. (b) There is a tradeoff between poll overhead and interrupt handling latency.

large overhead of 3-4x for hardware interrupts under load. LU displays massive improvement by using DIP, while SP does not show significant improvement. LU displays this behavior because LU executes a pipeline that overlaps different invocations of a loop in parallel. A single imbalanced CPU can cause a stall in the pipeline, but DIP is able to reduce the noise from interrupts on this CPU, mitigating the impacts of the stall. In SP, there are a significant number of parallel loops without too much speedup over the sequential version. In this case, any amount of noise or load imbalance will significantly harm the benchmark. The key takeaway is that the DIP model can help certain workloads which do not handle load imbalance well, when run on systems under load.

Do substantial, real-world applications like Postgres work with DIP? Yes. We apply Beandip to Postgres to demonstrate the full functionality of Beandip UX on a real-world application. We allow Postgres to use up to 48 worker threads on our testbed machine, and run pgbench and TPCC benchmarks from a neighboring machine which connect to this database. Figure 11 shows that the throughput of Postgres with DIP is 90% that of Postgres using hardware interrupts. If interrupts did not have to dispatch to be de-asserted (recall that x64 does not support interrupt stealing), the Beandip UX throughput could be improved even further. The takeaway is that real-world, multithreaded applications like Postgres work with DIP, and DIP does not significantly impact their performance.

6 Beandip KR Prototype

The purpose of this prototype is to evaluate a full Linux system on RISC-V, where both the Linux kernel and unmodified user space code have been compiled with our transformation. This prototype closely follows the DIP model, with some exceptions due to hardware limitations. Our analysis on x64 allows us to confidently say that DIP is *viable*, and this portion builds on this groundwork to show why DIP can be *beneficial*, when our prototype is not hindered by the limitations of x64 (lack of interrupt stealing, in particular).

6.1 Implementation

Returning once again to the DIP requirements (§3.3), we find that all requirements are met on RISC-V for a subset of all hardware interrupts. Specifically, the PLIC aggregates external device interrupts, makes these interrupts visible to software via the PLIC claim MMIO register (which can typically be mapped to user-space), and allows interrupt stealing for external interrupts when these hardware interrupts are disabled on a CPU. Thus, RISC-V meets the DIP requirements for external interrupts. However, in their current design, IPIs and timer interrupts are delivered via the CLINT, which cannot be made accessible from user-space. Nonetheless, RISC-V is still the closest approximation to ideal hardware for DIP given that it is the only modern architecture (to our knowledge) which has a single pending interrupt register for an aggregated set of interrupts which can be polled from user-space. Additionally, we claim that handling external interrupts is sufficient to demonstrate DIP applied to a full system, given that timer interrupts can be eliminated via Compiler Timing [20], and IPIs could be simulated via shared-memory mechanisms.

For this prototype, the Linux kernel is compiled statically with Clang LTO, allowing the Beandip transformation to be applied to all of the kernel code. Minimal changes are made directly to the kernel itself (<100 LOC) to disable dispatch of external hardware interrupts, modify the CPU idle function to prevent a *wait for interrupt* instruction, and add function call pathways into the PLIC interrupt handler for poll hits. Additional code is needed to map the PLIC claim register

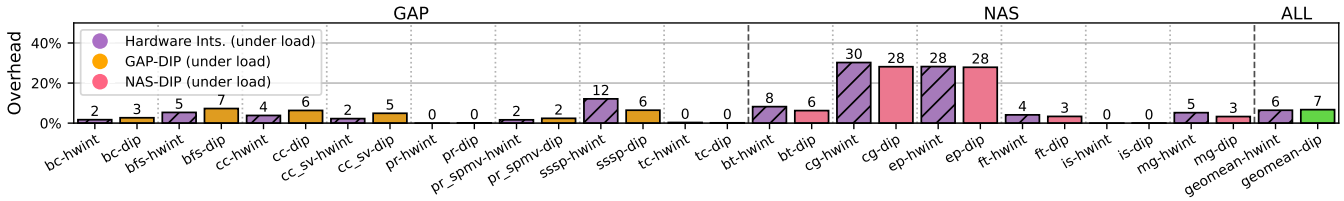


Figure 9. The best case for DIP is high interrupt load. BEANDIP UX (target interval 16,000) and hardware interrupts under heavy interrupt load are shown, all relative to hardware interrupts under no load. Ability of DIP to handle load is virtually identical to hardware interrupts across benchmarks, effectively making up for any overhead introduced by polling when load is applied

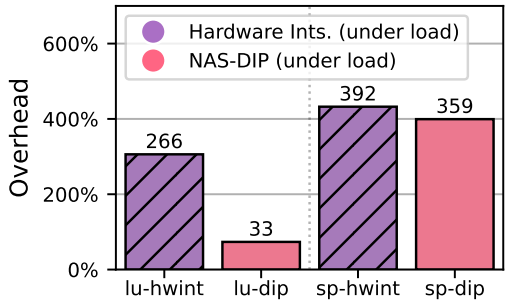


Figure 10. Two outliers exist that are heavily impacted by network load, one of which DIP helps significantly

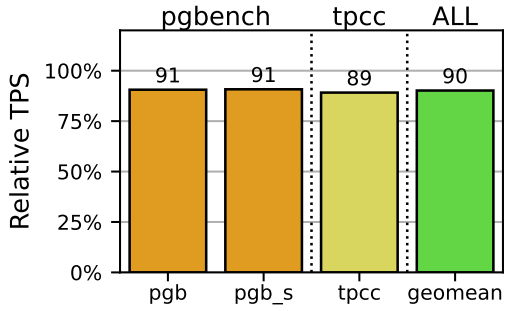


Figure 11. Postgres with BEANDIP UX achieves 90% TPS (transaction/sec) relative to Postgres with hardware ints.

into user-space, though this is very similar to the APIC mapping code of BEANDIP UX, and can be factored into a kernel module for a non-static build.

The compiler-injected kernel poll code is extremely simple, polling the PLIC claim register for the current CPU, then directly calling the PLIC interrupt handler on a poll hit. This follows the exact same pathway as a hardware interrupt would, but without the added overhead of a context switch. The compiler-injected userspace poll code reads the mapped PLIC claim register as well. Upon a poll hit, a system call is made to our ioctl handler in the kernel, which then calls the PLIC interrupt handler. Again, this is all done without the hardware interrupt actually firing.

6.2 Testbed and Setup

Our testbed here consists of a StarFive JH-7110 4-core 1.5 GHz processor on a VisionFive 2 SBC with 8 GB of memory.[4]

[4]We settled on this machine after thoroughly testing user-space mapping of PLIC MMIO for the 64-core SOPHON SG2042 on the Milk-V Pioneer board, without success.

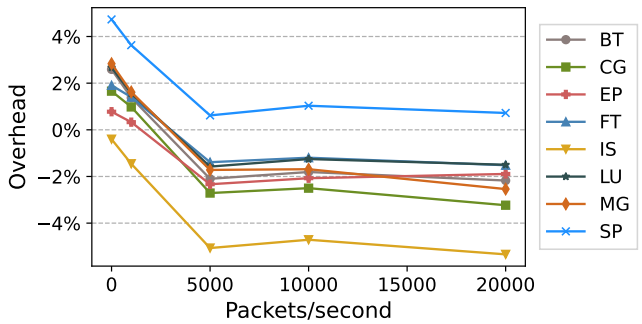


Figure 12. UDP packets of 100 bytes are sent at fixed rates, showing that BEANDIP KR can outperform baseline hardware interrupts under the same load. Lower is better, negative overhead is speedup.

Debian 12.7 with Linux kernel 6.1.31 is used, compiled with Clang LTO for both baseline and BEANDIP KR kernels.

We run our experiments on single-threaded NAS benchmarks (class A), locking the benchmark thread to CPU 0, where all external hardware interrupts are directed. To apply interrupt load to the system, a separate machine sends UDP packet at fixed target rates. Similar to our interrupt load experiment in §5.3, this models a noisy system, busy with interrupts, that is also running a compute-bound or memory-bound workload. The CPU needs to regularly process asynchronous events that are arriving, taking time away from the workload. A socket is opened on another thread of the RISC-V machine and locked to a free CPU, verifying that UDP packets are arriving.

In our benchmarks, a baseline system is one which uses hardware interrupts and does not have our compiler transformation applied to either the kernel or the user-space code. As with many BEANDIP UX experiments, the target polling interval is set to 5,000 cycles for both the kernel and the benchmarks. This target is mostly met, with mean achieved intervals on benchmarks not exceeding 10,000 cycles. Unlike in our BEANDIP UX prototype, we did not add runtime cycle counter checks to throttle polling rates here, as polls on this platform are much less expensive than on x64.

6.3 Results

Unlike on BEANDIP UX, the conditions are right for speedup to be achieved on BEANDIP KR. While x64 (the BEANDIP UX architecture) does not support interrupt stealing, RISC-V

does. This means that there is no time wasted in dispatching interrupts to allow for polling to proceed, a time sink which is unavoidable on x64. On RISC-V, the PLIC claim register can be polled and pending interrupts can be cleared entirely from software, meeting the interrupt stealing requirement.

Speedup can be achieved, as seen in Figure 12, with high load being the best case for DIP, relative to hardware interrupts. As load increases from UDP packets, the prototype acts as a software coalescing mechanism, allowing packets to accumulate before the next poll hit occurs. At this point, the network driver code is invoked, which processes all the packets that have accumulated in the buffer. We confirm that virtually all packets arrive at the receiving socket, indicating that the same amount of useful work is still being done during interrupt handling.

Compared to this software coalescing mechanism, our testbed SoC ethernet device does minimal hardware coalescing, allowing Beandip KR to outperform hardware interrupts. From 0-5,000 packets/second, one hardware interrupt typically arrives per packet. At the rate of 5,000-20,000 packets/second, the baseline receives around 4,000-5,000 hardware interrupts/second, whereas only 300-400 poll hits/second occur for Beandip KR. At 5,000 packets/second, the NIC itself begins interrupt coalescing, which limits the interrupt rate to 5,000 interrupts/second. This explains the flattening of the curves after 5,000 packets/second. Given higher interrupt rates, we would expect the linear trend to continue. Still, Beandip KR achieves speedups of up to 5%, due to having <10% of interrupts manifest as poll hits. These effects are compounded with the lack of costly interrupt dispatches for Beandip KR, where the greatest benefit is seen during kernel poll hits that require no context switch whatsoever. This makes a strong case for DIP as a model that can enable software coalescing and avoid costly hardware interrupts to achieve speedup under load.

7 Future Work

Exploiting the flexibility of DIP's polling interval In some applications, the interrupt rate varies (Figure 2). DIP's flexibility enables us to tune the polling interval to minimize the event handling latency within a constraint on the overhead. We are exploring dynamic tuning of DIP at run-time using interrupt rate prediction models.

Hardware support To our knowledge, no architecture supports the requirements of §3.3 perfectly. The RISC-V PLIC does not expose IPIs in the *claim* register, ARM does not expose a single register to user space, and x64 does not support interrupt stealing. We are exploring exposing a single interrupt CSR/MSR register to user space via hardware extensions, using Chipyard [2] to implement this on RISC-V.

8 Conclusion

We propose *Dispersed Interrupt Polling* (DIP), a viable alternative to hardware interrupts that deterministically distributes polls throughout kernel and application code, polling for *all* interrupts at regular intervals. This model removes the nondeterminism introduced by hardware interrupts, and has potential performance benefits, given the high cost of hardware interrupts on systems under load. Furthermore, the flexibility of the DIP model offers great potential, as it allows developers to make tradeoffs between overhead and interrupt handling latency as they see fit. Our prototype systems apply the DIP model to both unmodified user space code and the entirety of the Linux kernel on modern architectures. These prototypes demonstrate overhead which is often <10% without load relative to hardware interrupts, virtually no overhead with load, speedups of up to 5% when DIP acts as a software coalescing mechanism, and minimal impact to substantial applications like Postgres. These results were collected on current hardware/software combinations that require compromises in the DIP implementation, and can be expected to be even better on future hardware that adds minimal DIP support. Given that interrupt rates on common applications can easily exceed 10,000/second on a single CPU, this makes a strong case for DIP as an alternative to nondeterministic hardware interrupts.

References

[1] Tpc benchmark c. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf, 2010.

[2] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020.

[3] Berk Aydogmus, Linsong Guo, Danial Zuberi, Tal Garfinkel, Dean Tullsen, Amy Ousterhout, and Kazem Taram. Extended user interrupts (xui): Fast and flexible notification without polling. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2025)*, page 373–389, March 2025.

[4] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[5] Nilanjana Basu, Claudio Montanari, and Jakob Eriksson. 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 2021, page 1249–1263, New York, NY, USA, 2021. Association for Computing Machinery.

[6] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[7] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. Optimizing i/o performance of hpc applications with autotuning. *ACM Trans. Parallel Comput.*, 5(4), mar 2019.

[8] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI 2012)*, October 2012.

[9] J Bennett, P Dabbelt, C Garlati, GS Madhusudan, T Mudge, and D Patterson. Embench: An evolving benchmark suite for embedded iot computers from an academic-industrial cooperative, 2022.

[10] R. Brightwell, K.T. Pedretti, K.D. Underwood, and T. Hudson. Seastar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3):41–57, 2006.

[11] Franck Cappello, Amina Guermouche, and Marc Snir. On communication determinism in parallel hpc applications. In *2010 Proceedings of 19th International Conference on Computer Communications and Networks*, pages 1–8, 2010.

[12] Seung-Jun Cha, Seung Hyub Jeon, Yeon Jeong Jeong, Jin Mee Kim, and Sungin Jung. Os noise analysis on azalea-unikernel. In *2021 23rd International Conference on Advanced Communication Technology (ICACT)*, pages 81–84, 2021.

[13] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. Deterministic replay: A survey. *ACM Comput. Surv.*, 48(2), sep 2015.

[14] Juan A. Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miguel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B. Bartolini, Nitesh Mor, Krste Asanović, and John D. Kubiatowicz. Tessellation: Refactoring the OS around explicit resource containers with continuous adaptation. In *Proceedings of the* 50^{th} *ACM/IEEE Design Automation Conference (DAC 2013)*, pages 76:1–76:10, May/June 2013.

[15] AMD Corporation. AMD64 architecture programmer's manual: Volume 2: System programming, January 2023.

[16] Andrew Crotty, Viktor Leis, and Andrew Pavlo. Are you sure you want to use mmap in your database management system? In *CIDR 2022, Conference on Innovative Data Systems Research*, 2022.

[17] Daniel Bristot de Oliveira, Daniel Casini, and Tommaso Cucinotta. Operating system noise in the linux kernel. *IEEE Transactions on Computers*, 72(1):196–207, 2023.

[18] Peter Dinda, Xiaoyang Wang, Jinghang Wang, Chris Beauchene, and Conor Hetland. 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.

[19] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, December 1992.

[20] Souradip Ghosh, Michael Cuevas, Simone Campanoni, and Peter Dinda. Compiler-based timing for extremely fine-grain preemptive parallelism. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[21] Kyle Hale and Peter Dinda. 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.

[22] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3A: System Programming Guide, Part 1*, March 2023.

[23] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 466–481, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[25] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[26] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association.

[27] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. *SIGCOMM Comput. Commun. Rev.*, 44(4):175–186, aug 2014.

[28] Sohit Mehta. User interrupts: A faster way to signal. In *Proceedings of the Linux Plumbers Conference (LBC 2021)*, September 2021.

[29] Alessandro Morari, Roberto Gioiosa, Robert W. Wisniewski, Francisco J. Cazorla, and Mateo Valero. A quantitative analysis of os noise. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 852–863, 2011.

[30] Sarah Neuwirth and Arnab K. Paul. Parallel i/o evaluation techniques and emerging hpc workloads: A perspective. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 671–679, 2021.

[31] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *Proceedings of the* 16^{th} *USENIX Conference on Networked Systems Design and Implementation (NSDI 2019)*, February 2019.

[32] Xuehai Qian, Koushik Sen, Paul Hargrove, and Costin Iancu. Sreplay: Deterministic sub-group replay for one-sided communication. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Tristan Ravitch. Wllvm. https://github.com/travitch/whole-program-llvm, 2016.

[34] Smruti Ranjan Sarangi, Brian Greskamp, and Josep Torrellas. Cadre: Cycle-accurate deterministic replay for hardware debugging. *International Conference on Dependable Systems and Networks (DSN'06)*,

pages 301–312, 2006.

[35] Brian Suchy, Souradip Ghosh, Aaron Nelson, Zhen Huang, Drew Kersnar, Siyuan Chai, Michael Cuevas, Alex Bernat, Gaurav Chaudhary, Nikos Hardavellas, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. CARAT CAKE: Replacing paging via compiler/kernel cooperation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS)*, March 2022.

[36] The PostgreSQL Global Development Group. Postgresql 15.2 documentation. https://www.postgresql.org/docs/15/index.html, February 2023.

[37] Steven H. VanderLeest. Taming interrupts: Deterministic asynchronicity in an arinc 653 environment. In *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, pages 8A3–1–8A3–11, 2014.

[38] Nicholas C. Wanninger, Joshua J. Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C. Hale. Isolating functions at the hardware limit with virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 644–662, New York, NY, USA, 2022. Association for Computing Machinery.

[39] Editors Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA,*. RISC-V Foundation, document version 20191213 edition, Dec 2019.

[40] Vincent M. Weaver, Dan Terpstra, and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224, 2013.

[41] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 3, USA, 2012. USENIX Association.

[42] Heqing Zhu. *Data Plane Development Kit (DPDK): A Software Optimization Guide to the User Space-Based Network Applications*. CRC Press, 2020. Detailed DPDK materials are available from dpdk.org.