Out of Hand for Hardware? Within Reach for Software!

Zhihong Luo
UC Berkeley

Silvery Fu
UC Berkeley

Emmanuel Amaro
VMware Research

Amy Ousterhout
UC San Diego

Sylvia Ratnasamy
UC Berkeley

Scott Shenker
UC Berkeley & ICSI

ABSTRACT

Events that take 10s to 100s of ns like cache misses increasingly cause CPU stalls. However, hiding the latency of these events is challenging: hardware mechanisms suffer from the lack of flexibility, whereas prior software mechanisms fall short due to large overhead and limited event visibility. In this paper, we argue that with a combination of two emerging techniques – light-weight coroutines and sample-based profiling, hiding these events in software is within reach.

CCS CONCEPTS

• **Software and its engineering** → *Coroutines*; *Compilers*; Software system structures; Concurrency control.

KEYWORDS

CPU stall, coroutine, profile-guided yield instrumentation, asymmetric concurrency

ACM Reference Format:
Zhihong Luo, Silvery Fu, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. 2023. Out of Hand for Hardware? Within Reach for Software!. In *Workshop on Hot Topics in Operating Systems (HOTOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3593856.3595898

1 INTRODUCTION

To avoid wasting processor cycles while waiting for the result of some long event, an effective strategy is to hide the event latency by concurrently executing independent instructions. Applying this strategy to either hardware or software, people have arrived at satisfactory solutions to events with durations at *both ends* of the spectrum: for events that take a very small amount of time (*e.g.,* less than 10 ns), such as L1 misses and complex arithmetic instructions, hardware mechanisms like out-of-order executions can efficiently detect them and instantaneously interleave instructions to minimize CPU stalls [2, 59]; for events that run for sufficiently long (*e.g.,* over 1 μs), such as disk I/O and using offboard accelerators (*e.g.,* GPU), software mechanisms like OS process scheduling offer great flexibility with reasonable overhead and provide functionalities like on-demand scaling of concurrency and fine-grained control over application performance [60, 64].

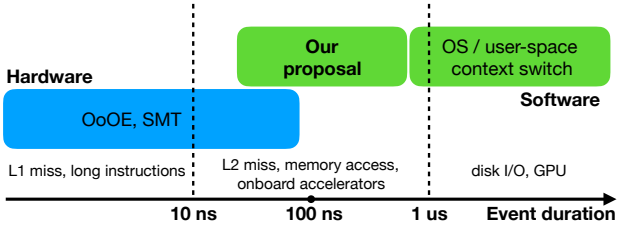

Figure 1: Hiding events of different durations: existing hardware and software mechanisms and our proposal; **OoOE:** out-of-order executions, **SMT:** simultaneous multithreading.

However, the solution is less clear for events with durations in the *middle* of the spectrum, ranging from 10s to 100s of ns, such as L2 cache misses, memory accesses and operations with onboard accelerators. Events in this range account for a significant portion of CPU stalls – some widely-used modern applications lose more than 60% of all processor cycles due to memory-bound CPU stalls [3, 13, 31, 62], and are getting prevalent – there is an increasing number of onboard accelerators in modern server processors [26, 32]. For these events, hardware mechanisms like simultaneous multithreading (SMT) (*e.g.,* Intel's Hyper-threading) suffer from limitations due to their lack of *flexibility*, which is manifested in two aspects: limited degrees of concurrency and negative impacts to application performance. In terms of degrees of concurrency, modern CPUs have only 2 to 8 threads per physical core, which is insufficient for SMT to fully hide the latency of events like memory accesses [28, 31, 53], especially for applications that have large memory footprints and thus frequently incur cache misses (*e.g.,* data analytics [5, 70, 71]). In terms of application performance, SMT is known to likely lead to significantly increased latencies [24, 55, 67, 68]. This is because SMT focuses solely on multiplexing instruction streams to best utilize core resources, without explicitly managing the impacts to application performance, which the hardware has little visibility to. While there are proposals [25, 67] that mitigate these issues by redesigning the hardware (*e.g.,* supporting a large number of software-controlled

HOTOS '23, June 22–24, 2023, Providence, RI, USA

ACM ISBN 979-8-4007-0195-5/23/06.
https://doi.org/10.1145/3593856.3595898

hardware threads), these proposals require significant hardware changes and are thus not feasible today.

Since relying on hardware to handle events of medium durations is unsatisfactory, how about hiding them in software? Unlike hardware mechanisms, software mechanisms have the flexibility to support high degrees of concurrency and minimize negative impacts to application latency: the former is due to using software contexts, and the latter is due to controlling application performance while hiding events for CPU efficiency. However, hiding these events in software is highly challenging due to software's lack of efficiency, in the form of *large switching overhead* and *limited event visibility*. In terms of switching overhead, for traditional threads of executions like OS processes and kernel threads, context switches take several hundreds of nanoseconds or even a few microseconds [14, 38], which is prohibitively expensive for hiding the target events. In terms of event visibility, a software mechanism must be able to detect the presence of an event in order to hide it, which is challenging for events like cache misses that are not exposed to software.

Fortunately, we see a way forward by mitigating the aforementioned inefficiency of software mechanisms via a novel combination of two emerging techniques: light-weight coroutines [17, 44, 63] and sample-based profiling [10, 35, 66]. First, by adopting cooperative multitasking, light-weight coroutines support fast context switchings that take only several nanoseconds [6, 36], allowing us to interleave coroutine executions with low overhead. Second, sample-based profiling leverages hardware performance counters available in modern CPUs to sample hardware events of interest in production with negligible overhead. The profiled information is then used to guide instrumentation of coroutines, so that an instrumented coroutine will appropriately yield to hide the latency of events. Fundamentally speaking, sample-based profiling provides software with the much needed visibility to hardware events, while allowing flexibility based on application characteristic. Note that both techniques require no changes to existing hardware and are getting adopted in production systems [27, 51, 56, 72], which makes them ideal building blocks for easily deployable software mechanisms.

To demonstrate the feasibility of this idea, we present a design proposal targeting L2/L3 cache misses, where we walk through the set of important design choices that one needs to make as they try to leverage light-weight coroutines and profile-guided instrumentations. Our proposed mechanism is carefully designed to meet three properties that we believe can facilitate adoption of the mechanism: transparent interface, general applicability and controllable latency. We elaborate on the rationale underlying our design choices in the hope of showing that our proposed design not only fulfills the desired properties, but more importantly exhibit many other possibilities future work can explore and investigate.

In the rest of this paper, we elaborate on the enabling techniques of our proposal (§2), present a design for hiding L2/L3 cache misses (§3) and discuss open questions (§4).

2 ENABLING TECHNIQUES

In our proposal, we leverage two enabling techniques to mitigate the aforementioned drawbacks of software mechanisms – light-weight coroutines to reduce switching overhead and sample-based profiling to obtain event visibility. Next, we elaborate on how they improve upon prior techniques and how these improvements facilitate our proposal.

Light-weight coroutines: coroutines are generalized subroutines whose execution can be suspended and resumed. Context switches of coroutines are orders of magnitudes cheaper than traditional threads of executions like processes and kernel threads. Being a user-space mechanism that resides in a single process, coroutine context switch requires no expensive system calls nor changes to the virtual memory mapping. Moreover, since the coroutine context switch is effected by a visible yield function call, it only needs to preserve a subset of registers (including instruction and stack pointer), defined by the calling convention, of the current coroutine and restore those registers of the resumed coroutine [36]. Thanks to these merits, recent coroutine implementations have brought the context switch latency down to less than 10 ns (*e.g.,* 9 ns for Boost's fcontext_t [6]). Moreover, there have been efforts on leveraging *compiler support* to further reduce the overhead [16, 46]. For instance, a compiler might determine a fewer number of registers that need to be preserved across a particular context switch. As we will discuss later, by instrumenting coroutines based on profiled data, our proposal is amenable to these compiler-side optimizations.

With the low switching overhead of coroutine, there have been recent works that interleave coroutine executions to hide memory accesses for pointer-based data structures in databases [23, 28, 53]. However, they do not address the issue of limited event visibility. Instead, they ask developers to decide where these events may happen (*e.g.,* loads that cause cache misses) and hard code event handlers at these locations (*e.g.,* issuing a prefetch instruction before switching to a different coroutine) at development time. This approach however requires significant engineering efforts – inferring the presence of short events is challenging and error-prone even for domain experts, and hinders wide adoption – manual rewriting is needed for legacy code. Moreover, as we will discuss in §3.3, yields inserted by developers are too sparse to allow fine-grained control over application performance.

Sample-based profiling: profile-guided optimizations (PGO), also called feedback-driven optimizations (FDO), is a compiler optimization technique that uses runtime information collected via profiling for improving program performance.

PGO has been proved highly effective for code optimizations [10, 48, 61]. Early efforts on PGO relied on instrumentation based profiling, which requires instrumenting the application to collect profile information. However, this approach not only complicates the build process, but also incurs significant CPU and memory overhead. More importantly, instrumentation-based profiling cannot easily support our proposal, because it is hard to obtain visibility into hardware events like L2/L3 cache misses with only instrumentation. Fortunately, to increase the adoption of PGO in production environments, recent work has instead focused on sample-based profiling [27, 33, 66], which relies on sampling using hardware performance counters available in modern CPUs, such as Intel's Precise Event Based Sampling (PEBS) [1] and Last Branch Records (LBR) [35]. Sample-based profiling requires no special build and incurs negligible run time overhead, both of which allows sample-based PGO to be widely deployed in production environments [10, 22, 48, 50, 51]. Most importantly, as we will elaborate later, sample-based profiling allows us to conveniently gather information on hardware events, *e.g.,* where and how frequently these events occur, which is then used for hiding these events in software.

3 A PROPOSAL

To illustrate how one can hide the latency of short events by intelligently combining light-weight coroutines and sample-based profiling, we next propose the design of an easily deployable software mechanism targeting L2/L3 cache misses.

3.1 Requirements

With a focus on deployability, we distill three requirements that we believe can facilitate adoption of the mechanism.

Transparent interface: the software mechanism should be transparent to both applications and developers. It should require no additional rewriting effort from the developer and should be applicable to any code structured in coroutines.

General applicability: the software mechanism should be applicable to a wide range of applications and implementations. Therefore, the mechanism must not depend on features or assumptions specific to certain programming languages, application domains, data structures *etc.* to properly function.

Controllable latency: the software mechanism should allow fine-grained control over application latency. It can thus be used with latency-sensitive applications to simultaneously achieve low latency and high CPU efficiency.

3.2 Profile-guided yield instrumentation

The proposed software mechanism follows the same procedure as prior systems that leverage PGO, which involves three logical steps: (i) running the original code (structured in coroutines) in production environments and collecting statistics about CPU stalls due to L2/L3 cache misses with sample-based profiling mechanism, (ii) instrumenting the coroutines so that they prefetch and yield to hide potential cache misses according to the profiled data and (iii) using the finalized code to interleave executions of instrumented coroutines at run time. Next, we elaborate on the set of design choices we make to enable transparent interface and general applicability in steps (i) and (ii). After that, we will introduce how we ensure controllable latency by supporting asymmetric concurrency in steps (ii) and (iii).

Hardware events to sample: for profiling, we first need to decide the set of hardware events to sample, so that the profiled data is useful to our mechanism. For hiding L2/L3 cache misses that cause CPU stalls, the ideal event would have informed us the number of stalled cycles due to L2/L3 cache miss for different load instructions. Unfortunately, to the best of our knowledge, such an event is not supported in today's CPUs.[1] To mitigate this issue, we propose to sample *multiple* events and combine their results, instead of relying on a single event. Specifically, we propose to sample both (i) load instructions that cause L2/L3 cache misses and (ii) the stalled cycles. We learn from (i) the set of load instructions that induce cache misses and correlate that with instructions causing CPU stalls from (ii), which leads us to load instructions that likely cause CPU stalls. Additional events can also be included to filter out stalls due to other reasons (*e.g.,* front-end stalls due to slow instruction fetching).

Besides the set of hardware events, there are other parameters to configure as well, such as the sampling frequency and the size of the in-memory buffer that temporarily stores sampled profiles. For these parameters, their corresponding trade-offs (*e.g.,* higher sampling frequency expedites profile collections at the cost of higher run time overhead) have been extensively studied in prior work [1, 47, 50], and our proposal can follow the established practices here.

Instrumentation level: after sampling hardware events in step (i), a key design decision we need to make for step (ii) is at what level in the compilation pipeline we perform instrumentation, ranging from source code [9], to the compiler's intermediate representations (IR) [10], to the post-linked binary [50]. Operating at each level comes with its pros and cons, and prior works on PGO make different choices depending on their needs. In our case, we propose to instrument at the *binary level* for the following two reasons. First, by instrumenting at the binary level, our mechanism can be applied to any application or implementation, as it does not require access to the source code nor restrict developers to any specific programming languages. Second, operating at

[1] A similar and supported event is stalled cycles *while there are* L2/L3 cache miss demand loads. However, this event does not indicate causal relationship between cache misses and stalls, and is not precise meaning that the exact instructions (loads in our case) that caused the event are unavailable.

the binary level allows us to surgically instrument at the correct locations. Specifically, since sample-based profiling collects data at the binary level, it is known that the closer a level is to the binary representation, the higher the accuracy with which the profiled data can be mapped back to that representation [10, 11, 50]. To see this, consider a function that is inlined at multiple locations. If the profiled data indicates that instrumentation is needed at one of the locations but not others, we can easily do that at the binary level, but will have difficulty retrofitting the data back to higher-level representations and correctly instrumenting at that level if function inlining has not been performed yet.

While instrumenting at the binary level brings benefits in terms of applicability and accuracy, it does suffer from some limitations. One of them is relinquishing potential optimization opportunities along the compilation pipeline. Fortunately, as we will discuss next, operating at the binary level still permits optimizations that can significantly improve the performance of our mechanism. Another general concern is the inability to perform operations that require high-level semantic information. Fortunately, the logic of instrumenting yields to hide L2/L3 cache misses is independent from the application logic or program structure.

Yield instrumentation: after deciding the sampled hardware events and the instrumentation level, we next discuss design problems that are directly related to instrumenting yields to hide L2/L3 cache misses. We will not go into details of the procedures in the instrumentation pipeline, such as disassembly and control flow graph (CFG) construction, for which our mechanism should be similar to existing binary optimizers [7, 50, 51]. Instead, we elaborate on three aspects specific to our use case: the *conditions* under which a yield will be inserted at a location, the *operations* to instrument at these locations, and *optimizations* to reduce overhead.

In terms of the conditions to insert yields, there is a trade-off: aggressive instrumentation minimizes CPU stalls due to uninstrumented cache misses, at the risk of incurring unnecessary overhead if a load turns out to be a cache hit. To make better decisions in the face of this trade-off, we propose to quantitatively model the gain and the cost of instrumenting at a specific load instruction. This requires some statistics that are either estimated from the collected profiles (*e.g.,* the likelihood of cache misses for a load instruction) or extracted from the machine characteristics (*e.g.,* the average latency of an L2/L3 cache miss). Based on the statistics and modelling, one could then decide whether to place yields based on different policies. A simple policy, for example, is to instrument yields if the likelihood of cache misses is above a threshold.

Once we decide to yield at a specific load instruction, the following operations are instrumented: (i) prefetching the requested cache line before yielding, (ii) saving registers to memory and setting the stack pointer and the program counter to the ones of the next coroutine and (iii) restoring registers from the memory (since the coroutine is resumed at this point). These instrumentations ensure that the coroutine can *correctly* yield to a different one to hide the cache misses. Various optimizations could then be applied to reduce the overhead due to instrumentations. One potential optimization is to identify registers whose values will be used later via a register liveness analysis [45, 52] and only preserve the values of these registers. This directly translates to less switching overhead. Another interesting optimization is *yield coalescing*, which is applicable when instrumenting multiple independent and adjacent loads. Specifically, instead of inserting a yield for every load, we could issue prefetches all together and instrument only a single yield to amortize the switching overhead. Independence of adjacent loads can be determined via dependence analysis [4, 43].

3.3 Asymmetric concurrency

Profile-guided yield instrumentation, as we described above, hopefully allows us to hide L2/L3 cache misses in a way that is transparent to developers and applicable to a wide range of applications. However, it does not support fine-grained control over application latency. To see this, consider a case where we need to ensure low latency of a high-priority coroutine, while improving CPU efficiency by interleaving with executions of other coroutines. To support this use case, what we need is for other coroutines to yield back to the high-priority coroutine *as soon as* they have run for long enough to hide the latency of L2/L3 cache misses. However, the instrumentation mechanism described so far, which we call *primary* instrumentation, places yields only at locations that likely have cache misses. As a consequence, adjacent yields can be arbitrarily far apart depending on the application, preventing a coroutine from timely relinquishing the CPU.

Fundamentally speaking, our proposal has to reconcile two seemingly conflicting needs: sparse instrumentation (*i.e.,* only inserting yields to hide cache misses) for improving CPU efficiency with minimal overhead, and dense instrumentation for managing the latency impact on yielded coroutines. As a solution, we propose to support *asymmetric* concurrency, which consists of two components. First, after primary instrumentation, we add a *scavenger* instrumentation phase, where we strategically place additional yields to ensure appropriate distance between adjacent yields. These yields are conditional, hence can be turned on and off to alter the mode of a coroutine at run time. Second, at run time, we leverage coroutines in scavenger mode to hide cache misses, while incurring minimal latency overhead to coroutines in primary mode. Considering the previously discussed case, we can now achieve both high CPU efficiency and low latency of the high-priority coroutine by running the high-priority

coroutine in the primary mode and other coroutines in the scavenger mode. Next, we elaborate on the challenges associated with these two components and discuss our proposals.

Scavenger instrumentation: at this phase, the user provides a target inter-yield interval that is bounded but sufficient to hide L2/L3 cache misses (*e.g.,* 100 ns), and our goal is to ensure that adjacent yields are separated approximately this far. Achieving this goal with only static analysis is challenging: the latency of a basic block is hard to predict [19, 54] and there can be multiple paths of vastly different lengths between two basic blocks [18, 41]. Inspired by efforts in trace scheduling [12, 20, 39], a technique that uses profiling information for static instruction scheduling, we propose to leverage profiling for scavenger instrumentation as well. Specifically, profiling mechanisms like Intel's LBR can extract information like the latency of a basic block and the common paths in the program [34, 35]. With profiled data, we could first insert yields to ensure timely yielding in the common case, then augment it with additional yields to bound the worst-case inter-yield interval based on static analysis.

After the scavenger phase, we now have the final instrumented binary, which contains both primary yields for hiding cache misses and (conditional) scavenger yields for timely yielding. Scavenger yields are carefully placed to ensure appropriate inter-yield distances, whereas primary yields may be too close to or far from each other as their locations are determined by the application memory access patterns.

Dual-mode execution: at run time, we propose dual-mode execution: (i) a primary coroutine yields to scavenger coroutines in the face of a potential cache miss, and (ii) scavenger coroutines will yield back to the primary once they have run for long enough to hide the cache miss. For (ii), our mechanism should scale up the number of scavenger coroutines *on demand*. Specifically, in the normal case, a single scavenger coroutine is sufficient – the coroutine will run for some time until it encounters a yield instrumented at the scavenger phase, at which point the coroutine can directly yield back to the primary coroutine. In other cases, multiple scavenger coroutines may need to be invoked before returning to the primary. This is because a scavenger coroutine may encounter a yield that was instrumented at the primary phase for hiding cache misses too early, in which case it will instead yield to another scavenger to consume more cycles. For example, for a coroutine that performs pointer chasing, when operating in the scavenger mode, it has to rely on other scavenger coroutines in order to fully utilize the CPU.

To summarize, we believe that some form of asymmetric concurrency is critical for ensuring low latency with high CPU efficiency. With our proposed design, we hope to shed light on the design space that future work can explore, which likely involves co-design of offline profiling, profile-guided instrumentation and runtime control.

4 DISCUSSION

Now that we have proposed a way of hiding short events in software, we discuss two important questions: (i) if we could make changes to the hardware, what would the final solution look like? and (ii) how could our proposal coexist with software mechanisms designed for other purposes?

4.1 Hardware support

To make our proposal feasible, we have restricted ourselves to techniques supported by today's hardware. An interesting question is then: what if we lift this restriction and envision a hardware-software co-design? To approach this question, we proceed with a software-centric view and look for the *minimal* hardware support that will significantly benefit our proposal. For this, we re-examine the two aspects that software mechanisms fall short in, *i.e.,* large switching overhead and limited event visibility, and discuss to what extent additional hardware support will be helpful to our proposal.

For switching overhead, we conjecture that it is *not* the most critical issue, given the possible software optimizations that could further reduce the overhead of coroutine switching [16, 46]. Specifically, while switching software contexts requires storing/restoring register states to/from memory, it supports high degrees of concurrency and fine-grained control, both of which are hard to obtain in hardware without a significant hardware redesign. Moreover, since our proposal targets events that last for 10s to 100s of ns, the sub-10 ns overhead of coroutine switching is acceptable.

In contrast, we believe that event visibility is the aspect that should receive more attention, where significant improvement may be achievable with modest hardware changes. Solely relying on profile-guided instrumentation for detecting and hiding events is sub-optimal due to its *static* nature – whether a coroutine will yield at a location or not is determined offline. Therefore, hardware support to expose events, *e.g.,* indicating whether a cache line is in L1/L2 cache, could be highly useful here, as it allows yields to be *conditional* on whether targeted events actually happen. While condition checking adds some overhead, profile-guided instrumentation can mitigate this issue by placing conditional yields only at locations that often but not always incur target events.

4.2 Software integration

Runtime scheduling: an interesting question is how to integrate our proposed mechanism with existing coroutine schedulers [21, 72] whose logic is agnostic to short events. One approach is to run our mechanism on the side of the scheduler and have the scheduler perform only a minimal set of additional tasks to support event hiding. For example, the scheduler could expose the set of coroutines in its ready queue, so that our mechanism knows that they can be switched

to when hiding events. A different approach is to have the scheduler explicitly consider these short events when scheduling tasks. This is conceptually similar to how I/O events receive special treatment in OS process scheduling. This approach could be appealing when performing fine-grained scheduling of very short (*e.g.,* μs-scale) tasks [15, 30, 49].

Coroutine isolation: there are two categories of isolation mechanisms suitable for coroutines: software-based fault isolation (SFI) and language-based isolation. SFI establishes a logical protection domain by inserting dynamic checks before memory and control-transfer instructions [58, 65, 69]. Language-based isolation relies on safe high-level languages for isolation through a combination of static and dynamic checks [37, 40, 57]. Language-based isolations can have lower runtime overhead by adopting restricted memory models and performing most of the checks at compile time [8, 29, 42]. However, adopting language-based isolations requires more engineering efforts, due to the need of developing based on restricted memory models and rewriting legacy code. Since our proposal is applicable to different programming languages, it can co-exist with either isolation mechanism. An interesting question is whether a co-design of SFI and our proposal can help reduce the runtime overhead of SFI.

5 CONCLUSION

With light-weight coroutines and sample-based profiling, hiding short events that last only 10s to 100s of ns in software is becoming feasible. By walking through a design proposal, we shed light on the challenges that arise from leveraging these two techniques, and hopefully offer some promising directions towards addressing these challenges.

REFERENCES

[1] Soramichi Akiyama and Takahiro Hirofuchi. 2017. Quantitative evaluation of intel pebs overhead for online system-noise analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*. 1–8.

[2] Haitham Akkary and Michael A Driscoll. 1998. A dynamic multithreading processor. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 226–236.

[3] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Memory hierarchy for web search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 643–656.

[4] Utpal Banerjee. 1997. *Dependence analysis*. Vol. 3. Springer Science & Business Media.

[5] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. 2018. Rock you like a hurricane: Taming skew in large scale analytics. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[6] Boost. 2022. Performance of Boost context switch. https://www.boost.org/doc/libs/1_79_0/libs/context/doc/html/context/performance.html.

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 265–275.

[8] Anton Burtsev, Dan Appel, David Detweiler, Tianjiao Huang, Zhaofeng Li, Vikram Narayanan, and Gerd Zellweger. 2021. Isolation in Rust: What is Missing?. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*. 76–83.

[9] Pohua P Chang, Scott A Mahlke, William Y Chen, and Wen-Mei W Hwu. 1992. Profile-guided automatic inline expansion for C programs. *Software: Practice and Experience* 22, 5 (1992), 349–369.

[10] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. 12–23.

[11] Dehao Chen, Neil Vachharajani, Robert Hundt, Xinliang Li, Stephane Eranian, Wenguang Chen, and Weimin Zheng. 2011. Taming hardware event samples for precise and versatile feedback directed optimizations. *IEEE Trans. Comput.* 62, 2 (2011), 376–389.

[12] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. 1988. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on computers* 37, 8 (1988), 967–979.

[13] Charlie Curtsinger and Emery D Berger. 2015. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 184–197.

[14] Francis M David, Jeffrey C Carlyle, and Roy H Campbell. 2007. Context switch overheads for Linux on ARM platforms. In *Proceedings of the 2007 workshop on Experimental computer science*. 3–es.

[15] HM Demoulin and J Fried. 2021. When Idling is Ideal: Optimizing Tail-Latency for Highly-Dispersed Datacenter Workloads with Persephone. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[16] Stephen Dolan, Servesh Muralidharan, and David Gregg. 2013. Compiler support for lightweight context switching. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–25.

[17] Roman Elizarov, Mikhail Belyaev, Marat Akhin, and Ilmir Usmanov. 2021. Kotlin coroutines: design and implementation. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 68–84.

[18] Mohammad Hossein Fazel Zarandi, Ali Akbar Sadat Asl, Shahabeddin Sotudian, and Oscar Castillo. 2020. A state of the art review of intelligent scheduling. *Artificial Intelligence Review* 53 (2020), 501–593.

[19] Christian Ferdinand and Reinhold Heckmann. 2004. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society: IFIP 18th World Computer Congress Topical Sessions 22–27 August 2004 Toulouse, France*. Springer, 377–383.

[20] Joseph A. Fisher. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE transactions on computers* 30, 07 (1981), 478–490.

[21] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. 2019. Neptune: Scheduling suspendable tasks for unified stream/batch applications. In *Proceedings of the ACM symposium on cloud computing*. 233–245.

[22] Google. 2020. Propeller: Profile Guided Optimizing Large Scale LLVM-based Relinker. https://github.com/google/llvm-propeller.

[23] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: coroutine-oriented main-memory database engine. *Proceedings of the VLDB Endowment* 14, 3 (2020), 431–444.

[24] Joel Hruska. 2012. Maximized performance: Comparing the effects of Hyper-Threading, software updates. https://www.extremetech.com/computing/133121-maximized-performance-comparing-the-effects-of-hyper-threading-software-updates.

[25] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2021. A case against (most) context switches. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 17–25.

[26] Intel. 2022. Intel Accelerator Engines. https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/overview.html.

[27] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. 2022. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 747–764.

[28] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting coroutines to attack the" killer nanoseconds". *Proceedings of the VLDB Endowment* 11, 11 (2018), 1702–1714.

[29] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

[30] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 345–360.

[31] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[32] Patrick Kennedy. 2022. Intel Xeon Sapphire Rapids Shows Built-in Accelerators at Innovation 2022. https://www.servethehome.com/intel-xeon-sapphire-rapids-shows-built-in-accelerators-at-innovation-2022/.

[33] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-spy: Context-driven conditional instruction prefetching with coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 146–159.

[34] Andi Kleen. 2016. Advanced usage of last branch records. https://lwn.net/Articles/680996/.

[35] Andi Kleen. 2016. An introduction to last branch records. https://lwn.net/Articles/680985/.

[36] Oliver Kowalke and Nat Goodspeed. 2018. fiber_handle-fibers without scheduler. (2018).

[37] Dexter Kozen. 1999. Language-Based Security: Invited Lecture. In *Mathematical Foundations of Computer Science 1999: 24th International Symposium, MFCS'99 Szklarska Poręba, Poland, September 6–10, 1999 Proceedings 24*. Springer, 284–298.

[38] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*. 2–es.

[39] P Geoffrey Lowney, Stefan M Freudenberger, Thomas J Karzes, WD Lichtenstein, Robert P Nix, John S O'donnell, and John C Ruttenberg. 1993. The multiflow trace scheduling compiler. *The journal of Supercomputing* 7 (1993), 51–142.

[40] Sergio Maffeis and Ankur Taly. 2009. Language-based isolation of untrusted Javascript. In *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE, 77–91.

[41] Abid M Malik, Jim McInnes, and Peter Van Beek. 2008. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. *International journal on artificial intelligence tools* 17, 01 (2008), 37–54.

[42] Nicholas D Matsakis and Felix S Klock. 2014. The rust language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.

[43] Dror E Maydan, John L Hennessy, and Monica S Lam. 1991. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 1–14.

[44] Ana Lúcia De Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31, 2 (2009), 1–31.

[45] Robert Muth. 1998. Register liveness analysis of executable code. *Manuscript, Dept. of Computer Science, The University of Arizona, Dec* (1998).

[46] Gor Nishanov. 2018. C++ Extensions for Coroutines. (2018).

[47] Aleix Roca Nonell, Balazs Gerofi, Leonardo Bautista-Gomez, Dominique Martinet, Vicenç Beltran Querol, and Yutaka Ishikawa. 2018. On the applicability of PEBS based online memory access tracking for heterogeneous memory management at scale. In *Proceedings of the Workshop on Memory Centric High Performance Computing*. 50–57.

[48] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing function placement for large-scale data-center applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 233–244.

[49] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads.. In *NSDI*, Vol. 19. 361–378.

[50] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[51] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. 2021. Lightning bolt: powerful, fast, and scalable binary optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 119–130.

[52] Mark Probst, Andreas Krall, and Bernhard Scholz. 2002. Register liveness analysis for optimizing dynamic binary translation. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE, 35–44.

[53] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with coroutines: a practical approach for robust index joins. *Proceedings of the VLDB Endowment* 11, CONF (2017), 230–242.

[54] Peter Puschner and Alan Burns. 2000. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems* 18, 2-3 (2000), 115–128.

[55] Steven E Raasch and Steven K Reinhardt. 1999. Applications of thread prioritization in SMT processors. In *Proc. of the Workshop on Multithreaded Execution And Compilation*. Citeseer.

[56] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro* 30, 4 (2010), 65–79.

[57] Fred B Schneider, Greg Morrisett, and Robert Harper. 2001. A language-based approach to security. *Informatics: 10 Years Back, 10 Years Ahead* (2001), 86–101.

[58] David Sehr, Robert Muth, Cliff L Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. 2010. Adapting software fault isolation to contemporary CPU architectures. (2010).

[59] John Paul Shen and Mikko H Lipasti. 2013. *Modern processor design: fundamentals of superscalar processors*. Waveland Press.

[60] Abraham Silberschatz, James L Peterson, and Peter B Galvin. 1991. *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc.

[61] Michael D Smith. 2000. Overcoming the challenges to feedback-directed optimization (keynote talk). In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*. 1–11.

[62] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. 2019. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*. 513–526.

[63] Lukas Stadler, Thomas Würthinger, and Christian Wimmer. 2010. Efficient coroutines for the Java platform. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*. 20–28.

[64] William Stallings. 1998. *Operating systems internals and design principles*. Prentice-Hall, Inc.

[65] Gang Tan et al. 2017. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends® in Privacy and Security* 1, 3 (2017), 137–198.

[66] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009: Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*. Springer, 157–173.

[67] Dean M Tullsen and Jeffery A Brown. 2001. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. IEEE, 318–327.

[68] Antonio Valles, Matt Gillespie, and Garrett Drysdale. 2009. Performance insights to Intel® hyper-threading technology. *Source:< https://software. intel. com/enus/articles/performance-insights-to-intel-hyper-threadingtechnology* (2009).

[69] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*. 203–216.

[70] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[71] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J Freedman. 2018. Riffle: Optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[72] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. 2021. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 195–211.