Harvesting Memory-bound CPU Stall Cycles in Software with MSH

Zhihong Luo, Sam Son, and Sylvia Ratnasamy, *UC Berkeley;*
Scott Shenker, *UC Berkeley & ICSI*

https://www.usenix.org/conference/osdi24/presentation/luo

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

Harvesting Memory-bound CPU Stall Cycles in Software with MSH

Zhihong Luo
UC Berkeley

Sam Son
UC Berkeley

Sylvia Ratnasamy
UC Berkeley

Scott Shenker
UC Berkeley & ICSI

Abstract

Memory-bound stalls account for a significant portion of CPU cycles in datacenter workloads, which makes harvesting them to execute other useful work highly valuable. However, mainstream implementations of the hardware harvesting mechanism, simultaneous multithreading (SMT), are unsatisfactory. They incur high latency overhead and do not offer fine-grained configurability of the trade-off between latency and harvesting throughput, which hinders wide adoption for latency-critical services; and they support only limited degrees of concurrency, which prevents full harvesting of memory stall cycles.

We present MSH, the first system that transparently and efficiently harvests memory-bound stall cycles in software. MSH makes full use of stall cycles with concurrency scaling, while incurring minimal and configurable latency overhead. MSH achieves these with a novel co-design of profiling, program analysis, binary instrumentation and runtime scheduling. Our evaluation shows that MSH achieves up to 72% harvesting throughput of SMT for latency SLOs under which SMT has to be disabled, and that strategically combining MSH with SMT leads to higher throughput than SMT due to MSH's capability to fully harvest memory-bound stall cycles.

1 Introduction

CPU cores are valuable resources in datacenter infrastructure. To meet the ever-growing computation demand, there have been extensive software efforts in *harvesting* idle CPU cycles and keeping cores fully utilized [7, 43, 52, 88, 94]. While differing in mechanisms, these works share a similar harvesting scheme: "scavenger" instances (*e.g.,* spot VMs, batch jobs) temporarily run on cores that primary instances are not actively using. Their common performance goal is to have scavenger instances fully utilize the idle cycles without slowing down primary instances. Minimizing negative performance impacts is particularly important for latency-critical services as their increased latencies directly affect user experience.

Unlike prior efforts that harvest cores that are idle for a relatively long period of time, *e.g.,* allocated but unused cores of the primary VM, we focus on memory-bound CPU *stall* cycles. These are cycles that cores transiently stall while waiting for memory accesses to finish. Although each lasts only a few hundred nanoseconds, memory-bound stalls can happen frequently and account for a significant portion of CPU cycles [10, 23, 45, 78]: more than 60% for some widely-used modern applications, which implies substantial benefits harvesting these stall cycles. However, the current hardware harvesting mechanism, simultaneous multithreading (SMT), is unsatisfactory. First, SMT is known to likely lead to significantly *increased latencies*, as it focuses solely on multiplexing instruction streams to best utilize core resources [37, 74, 83, 84]. Moreover, SMT does not allow fine-grained *control* over the tradeoff between primary latency and scavenger throughput, which is needed to maximize CPU utilization under a latency SLO. As a result, for latency-critical services, a common compromise is thus to avoid using SMT for better performance, at the cost of wasting stall cycles [18, 19, 55, 69]. Lastly, there are cases where SMT can not *fully harvest* memory-bound stall cycles: modern CPUs often support only limited degrees of concurrency (*e.g.,* 2 threads per physical core in the case of Intel's Hyperthreading), which are insufficient when concurrent threads frequently incur cache misses [44, 45, 72].

In view of the significance of memory-bound CPU stalls and the drawbacks of SMT as the hardware harvesting mechanism, our goal is to design a system that harvests these stall cycles in software. This system should meet several requirements. First, it should be *transparent* to applications and require no additional rewriting efforts from developers. As a result, it will resemble SMT in terms of being conveniently applicable to any code, including legacy code. The other requirements then demand improving upon the drawbacks of SMT. Specifically, it should *efficiently* and *fully* harvest the memory-bound stall cycles, and it should do so while introducing *minimal* latency overhead to the primary instance.

A recent proposal [57] discusses the possibility of transparently hiding the latency of cache misses in software with the combination of light-weight coroutines [25, 28, 64, 79] and sample-based profiling [17, 47, 82]. The former allows interleaving of primary and scavenger coroutines with a switching overhead much smaller than traditional threads of executions like processes and kernel threads; whereas the latter makes it possible to do it transparently, as we could identify likely

locations of cache misses via profiling. This is a key realization that our work builds upon. However, there remains to be a set of challenges toward building a software system that harvests memory-bound CPU stall cycles and meets the aforementioned requirements. First, to improve harvesting efficiency, we have to minimize the amount of register savings and restorations for each yield, while ensuring the correctness of program executions. Second, to introduce minimal latency overhead, scavengers need to yield back the core soon after they have consumed enough stall cycles, which is challenging given that programs have complex and dynamic control flows. Third, to fully harvest stall cycles, we need to detect when a higher degree of concurrency is needed and properly interleave the executions of multiple scavengers. Lastly, it is challenging to transparently interleave scavenger executions with a primary binary that has an internal threading structure.

To overcome these challenges, we present Memory Stall Software Harvester (MSH), the first system that transparently and efficiently harvests memory-bound CPU stall cycles in software. MSH makes full use of stall cycles while incurring only minimal latency overhead. MSH fulfills all the requirements with a novel co-design of *profiling, program analysis, binary instrumentation and runtime scheduling*. To use MSH, users simply provide unmodified primary binaries and a pool of scavenger threads, and MSH takes care of running scavenger threads with stall cycles of the primary binaries.

Internally, MSH operates in two logical steps. First, after profiling the primary and scavenger code, MSH statically instruments them at the binary level, by leveraging information obtained via profiling and program analysis. Specifically, for both primaries and scavengers, MSH inserts a prefetch instruction followed by yielding to either a primary or a scavenger coroutine (configured in runtime, discussed below), before selected load instructions that frequently incur cache misses according to profiled data. In addition, MSH places additional yields in scavengers to ensure that they timely relinquish their core. The first two of the aforementioned challenges are resolved in this step. For the primary binaries, MSH carries out various optimizations to reduce the amount of register savings and restorations for each yield by analyzing register usage and program structures. For the scavenger, MSH conducts a forward data flow analysis that also takes in profiled data to decide additional yield points, so that the distance between consecutive yields is bounded to a configurable threshold.

In the second logical step, when executing a primary binary, MSH sets up and dynamically assigns scavengers to active primary threads. The last two challenges regarding scavenger scheduling and concurrency scaling are tackled in this step. MSH intercepts function calls that change the status of primary threads and efficiently adjusts the scavenger assignment. This allows MSH to transparently schedule scavengers on top of the primary's threading structure. To support on-demand concurrency scaling, MSH performs two operations: assigning multiple scavengers to a primary thread and configuring scavengers so that they yield to the right target. For the former, MSH decides the number of scavengers assigned to a primary thread by estimating and bounding the likelihood of not full harvesting stall cycles. For the latter, MSH instruments yields in scavengers that are close to each other to yield to the next scavenger instead of the primary thread. MSH's runtime then takes care of correctly setting up the targets of these yields.

We implement MSH's offline parts on top of Bolt [67], an open-source binary optimizer built on the LLVM framework, and MSH's runtime as a user-level library[1]. We evaluate MSH with unmodified syntactic and real applications and show that MSH is general enough to harvest stall cycles from all of them. Compared with SMT, MSH offers superior harvesting performance in three aspects: first, MSH incurs minimal latency overhead and achieves up to 72% harvesting throughput of SMT, for latency SLOs under which SMT has to be disabled. Second, as a configurable software solution, MSH enables users to have fine-grained control over the tradeoff between primary latency and scavenger throughput. Third, MSH can fully harvest memory-bound stall cycles via concurrency scaling, achieving up to 2x higher throughput than SMT when scavengers frequently stall. Moreover, we show that by strategically combining MSH with SMT, one could achieve higher throughput than SMT due to MSH's ability to fully harvest memory-bound stall cycles. Lastly, we extensively evaluate MSH's main components and show that they play a vital role in achieving MSH's superior performance.

In summary, the contributions of this paper are: (i) a transparent and efficient approach to harvest memory-bound CPU stall cycles in software; (ii) the detailed design and implementation of a system (MSH) based on this approach, which involves a co-design of profiling, program analysis, binary instrumentation, and runtime scheduling; (iii) an evaluation with real applications showing that compared with SMT, MSH can deliver high scavenger throughput under stringent primary latency SLOs and fully harvest memory-bound stall cycles. In addition to presenting the design, implementation, and evaluation of MSH, we extensively discuss other related aspects in §8. These include isolation mechanisms that can be integrated with MSH to ensure memory safety, hardware support that can enhance MSH's performance and so on. Our hope here is to motivate greater efforts in delivering these critical aspects.

2 Background and Motivation

Memory-bound stalls: Memory-bound stalls, where cores stall and wait for memory accesses to finish, were reported to be a dominant source of CPU overhead for datacenter workloads. To see this, we perform a top-down analysis [90] on two latency-critical applications. This analysis classifies CPU pipeline slots into four categories: retiring, frontend-bound, bad speculation and backend-bound. The last three correspond to different overhead, and backend-bound stalls can be further divided into core-bound or memory-bound stalls. Our

[1] MSH is publicly available at https://github.com/sosson97/msh.

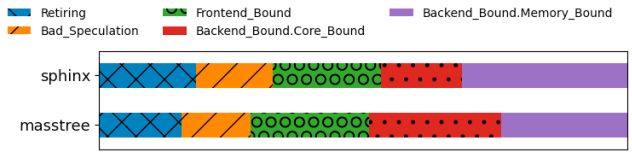

Figure 1: Top-down analysis of Sphinx and Masstree; memory stalls account for 25% and 31% of cycles respectively.

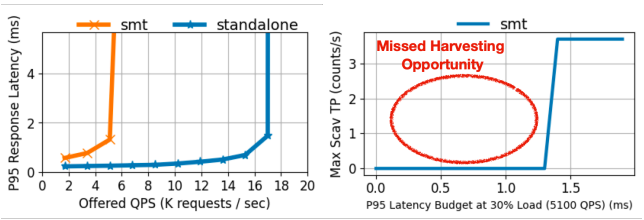

Figure 2: (a) P95 latency of Masstree when running by itself vs. co-locating with a Scan scavenger; (b) SMT is unable to harvest stall cycles under low latency SLOs.

analysis confirms the dominance of memory-bound stalls, as they account for 25% and 31% of total cycles for Masstree and Sphinx respectively (Figure 1). While there have been extensive efforts on *reducing* memory stalls, it is generally infeasible to eliminate them (§7). In this work, we focus on the alternative approach of *harvesting* these stall cycles to execute useful work, where simultaneous multithreading (SMT) is the representative hardware mechanism.

Drawbacks of SMT: However, SMT, as a harvesting mechanism, suffers from three main drawbacks[2] that we next show:

- **Latency overhead:** SMT focuses solely on multiplexing instruction streams to best utilize CPU cores. As a result, it significantly increases the primary latency if the scavenger creates notable *contention* on core resources. This is problematic as it is common to co-locate latency-critical tasks that have stringent latency SLOs, with best-effort tasks that are resource-hungry. To see the latency overhead of SMT, we measure the latency of Masstree while running a synthetic Scan scavenger on its sibling cores. Scan is a representative of contending scavengers: by iterating a 4MB array and computing the sum, it consumes L1/L2 caches and core resources like ALU. As shown in Figure 2-(a), compared with running on dedicated cores, harvesting stall cycles via SMT leads to 92x higher latency of Masstree at 40% load. Such a behavior is widely observed in prior studies, thus it is common to avoid using SMT for latency-critical services at the cost of wasting cycles.
- **Lack of Configurability:** Related to the large latency overhead, another drawback of SMT that hinders its uses for latency-critical services is the lack of fine-grained configurability. Given a latency SLO, what is needed to maximize CPU utilization is a knob that controls the *extent* of resource sharing and hence the tradeoff between primary latency and scavenger throughput. However, with SMT, one can only decide whether to turn it on or off, which is too coarse-grained to be useful. To see this, we compute the maximum achievable Scan scavenger throughput under different Masstree latency SLOs for the experiment above. Here we set the SLO to be the latency under 30% load. An ideal mechanism should gracefully harvest cycles proportional to the latency budget given. In contrast, as shown in Figure 2-(b), with SMT, one has to turn off SMT and effectively achieve zero scavenger throughput when the latency SLO is lower than SMT latency. Even after SMT is on, it can not harvest more cycles when looser latency SLOs are given. Neither of these two ends is desirable.
- **Incomplete harvesting:** Lastly, SMT often can not fully harvest memory-bound stall cycles, especially when concurrent threads frequently incur cache misses. This is because the mainstream 2-wide SMT does not have sufficient degrees of concurrency to harvest the bulk of memory stalls. Note that while increasing the width of SMT helps with this issue, it requires dedicating more hardware resources and worsens the already problematic latency overhead issue.

We aim to design a software system that harvests memory-bound stall cycles, is as generally applicable and convenient to use as SMT, and improves upon the drawbacks of SMT.

Software opportunities: There are two capabilities a software mechanism needs for harvesting memory stall cycles: (i) transparently detecting the presence of memory stalls and (ii) efficiently interleaving the executions of primaries and scavengers. The former is challenging, because cache misses are not exposed to software, and manually identifying stalls is burdensome and error-prone. The latter requires much smaller switching overhead than traditional threads of execution like kernel threads. A recent proposal [57] discusses the opportunity of enabling these two capabilities via a combination of light-weight coroutines and sample-based profiling:

- **Sample-based profiling:** By using hardware performance counters in modern CPUs, such as Intel's PEBS [3] and LBR [47], one could profile binaries with no special build and negligible run time overhead. Thanks to these merits, sample-based profiling has been widely used in production for profile-guided optimizations (PGO) [17, 30, 66–68].
- **Light-weight coroutines:** Context switches of coroutines are orders of magnitudes cheaper than traditional threads of execution. This is because as a user-space mechanism within a single process, coroutine context switch requires no system calls nor changes to virtual memory mappings.

Building on these two techniques, MSH is the first software system that transparently and efficiently harvests memory-bound stall cycles. Next, we present an overview of MSH.

[2]These drawbacks apply to SMT of most modern processors (*e.g.,* Intel's and AMD's), with IBM Power as an exception, discussed further in §7.

3 MSH Overview

In this section, we discuss MSH's overarching goals, deployment scenarios, high-level approach as well as overall flow.

Goal: Our goal is to transparently harvest memory-bound stall cycles from any application, while overcoming SMT's performance limitations. We thus distill four requirements that MSH as a software harvesting system should meet:

- **Transparent:** The system should be transparent to applications. It thus requires no rewriting effort from developers and is applicable to any code, including legacy code.
- **Efficient:** The system should efficiently utilize the stall cycles for scavenger executions, which demands extremely low overhead from the harvesting machinery.
- **Latency-aware:** The system should incur minimal latency overhead and allow fine-grained control over the trade-off between primary latency and scavenger throughput.
- **Full-harvesting:** The system should fully harvest stall cycles by interleaving sufficient scavenger executions, especially when scavengers also incur frequent cache misses.

Deployment scenario: System operators can use MSH to harvest stall cycles of any application written in compiled languages. MSH handles scavenger's offline instrumentations and runtime executions. MSH assumes that it is safe to run these scavengers alongside the primaries [92], *e.g.,* they are crash-free and access memory safely. Ensuring safety properties with techniques like verification and information flow control [14, 35, 62] is left to future work. MSH can be seamlessly integrated with existing profiling systems deployed for PGO [17, 30, 68]. MSH is well suited for when latency-critical and best-efforts tasks are co-located in the same machine, a common arrangement in production [27, 55, 61, 93]. In this case, latency-critical tasks serve as the primary, whose stall cycles are harvested for the best-effort tasks.

Approach: MSH uses a novel co-design of binary instrumentation, profiling, program analysis, and runtime scheduling, each of which plays a role in meeting the requirements above:

- **Binary instrumentation:** MSH instruments primaries and scavengers so that they are amenable to stall cycle harvesting. Operating at the binary level provides visibility of low-level information, *e.g.,* register usage and basic block control flows, which is needed by MSH's program analysis.
- **Profiling:** With sample-based profiling, MSH decides locations to harvest stall cycles without requiring efforts from developers. Profiling also allows MSH to use dynamic information, *e.g.,* basic block latency and branching probability, to achieve high accuracy in its program analysis.
- **Program analysis:** MSH leverages program analysis to achieve efficiency, full-harvesting and latency-awareness. For efficiency, MSH minimizes the amount of register savings and restorations for yields. For full-harvesting, MSH directs yields in scavengers that are close to each other to another scavenger. For latency-awareness, MSH bounds the latency between adjacent yields in scavengers.
- **Runtime scheduling:** MSH's runtime schedules scavenger executions on top of the primary's internal threading structure. It enables MSH to fully harvest stall cycles with available scavengers, by assigning multiple scavengers to a primary thread to scale up concurrency and migrating scavengers from blocked primary threads to active ones.

Overall Flow: MSH performs both offline and run-time operations (Figure 3). In the offline phase, MSH transforms the primary and scavenger binaries so that they are amenable to stall cycle harvesting. Specifically, MSH first profiles the binaries and obtains information needed by program analysis and later binary instrumentation: load instructions that incur cache misses, indicating where CPU stalls happen; basic block latencies and execution counts as well as branching probability, which are used by the primary and scavenger instrumentations. After profiling, MSH analyzes the binaries and extracts information that later guides the instrumentations. For each yield site, where a yield is inserted to harvest stall cycles of a delinquent load, MSH identifies a minimal amount of register savings and restorations that still ensures program correctness, by analyzing register usage and program structures. For scavengers, MSH conducts a data flow analysis to decide the locations of additional yields, so that the expected inter-yield latency is bounded. Most of the analysis is designed to be intra-procedural, the complexity of which thus scales only sublinearly with program sizes. Lastly, based on the analysis results, MSH instruments the binaries.

The instrumented primary binaries contain so-called "primary" yields to expose CPU stalls: each primary yield is inserted before a selected load instruction and prefetches the cache line before yielding to a scavenger. As for instrumented scavengers, they also contain primary yields before selected load instructions, with default yield targets being a primary thread. The special case is when primary yields are close to each other: the target of these "special" primary yields is set to another scavenger to scale up concurrency. Scavengers also contain so-called "scavenger" yields, which are placed to ensure that scavengers relinquish their cores in a timely manner. We present the design of primary and scavenger instrumentations in §4.1 and §4.2.

At runtime, MSH interleaves the executions of instrumented primaries and scavengers by dynamically assigning scavengers to active primary threads, which means that MSH does not require pre-determined or static pairings of primaries and scavengers. To do that, MSH tracks the status of primary threads by intercepting relevant function calls and adjusts scavenger assignment accordingly. When a new thread is created, MSH either steals the scavengers of a blocked thread or fetches scavengers from the scavenger pool. If a thread is blocked or ended, MSH marks its scavengers as stealable.

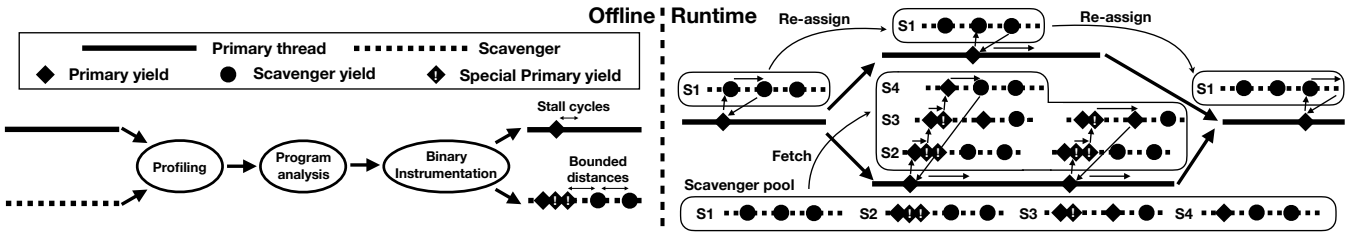

Figure 3: MSH system overview. Offline: MSH profiles and analyzes primaries and scavengers. It then instruments the primaries to yield control to scavengers at likely memory stall sites, with scavengers returning control to primaries within a bounded time. Runtime: MSH sets up a scavenger pool and dynamically assigns scavengers to each active primary thread.

When a thread later resumes, it will first attempt to reuse its previously assigned scavengers, before falling back to getting new scavengers like the thread creation case. Multiple scavengers could be assigned to a primary thread to scale up concurrency. MSH's runtime performs all these operations efficiently, and its design is later presented in §4.3.

4 Design

MSH consists of three components: primary instrumentation (§4.1), scavenger instrumentation (§4.2) and a runtime (§4.3).

4.1 Primary Instrumentation

Primary instrumentation allows MSH to prefetch and yield before load instructions that incur cache misses to expose stall cycles. This should be transparent – requiring no assistance from developers, and efficient – leaving most stall cycles for scavengers. MSH achieves transparency by selecting yield sites based on profiled data, and efficiency by minimizing register savings/restorations for each yield via program analysis.

Profile-guided yield instrumentation: MSH selects locations that both account for a significant portion of memory-bound stalls and have a high likelihood of L3 cache misses: the former indicates substantial stall cycles, and the latter allows less impact to the primary's latency. To support this, MSH obtains two pieces of information via profiling: load instructions with L2/L3 cache misses and execution counts of basic blocks.. MSH then adopts a two-step selection logic. First, MSH sorts load instructions whose cache miss rates are higher than a threshold by their frequencies. Second, MSH estimates the latency overhead for each load instruction by multiplying its frequency with its cache hit rate and the memory access latency. MSH then goes down the sorted list, includes a load instruction if the aggregate overhead falls below a provided bound, and skips otherwise. This selection logic maximizes harvesting opportunities by prioritizing frequent load instructions, while limiting the overall latency overhead. Both the cache miss threshold and overhead bound are configurable parameters that affect the tradeoff between primary latency and scavenger throughput (§6.4).

For each selected load instructions, MSH instruments a prefetch instruction for the same address, followed by a yield that consists of two parts: register savings/restorations and control passing. The former accounts for most of the yielding overhead, and as we will describe next, MSH minimizes it while ensuring correctness of program executions. For control passing, MSH instruments the primary to swap its instruction and stack pointer with the ones of an assigned scavenger that the primary reads from a per-thread data structure (§4.3). The instrumented code also reads a flag that indicates whether to bypass the yield and directly resumes. This allows the runtime to turn off stall cycle harvesting for instrumented primaries and avoid the latency overhead of scavenger executions.

Yield cost minimization: Minimizing the yield cost is important for two reasons. First, it improves harvesting efficiency: the less cycles spent on the yielding machinery, the more cycles available for executing scavengers while the primary stalls. Moreover, it reduces the latency impacts to the primary, especially when an instrumented load instruction results in a cache hit and only stalls for a short amount of time.

Register savings and restorations are the dominant cost for yields. MSH thus performs various optimizations to reduce them while ensuring correctness of the program executions. To avoid preserving every register, MSH first leverages register liveness analysis [71], a form of data-flow analysis that determines for each program point the set of "live" registers whose values will likely be used later. Given that register liveness is conservative, meaning that a register will be identified as live as long as there is any potential program path that may read its current value, by preserving live registers at the yield site, we are guaranteed to not violate the program correctness.

While saving only live registers reduces the yielding overhead, the cost saving is small for functions with non-trivial control flows, where most registers are considered live. To further reduce the cost, MSH builds on an observation: besides *what* registers to preserve, *where* these register savings and restorations take place also plays an important role in the yielding overhead. In particular, the naive approach of placing register savings and restorations at yield sites leads to *unnecessary* overhead. This is because there can be multiple yields between a definition of a register and its corresponding uses that repeatedly save and restore the register's value as the register is indeed live. To fix this, the key insight is to *align* register savings/restorations with register definitions/uses. Intuitively, if we were to save/restore the register at

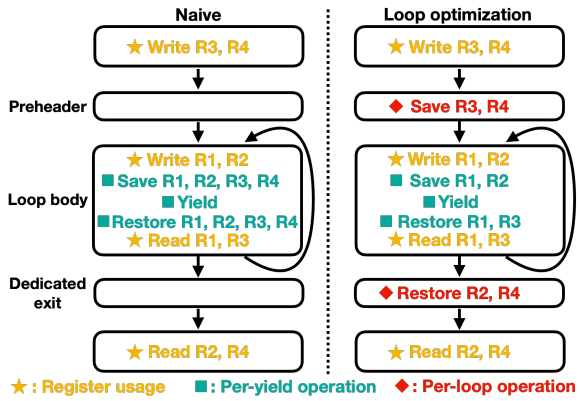

Figure 4: Loop optimization in primary instrumentation.

its definition/use sites, we can remove the redundancy due to having multiple yields in between the definition-use pairs, while still correctly preserving program semantics.

However, placing register savings/restorations at its definition/use sites for arbitrary program structures is highly complicated and potentially undesirable. Specifically, for correctness, one needs to identify all the definition sites, whose definitions are likely to reach the yielding point, as well as all the use sites that likely read these definitions. Instrumenting at all these scattered locations requires a substantial amount of work. Moreover, it is inevitable that some definition-use pairs have paths that do not go through the yield point. This means that there is register saving/restoring overhead even when the function does not yield, which could lead to overall increased overhead, if these cases happen frequently.

Instead of handling arbitrary program structures, MSH focuses on *loops*: it is often the case that a large portion of yields reside in loops, which make them valuable targets for optimizations. More importantly, the unique structure of loops allows MSH to perform *per-loop* register savings or restorations. As shown in Figure 4, most loops can be restructured to have a preheader and some dedicated exits: the former dominates the loop body whereas the latter post-dominate it. As a result, any paths traversing the loop will enter the preheader and leave one of the exits. MSH can thus simply place register savings and restorations at the preheader and exits, respectively, to ensure correctness for yields within the loop. Moreover, as long as more than one loop iteration goes through the yielding point, such a placement leads to strictly fewer register savings and restorations than the yield-site placement. In practice, this improvement is significant as the operation now happens once per loop instead of once per iteration. For registers that only have either uses or definitions within the loop body (R2 and R3 in Figure 4), MSH adopts a hybrid approach that places either saving or storing at the preheader/exit and the other at the yield site. To enable such loop optimizations, besides register liveness analysis, MSH performs reaching definition analysis to track the relevant definitions and uses for live registers, as well as loop simplification to transform feasible loops.

Besides when there are yields directly within a loop, MSH optimizes for another common case, where a function called within loops contains a single yield point. In particular, for a function that has unused callee-saved registers, we need to preserve values of these registers at the function boundary to abide by the calling convention. However, when such functions are called in loops, they incur redundant overhead due to per-iteration saving and restoration. To address this issue, MSH performs an optimization that we call "pseudo-inlining": MSH effectively inlines the target function by creating a copy of the function, for which the values of unused callee-saved registers are not preserved, and redirecting calls in loops towards this copy. MSH then leverages its loop optimization technique to save and restore the values of these unused callee-saved registers at the loop granularity as much as possible. MSH ensures that the original copy complies with the calling convention, so that other calls to the function take place correctly. Pseudo-inlining thus enables loop optimizations as if the function were inlined, while being easy to implement and creating minimal code expansion since the copy is shared.

In summary, MSH is strategic about what registers to preserve and where operations take place. It achieves the former by identifying live registers and the latter by exploiting per-loop operations. This reduced yield cost then leads to lower primary latency and higher harvesting efficiency (§6.4).

4.2 Scavenger Instrumentation

Scavenger instrumentation allows full stall cycle harvesting, while incurring minimal latency overheads. To minimize latency overhead, MSH places scavenger yields to bound inter-yield distances. To fully harvest stall cycles, primary yields that are too close to each other are directed to another scavenger. Next, we describe the mechanism in detail.

Primary yields: MSH instruments yields for stalling load instructions within scavengers in the same way as primary instrumentation: identifying yield sites via profiling and adopting optimizations to reduce yield costs. By default, these primary yields relinquish the core back to the primary. The special case is when some yields are too close to each other to fully harvest stall cycles (*e.g.,* yields within tight loops). These special primary yields will continue to the next scavenger. To support this, the per-thread data structure managed by runtime contains two targets (*i.e.,* primary thread and next scavenger) for each scavenger (§4.3). Normal and special primary yields are thus instructed to read different targets.

Scavenger yields: With only primary yields, it could take arbitrarily long for scavengers to yield back. MSH thus bounds inter-yield distances via a data-flow analysis that (i) calculates the average distances between a basic block and the current set of scavenger yields and (ii) inserts yields if some distance is over the bound. Note that the accuracy of bounding inter-yield distances affects the latency overhead, but not the correctness of the primary's execution. We next describe

the state, transfer function and join operation of the analysis:

- **State:** The state of our analysis is a list of yields and their average *uninstrumented* distances (in terms of time/cycles) to the current program point. If the scavenger were to yield here, these are the expected amount of time the scavenger has consumed before relinquishing the core since the previous yield points. Note that only yields with paths to the current program point that do not contain any other yield are included in the list. Input and output states of a basic block thus represent the uninstrumented distances before and after the basic block execution. MSH focuses on these states as they directly allow bounding inter-yield distances.
- **Transfer function:** This determines how the output state of a basic block is calculated based on its input state. If no new yields are added, the output state is simply the input incremented by the average latency of the basic block. This average latency can be computed with latency samples from profiling, or estimated as the product of the number of instructions and the scavenger's CPI. If any of the incremented distance is larger than the bound, MSH looks for a subset of its incoming edges to instrument yields. As described below, this will change the input (and consequently output) state of the basic block to contain new yield points and hopefully keep all the distances in the output within bound. If no such subset can be found, MSH inserts a yield at the end of the basic block and sets the output state to have only this yield point with zero distance.
- **Join operation:** This determines how the input state of a basic block is calculated based on the output states of its predecessors. For predecessors whose incoming edges are not instrumented, yields in their output states are all included in the basic block's input state, with distances being weighted averages of the corresponding distances in predecessors' output states. The weights are proportional to hotness of incoming edges, obtained via profiling. For instrumented incoming edges, the predecessor's output state will not propagate, instead the inserted yield is added to the basic block's input state with zero distance.

For the analysis, MSH ignores back edges (loops are handled later) and sorts basic blocks topologically, so that output states of predecessors are available before a basic block's turn. MSH sets the input state of the entry basic block to be a pseudo-yield named "function-start" with zero distance. MSH then iteratively computes all the states with the transfer function and join operation. Here, there are two aspects that require careful treatments – loops and function calls:

- **Loops:** For each loop, MSH computes the expected uninstrumented distance as a weighted average of the distances of all uninstrumented paths from the header basic block to the latch basic block, where weights correspond to path hotness. If the distance is zero (*i.e.,* all paths have yields), no loop instrumentation is needed. Otherwise, MSH instruments the back edge so that it yields every bound divided by distance iterations. To do this, MSH uses an induction register if available; otherwise MSH maintains a counter with unused registers or in per-thread data structures.
- **Function calls:** One aspect omitted so far is the treatment of function calls. For calls whose callee are unknown or external, MSH treats them as normal instructions. For uninstrumented external library calls that are known to be expensive, we adopt the standard practice of instrumenting right before and after the calls [13, 58]. Instead, for calls to local functions, MSH considers whether there are uninstrumented paths (*i.e.,* from entry to exits) in the callee – if yes, distances in the basic block's output state are incremented by the average uninstrumented latency of the callee; otherwise, since previous yields will be terminated in this call, MSH resets the output state to have only a pseudo-yield for the call with zero distance. The uninstrumented latency of a callee is computed with the distances for the function-start entry in the output states of its exit basic blocks. To use callee's analysis results, MSH builds a function call graph, ignores some calls to break loops, and analyzes functions in a topological order.

In summary, MSH can scale up concurrency to fully harvest stall cycles (§6.2) and manage latency impacts by enforcing inter-yield distance bounds via data-flow analysis (§6.4).

4.3 MSH Runtime

MSH intercepts function calls and assigns scavengers to active primary threads with minimal runtime overhead using tailored data structures. Next, we present the runtime design.
Function interception: MSH intercepts three types of functions: (i) functions starting a thread: *e.g.,* pthread_create, (ii) functions (likely) blocking a thread, *e.g.,* pthread_mutex_lock, and (iii) functions terminating a thread, *e.g.,* returning from the thread's start routine. Note that if there are unintercepted function calls that alter thread status, MSH's correctness is unaffected: *e.g.,* if a thread gets blocked silently (from the view of MSH), its scavengers will stay with the blocked thread, and harvestings will continue normally once the thread resumes.
Runtime operations: MSH performs different operations before/after intercepted calls to adjust scavenger assignment:

- **Scavenger initialization:** MSH initializes a new scavenger before assigning it to a primary thread, which includes loading the scavenger code, allocating its stack space and setting the return address for MSH to track when it finishes.
- **Scavenger assignment:** MSH assigns scavengers to a primary thread by configuring yield targets. The target for primary threads is a scavenger, and the target for scavengers is a primary thread by default, or another scavenger for special yields. MSH assigns more scavengers to a thread until the product of special yield ratios for scavengers is below a threshold or the scavenger number reaches a maximum.
- **Scavenger stealing:** When a primary thread needs scavengers, MSH first attempts to "steal" existing scavengers.

```
bool steal_scavengers(per_thread_ctx *t) {
    for (per_thread_ctx *it: thread_list) {
        if (CAS(it->stealable, true, false)) {
            it->stolen = migrate_scavengers(t, it);
            it->stealable = true;
            if(!need_more_scavengers(t))
                return true;
        }
    }
    return false;
}
void get_scavengers(per_thread_ctx *t) {
    if(!steal_scavengers(t)) {
        fetch_scavengers_from_pool(t);
    }
}
void enter_blockable_call(per_thread_ctx *t) {
    t->stealable = true;
}
void exit_blockable_call(per_thread_ctx *t) {
    while (!CAS(t->stealable, true, false)) {}
    if (t->stolen) {
        get_scavengers(t);
        update_yield_targets(t->yield_contexts);
        t->stolen = false;
    }
}
```

Listing 1: Pseudocode for key functions of MSH's runtime.

MSH ensures that each scavenger is assigned to at most one active thread at any time, by marking the scavengers of a thread as stealable before the thread gets blocked or terminated and only re-assigning stealable scavengers.

- **Scavenger fetching:** When there are no stealable scavengers, MSH fetches new scavengers from a pool. These scavengers should be initialized before getting assigned.

For functions starting a thread, MSH obtains scavengers via stealing or fetching and initializes them if necessary before assigning them to the thread. For functions (potentially) blocking a thread, MSH marks the thread's scavengers as stealable before the function call. After the call, MSH first attempts to reuse the scavengers previously assigned to this thread. If some scavengers were stolen, MSH obtains new scavengers with the same logic as the one for thread creation functions. Having "sticky scavengers" is good for cache locality, as scavengers mostly remain in the same core unless the primary thread gets migrated by the kernel. Lastly, for functions terminating a thread, before the thread destruction, MSH marks its scavengers as stealable.

Data structures: MSH tailors its data structures to prioritize critical events that are short but take place frequently, as overhead added to them likely leads to performance degradation. We identify two critical events: (i) primary and scavenger yielding and (ii) primary threads quickly resuming after blocking calls. (i) requires primaries and scavengers to quickly check their yield targets. (ii) occurs because a likely blocking function may not block after all (*e.g.,* synchronization calls).

MSH's data structures are shown in Figure 5. For event

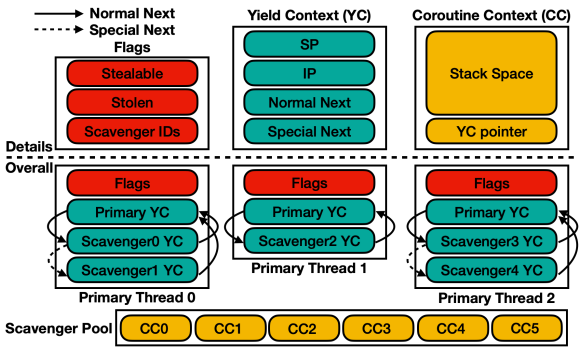

Figure 5: Data structures managed by MSH runtime. Some fields are omitted due to space constraints.

(i), the goal is allowing primaries and scavengers to quickly check their yield targets. A naive design is to have a per-application data structure that stores the context for each primary thread and scavenger. Such a context includes its stack and instruction pointers, and a runtime allocated stack in the case of a scavenger. Each primary thread has a per-thread data structure that stores pointers to contexts. Such a design, while intuitive, adds indirection overhead for yields: each primary thread or scavenger first reads its pointer in the per-thread data structure, in order to read its target's stack and instruction pointers (in a different cache line) from the per-application structure. Given the high frequency and small time budgets of yields, such a design is undesirable.

In contrast, MSH adopts a design that effectively removes the indirection overhead for yields. MSH divides a scavenger context into two parts: a "yield context", containing information needed for yielding to the scavenger, *i.e.,* its stack and instruction pointers; and a "coroutine context", containing other relevant information, *e.g.,* the scavenger stack and a pointer to the yield context. The coroutine context of each scavenger is stored in a per-application data structure, as it is in the naive design. As for the yield context, it is augmented with indexes of its targets (so effectively pointers), and the augmented yield contexts of the primary thread and its scavengers are stored contiguously on the primary's per-thread data structure. With this arrangement, each primary thread or scavenger yields by reading two yield contexts, one of itself and the other of its target. MSH minimizes the size of yield contexts, so that these two yield contexts often reside in the same cache line, resulting in little overhead. Moreover, since scavenger stacks reside in the shared data structure, MSH can easily migrate scavengers by setting up the targets in the per-thread data structures, without having to copy their stacks.

For event (ii), MSH strives to minimize the overhead for when a primary thread quickly resumes with no blocking and no scavengers stolen. A naive design is to maintain the status of each scavenger, whether it is stealable or has been stolen, in a per-application data structure. This makes scavenger stealing simple by just looking for stealable scavengers and

changing their status to stolen. However, such a design complicates the operations that a primary thread needs to perform before and after a (likely) blocking call, which includes reading and setting the status of all the assigned scavengers. This process is unnecessarily expensive when there is no blocking.

In contrast, MSH optimizes for this case by leveraging two per-thread flags: a "stealable" flag indicating whether this thread is blocked, and a "stolen" flag indicating whether some scavengers were stolen. As shown in Listing 1, before a primary thread enters a blocking function, it simply sets the stealable flag to be true. If it does not get blocked, it (i) waits for the stealable flag to become true (explained later), which will be immediate in this case, and (ii) resumes its execution if the stolen flag is false. As a result, a primary thread that quickly resumes at a blocking function only performs a read, a write, and a CAS operation on a single cache line, which is significantly less work than the baseline design.

To steal scavengers, a new thread attempts to compare-and-swap the stealable flags of other threads from true to false. If succeeded, this means that (i) that thread is blocked and (ii) no other thread is stealing from this thread. The new thread then steals the blocked thread's scavengers by looking at their yield contexts – if a scavenger's yield context is valid, it copies the yield context to its own per-thread structure before invalidating the context. The new thread ends its stealing by setting both the stolen and stealable flags of the blocked thread as true. Once the blocked thread resumes, it finds out that some of its scavengers get stolen via the stolen flag, which triggers the slow path of replacing its stolen scavengers with new ones. In essence, by using per-thread flags, MSH expedites the cases where the per-thread flags are untouched due to short or no blocking. The cost of more complex scavenger stealing is acceptable given that stealings happen infrequently.

To sum up, MSH is capable of dynamically assigning scavengers to primary threads for unmodified multi-threaded applications (§6.2) and does so with minimal overhead (§6.4).

5 Implementation

We prototype MSH's offline parts on top of Bolt [67], a binary optimizer, as well as perf [24], a sample-based profiler; and MSH's runtime as a user-level library. Next, we describe how the four main components are implemented:

Offline profiling: MSH adopts the same set of profiling practices as prior sample-based profiling works [17, 30, 31, 41, 66–68]: sampled inputs are used for profiling, and in the case of input changes leading to notable performance degradations, different profiling runs happen in the background. In practice, MSH's performance is observed to be consistent across different inputs. This is because programs often have a fixed set of delinquent load instructions that trigger cache misses, an insight that has been observed and exploited in cache prefetching works [11, 41, 54]. MSH parallelizes profile processing across multiple cores to speed up the process.

Primary instrumentation: There are three phases: a profiling phase, where we profile load instructions causing cache misses via PEBS and basic block execution counts via LBR, and parse profiled data; an analysis phase, where program analysis results (*e.g.,* what registers to save) are annotated in relevant program points (*e.g.,* load instructions, loops); and an instrumentation phase, where binaries are finally altered. We reuse register liveness and reaching definition analysis from Bolt, and implement loop optimizations and pseudo-inlining.

Scavenger instrumentation: This takes place in the same three phases. In the profiling phase, we obtain the basic block latency via LBR. Given that LBR reports the latency between different branching instructions, which does not always correspond to a basic block's latency, we implement a script to map LBR samples to basic blocks. In the analysis phase, we construct call graphs and implement the data-flow analysis.

MSH Runtime: We use the LD_PRELOAD dynamic linker feature [73] to override pthread functions, and implement in a shared library MSH's runtime operations before/after calling the original pthread functions. For per-thread data structures, the runtime sets their base addresses in the GS segment register upon thread creations, so that they can be accessed by primaries and scavengers via GS-based addressing [53].

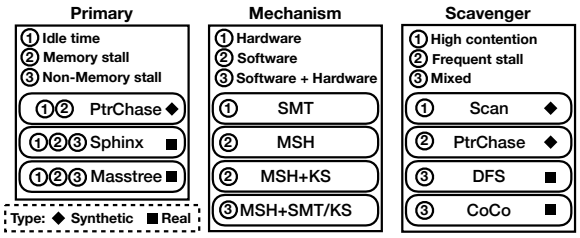

Figure 6: Primaries, scavengers and mechanisms evaluated.

6 Evaluation

In this section, we present our evaluation setup (§6.1) and investigate three key questions regarding MSH: (i) how well does MSH perform compared to SMT? (§6.2), (ii) how does MSH change the landscape of cycle harvesting? (§6.3) and (iii) how do different components of MSH contribute to its performance? (§6.4). We answer (i) and (ii) by evaluating different mechanisms with both synthetic workloads and real applications, (iii) by carefully testing the specific component.

6.1 Evaluation Setup

As shown in Figure 6, we carefully select primaries, scavengers and mechanisms to allow a comprehensive understanding of MSH's behaviors and the cycle harvesting landscape.

Harvestable cycles: To set up evaluations, it is important to realize that there are three main classes of harvestable cycles. The first class is idle time, which occurs at low loads when an application does not have enough work for its cores. Software mechanisms like kernel scheduling (KS) focus on harvesting these cycles. As the load increases, idle time reduces and CPU stalls become the main harvestable cycles. CPU stalls

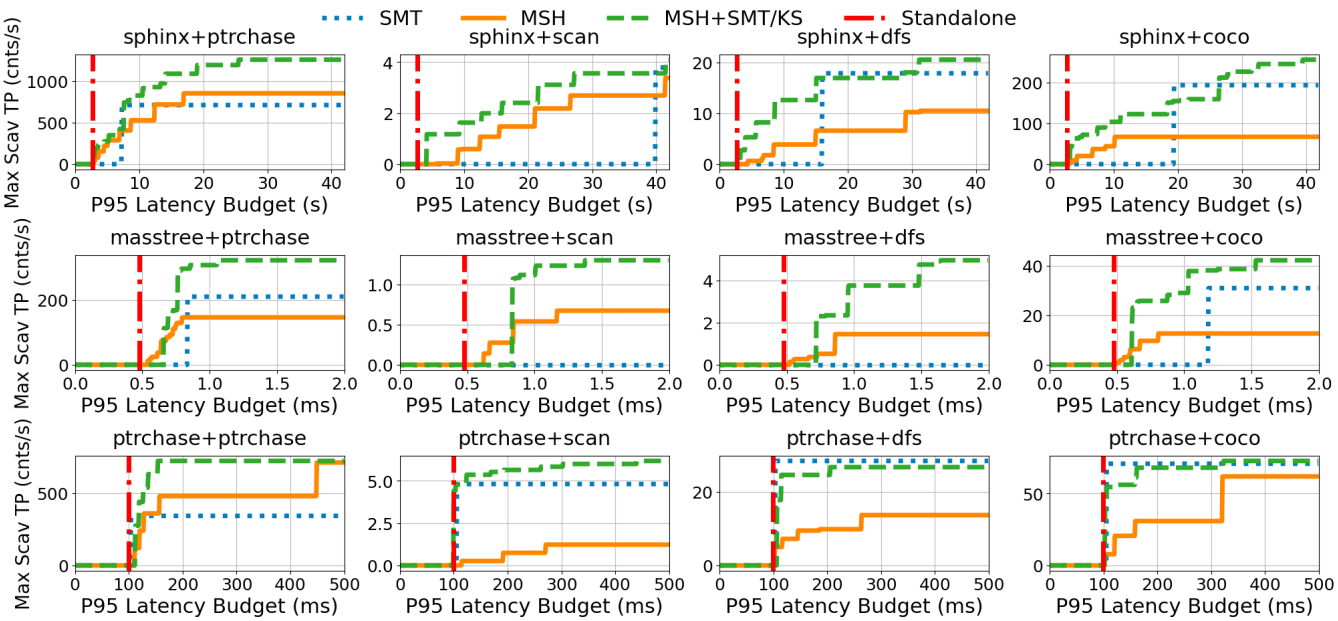

Figure 7: Maximum scavenger throughput vs. P95 Latency budget at 80% load. The red line denotes the standalone latency.

can be divided into either memory stalls, which often account for a significant portion of cycles (§2) and can be efficiently harvested by MSH, or non-memory stalls (*e.g.,* core-bound or frontend stalls), which remain to be private territory of SMT.

Primaries: For primaries, we include a synthetic pointer-chasing workload (PtrChase), which has most of its active cycles bounded by memory. It thus allows us to study how well MSH harvests memory stalls in comparison to SMT. We also have two real latency-critical applications: Masstree [59], an in-memory key-value store, and Sphinx [87], a speech recognition system. With these workloads, we evaluate harvesting mechanisms on realistic mixes of memory and non-memory stalls. Masstree and Sphinx are configured to use the same dataset as Tailbench [46] with 6 and 24 threads respectively. PtrChase has 8 threads, each iterating over its own 16MB array via random pointer chasing upon new requests.

Mechanisms: SMT harvests all three classes of harvestable cycles, but suffers from high latency overhead, lack of configurability, and incomplete harvesting (§2). MSH harvests memory-bound stalls and overcomes the drawbacks of SMT. Building on MSH's superior performance, we complement it with KS and SMT to also harvest idle time and non-memory stalls: KS adds little overhead to MSH but allows idle time harvesting; MSH+SMT/KS enables SMT with MSH if the primary latency meets the SLO, disables SMT and runs KS otherwise. This allows exploiting SMT's ability to harvest non-memory stalls, while managing its latency impacts.

SMT[3] runs scavengers on the sibling cores of the primary. MSH interleaves scavenger executions within the primary. MSH+KS schedules scavengers to run on the primary's logical cores with lower real-time priority, so that these scavengers run when the primary is idle. MSH+SMT/KS runs other scavengers on sibling cores when SMT is enabled.

Scavengers: SMT performs poorly for scavengers that contend for core resources or frequently stall, causing large latency overhead and incomplete harvesting respectively. We thus include synthetic workloads with such behaviors: Scan – creating contention by scanning a 4MB array and computing the sum; PtrChase – frequently stalling due to iterating through a 16MB array in random order via pointer chasing, to evaluate whether MSH can handle such challenging cases. We also include two graph analysis workloads: DFS and Connected Component (CoCo), from the CRONO benchmark [1] as representatives of scavengers with mixed behaviors.

Testbed and Metrics: We conduct experiments using a dual-socket server with 56-core Intel Xeon Platinum 8176 CPUs operating at 2.1 GHz[4]. We measure at different loads the 95 percentile primary latency as well the scavenger throughput in terms of the number of scavengers finished per second.

6.2 MSH performance

Summary: We extensively evaluate MSH and show that it provides three main performance benefits over SMT:

- MSH can harvest up to 72% scavenger throughput of SMT, for latency SLOs under which SMT has to be disabled.
- MSH can further trade off primary latency for higher scavenger throughput if looser latency SLOs are given.

[3] We focus on Intel's SMT implementation (*i.e.,* Hyper-threading) in our evaluation. As we will discuss in §7, drawbacks of SMT stem from the lack of (software-controllable) prioritizations and the limited degrees of concurrency, which are common among most commercial SMT implementations. We thus expect our results to be representative of common SMT behaviors.

[4] Applications use memory from the local node in our evaluation. Under a NUMA setup, MSH can be configured to efficiently harvest the longer stalls caused by remote accesses, *e.g.,* by using larger inter-yield distances (§6.2).

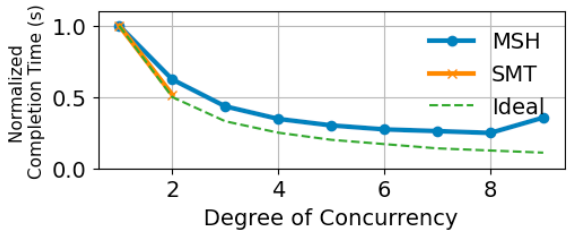

Figure 8: Time to completion for a fixed number of pointer-chasing jobs with different degrees of concurrency.

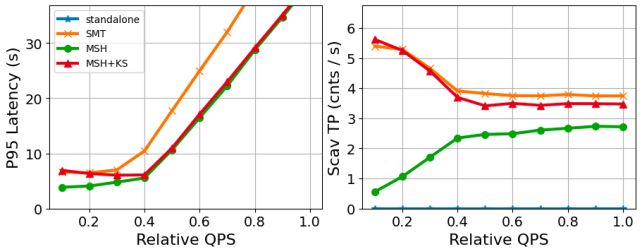

Figure 9: SMT, MSH and MSH+KS for Sphinx+Scan.

- Unlike SMT, MSH can fully harvest memory stalls when scavengers stall and achieve up to 2x higher throughput.

MSH provides these benefits with its capabilities like fine-grained configurability and concurrency scaling, which we will elaborate further on §6.4. Here we focus on presenting MSH's performance characteristics in comparison to SMT.

The whole picture: As shown in Figure 7, for each of the primary and scavenger combinations, we report the maximum achievable scavenger throughputs under different primary latency SLOs, which is defined as the latency budget at 80% loads. Note that, the comparisons among harvesting mechanisms remain unchanged for different latency metrics (e.g. average, 99 percentile) at other loads (other than 80%). As discussed below, MSH can be flexibly configured to achieve different scavenger throughputs depending on the primary latency budgets. These results thus allow us to have a holistic understanding of MSH's performance in comparison to SMT. Here one could make several key observations:

First, MSH harvests substantial stall cycles for latency SLOs under which SMT effectively achieves zero scavenger throughput (*i.e.,* disabled). This is especially valuable when contentious scavengers cause significant slowdown for SMT: *e.g.,* for Sphinx with Scan, MSH achieves up to 72% of SMT scavenger throughput with lower than SMT primary latency. Such behaviors exist for Sphinx and Masstree with all the evaluated scavengers, indicating the general usefulness of MSH as a harvesting mechanism under stringent latency SLOs.

Second, unlike SMT, which achieves the same scavenger throughput regardless of the latency SLO given, MSH can trade off primary latency for higher scavenger throughput. This capability, together with the aforementioned ability to harvest stall cycles under stringent latency SLOs, makes MSH a highly elastic harvesting mechanism that can be combined with other mechanisms, as we will describe in §6.3.

Lastly, MSH can fully harvest memory stalls even when scavengers frequently stall. Specifically, for the PtrChase scavenger, with both Sphinx and PtrChase primaries, MSH manages to achieve higher scavenger throughput than SMT without incurring much latency overhead. Given that SMT harvests both idle time and non-memory stalls, which MSH does not handle, this indicates that MSH can better harvest memory stalls with higher degrees of concurrency.

Full harvesting: To verify this, we conduct an experiment with a fixed number of jobs, where each job traverses a 128 MB array via random pointer chasing and thus frequently incurs memory stalls. We then measure the total completion time of these jobs with a single physical core. For SMT, we either run one job at a time or co-locate two concurrent jobs. For MSH, we interleave these jobs with various degrees of concurrency. The normalized completion times are shown in Figure 8. In the ideal case, the completion time is one over the concurrency degree. Although SMT-2 is close to ideal thanks to hardware efficiency, it does not have enough concurrency to further harvest memory stalls. In contrast, while having larger interleaving overhead, MSH reduces SMT's completion time by roughly a half (*i.e.,* 2x throughput) with a concurrency degree of eight. This shows that compared with SMT, MSH can harvest more memory stalls via concurrency scaling. When the degree of concurrency goes beyond eight, the completion time of MSH increases due to the aggregate yielding overhead outweighing the benefits of additional multiplexings.

6.3 Cycle Harvesting Landscape

With various desirable properties, MSH can be efficiently combined with other harvesting mechanisms to re-shape the CPU cycle harvesting landscape. To see this, we evaluate two compound mechanisms that leverage MSH for memory stalls: MSH+KS and MSH+SMT/KS, and compare that with SMT.

- **MSH+KS:** KS complements MSH with idle time harvesting. MSH+KS thus achieves much higher scavenger throughput than MSH at low loads, while adding small latency overhead (Figure 9). As the load increases, idle time reduces, and MSH+KS behaviors converge to MSH's. Note that MSH in this figure is only one configuration.

- **MSH+SMT/KS:** MSH+SMT/KS strives to utilize SMT's ability to harvest non-memory stalls, and falls back to KS if SMT incurs excessive latency overhead. As shown in Figure 7, MSH+SMT/KS delivers superior performance, with higher scavenger throughput than SMT under *almost all* latency SLOs. The reason is that: (i) for scavengers that frequently stall, SMT can be safely enabled with minimal latency overhead, the combination of SMT and MSH can harvest idle time, non-memory and memory stalls to the full extent; (ii) for contentious scavengers, the combination

of KS and MSH then efficiently harvests both idle time and memory stalls for latency SLOs where SMT is disabled.

6.4 Performance Breakdown

Summary: We test MSH's configurability and performance of its components, the results of which are outlined below:

- **Configurability:** MSH offers fine-grained control over the latency-throughput trade-off via (i) yield site selections in primary instrumentation, (ii) inter-yield distances in scavenger instrumentation and (iii) concurrency degrees in runtime. Since the effects of concurrency scaling have been studied in Figure 8, we focus on the other two knobs. We measure the primary latency and scavenger throughput for Sphinx and Scan with different configurations, with results shown in Figure 10. For the primary, MSH estimates the overhead of each load instruction with its cache miss rate and bounds the aggregate overhead when selecting yield sites (§4.1). We increase this overhead bound from 5% to 15% and observe a clear latency-throughput trade-off as more yields are instrumented. For the scavenger, increasing the target inter-yield distance also leads to higher scavenger throughput at the cost of larger primary overhead. Besides the latency-throughput trade-off, such configurability allows MSH to mitigate some inherent issues of instruction interleaving, such as increased memory contention and effectively partitioned caches, by controlling the extent and locations of interleaving.
- **Primary instrumentation:** MSH reduces the yield cost by minimizing the amount of register savings and restorations per yield. To measure how this affects its harvesting performance, we conduct an experiment with Sphinx and Scan, where we measure Sphinx's latency for different inter-yield distances of Scan, with and without our optimizations. As shown in Figure 11, reduced yield costs do lead to up to 23% lower primary latency. Note that the improvement first increases with scavenger inter-yield distances before dropping, because (i) the larger yield cost (without optimizations) does not affect the primary latency until the duration of the interleaved scavenger execution (*i.e.,* inter-yield distance plus yield cost) exceeds the cache hit latency, and (ii) as the inter-yield distance further increases, yield cost plays a smaller part in the overall overhead.
- **Scavenger instrumentation:** MSH accurately enforces target inter-yield distances via its data-flow analysis (Figure 12-(a)). As for overhead, a unique source of overhead for scavengers is the loop instrumentation overhead – using an in-memory iteration counter is expensive for tight loops. MSH thus attempts to reuse induction registers or maintain a counter with unused registers before spilling to memory. This optimization reduces the overhead by 130% and 15% for CoCo and DFS respectively (Figure 12-(b)).
- **MSH runtime:** MSH harvests stall cycles via dynamic scavenger assignment. It does so with low overhead: 10 ns for thread resuming with unstolen scavengers, which does not cause noticeable impacts on our evaluated applications.
- **Profiling overhead:** Even with sample-based profiling using hardware performance counters, sampling events at high frequencies can still slow down the primary application. In MSH, we confirm that accurately capturing delinquent load instructions incurs minimal profiling overhead. Specifically, for Masstree, using the default sampling frequency and following the yield site selection logic (§4.1), MSH selects the *same* set of load instructions as if it were to sample 100x more frequently. As a result, while using a 100x higher sampling rate would slow down the application by 25%, the slowdown from MSH's profiling is negligible.
- **Analysis complexity:** MSH instruments only selective loads and performs mostly intra-procedural analysis, which finishes less than a minute for all the evaluated workloads.

7 Related Work

Reducing memory stalls: Orthogonal to harvesting efforts like MSH, there has been extensive research on reducing memory stalls. Beyond out-of-order executions, there are two lines of techniques based on *load slices*, *i.e.,* instructions that generate the address of a load instruction. One technique is prefetching [2, 4, 8, 12, 22, 39, 42, 56], where the cache line is prefetched after the end of its load slice; and the other technique is criticality-aware instruction scheduling [5, 6, 16, 77], where the processor prioritizes the executions of load slices, which requires hardware changes. For both techniques, there is a trade-off between capability and deployability. Simple techniques like stream prefetchers [39, 75] and prefetch insertion via static analysis [4, 20] have limited capability (*e.g.,* unable to handle complex access patterns); whereas advanced proposals like runahead prefetchers [26, 33] often have requirements that hinder wide adoptions (*e.g.,* excessive hardware complexity, source code modification). Moreover, a key requirement for both techniques to reduce stalls is that load slices end *sufficiently ahead* of the load instruction. As a result, for cases where load slices are close to the load instruction, neither technique can help. In contrast, MSH is easily deployable, requiring no hardware changes nor rewriting efforts, and harvests stall cycles for any access pattern.

SMT: For the three drawbacks of SMT (*i.e.,* latency overhead, lack of configurability and incomplete harvesting), the first two stem from the lack of prioritizations, whereas the last one is due to limited degrees of concurrency. Most modern processors from Intel and AMD have these two issues, which leads to unsatisfactory harvesting performance (§6.2). An exception is IBM Power processors [50, 63], as they (i) support assigning hardware threads with priorities that determine the ratio of physical core decode slots allotted to them, and (ii) have wider SMT with up to eight threads per core, at the cost of more complex and resource-consuming SMT design.

Given this context, the value of MSH is two fold. First,

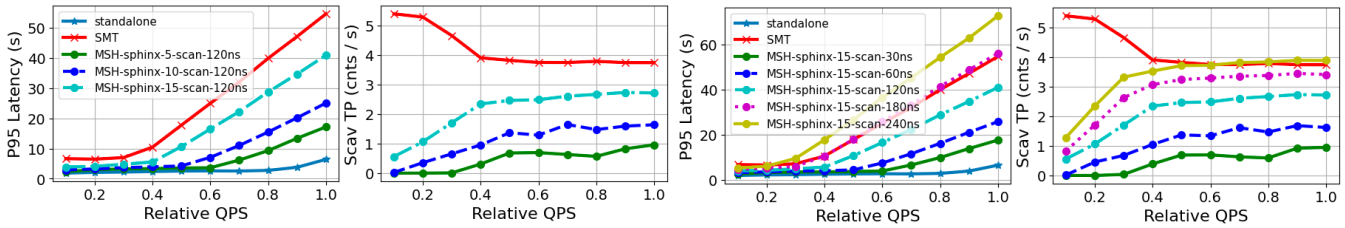

Figure 10: The effects of the aggregate yield overhead bound (left) and the scavenger inter-yield distance (right) on the primary latency and the scavenger throughput in Sphinx+Scan.

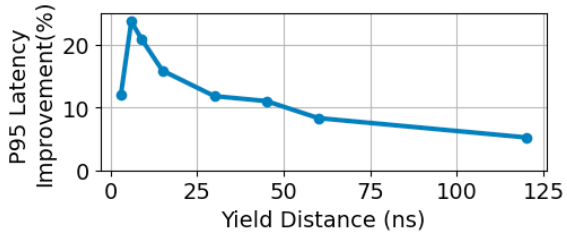

Figure 11: Latency improvement made by the yield cost optimizations in the primary instrumentation on Sphinx+Scan.

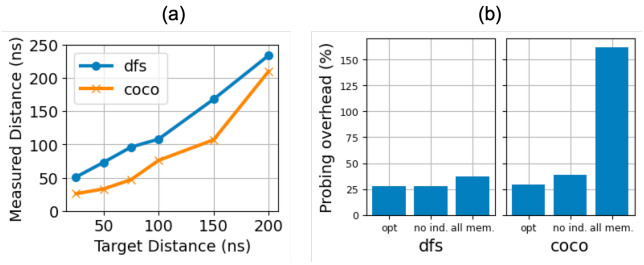

Figure 12: (a) Inter-yield distance of scavenger instrumentation; (b) overhead of loop instrumentation: *opt* uses induction registers and unused registers, *no ind.* uses only unused registers, and *all-mem* uses in-memory iteration counters.

for most modern processors, MSH allows harvesting memory stall cycles in software without the drawbacks of their SMT mechanisms. Second, for processors like IBM Power and Cray Threadstorm [48, 49] that support massive multithreading and fine-grained parallelism, MSH raises the question of whether certain functionality should be implemented in hardware or software, *e.g.,* concurrency scaling in MSH happens *on-demand*, without requiring dedicated thus likely wasted resources, such as die area and power.

Software efforts: Some work focuses on utilizing SMT with latency-critical services, by disabling it when high latency or resource interference is detected [29, 60, 70, 89]. However, they do not address SMT's high latency overhead and lack of configurability, and are thus unable to harvest stall cycles when SMT violates latency SLOs. As for software harvesting efforts, prior work shows that if done correctly, prefetching and yielding before load instructions can lead to increased throughput for memory-intensive workloads [21, 34, 44, 72]. However, they either require manual identification of yield sites and source code modification, or instrument every load instruction at the cost of high latency. Moreover, none of them can enforce low latency overhead and full harvesting from diverse scavengers, which MSH provides with scavenger instrumentation and runtime operations. In short, MSH is the first software system that enables transparent and general memory stall harvesting with competitive performance.

8 Discussion

Isolation mechanism: In MSH, the primary and its scavengers reside in the same process to benefit from fast yielding, which necessitates mechanisms other than hardware isolation to ensure memory safety under this setup. This turns out to be an extensively studied problem, with solutions falling into two main categories: (i) software-based fault isolation (SFI) [76, 81, 86], which establishes logical protection domains by inserting dynamic checks at the binary level; and (ii) language-based isolation, where a program is accepted in the form of a safe language (*e.g.,* WebAssembly [32, 36, 85], Rust [15, 51, 65, 92]) and validated by the type checker and compiler. Operating at the binary level, MSH easily coexists with either isolation mechanism: SFI can be a better fit as it is applicable to code written in different languages, including legacy code, which is a merit that MSH shares. Moreover, a recent work [91] shows lower runtime overhead with a lightweight SFI implementation than existing language-based solutions. Integrating MSH with some isolation mechanism and evaluating the resulting system is left for future work.

Further evaluation: In §6, we demonstrated and dissected the desirability of MSH as a harvesting mechanism. Next, we discuss directions for more thorough evaluation of MSH.

- **Additional workloads:** We focus on evaluating a set of representative workloads with distinct characteristics, *e.g.,* scavengers that either create large contentions, or frequently stall, or exhibit mixed behaviors. This appraoch allows us to interpret the performance differences caused by (i) the distinct characteristics of the primary-scavenger pairs and (ii) the differences in harvesting mechanisms. One could extend with more real workloads
- **Cache prefetching:** As discussed in §7, MSH can harvest memory stalls that are not hidden by cache prefetching, and prefetching techniques that are easy to deploy usually have limited capability. It will thus be interesting to evaluate the

effectiveness of MSH with software prefetching techniques used in production [40]. That being said, most delinquent loads in our evaluated workloads exhibit pointer-chasing behaviors, which are inherently challenging to prefetch.

- **Datacenter efficiency:** The effect of MSH on the overall CPU efficiency of a datacenter is hard to estimate, as it depends on various factors such as workload characteristics, colocation arrangements, and SLO policies. This necessitates large-scale evaluation and profiling [45].

Efficacy of profiling: In terms of profiling overhead, we have shown that MSH can capture delinquent load instructions with a low sampling rate (§6.4). The other natural question is whether profiling is consistently effective for the purpose of harvesting stall cycles in MSH. Similar to prior works that leverage profiling for cache prefetching [40, 41, 95], we show positive results with our evaluated workloads (§6.2). One conjecture is that, while whether a particular load invocation will trigger a cache miss is highly random, the two pieces of information MSH needs from profiling – namely, (i) the set of load instructions that account for a significant portion of memory stalls and (ii) their likelihoods of cache misses, are often stable across runs and inputs. Evaluating a wider range of applications can help further validate this conjecture.

Hardware support for MSH: We identify two aspects that MSH can benefit from hardware support. First, an overhead that MSH inevitably incurs is when an instrumented load causes cache hits. MSH mitigates this with the selection logic in primary instrumentation, which enforces a lower bound on cache miss rate and an upper bound on aggregate overhead (§4.1). To do better, what is needed is *dynamic visibility* of cache misses, *e.g.,* indicating if a cache line is in L2 cache. This allows yields to be conditional on whether cache misses actually happen. We expect conditional checking overhead to be on the scale of L2 cache latency, much faster than scavenger executions configured to harvest memory stalls.

Another aspect that hardware can offer support is reducing yield overhead. MSH minimizes the amount of register savings and restorations for each yield, which leads to lower latency overhead (§6.4). One useful hardware feature here is to save/restore multiple registers to/from memory with a single instruction for lower instruction fetch costs, which is already provided in ARM with LDM/STM instructions [9]. Prior works also propose hardware support for fast saving and restoration of process state during context switches [38, 80].

9 Conclusion

We presented MSH, a software system that transparently and efficiently harvests memory stall cycles. With a co-design of profiling, program analysis, binary instrumentation and runtime scheduling, MSH fully harvests stall cycles, while incurring minimal latency overhead and offering fine-grained control of the latency-throughput tradeoff. MSH is thus a preferable solution for harvesting memory stalls and brings valuable changes to the CPU cycle harvesting landscape.

References

[1] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *2015 IEEE International Symposium on Workload Characterization*, pages 44–55. IEEE, 2015.

[2] Sam Ainsworth and Timothy M Jones. An event-triggered programmable prefetcher for irregular workloads. *ACM Sigplan Notices*, 53(2):578–592, 2018.

[3] Soramichi Akiyama and Takahiro Hirofuchi. 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*, pages 1–8, 2017.

[4] Hassan Al-Sukhni, Ian Bratt, and Daniel A Connors. Compiler-directed content-aware prefetching for dynamic data structures. In *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 91–100. IEEE, 2003.

[5] Mehdi Alipour, Stefanos Kaxiras, David Black-Schaffer, and Rakesh Kumar. Delay and bypass: Ready and criticality aware instruction scheduling in out-of-order processors. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 424–434. IEEE, 2020.

[6] Mehdi Alipour, Rakesh Kumar, Stefanos Kaxiras, and David Black-Schaffer. Fiforder microarchitecture: Ready-aware instruction scheduling for ooo processors. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 716–721. IEEE, 2019.

[7] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, et al. Providing {SLOs} for {Resource-Harvesting}{VMs} in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751, 2020.

[8] Murali Annavaram, Jignesh M Patel, and Edward S Davidson. Data prefetching by dependence graph precomputation. *ACM SIGARCH Computer Architecture News*, 29(2):52–61, 2001.

[9] ARM. Arm developer suite assembler guide. https://developer.arm.com/documentation/dui0068/b/Writing-ARM-and-Thumb-Assembly-Language/Load-and-store-multiple-register-instructions/ARM-LDM-and-STM-instructions, 2023.

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

[11] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 513–526, 2020.

[12] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Bingo spatial data prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 399–411. IEEE, 2019.

[13] 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*, pages 1249–1263, 2021.

[14] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. Checking memory safety with blast. In *International Conference on Fundamental Approaches to Software Engineering*, pages 2–18. Springer, 2005.

[15] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1–19, 2020.

[16] Trevor E Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. The load slice core microarchitecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, pages 272–284, 2015.

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

[18] Ruobing Chen, Haosen Shi, Yusen Li, Xiaoguang Liu, and Gang Wang. Olpart: Online learning based resource partitioning for colocating multiple latency-critical jobs on commodity computers. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 347–364, 2023.

[19] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[20] William Y Chen, Scott A Mahlke, Pohua P Chang, and Wen-mei W Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 69–73, 1991.

[21] Shenghsun Cho, Amoghavarsha Suresh, Tapti Palit, Michael Ferdman, and Nima Honarmand. Taming the killer microsecond. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 627–640. IEEE, 2018.

[22] Jamison D Collins, Hong Wang, Dean M Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P Shen. Speculative precomputation: Long-range prefetching of delinquent loads. *ACM SIGARCH Computer Architecture News*, 29(2):14–25, 2001.

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

[24] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

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

[26] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th international conference on Supercomputing*, pages 68–75, 1997.

[27] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 727–741, 2023.

[28] Roman Elizarov, Mikhail Belyaev, Marat Akhin, and Ilmir Usmanov. 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, pages 68–84, 2021.

[29] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[30] Google. Propeller: Profile guided optimizing large scale llvmbased relinker. https://github.com/google/llvm-propeller, 2020.

[31] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 692–708, 2023.

[32] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[33] Milad Hashemi, Onur Mutlu, and Yale N Patt. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

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

[35] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *Software safety and security*, pages 319–347. IOS Press, 2012.

[36] Pat Hickey. How fastly and the developer community are investing in the webassembly ecosystem. https://www.fastly.com/blog/how-fastly-and-developer-community-invest-in-webassembly-ecosystem/, 2020.

[37] Joel Hruska. 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, 2012.

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

[39] Ibrahim Hur and Calvin Lin. Memory prefetching using adaptive stream detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 397–408. IEEE, 2006.

[40] Akanksha Jain, Hannah Lin, Carlos Villavieja, Baris Kasikci, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. Limoncello: Prefetchers for scale. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 577–590, 2024.

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

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

[43] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments. In *Proceedings of the ACM symposium on cloud computing*, pages 272–285, 2019.

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

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

[46] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

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

[48] Petr Konecny. Introducing the cray xmt. In *Proc. Cray User Group meeting (CUG 2007). Seattle, WA: CUG Proceedings*, 2007.

[49] Andrew Kopser and Dennis Vollrath. Overview of the next generation cray xmt. In *Cray User Group Proceedings*, pages 1–10, 2011.

[50] Hung Q Le, JA Van Norstrand, Brian W Thompto, José E Moreira, Dung Q Nguyen, David Hrusecky, MJ Genden, and Michael Kroener. Ibm power9 processor core. *IBM Journal of Research and Development*, 62(4/5):2–1, 2018.

[51] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 234–251, 2017.

[52] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-chun Feng, Mark Gardner, and Zhe Zhang. Moon: Mapreduce on opportunistic environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 95–106, 2010.

[53] Linux. Using fs and gs segments in user space applications. https://www.kernel.org/doc/html/next/x86/x86_64/fsgs.html, 2023.

[54] Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. Crisp: critical slice prefetching. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 300–313, 2022.

[55] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[56] Chi-Keung Luk and Todd C Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 222–233, 1996.

[57] Zhihong Luo, Silvery Fu, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. Out of hand for hardware? within reach for software! In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 30–37, 2023.

[58] Zhihong Luo, Sam Son, Dev Bali, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. Efficient microsecond-scale blind scheduling with tiny quanta. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 305–319, 2024.

[59] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[60] Artemiy Margaritov, Siddharth Gupta, Rekai Gonzalez-Alberquilla, and Boris Grot. Stretch: Balancing qos and throughput for colocated server workloads on smt cores. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–27. IEEE, 2019.

[61] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible colocations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

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

[63] Alessandro Morari, Carlos Boneti, Francisco J Cazorla, Roberto Gioiosa, Chen-Yong Cher, Alper Buyuktosunoglu, Pradip Bose, and Mateo Valero. Smt malleability in ibm power5 and power6 processors. *IEEE Transactions on Computers*, 62(4):813–826, 2012.

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

[65] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. {RedLeaf}: isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39, 2020.

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

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

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

[69] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206. IEEE, 2020.

[70] Aidi Pi, Xiaobo Zhou, and Chengzhong Xu. Holmes: Smt interference diagnosis and cpu scheduling for job co-location. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pages 110–121, 2022.

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

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

[73] Kevin Pulo. Fun with ld_preload. In *linux. conf. au*, volume 153, page 103, 2009.

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

[75] Suleyman Sair, Timothy Sherwood, and Brad Calder. A decoupled predictor-directed stream prefetching architecture. *IEEE Transactions on Computers*, 52(3):260–276, 2003.

[76] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary {CPU} architectures. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[77] Andreas Sembrant, Trevor Carlson, Erik Hagersten, David Black-Shaffer, Arthur Perais, André Seznec, and Pierre Michaud. Long term parking (ltp) criticality-aware resource allocation in ooo processors. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 334–346, 2015.

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

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

[80] Jovan Stojkovic, Chunao Liu, Muhammad Shahbaz, and Josep Torrellas. μmanycore: A cloud-native cpu for tail at scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

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

[82] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 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*, pages 157–173. Springer, 2010.

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

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

[85] Kenton Varda. Webassembly on cloudflare workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/, 2018.

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

[87] Willie Walker, Paul Lamere, Philip Kwok, Bhiksha Raj, Rita Singh, Evandro Gouvea, Peter Wolf, and Joe Woelfel. Sphinx-4: A flexible open source framework for speech recognition, 2004.

[88] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. Smartharvest: Harvesting idle cpus safely and efficiently in the cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 1–16, 2021.

[89] Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. Elfen scheduling:{Fine-Grain} principled borrowing from {Latency-Critical} workloads using simultaneous multithreading. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 309–322, 2016.

[90] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44. IEEE, 2014.

[91] Zachary Yedidia. Lightweight fault isolation: Practical, efficient, and secure software sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 649–665, 2024.

[92] 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. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 195–211, 2021.

[93] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391, 2013.

[94] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. {History-Based} harvesting of spare cycles and storage in {Large-Scale} datacenters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 755–770, 2016.

[95] Yuxuan Zhang, Nathan Sobotka, Soyoon Park, Saba Jamilan, Tanvir Ahmed Khan, Baris Kasikci, Gilles A Pokam, Heiner Litz, and Joseph Devietti. Rpg2: Robust profile-guided runtime prefetch generation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 999–1013, 2024.