Topics in Extreme System Design

Zhihong Luo

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-89
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-89.html

May 16, 2025

Copyright © 2025, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Topics in Extreme System Design

By

Zhihong Luo

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Scott Shenker, Co-Chair
Professor Sylvia Ratnasamy, Co-Chair
Assistant Professor Natacha Crooks
Assistant Professor Aurojit Panda

Spring 2025

Topics in Extreme System Design

Copyright © 2025

by

Zhihong Luo

Abstract

Topics in Extreme System Design

by

Zhihong Luo
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Scott Shenker, Co-Chair

Professor Sylvia Ratnasamy, Co-Chair

This dissertation presents a series of systems that push the boundaries of what is considered feasible and widely accepted in the design of datacenter and cellular infrastructures. The first part focuses on improving datacenter efficiency through finer-grained software mechanisms. It introduces software systems that harvest memory-bound CPU stall cycles for useful work and perform object-level memory management in tiered memory systems. The second part explores how cellular network functionality can be expanded through clean-slate architectural redesigns. It introduces systems that challenge entrenched assumptions around network control and user privacy, demonstrating the feasibility of more open and privacy-preserving models.

The first system, MSH, targets the underutilization of memory-bound CPU stall cycles, which represent a significant inefficiency in datacenter workloads. Traditional hardware-based approaches like simultaneous multithreading (SMT) are limited in configurability and concurrency, especially for latency-sensitive services. MSH is a software system that transparently and efficiently harvests stall cycles using a co-design of profiling, static analysis, binary instrumentation, and runtime scheduling. It achieves high throughput with minimal latency overhead, outperforming SMT in scenarios where SMT cannot be used due to the latency requirement.

The second system, Fava, addresses the challenge of effective data placement in tiered memory systems combining fast local memory with slower disaggregated options such as CXL-attached memory. Unlike prior approaches that operate at the page or cache-line level, Fava enables object-level memory management in managed-language environments. It accurately tracks object hotness with minimal overhead and leverages a hybrid mechanism combining object colocation with page migration. As a result, Fava significantly improves local memory utilization and reduces

application slowdowns compared to state-of-the-art systems.

The third system, CellBricks, introduces a novel cellular architecture that lowers the barrier to entry for new network operators. By moving key functionality such as mobility and user management out of the network and into end hosts, CellBricks enables users to dynamically access service from a variety of operators, including small-scale and untrusted ones. This design fosters greater competition and flexibility while maintaining performance comparable to traditional infrastructure.

Finally, LOCA tackles the long-standing issue of location privacy in cellular networks. Today's architectures allow operators to track both the identity and location of users, posing serious privacy risks. LOCA decouples location information from identity while preserving support for identity-based services such as billing, lawful intercept, and emergency access. Leveraging MVNO-based deployments, LOCA redesigns key cellular protocols using cryptographic primitives to deliver strong privacy guarantees without sacrificing scalability or service quality.

Together, these four systems demonstrate how rethinking software mechanisms and architectural designs can lead to meaningful gains in efficiency, performance, openness, and privacy across modern computing infrastructure.

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

This dissertation is the result of collaboration with incredible individuals, including Sam Son, Silvery Fu, Shaddi Hasan, Natacha Crooks, Sylvia Ratnasamy, and Scott Shenker. I am deeply grateful to all those who supported me throughout this journey. First and foremost, I thank my advisors, Scott Shenker and Sylvia Ratnasamy. Scott, thank you for sharing your wisdom not only in research but in all aspects of life. Your encouragement to stay true to myself and pursue what I truly want has left a lasting impact. I deeply admire your clarity of thought, honesty, and your ability to pinpoint areas where I could improve. Sylvia, thank you for your patience in guiding me through the early chapters of my Ph.D. when I was unsure how to find my place in academia. You showed me how to precisely identify the arguments worth making in research, and your rigor has been a constant source of inspiration.

I also want to thank the senior collaborators and mentors who taught me valuable lessons. Aurojit Panda, thank you for generously sharing your deep knowledge when I was searching for research ideas in unfamiliar domains. Natacha Crooks, thank you for guiding me in thinking rigorously about privacy-preserving systems. Amy Ousterhout, thank you for helping me understand how to analyze trade-offs in system design through meticulous experimentation. Shaddi Hasan, thank you for introducing me to the arcane world of cellular protocols when I first started.

To my friends and peers, thank you for your companionship and support throughout this journey. Jingqi Li, Emmanuel Amaro, Christopher Branner-Augmon, Lloyd Brown, Tenzin Ukyab and Wen Zhang, thank you for all the encouragement and the memorable conversations about life beyond research. Sam Son, thank you for being a close collaborator and for the reflections on life we shared. Silvery Fu, thank you for supporting me through the highs and lows of my Ph.D. and for helping me grow as both a reliable collaborator and a friend.

To my family, thank you for your unwavering love and support throughout this journey. My parents, Junsheng Luo and Taolian Luo, thank you for instilling in me perseverance and a goal-driven mindset, qualities that have deeply shaped how I approach challenges, both in work and in life. Last and foremost, to my wife Yichen Zhang: your patience, intelligence, and deep understanding of who I am have helped me grow in ways I could not have on my own. You have shown me how to better understand others and direct my determination toward what truly matters. You have encouraged me to reflect honestly on what I value, to define my goals with clarity and openness, and to pursue them with intention and authenticity. Your insight, kindness, and honesty have shaped me into not only a better researcher but, far more importantly, a better person. I am truly grateful for your presence in my life and for everything we have shared along the way.

Chapter 1

Memory Stall Software Harvesting

1.1 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 [315, 27, 188, 154, 342]. 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 [36, 79, 160, 277]: 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 [301, 299, 138, 253]. 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 [193, 58, 57, 238]. 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 [160, 157, 248].

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 [200] discusses the possibility of transparently hiding the latency of cache misses in software with the combination of light-weight coroutines [218, 94, 279, 86] and sample-based profiling [168, 56, 295]. 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 [236], 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

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

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 §1.8. 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.

1.2 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 [336] 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 analysis confirms the dominance of memory-bound stalls, as they account for 25% and 31% of total cycles for Masstree and Sphinx respectively (Figure 1.1). While there have been extensive efforts on *reducing* memory stalls, it is generally infeasible to eliminate them (§1.7). 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

[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 §1.7.

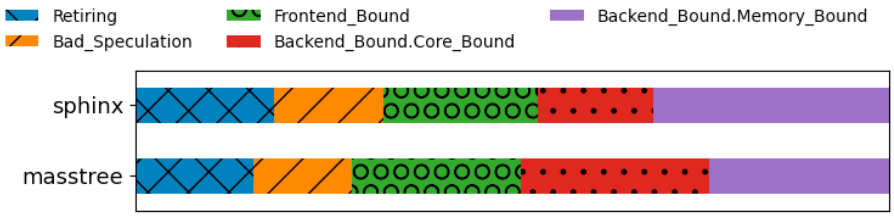

Figure 1.1: Top-down analysis of Sphinx and Masstree; memory stalls account for 25% and 31% of cycles respectively.

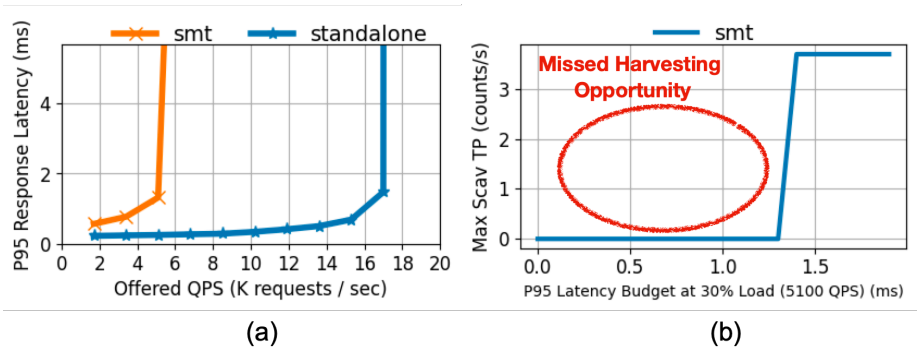

Figure 1.2: (a) P95 latency of Masstree when running by itself vs. co-locating with a Scan scavenger; (b) SMT is unable to harvest stall cycles under low latency SLOs.

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 1.2-(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 1.2-(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 [200] 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 [19] and LBR [168], 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) [232, 56, 236, 116, 237].
- **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 trans-

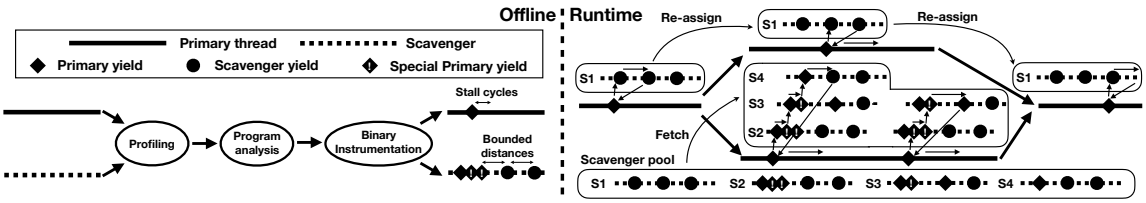

Figure 1.3: MSH system overview. Offline: MSH profiles and analyzes primaries and scavengers. It then instruments the primaries to yield control to scavengers at likely memory stall sites, with scavengers returning control to primaries within a bounded time. Runtime: MSH sets up a scavenger pool and dynamically assigns scavengers to each active primary thread.

parently and efficiently harvests memory-bound stall cycles. Next, we present an overview of MSH.

1.3 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 [340], *e.g.,* they are crash-free and access memory safely. Ensuring safety properties with techniques like verification and information

flow control [132, 211, 44] is left to future work. MSH can be seamlessly integrated with existing profiling systems deployed for PGO [116, 56, 237]. MSH is well suited for when latency-critical and best-efforts tasks are co-located in the same machine, a common arrangement in production [92, 193, 341, 208]. 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 1.3). 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 §1.4.1 and §1.4.2.

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. 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 §1.4.3.

1.4 Design

MSH consists of three components: primary instrumentation (§1.4.1), scavenger instrumentation (§1.4.2) and a runtime (§1.4.3).

1.4.1 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 (§1.6.4).

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 (§1.4.3). 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 [244], 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 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 1.4, 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 1.4), MSH adopts a hybrid approach that

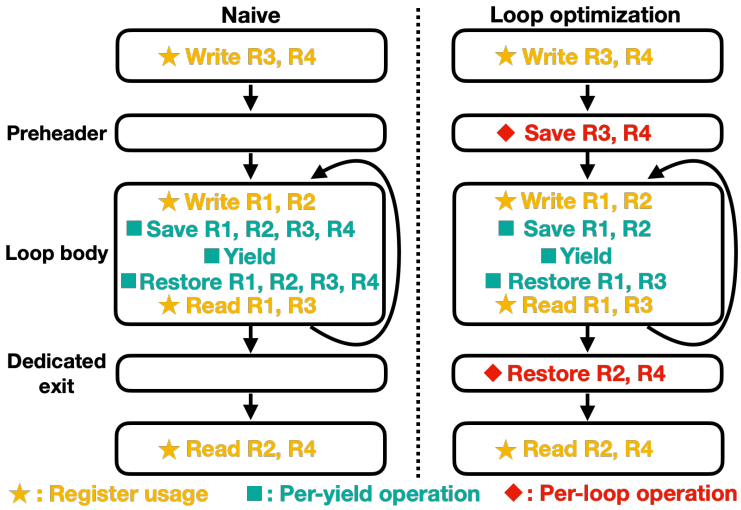

Figure 1.4: Loop optimization in primary instrumentation.

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 (§1.6.4).

1.4.2 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 (§1.4.3). 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 [41, 203]. 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 (§1.6.2) and manage latency impacts by enforcing inter-yield distance bounds via data-flow analysis (§1.6.4).

1.4.3 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. 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

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

```
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;
    }
}
```

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 1.5. For event (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

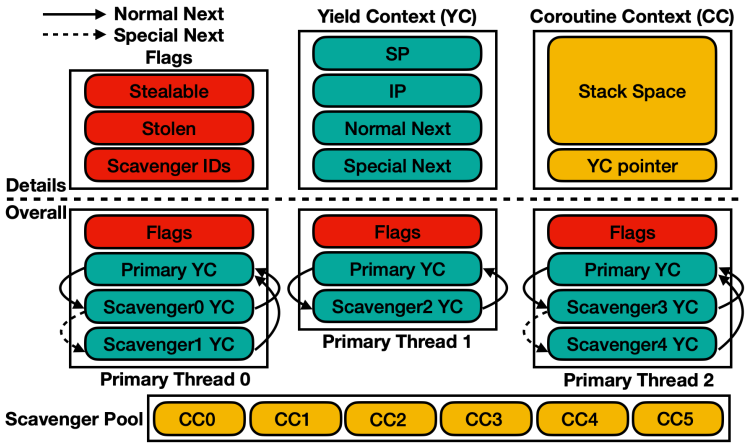

Figure 1.5: Data structures managed by MSH runtime. Some fields are omitted due to space constraints.

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 (§1.6.2) and does so with minimal overhead (§1.6.4).

1.5 Implementation

We prototype MSH's offline parts on top of Bolt [236], a binary optimizer, as well as perf [84], 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 [232, 152, 125, 56, 236, 116, 237]: 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 [152, 190, 37]. 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 [249] 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 [189].

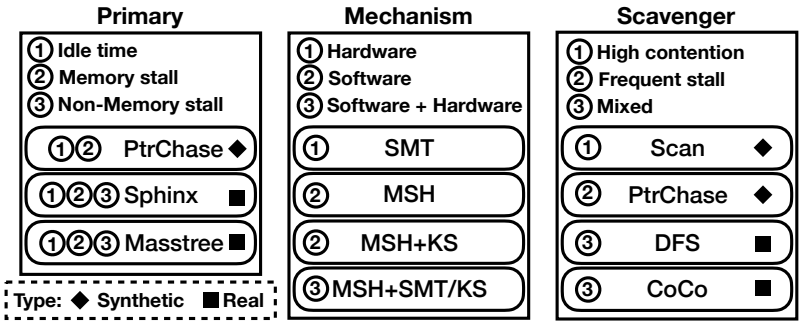

Figure 1.6: Primaries, scavengers and mechanisms evaluated.

1.6 Evaluation

In this section, we present our evaluation setup (§1.6.1) and investigate three key questions regarding MSH: (i) how well does MSH perform compared to SMT? (§1.6.2), (ii) how does MSH change the landscape of cycle harvesting? (§1.6.3) and (iii) how do different components of MSH contribute to its performance? (§1.6.4). We answer (i) and (ii) by evaluating different mechanisms with both synthetic workloads and real applications, (iii) by carefully testing the specific component.

1.6.1 Evaluation Setup

As shown in Figure 1.6, 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 can be divided into either memory stalls, which often account for a significant portion of cycles (§1.2) 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 [206], an in-memory key-value store, and Sphinx [311], a speech recognition system. With these workloads, we evaluate harvesting mechanisms on realistic mixes of memory and non-memory stalls. Masstree

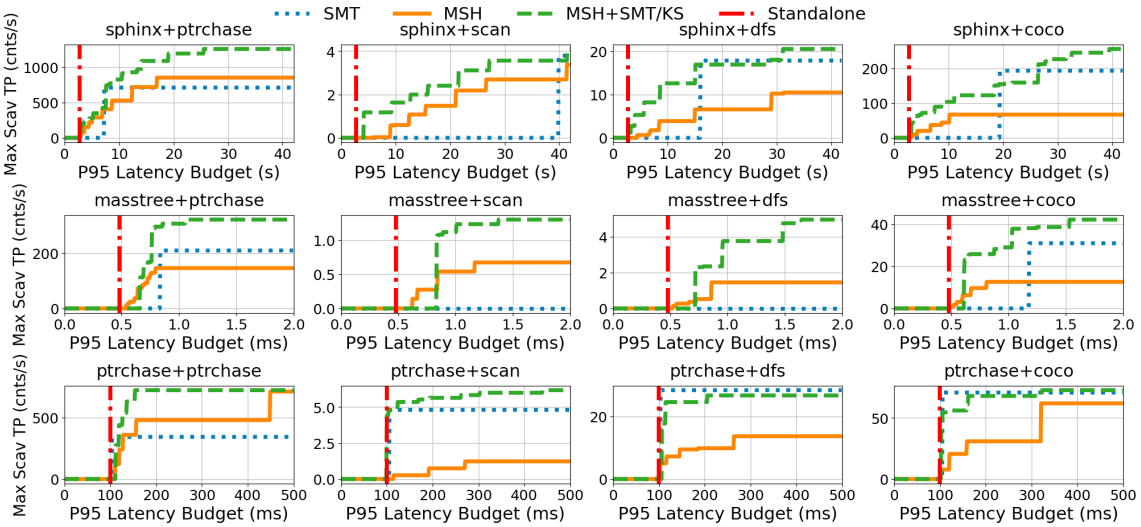

Figure 1.7: Maximum scavenger throughput vs. P95 Latency budget at 80% load. The red line denotes the standalone latency.

and Sphinx are configured to use the same dataset as Tailbench [162] 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 (§1.2). 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.

[3]We focus on Intel's SMT implementation (*i.e.,* Hyper-threading) in our evaluation. As we will discuss in §1.7, 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.

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 [17] 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.

1.6.2 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.
- 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 §1.6.4. Here we focus on presenting MSH's performance characteristics in comparison to SMT.
The whole picture: As shown in Figure 1.7, 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:

[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 (§1.6.2).

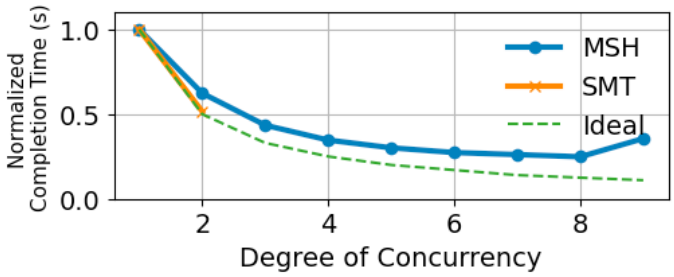

Figure 1.8: Time to completion for a fixed number of pointer-chasing jobs with different degrees of concurrency.

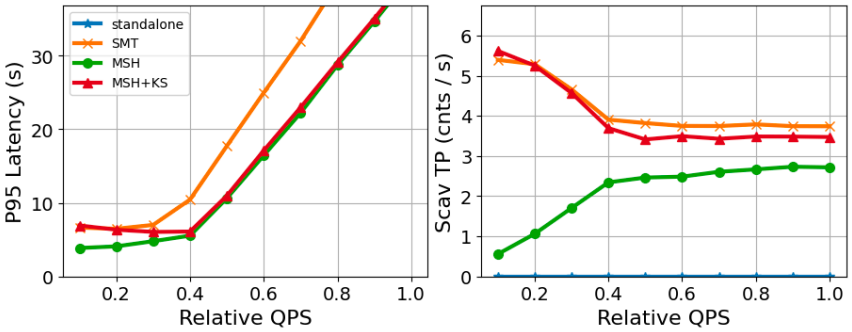

Figure 1.9: SMT, MSH and MSH+KS for Sphinx+Scan.

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 §1.6.3.

Lastly, MSH can fully harvest memory stalls even when scavengers frequently stall. Specifically, for the PtrChase scavenger, with both Sphinx and PtrChase pri-

maries, 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 1.8. 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.

1.6.3 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 1.9). 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 1.7, 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

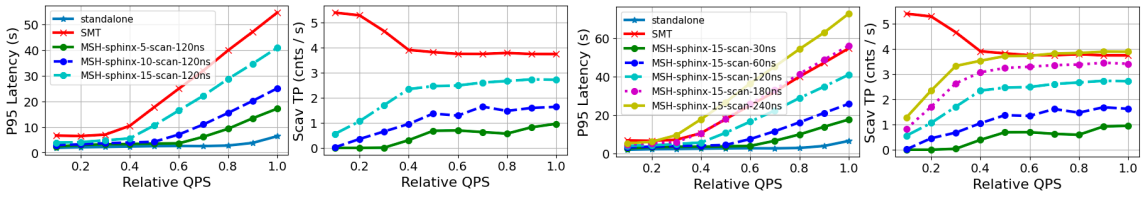

Figure 1.10: The effects of the aggregate yield overhead bound (left) and the scavenger inter-yield distance (right) on the primary latency and the scavenger throughput in Sphinx+Scan.

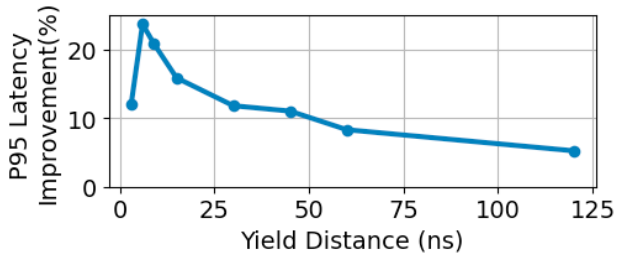

Figure 1.11: Latency improvement made by the yield cost optimizations in the primary instrumentation on Sphinx+Scan.

disabled.

1.6.4 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 1.8, 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 1.10. 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 (§1.4.1). 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 over-

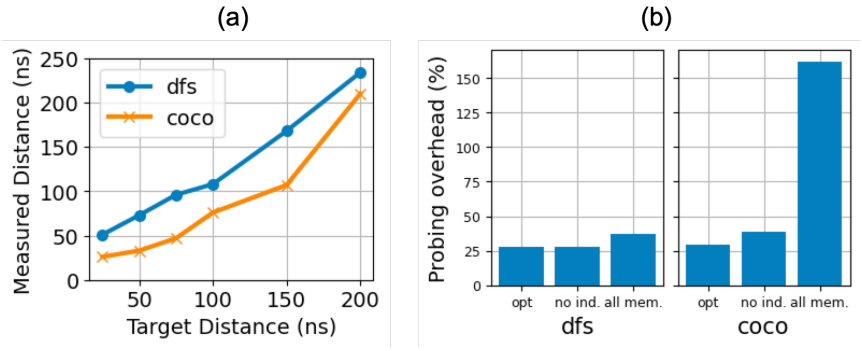

Figure 1.12: (a) Inter-yield distance of scavenger instrumentation; (b) overhead of loop instrumentation: *opt* uses induction registers and unused registers, *no ind.* uses only unused registers, and *all-mem* uses in-memory iteration counters.

head. 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 1.11, 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 1.12-(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 1.12-(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 (§1.4.1), 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.

1.7 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 [199, 20, 143, 18, 38, 153, 28, 66], where the cache line is prefetched after the end of its load slice; and the other technique is criticality-aware instruction scheduling [50, 21, 270, 22], 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 [143, 262] and prefetch insertion via static analysis [20, 59] have limited capability (*e.g.,* unable to handle complex access patterns); whereas advanced proposals like runahead prefetchers [90, 129] 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 (§1.6.2). An exception is IBM Power processors [216, 178], 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, 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 [169, 170] 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 [107, 335, 207, 241]. 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 [248, 157, 131, 61]. 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.

1.8 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) [309, 290, 268], 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 [127, 305, 135], Rust [46, 184, 222, 340]) 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 [337] 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 §1.6, 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 §1.7, 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 [151]. 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 [160].

Efficacy of profiling: In terms of profiling overhead, we have shown that MSH can capture delinquent load instructions with a low sampling rate (§1.6.4). 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 [152, 151, 343], we show positive results with our evaluated workloads (§1.6.2). 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 (§1.4.1). 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 (§1.6.4). 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 [32]. Prior works also propose hardware support for fast saving and restoration of process state during context switches [282, 142].

1.9 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.

Chapter 2

Object-level Tiered Memory Management

2.1 Introduction

As application memory demands continue to increase, there is increasing interest in overcoming the physical and economic limitations on server memory capacity. This typically involves supporting different tiers of memory, ranging from fast local memory to various forms of memory slower to access. These longstanding efforts have recently been given a boost by the advent of Compute Express Link (CXL) technology, which offers significantly higher performance than previous memory expansion methods; its access latencies are only 2-4 times [191] higher than those of local DRAM.

There is already an extensive body of research on tiered memory systems (see, for example, [180, 256, 307, 330, 92, 210, 185, 345]), which we discuss in §2.7. Although their detailed designs vary, these tiered-memory management systems typically leverage NUMA page migration and apportion pages to fast or slow memory based on page access patterns measured using techniques such as page table scans or hardware-assisted sampling. Placing "hot" pages in local memory increases the chance that a high fraction of memory references can be handled by local memory, resulting in less slowdown.

However, such techniques face the fundamental problem that the data on a single page may have very different levels of access rates, something we call intrapage hotness skew, so even an optimal placement of pages into fast and slow memory cannot achieve the desired level of performance (§2.2.1). This problem of intrapage hotness skew is exacerbated by the fact that memory-intensive applications rely extensively on hugepages for reducing TLB pressure [130, 231].

To address this fundamental problem, we must manage memory at a sub-page granularity. Some recent systems manage tiered-memory at cacheline granularity, with the help of a new hardware module that essentially treats local memory as an

L4 cache [345, 139, 181]. As we explain in §2.7, this approach requires a large amount of SRAM to maintain the necessary cache metadata, which in turn adds excessive hardware cost such as die area and power. The current hardware prototype thus makes a compromise and only supports a fixed 1:1 tiering ratio with a direct-mapped cache, which however limits the applicability and practicality of these solutions.

Given the drawbacks of current software and hardware approaches, we ask the question: *Can we design a software tiered-memory system that efficiently manages data placement at a sub-page level?* We argue that we can, if we focus on systems that already provide sophisticated *object*-level memory management, such as in a JVM.

Designing an object-level tiered-memory management system presents three unique challenges (§2.2.2). First, tracking object usage with high accuracy and low overhead is more challenging than tracking pages because there are more of them. For instance, while PEBS [148] is a state-of-the-art tool for tracking page-level hotness, accurately tracking object-level hotness with it requires extremely high sampling rates, resulting in excessive CPU overhead and cache pollution. Second, unable to leverage the OS's efficient page migration mechanisms, object-level systems require their own efficient object relocation mechanisms. Lastly, while most objects are small, some large objects span multiple pages (such as large arrays) and migrating them at the object level is actually coarser-grained than page-level migration.

To solve these problems, we present Fava, a JVM-based object-level tiered-memory management system. Fava tracks object hotness using a novel low-overhead hotness tracking mechanism, then periodically relocates objects to resolve intrapage skewness. The use of JVM for object-level management limits the applicability of Fava, but JVM-based languages such as Java and Scala are widely used for various system software and applications today [338, 297]. To the best of our knowledge, Fava is the first work to significantly reduce application slowdown in tiered-memory through object-level management in software.

As a preview of our design, to overcome the first challenge of scalably tracking object hotness, Fava introduces *profiling-guided object hotness tracking*. Fava identifies the load instructions responsible for the majority of L3 cache misses and instruments only those instructions with hotness tracking logic that increments a counter in the object header. This approach enables Fava to track object hotness with high accuracy and low overhead. Moreover, Fava performs this profiling and instrumentation process online, based on PEBS and Just-in-time compilation, eliminating the need for separate profiling runs or offline instrumentation.

To handle the challenges of object relocation and large objects, Fava adopts a hybrid approach that combines *object colocation* with *page migration*. In particular,

Fava uses the JVM's object relocation mechanisms to pack hot objects on pages, while delegating data migration across tiers to the underlying page-based system. This arrangement enables Fava to efficiently relocate objects to resolve intrapage skewness, while migrating large objects like arrays at the page granularity. Also, to best utilize local memory, Fava periodically scans the object graph and dynamically determines a cutoff for hot objects according to the tiering ratio and the overall object hotness distribution. This allows Fava to achieve a near-optimal utilization of the available local memory.

We implement Fava on top of OpenJDK 21 and evaluate it using three memory-intensive Java applications: a key-value cache, a graph algorithm, and a tree-based index, with a mixture of workloads including the large-scale production trace. Our evaluation demonstrates that Fava reduces application slowdowns by up to 52–83% compared to a state-of-the-art page-level system, Memtis [180], across workloads of varying scales. Furthermore, we show that these improvements result from the accurate and efficient object management enabled by Fava's key design ideas, and that Fava effectively adapts to dynamic workloads using the production trace.

In summary, the contributions of this paper are: (i) a discussion of the need and challenges for object-level tiered memory management; (ii) a JVM-based approach that effectively manages tiered memory placement at the object level; (iii) the detailed design and implementation of a system (Fava) based on this approach; (iv) an evaluation with real applications showing that Fava substantially improves the local memory efficiency and reduces application slowdowns compared to the state-of-the-art page-based system.

2.2 Background and Motivation

In this section, we investigate the problem of space waste in virtual memory-based systems and underscore the challenges for effective object-level tiered memory management.

2.2.1 Space Inefficiency in Page-level Systems

Tiered memory systems aim to manage a new memory hierarchy consisting of fast, locally attached memory and slower, out-of-socket memory connected via interconnects such as CXL. We will refer to the former as *local memory* and the latter as *slow memory* throughout the paper. A main factor driving the design of these systems is the tight performance budget; CXL-connected memory modules have an access latency of approximately 200–400ns [191]—only 2-4x slower than locally attached DDR5 memory. Consequently, even in the worst-case scenario where all data resides in slow memory, the application slowdown can be less than 50% for many applica-

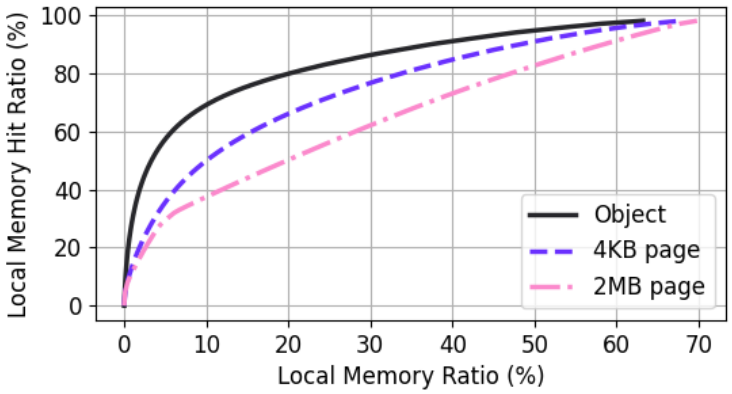

Figure 2.1: Ideal local memory hit ratio of L3 cache misses achievable when object, 4KB page, and 2MB page are used as a migration unit. The workload is a key-value cache whose access pattern follows Zipifian distribution.

tions [185, 345, 285]. This implies that the managing mechanisms need to be highly efficient, as their overhead could easily negate the benefits of better data placement. For this reason, existing tiered memory systems are built on top of virtual memory management [180, 256, 307, 330, 92, 210] in operating systems as it provides an efficient data relocation mechanism without requiring program changes through page table and hardware such as the MMU and TLB [134].

However, since common page sizes (4KB and 2MB) in modern architectures/OS are much larger than most program objects, without careful object placement, each page typically contains many objects with different levels of hotness, leading to a fundamental inefficiency in how the limited amount of local memory is used; we call this *space inefficiency*.

To demonstrate that this space waste makes page-level migration suboptimal under realistic scenarios, we evaluate the quality of *ideal* placement decisions when migrating at the units of 4KB pages, 2MB pages, and objects for different tiering ratios. The workload here is a key-value cache with Zipfian 0.99 access pattern (see §2.6.1 for the detailed setup). We measure the local memory hit ratio (*i.e.,* ratio of L3 cache misses that are served in local memory) under the ideal placement where local memory is populated with the hottest 4KB, 2MB, or objects. For the objects case, we place objects in the order of their access frequency, so that the hit ratios for ideal object placements can be estimated by looking at the hottest pages. We can then make the following observations:

Object vs. 4KB Figure 2.1 shows that ideal object placements achieve 19.6% and

15.4% higher local memory hit ratios than 4KB pages at local memory ratios of 10% and 20%, respectively. This difference stems from the inefficient packing inherent to page-level migration: a single 4KB page holds a dozen key-value pairs with varying memory access frequencies in this example. Thus, without careful object placement, pages placed in local memory inevitably include cold objects that result in lower hit ratios.

Object vs. 2MB Memory-intensive applications often employ 2MB pages, as they provide 10-15% performance improvements by reducing TLB pressure [130, 231]. The use of huge pages, however, exacerbates the space waste. Each 2MB page contains about 7000 key-value pairs, making most pages similarly warm according to their aggregate hotness. As a result, the hit ratio in the 2MB curve increases almost linearly after local memory ratio = 5%, much lower than the others.

Takeaways This experiment demonstrates two key performance opportunities offered by better object placement:

- **Efficient Packing**: By more efficiently packing hot objects into local memory, one could significantly improve hit ratios compared to page-based approaches.
- **Better Hugepage Usage**: Without careful object placement, the performance benefits (*i.e.,* reducing TLB pressure and faster translation) of using 2MB pages comes at the cost of even worse space efficiency. One can resolve this dilemma by operating with huge pages that are densely packed with hot objects.

2.2.2 Challenges of Object-level Management

While it is clear that object-level management can significantly improve space efficiency, it turns out to be highly challenging to design such a system. To see this, we next elaborate on three challenges in object-level management.

Hotness Tracking Mechanism The hotness tracking mechanisms used by virtual memory-based systems can be categorized into three types: page fault, page table scan, and hardware-assisted sampling. The first two mechanisms are inherently limited to tracking page access patterns, so we focus on hardware-assisted sampling, such as Intel PEBS [148] and AMD IBS [87] , which is also used by state-of-the-art page-level tiered memory systems [180, 256].

These sampling methods track events at cacheline level and are extensible to sub-page granularity in theory. However, mapping raw sample addresses to objects is challenging, as samples contain only memory addresses. More importantly, these methods suffer from *low coverage* and *high overhead* when tracking object-level hotness. As noted in prior works [330, 181], the combination of sampling and a large

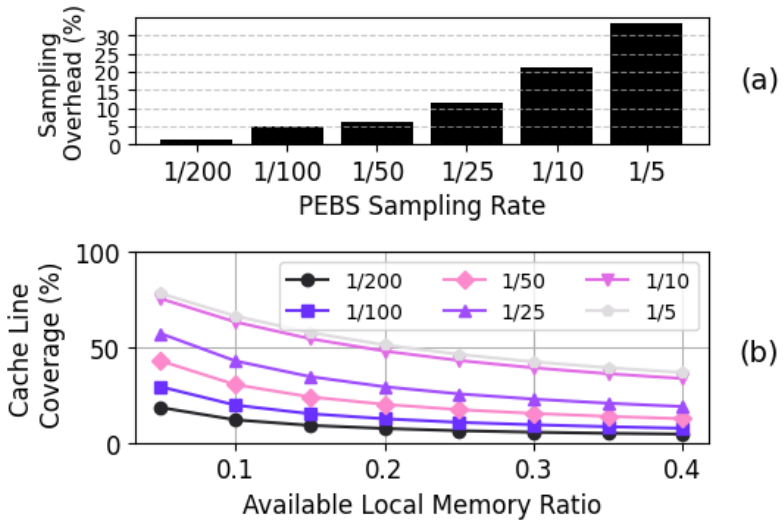

Figure 2.2: (a) The impact of PEBS sampling rates on end-to-end run time. (b) Cacheline coverage of PEBS. We count how many cachelines in local memory appear in the PEBS samples in 1 min run and compute the coverage.

object count demands extremely high sampling rates to achieve even moderate coverage, incurring unacceptable overhead. To demonstrate this, we use the object-level migration setup from §2.2.1, where objects are ordered by access frequency. Ideally, a hotness tracker should identify all objects within the first N bytes as hot, where N is the local memory size. We collect L3 cache miss samples using PEBS and measure how much of the first N bytes PEBS observes (at cacheline granularity) across various Ns. As shown in Figure 2.2, PEBS requires very high sampling rates—incurring over 20% runtime overhead—to capture more than 60% of cachelines in hot data at a 10% local memory ratio. Worse, coverage drops as local memory increases; at lower sampling rates, PEBS fails to reach even 40% coverage with 10% local memory.

While their target is not tiered memory management, prior works on object reorganization [332, 227] also propose a object usage tracking method of maintaining a single hotness bit for each object, which is reset every GC and set upon its first access thereafter. While this approach can cover many objects with reasonable overhead, it suffers from *low accuracy*. This is because (i) it does not provide information about the hotness of accessed objects and (ii) it includes accesses handled by the cache, which are irrelevant in a tiered memory setup. Consequently, as we will demonstrate in §2.6.4, this approach fails to enable systems to make accurate object placement decisions for varying local memory sizes.

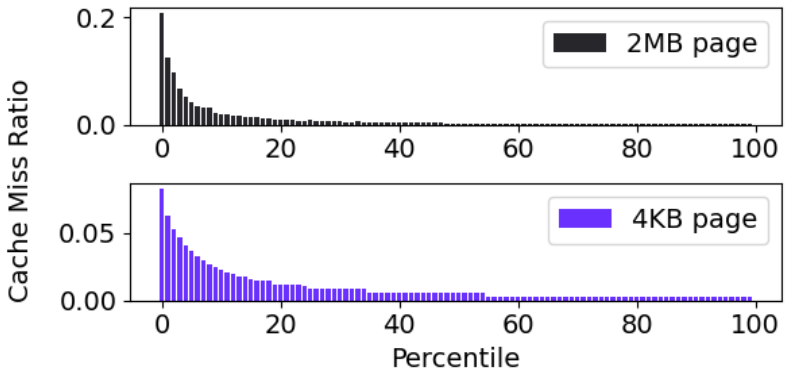

Figure 2.3: Page-level skewness in a large array

Relocation Mechanism Object-level management cannot leverage the efficient page relocation mechanisms provided by the OS, and therefore requires a dedicated object relocation mechanism. One approach is to eagerly update all references, *i.e.* pointers, to an object when relocated. However, this is not only costly—due to the need to identify all such references—but also difficult to implement correctly due to potential race conditions. An alternative approach that avoids this complexity is to introduce an indirection layer: all objects are accessed through a handle, and the runtime system updates the handle-to-object mapping after relocation. However, this method incurs additional memory accesses in object accesses, adding substantial runtime overhead in a tiered memory setup. We elaborate on this issue in § 2.2.3.

Large Objects While most objects are smaller than page sizes, large objects that span multiple pages still exist—arrays being a typical example. In such cases, migrating these objects entirely to either local or slow memory is actually much coarser grained compared to page-level management, leading to inefficient use of local memory. To illustrate this point, we measure the skewness in a large binary search tree implemented using an array. We assume that accesses to tree nodes follow a Zipfian distribution skewed toward larger keys and measure the cache miss counts in each 4KB/2MB page containing the array over a 1-minute run. As shown in Figure 2.3, pages with lower-level or larger-key nodes are more frequently accessed, and thus placing this array entirely in local or slow memory is clearly a bad choice.

2.2.3 Object-level Management: CXL vs. Far Memory

Object-level management has been studied in RDMA-based far memory systems [261, 291, 125]. However, their designs do not address the aforementioned challenges of object-level management for CXL-based tiered memory. In particular, indirection,

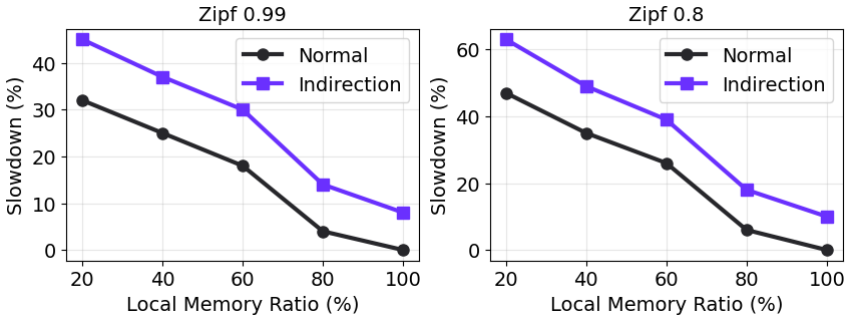

Figure 2.4: The overhead of indirection in Cache application. We measure the slowdown relative to an all-local case under various CXL tiering configurations

a necessary and critical element of their design that naturally supports both object hotness tracking and relocation, is unsuitable in our context.

Since remote data in far memory cannot be directly accessed by unmodified programs, object-level far memory systems use indirection to catch remote access exceptions. For example, AIFM [261] enforces access through a *RemoteablePointer*, embedding the runtime logic that checks object location and performs copy if needed within its dereference operation. This indirection layer not only simplifies relocation but also streamlines hotness tracking, as the tracking logic can be naturally added to the indirection path.

However, indirection is unnecessary in CXL-based systems and can lead to inaccurate hotness tracking. CXL allows data in any memory tier to be accessed directly via regular memory instructions, eliminating the need for indirection. Furthermore, while indirection lookups are designed to catch far memory accesses, they are not directly tied to actual memory accesses, i.e. L3 cache misses. As a result, indirection lookup can no longer serve as a reliable proxy for object hotness in the context of tiered memory.

More importantly, the overhead of extra memory accesses introduced by indirection is relatively small in far memory systems—where remote access latency dominates performance (20–30× slower than local memory)—but can have a significant impact in CXL-based systems. To quantify this cost, we implement a C++ port of Cache application, modifying it so that keys, values, and metadata objects are accessed through an indirection layer. We then compare its performance with the case without indirection, under ideal object placement. As shown in Figure 2.4, the indirection variant exhibits 8-13%p and 10-15%p higher slowdown than the normal version under Zipf 0.99 and Zipf 0.8 workloads, respectively—consistent with

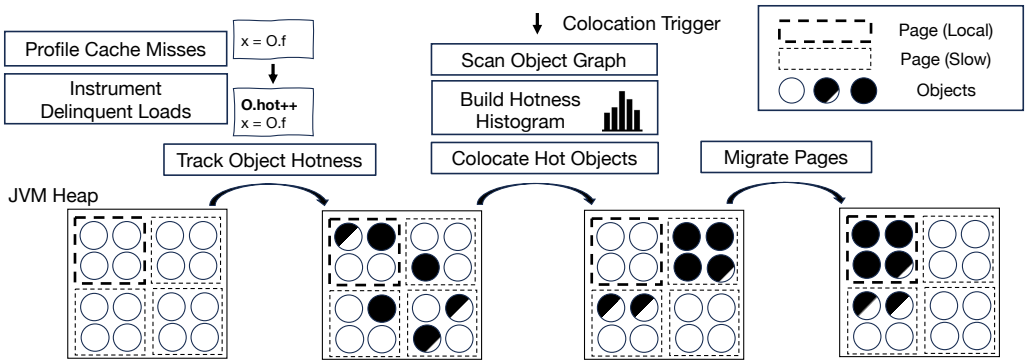

Figure 2.5: Overall workflow of Fava. The diagram shows how object placement in JVM heap is changed as Fava operates. The shade in objects represents the relative hotness of them.

prior findings in non-tiered memory case [316]. Notably, its performance at a 60% local memory ratio is similar to that at 20%, showing that this level of overhead is impactful in CXL-based systems.

2.3 Fava Overview

Fava is a Java Virtual Machine(JVM)-based object-level tiered memory management system that achieves near-ideal local memory hit ratio by addressing the space inefficiency problem. Fava accurately tracks the object accesses related to tiered memory usage and relocates the objects in memory to maximize the hot bytes in fast local memory. Fava's design focuses on addressing the key challenges outlined in §2.2.2.

Hotness Tracking at Object-level The core of Fava is its *profiling-guided object hotness tracking*, which is based on two key observations. First, when a program accesses an object, its accesses can be cheaply recorded in the unused bits of its header. This is because recording access requires only a few simple instructions and, more importantly, the object header is most likely already loaded in the L1 cache during object access. However, this observation alone is insufficient to achieve our goal. Monitoring all accesses can still be costly even with very simple instrumentation, as shown by existing frameworks [198, 223]. Furthermore, most object accesses are served in the CPU cache, making them irrelevant to tiered-memory architectures. The second observation is that object accesses relevant to tiered memory usage—namely, load instructions that cause L3 cache misses—can be identified with low overhead using modern hardware-assisted sampling techniques such as PEBS, as demonstrated by prior works in profiling-guided optimization [204, 153].

Building on these observations, Fava first identifies delinquent load instructions

responsible for the majority of L3 cache misses and monitors only the accesses triggered by these instructions. This approach is *efficient* due to its use of object headers and instruction filtering. It achieves *high accuracy and coverage* simultaneously by focusing exclusively on accesses relevant to tiered-memory usage, while monitoring all accesses made by delinquent instructions.

Fava implements the hotness tracking logic by instrumenting each delinquent load with a snippet of assembly code that increments the hotness counter. For this, Fava performs dynamic instrumentation during JVM's Just-in-time (JIT) compilation process. Specifically, Fava maps the address of delinquent instructions to their corresponding Java bytecode locations and recompiles Java methods containing these instructions with its counter increment logic through JIT compilation. Fava addresses several challenges in this recompilation process (§2.4.1,2.4.2), successfully automating the instrumentation. Note that both the instruction profiling and instrumentation in Fava are performed online without requiring separate profiling runs.

To support these hotness counters, Fava repurposes unused bits in the header of each JVM object. Compared with the conventional approach of maintaining hotness metadata in a separate memory space, this "distributed" counter design offers significant benefits in both space overhead and tracking efficiency, by avoiding additional cache misses, as described earlier. Fava demonstrates that it is possible to *capture many real-world access patterns*, such as hotspots or Zipfian distributions, using this counter.

Migration at Object-level To support efficient object relocation without relying on indirection, Fava *colocates hot objects in batch*. For this, Fava periodically scans the object graph to read the hotness of all objects and relocate hot objects into contiguous memory—similar to how the JVM relocates live objects during garbage collection. This design amortizes the cost of updating object references, as all relocated references can be updated in a single heap scan, and enables colocation decision based on a global view of object hotness derived from the scan (see the last paragraph).

To address the challenges of managing large objects, Fava adopts a hybrid approach that combines object-level colocation with page-level migration. Consider a large array as an example: managing such objects purely at the object level would require partitioning them into sub-arrays and tracking the hotness of each part separately. This requires accurately inferring array indices during instrumentation and can incur unnecessary overhead, *i.e.* accessing some parts of an array may not trigger cache misses. Instead, Fava incorporates page migration: the JVM handles *colocating* hot objects into physically contiguous regions, while the actual movement of pages between local and slower memory is delegated to the underlying page-based system.

This hybrid strategy allows Fava to handle skew in large objects efficiently via page migration while eliminating space waste by packing small hot objects into the same pages.

Lastly, to ensure that colocated pages are populated with the hottest objects, Fava dynamically selects a hotness cutoff based on the distribution of object hotness. A naive approach that colocates all objects with non-zero counters would dilute page hotness density, especially when their total size exceeds the available local memory. To avoid this, Fava builds a hotness histogram by scanning the object graph during each batched colocation pass and then determines a cutoff dynamically to colocate only sufficiently hot objects. This approach enables Fava 's hybrid system to maintain a near-optimal local memory hit ratio across a range of local memory sizes.

Overall Workflow Fava operates in the following logical steps, as illustrated in Figure 2.5. First, Fava's online profiler (§2.4.1) collects cache miss samples using PEBS with a low sampling rate, and identifies delinquent instructions to instrument based on the samples. Second, the profiler maps the address of the delinquent instruction to bytecode within a Java function. It then leaves a special marker with the bytecode and requests its re-compilation to the JIT compiler via JVM's deoptimization framework (§2.4.2). The compiler instruments the code for the marked bytecode by inserting logic to increment a hotness counter (`O.hot++` in the figure). This instrumented assembly updates counters in each object, recording the hotness distribution in the headers (§2.4.2).

Once the counters are populated over a sufficient period, the object colocation procedure is triggered by a background thread. The procedure begins by scanning the object graph, similar to how GC operates, and reading the hotness counters stored in each object. Next, it builds a hotness histogram and determines the hotness cutoff. The colocation thread then performs colocation, relocating objects with the counter exceeding the cutoff to the relocation target regions (§2.4.3). Finally, the underlying page migration-based system detects that OS pages in the target regions have become hot and thus migrates them to local memory.

2.4 Design

Fava consists of three main components: online profiling (§2.4.1), hotness tracking (§2.4.2) and object colocation (§2.4.3).

2.4.1 Online Profiling

The goal of the online profiler is to identify the instructions that are the source of the majority of L3 cache misses during runtime. For this purpose, the profiler should accurately identify all delinquent instructions with low overhead.

To meet these requirements, Fava's profiler uses hardware-assisted sampling like PEBS to collect L3 cache miss samples, which contain instruction pointer (IP) of the triggering instructions. The key insight here is that, unlike events like L1/L2 misses, L3 cache misses often stem from a small number of instructions. As a result, our profiler can thus capture these delinquent instructions with both high accuracy and low overhead, by setting PEBS at a relatively low sampling rate. To embed our profiling logic in JVM, during the JVM initialization process, Fava sets up PEBS events and creates a ring buffer for each core to record L3 miss samples. Then, it spawns a buffer-monitoring thread that regularly wakes up and consumes the PEBS samples from the buffers. The thread reads the IP field of the samples and records the count of appearance of each instruction in a hashmap.

One approach to determine delinquent instructions using PEBS samples is to count the occurrences of each instruction in the samples from the beginning and identify delinquent loads based on the ratios between these counts. However, this approach does not account for the recency of accesses and is vulnerable to short, bursty access patterns. An alternative approach that considers recency is to set a sampling window and make decisions based on the samples within a single window. However, this method can cause decisions to oscillate depending on the samples in each window, leading to unnecessary re-instrumentation requests.

To maintain a stable deliqneunt instruction list while considering recency, Fava's profiler determines delinquent instructions based on the exponential moving average of sample counts over windows. Specifically, the monitoring thread counts the samples for each instruction and records them in a local instruction table, which maps IP to the occurrences. Once observing a predefined number of samples, it ends the current sampling window, and merges the counts in the local table into a global hash table. During the merging process, the original counts in the global table are halved, implementing the exponential moving average. Finally, the profiler computes the ratio of each instruction's count to the total counts in the global table, selecting those with a threshold of $N\%$ as delinquent instructions.

2.4.2 Hotness Tracking

The goal of this part is to instrument instructions in the delinquent load list with hotness tracking logic.

2.4.2.1 Instrumenting Delinquent Instructions

JIT-based Instrumentation Unlike native applications, Java applications generate machine code at runtime via the interpreter or Just-in-Time (JIT) compiler, making it impossible to directly instrument their binary files. Therefore, we employ

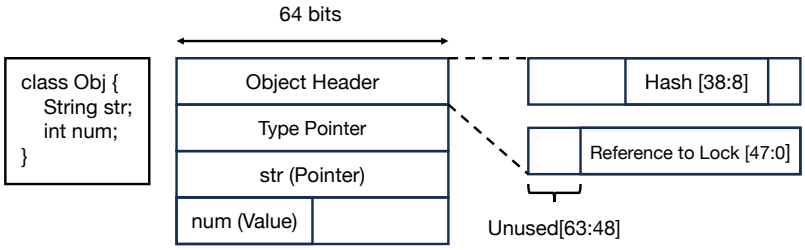

Figure 2.6: Java object layout in 64 bit system. The upper 16 bits of the header are unused.

Bytecode	**Access Type**	**Example Code**
getfield	Fields	`x = O.f`
checkcast	Type pointer	`String x = O.get(k)`
isinstanceof	Type pointer	`if (x == null) {...}`
invokevirtual	Type pointer	`O.put(k,v)`
arraylength	A fixed field	`if (arr.length == 5) {...}`

Table 2.1: The list of Java bytecodes profiled by Fava.

JIT compiler-based instrumentation. Fava maps the IP of delinquent instructions to Java bytecode, marks these bytecodes as targets, and modifies the compiler to insert hotness tracking logic when translating those target bytecodes.

However, this approach requires careful handling of a key factor: the function's call context. Whether an instruction is delinquent often depends on its position in the call stack, so indiscriminately marking bytecodes in frequently used methods as instrumentation targets can introduce unnecessary overhead. This issue is exacerbated by the fact that Java programs frequently use many short utility functions in diverse contexts. To address this, when Fava runs into delinquent instructions, it records the corresponding caller contexts, namely the pairs of bytecode index and caller in the call stack, along with the marker. When the JIT compiler encounters the special marker during parsing, it checks the current callers in the call stack against the recorded pairs and performs instrumentation only if all matches are confirmed.

Fava implements JIT-based instrumentation by leveraging the reverse mapping from compiled code to Java bytecodes maintained in the JVM for debugging purposes. Since this mapping provides both the call context and the basic PC-to-bytecode mapping, Fava reuses this information extensively.

Hotness Tracking Logic The efficiency of hotness tracking logic is vital in Fava's design. Fava achieves this goal via the use of header and the minimal tracking logic

Listing 2 Fava's hotness tracking logic in x86 assembly. It reads the counter field in the header, increments, and writes it back. If the counter reaches the limit, the header update is skipped. `movzwq`, `movw` represent 16-bit `mov` operation.

```
1  inc_counter(Register scr, Address header_addr) {
2      movzwq %scr,0x6(header_addr) // scr = obj->counter
3      cmp %scr,0xffff
4      je equal // if (scr == 2^16-1), skip
5      inc %scr // scr++
6      movw 0x6(obj),%scr // obj->counter = scr
7  equal:
8      ... // delinquent load instruction
9  }
```

implementation.

To record the accesses by delinquent loads, Fava uses 16 unused bits in the object header, as shown in Figure 2.6. This approach avoids the extra space required for separate counters by reusing the additional space already reserved by the JVM. Maintaining separate counters can consume hundreds of megabytes due to the scale of objects. In addition, read-and-modify counters in a separate space likely causes additional cache misses, incurring execessive tracking overhead when used with very high frequency.

Listing 2 shows Fava's hotness tracking logic. We highlight two aspects here. First, this implementation is highly efficient because it spans only 5 lines of assembly code, and since the branch direction changes only when the counter exactly reaches its limit, the control flow overhead is completely mitigated by the branch predictor. Second, this logic assumes that a scratch register and the address of the object header are provided. This requirement can always be met in the JIT compiler. For the scratch register, the register used to store the result of the load can be freely used at the time of hotness tracking. For the object header, we observe that the memory operand of load instructions is offset by a fixed number of bytes from the object header, depending on the bytecode (Table 2.1). For example, in the `getfield` bytecode, the memory operand of the load instruction takes the form `[REG + offset]`, where `REG` contains the address of the object header, and `offset` specifies the displacement of the target field relative to the header. Fava modifies the JIT compilation of each bytecode based on similar characteristics, ensuring that the address of the object header is correctly provided to the hotness tracking logic.

Listing 3 Pseudocode showing the difference of sampling (above) and periodic activation (below). `T` represents a thread local storage.

```
// Sampling: application thread
if (T.count++ == N) {
    ... // hotness tracking logic
}
... // delinquent load instruction
```

```
// Periodic activation: application thread
if (T.count == N) {
    ... // hotness tracking logic
}
... // delinquent load instruction

// Periodic activation: background thread
while (true) {
    T.count++;
    sleep(1ms);
}
```

2.4.2.2 Refining Hotness Tracking

Now that we have shown how Fava supports profiling-guided hotness tracking, we next present several complementary mechanisms that allow the tracking to efficiently and correctly work across applications. First, although lightweight, our hotness tracking mechanism can still incur noticeable overhead when fully turned on. Fava thus provides a knob to balance the overhead-accuracy trade-off, by controlling how often the tracking logic is enabled. Second, Fava efficiently resolves the conflicts between the hotness tracking mechanism and Java's locking primitives that rely on the object header to detect locking contention. Lastly, as the hotness tracking logic only increments the counter until reaching the limit, Fava periodically refreshes the object hotness counters to reflect recency of accesses and handle the saturated counters. We next present these mechanisms in detail.

Periodic Activation One conventional way to provide a knob for the overhead-accuracy trade-off is to employ uniform sampling, tracking only one event out of N, where N is sampling rate, by adding a conditional check and sampling counter. However, using sampling in our hotness tracking method significantly degrades both

overhead and accuracy for two reasons. First, the conditional check for sampling is entirely unpredictable by the branch predictor, adding significant misprediction overhead. Second, sampling loses the ability to track objects with locality. For example, objects associated with a single key-value pair in a hashmap are typically allocated contiguously and used together. Tracking their usage through uniform sampling can produce inconsistent hotness values for these objects, potentially leading to relocation decisions that disrupt their locality.

To address these issues, Fava employs periodic activation of the hotness tracking logic as the overhead-accuracy knob. Listing 3 highlights the difference between sampling and periodic activation. Periodic activation also adds a conditional check before invoking the hotness tracking logic, similar to sampling. However, the condition is updated periodically by a background thread, rather than by the application thread, and much less frequently (e.g., every 1 ms). This approach enables the branch predictor to handle the condition check correctly in nearly all cases and allows the hotness tracking logic to continue tracking objects with the locality.

Handling Locking Primitives The JVM includes internal locking primitives to support object-level locking in Java applications. When locking an object, a thread creates a locking primitive object and stores a reference to it in the object header. This installation uses a compare-and-swap (CAS) operation to handle contention during the locking process. The CAS operation is expected to fail only when there are contending threads. However, our hotness tracking logic continuously modifies the object header, which can lead to false CAS operation failures.

Modifying the hotness tracking logic to acquire a lock or use expensive atomic operations to resolve this issue would significantly harm its efficiency. To address this, Fava adopts an optimistic retry approach, assuming that false failures are rare. This approach keeps the hotness tracking logic intact and introduces a check-and-retry mechanism after each CAS operation for locking. When a CAS operation fails, the thread checks whether the lower 48 bits of the object header remain unchanged before and after the CAS attempt. If no changes are detected, the failure is classified as false, and the thread retries the CAS operation.

Refreshing Counters To minimize overhead, our hotness tracking logic is intentionally designed to be simple and only incrementing the counter monotonically. As a result, it fails to capture the recency of accesses and quickly react to hotness shifts. To address this issue, Fava periodically scans the object graph and decays all the counter values. The decaying period and rate are both configurable parameters. In addition, Fava triggers the refreshing if substantial counter saturations are observed, to ensure the accuracy of the hotness tracking.

2.4.3 Hot Object Colocation

Using the hotness information of objects, Fava identifies hot objects and colocates them to maximize local memory utilization as hot pages get migrated. We next elaborate on the hot object and region selection policies of Fava.

Hot Object Selection To maximally utilize local memory via colocation, one has to carefully select the set of hot objects. If hot objects are selected too aggressively, *e.g.,* all objects with non-zero hotness counters are considered hot, this will dilute the hotness density of the pages. Conversely, selecting objects too conservatively will leave some hot objects uncolocated and thus also result in suboptimal utilization. Intuitively, the ideal policy should select just enough hot objects to populate the available local memory.

To achieve this, Fava computes a histogram of the hotness distribution and determines the cutoff for hot objects based on the histogram and the available local memory size. Specifically, to compute the histogram, Fava scans the entire object graph, reads the hotness counter of each object, and calculates the histogram bin index based on the counter value. If the computed bin index is i, the object's size in bytes is added to bin i. Once the scan is complete, Fava computes the cumulative sum of bytes in each histogram bin, starting from the bins containing the hottest objects and proceeding in descending order of hotness. If the cumulative sum exceeds the local memory size at bin i, then $i + 1$ is selected as the hotness cutoff, meaning that objects in bins $i + 1$ and higher are classified as hot in the current colocation cycle.

Fava uses exponential bins, where the i-th bin contains objects with counters in the range [2i, 2i+1), to make fine-grained cutoff decisions with a small number of bins. In contrast, if uniform bins are used, one has to maintain many more bins, to avoid inaccurate cutoff decisions caused by most objects falling into the lower bins.

Region Selection Once the set of hot objects are selected, Fava aims to colocate them so that the underlying page migration system can effectively migrate these objects in the unit of page to local memory. To achieve this, Fava adopts region-based colocation, a strategy commonly used in modern GC [230, 333]. This approach partitions the heap into fixed-size regions, selects regions based on a policy, and relocates target objects, hot objects in our case, from the selected regions to empty regions. For our purpose, the goal is to design a region selection policy that effectively improves local memory utilization with reasonable relocation overhead.

The key observation here is that selecting regions with either too many or too few hot objects is undesirable. For a region that is already hot, the room for improvement in terms of local memory utilization is too little to justify the high overhead of relocating the large number of hot objects within it. Meanwhile, regardless of

the number of hot objects within a region, one has to pay the constant overhead of region scanning, which makes selecting a region that contains too few hot objects not cost-effective.

Base on this observation, Fava adopts a simple yet effective region selection policy based on two configurable watermarks. With this policy, only regions with bytes of hot objects in between these two watermarks are selected for object relocations. This allows us to strike a balance between improvement of local memory utilization and the associated overhead. In addition, this policy can be efficiently implemented, as the number of hot bytes in a region can be recorded during the object graph scanning.

Page Migration After object colocation, the corresponding pages will be identified as hot and get promoted to local memory by the underlying page-based system. Fava is designed to be only loosely coupled with the virtual memory-based system and require no explicit signaling or interaction with it, which in turn allows greater interoperability.

2.5 Implementation

We implemented Fava in OpenJDK 21, focusing on modifying the JVM's C2 compiler [235], as it generates the most optimized code, and integrating the object colocation logic into generational ZGC [333, 161]. The online profiler is implemetned as a native background thread within the JVM. By default, we used a PEBS sampling rate of 1/2000 and set threshold to 1% to capture delinquent loads and did not observe any noticeable overhead from the sampling.

To instrument hotness tracking, we modified the C2 compiler in two ways. First, we used the `ScopeDesc` data structure, which maintains bytecode and inlining context for debugging and deoptimization, to map the IP of delinquent loads back to their original bytecodes. Second, we altered the bytecode parsing and code generation phases to insert our instrumentation logic before delinquent loads. Specifically, when the C2 compiler parses a bytecode associated with a delinquent load, it generates a special `Load` IR node, treated identically to a regular `Load` node throughout compilation. During the final code generation phase, we modified the compiler to inject our hotness tracking assembly before the load instruction when encountering the special `Load` node. This approach enabled effective instrumentation without interfering with existing C2 compiler optimizations.

Lastly, we implemented object colocation as a new full GC pass in generational ZGC. ZGC is a region-based garbage collector that partitions the heap into 2MB regions, and its full GC involves three phases: marking, region selection, and relocation. During the marking phase, GC threads traverse the object graph, which we

modified to also check each object's hotness counter and update per-region hotness histograms. At the end of the marking phase, we aggregate the per-region histograms into a global histogram and determine the object hotness cutoff from it. In the region selection phase, we use the per-region histograms and the cutoff to compute the number of hot bytes in each region, then select target regions according to our region selection policy. Finally, during the relocation phase, if an object's hotness counter exceeds the cutoff, we relocate it into a region reserved for colocation. To maintain a consistent view of object hotness, we temporarily disable hotness tracking, by adjusting the periodic activation, during the relocation phase.

2.6 Evaluation

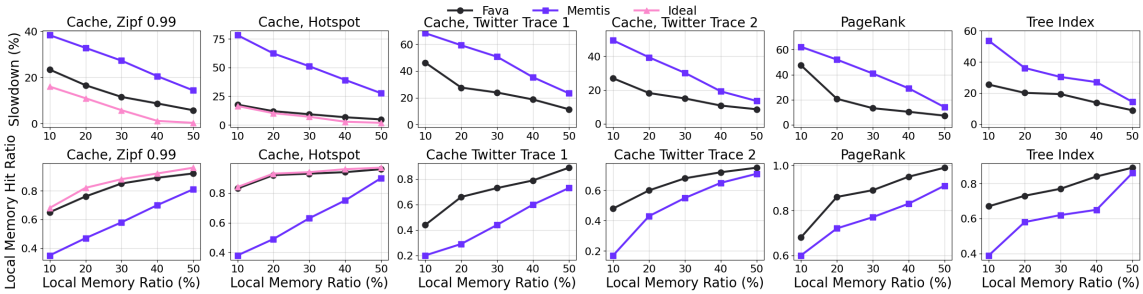

Figure 2.7: Slowdown over all-local case (lower is better) and local memory hit ratio (higher is better) at different local memory ratios in the workloads. The ideal scenario performance is also shown for Cache with synthetic workloads.

In this section, we present our evaluation setup (§2.6.1) and investigate the following key questions regarding Fava: (i) How does Fava improve workload performance under tiered memory by mitigating space waste?(§2.6.2) (ii) How does the design of Fava's hotness tracking contribute to its performance? (§2.6.3) (iii) How does Fava's hot object selection policy impact its performance? (§2.6.4) (iv) How does the hybrid approach enable Fava to handle large objects? (§2.6.6).

2.6.1 Evaluation Setup

Workloads We evaluate three memory-intensive Java applications using both synthetic and realistic workloads. Table 2.2 describes the detailed setup.

- *Cache* is an in-memory key-value store based on Ehcache [93]. We test it with two synthetic workloads(Zipfian with 0.99 skew, Hotspot with 10% of keys receiving 90% of accesses, commonly used in prior works on KV cache [81, 47, 71]) and two Twitter production traces [334, 300]. While synthetic

Application	Workloads	Heap Size
Cache	Synthetic (Zipf, Hotspot), Value size: 256B	16GB
	Twitter Production Trace 1[300], Avg. value size: 44104B	180GB
	Twitter Production Trace 2[300], Avg. value size: 3025B	120GB
PageRank	LiveJournal Graph[182]	16GB
Tree Index	Zipfian, Record size: 128B	16GB

Table 2.2: Specification of evaluated workloads.

workloads have static hotness distribution and fixed key size, the production traces have variable value sizes and a dynamic hotness distribution that changes over time.

- *PageRank* uses JGraphT [155] to compute PageRank on the LiveJournal graph [183, 182]. We repeat 50 iterations.
- To test scenarios where objects are mixed with large arrays exhibiting skewness, we evaluate *TreeIndex* that mixes large arrays and objects: records are stored in arrays indexed by a red-black tree, with a Zipfian prefix introducing intra-array skewness [113]. We set each key prefix to contain 64 records.

These workloads have different heap sizes, and we vary the tiering ratios in our experiments. This allows comprehensive evaluations of Fava's effectiveness under different tiering configurations seen in *e.g.,* multi-tenant scenarios [185, 345].

Systems We use Memtis [180, 212] as both a baseline for comparison and the underlying page migration system for Fava. Memtis is a state-of-the-art virtual memory-based tiered memory system that uses the exponential moving average of PEBS samples of each page as a hotness metric and split hugepages when hugepage-level skewness is detected. For our evaluation, we enable Memtis' dynamic hugepage split feature and tune its parameters to achieve best performance for the workloads. In the experiments, we compare two systems: *Memtis*, where Memtis is used alone, and *Fava* where Fava's JVM performs hotness tracking and colocation while Memtis works as the underlying page migration system.

Methodology and Testbed As of writing, there are no commercially available CXL memory modules, so we use a dual-socket system to emulate a CXL-based tiered memory architecture, following the methodology in prior works [180, 25]. We conduct our experiments in a c6420 machine on CloudLab [91], equipped with Intel Xeon Gold 6142 CPUs with 192GB DDR4 main memory in each socket. We enable transparent

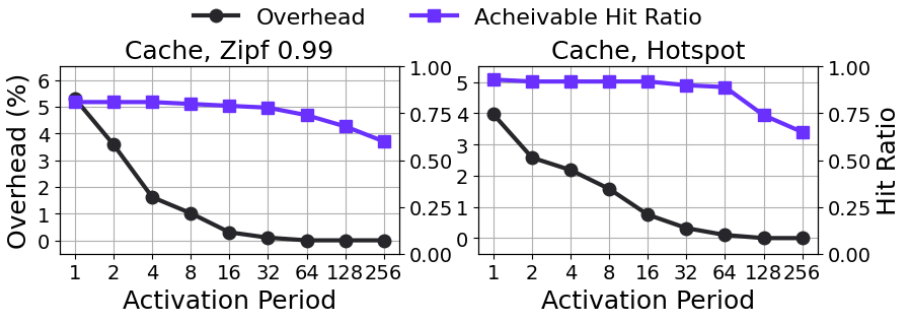

Figure 2.8: Overhead of hotness tracking logic and the ideally achievable hit ratio with different activation periods. Activation period N means that it's enabled for 1ms out of Nms.

hugepages (THP) in all experiments, as doing so consistently improves performance. A modified kernel from Memtis is used to control memory usage between the nodes.

2.6.2 Fava Performance

Summary: Compared to Memtis, Fava reduces the application slowdowns in tiered memory by up to 82%, 54%, 67% and 53% for the Cache with Hotspot and with Twitter trace, PageRank, and TreeIndex workloads, respectively, due to the 3.96x, 2.82x, 1.19x and 1.72x higher local memory hit ratios.

We evaluate application performance under local memory ratios between 10% and 50%, which represent the intended operating range of Fava. We report the application slowdown compared to an "all-local" scenario—i.e., when the application runs entirely on the local memory of NUMA node 0. For performance metric, we use average latency under a fixed load for Cache and TreeIndex and the iteration time for PageRank. In addition, to verify that performance improvements arise from better utilization of local memory, we measure the local memory hit ratio, using PEBS events on L3 misses to the local and remote NUMA nodes.

Cache The Cache workload consists of a large reference array for the hashmap and many small objects associated with key-value pairs including cache metadata. Since the hashmap array is accessed very frequently, Memtis effectively identifies pages containing the array and places them in local memory. However, pages containing small objects appear similarly hot to Memtis, primarily due to the use of 2MB huge pages. Additionally, because most 4KB pages within each huge page are utilized, Memtis rarely performs dynamic huge page splits in this workload.

The first two columns of Figure 2.7 show the slowdown over the all-local case and the local memory hit ratio of Memtis and Fava for the Cache workload with Zipfian

and Hotspot distributions, respectively. Also, we include the graph for "ideal", used in §2.2.1, for Cache, representing the performance and the hit ratio when key-value pairs are inserted in the order of their hotness. As shown in the local memory hit ratio graphs, Memtis's ratio increases rapidly at a local memory ratio of only 10%, when the hashmap array is included. Beyond that point, the hit ratio grows slowly.

In contrast, the graphs show that Fava effectively identifies and packs hot key-value pair objects in the available local memory space, reducing the slowdown compared to Memtis by 39-65% for the Zipfian distribution. The improvement is even greater in the Hotspot distribution, where Fava captures all hot objects and achieves a hit ratio over 90% with a memory ratio of 10%, reducing slowdowns by 77-83%. The third and forth columns show the performance of Fava and Memtis with the Twititer production trace. The trend is similar; Fava reduces the slowdown by 32-53%, 36-54% for the trace 1 and 2, respectively, demonstrating its effectiveness in large and dynamic workloads.

PageRank Most of the memory in PageRank is used by three types of objects: Node objects that store keys and ranks, adjacency lists for each Node, and a hash table that maps Nodes to their adjacency lists. The array of the hash table and the set of Node objects with high degrees account for the majority of cache misses. The third column of Figure 2.7 shows the slowdown and the hit ratio of the two systems for PageRank. In this workload, 10% of local memory is sufficient to accommodate the pages of the hash table array and the hottest Node objects, even with intra-page skewness. As a result, the performance and hit ratio gap between Memtis and Fava is smaller at local memory ratio 10% compared to the Cache workload. However, Memtis wastes some space in accommodating the hottest objects, leaving less space for other similarly hot objects, while Fava uses the minimum space to accommodate hottest nodes. This leads to a constant gap in the hit ratios between Memtis and Fava, starting from 20% of local memory ratio. Consequently, Fava reduces the application slowdown by 23-67%.

TreeIndex The hottest objects in this workload are the hot nodes of the red-black tree. As shown in the last column of Figure 2.7, Memtis achieves a 39% local memory hit ratio at 10% local memory ratio, even though this workload lacks the very hot large array seen in previous ones. This is due to the tree nodes being partially clustered in the search tree structure; for instance, low-level nodes in the tree are mostly hot. However, Fava identifies and colocates hot objects even at higher levels, reducing the slowdown by 36-53%.

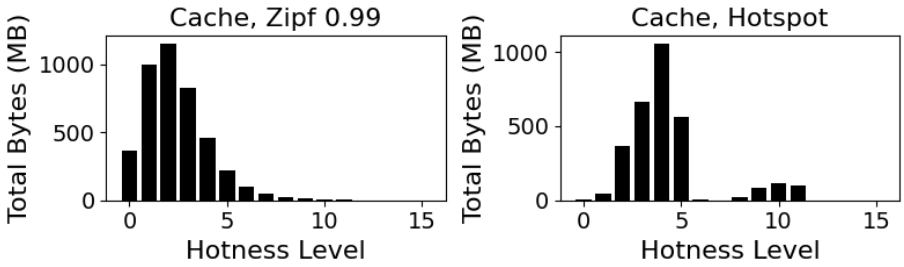

Figure 2.9: Hotness distribution stored in the counters. Bar at hotness level i represents the total bytes of objects whose hotness counter is in $[2^i, 2^i + 1)$.

2.6.3 Hotness Tracking Logic

Summary: Fava's hotness tracking logic incurs only about 4–5.5% overhead when it is always turned on*, and the use of periodic activation effectively eliminates this overhead without hurting tracking accuracy.*

We measure the accuracy and overhead of Fava's hotness tracking logic using the Cache workload under different activation settings. We use synthetic workloads to isolate the instrumentation overhead from dynamic behavior. For accuracy, we measure the hit ratio at 20% local memory. For overhead, we run the Cache at a 100% local memory ratio only with profiling/instrumentation enabled and measure the slowdown relative to the all-local baseline.

The results are shown in Figure 2.8. Here, the activation period N means enabling the hotness tracking logic for 1 ms every N ms. Thus, $N = 1$ corresponds to continuous activation (no periodic deactivation). We highlight three points. First, even without periodic activation, the overhead remains low—only about 5.5, 4% in each distribution—due to the efficient design of the hotness logic. Second, periodic activation effectively eliminates this overhead with virtually no reduction in accuracy in $N = 8, 16, 32$ range, demonstrating the value of this technique. Finally, we note that accuracy only begins to degrade when the activation period becomes very large ($N \geq 64$), suggesting that our method can effectively handle scenarios with higher instrumentation overhead.

2.6.4 Hot Object Selection Policy

Summary: Fava's hot object selection logic reduces slowdowns of naive policy by up to 43%, 79% in Cache with the Zipfian and Hotspot distributions. The effect is more important when available local memory sizes are limited. This is attributed to the hotness counters differentiating hotness of objects.

Hotness Distribution stored in Counters The effectiveness of our selection

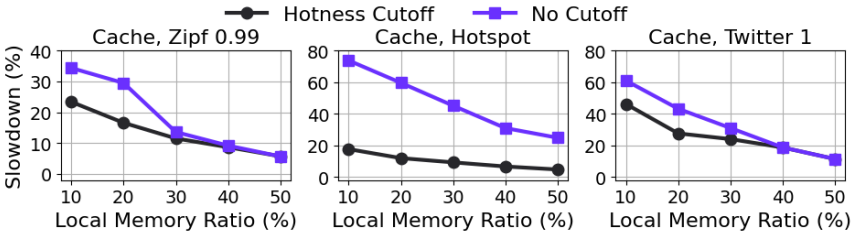

Figure 2.10: The effects of hot object selection policy.

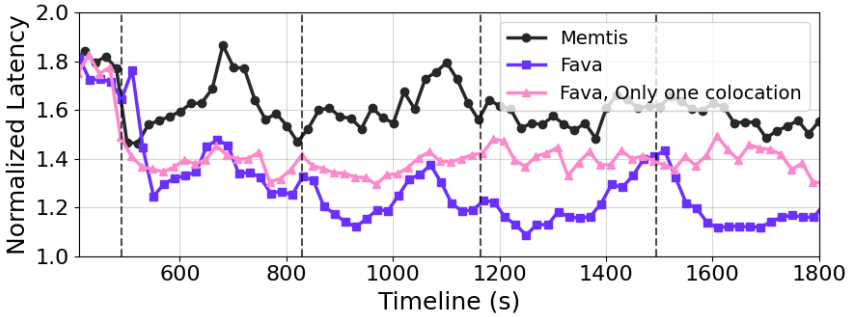

Figure 2.11: Request latency over time (normalized to the average latency of the all-local case; lower is better) in Cache workload with Twitter production Trace 1. Local memory ratio is 20%. Dotted lines indicate the start of colocation.

policy depends on how accurately the hotness counters reflect the access pattern. To show that our hotness tracking logic fulfills this requirement, we examine the hotness histogram in the Cache workload with static distribution. As shown in Figure 2.9, our hotness counters successfully categorize objects into multiple levels of hotness, allowing our policy to gradually include objects for relocation as more local memory becomes available. This trend is clear in the Hotspot case, where the counters for the hottest 10% of objects are significantly higher than for others. This separation enables the hot object selection policy to colocate only the hottest items when local memory is limited.

Impact on Performance To assess the object selection policy's impact on end performance, we evaluate a variant of Fava, where a baseline no-cutoff policy that colocates all objects with non-zero counters is adopted. Note that this baseline approach is equivalent to using single-bit hotness counters for the accessed objects. As shown in Figure 2.10, the policy does impact Fava's overall performance. For the Hotspot distribution, the naive policy mixes warm objects with hot ones in

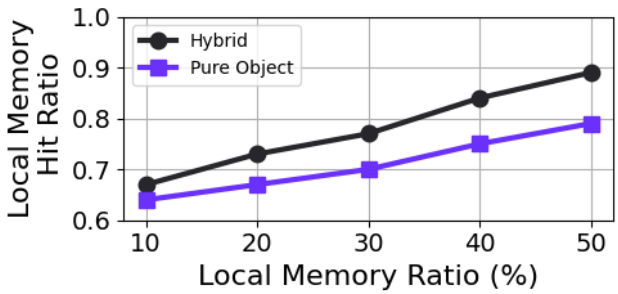

Figure 2.12: The effect of hybrid approach in TreeIndex.

the pages, which leads to worse performance. Specifically, Fava with the dynamic selection policy incurs 78-80% smaller application slowdowns than the naive variant. A similar trend is observed in the Zipfian and Twitter trace 1 at lower local memory ratios (10% and 20%). The policy makes little difference when there is enough (30% and above) local memory to accommodate all objects in histogram.

2.6.5 Dynamic Behavior

Summary: Fava maintains superior performance over time by adapting to changes in hotness distribution.

We evaluate Fava 's ability to adapt to hotness changes using Cache workloads with Twitter production trace 1. To demonstrate this, we compare the performance of Fava against Memtis and a variant of Fava where colocation occurs only once at the beginning. Figure 2.11 shows the performance of the three systems over time. Both Fava and the one-time colocation variant initially outperform Memtis following the first colocation. However, the one-time variant fails to colocate objects that appear later, resulting in worse performance than Fava after the second colocation point. In contrast, Fava continues to perform colocations periodically and sustains lower latency over time, demonstrating its adaptability.

2.6.6 Hybrid Approach

Summary: By handling large objects via page migration, Fava's hybrid approach achieves 10% higher local memory hit ratio than even the optimal object placement in TreeIndex.

To assess the benefit of the hybrid approach, we compare the local memory hit ratio of Fava with that of the optimal object placement, which is computed based on the cache miss samples. As shown in Figure 2.12, Fava achieves a hit ratio up to 10% higher than the optimal object placement. The gap is smaller at lower local memory ratios, as the local memory is predominantly populated with hot tree node

objects, but as more local memory is available, Fava's ability to only migrate hot pages of large arrays in local memory leads to a bigger advantage over the optimal object placement, where the entire arrays have to reside in local memory, leading to substantial space waste.

2.7 Related Work

Software Tiered-memory Systems To leverage the efficiency of page table indirection, existing software tiered memory management systems are built on top of the operating system's virtual memory [180, 256, 307, 330, 92, 210, 185, 345, 331, 167, 209] , particularly NUMA migration. Although each system employs different methods to track page hotness, they all face a fundamental challenge: space inefficiency at the page level, which is the primary focus of our work.

Memtis [180] identifies space inefficiencies in hugepage usage and proposes a dynamic hugepage split mechanism. However, to avoid the heavy overhead of the split, Memtis performs it only under very clear conditions and remains limited to 2MB or 4KB units. In contrast, Fava colocates hot objects in contiguous physical space, resolving the space inefficiency issue. Colloid [307] demonstrates that bandwidth contention is another factor to tiered memory performance. In this work, we focus on scenarios with low bandwidth contention to isolate and address the space inefficiency problem.

Hardware Tiered-memory Systems To enable efficient fine-grained tiered memory management, hardware-based approaches have been proposed that treat main memory as an L4 cache for slow memory modules and migrate data at cache line granularity [181, 345, 139]. However, these solutions suffer from fundamental limitations associated with having to maintain a large amount of metadata. In particular, maintaining a cache necessitates metadata to handle replacement policies and locate cache lines, and for hardware cache, this information is typically stored in expensive SRAM to ensure fast lookups. Since main memory is orders of magnitude larger than conventional CPU caches, scaling such an approach would either demand excessive hardware resources or push metadata into DRAM, thereby increasing lookup overheads [252]. solutions. In contrast, Fava can deliver high performance at different local memory ratios without requiring specialized hardware.

Far Memory Prior work on far-memory systems [261, 313, 312, 24, 123, 49, 125, 291] shares a similar goal with tiered memory systems: minimizing the overhead of slow-tier accesses. However, because remote accesses introduce much longer network latencies compared to CXL memory, far-memory research mainly focuses on hiding latency through techniques like prefetching or software stack optimization. Object-level management efforts in far memory were discussed in §2.2.3.

Object Reorganization The idea of observing object usage and relocating objects has also been used for other purposes. For instance, HCSGC [332] enhances CPU cache efficiency by relocating objects based on their access order. Polar [227] marks any object touched since the last GC cycle, and then proactively relocates those not touched to far memory. These works rely on a simple one-bit usage counter—sufficient for their respective goals—, but our analysis in § 2.6.4 shows that this approach fails to capture complex access distributions, unsuitable for tiered memory management.

2.8 Conclusion

We present Fava, a system that manages data placement in tiered memory at the object level. By removing the space waste caused by intra-page skewness, Fava achieves near-optimal local memory utilization. Fava achieves this through accurate, lightweight object hotness tracking and hot object colocation. As a result, Fava significantly reduces application slowdown compared to page-based tiered memory systems.

Chapter 3

Democratizing Cellular Access

3.1 Introduction

Cellular networks play an increasingly important role in the Internet ecosystem: they serve over 5B subscribers, source over 50% of web traffic, and are expected to see dramatic growth due to new applications enabled by 5G, IoT, and edge computing [269, 64]. Given their central role, it is vital that the cellular market be open to innovation and competition. Unfortunately, this is not the case today as the cellular market is dominated by a small number of providers; *e.g.,* 3 carriers account for over 98% of US subscribers [280] and there is mounting concern that the monopolistic nature of this market will negatively impact innovation, pricing, security and, ultimately, the user's experience. [214, 213, 106, 239, 302, 164, 112, 104]

In this paper, we explore an alternate cellular architecture that allows a potentially large number of competing cellular providers to coexist. We start with the observation that, to lower the barrier to entry, we must ensure that providers of *any* scale – small to large – can compete equally within the cellular ecosystem.

We use "scale" to refer to the geographic area that a provider covers. E.g., a small-scale provider might offer coverage over a modest geographic area spanning just one or a few cell towers (*e.g.,* in a campus, mall, city downtown or rural area) while a large-scale provider might offer nation-wide coverage (as today's leading providers do). By "compete equally", we mean that a user should have no reason to discriminate between providers based on the geographic scope of their infrastructure. Instead, we'd like to enable a user to consume cellular service from *any provider that is available at that place and time* with no concern for whether that provider offers coverage in *other* locations. Doing so levels the playing field for all providers: small or large, new or incumbent.

This ideal scenario described is very different from current practice. Today, a user's choice of provider is influenced by the provider's coverage area (*e.g.,* [243]), in addition to price and other factors. Thus, the viability and success of a provider

depends on its deployment scale. Building a cellular network is slow and capital intensive; hence expecting new entrants to roll out a large-scale network before they can enter the market significantly raises their barrier to entry. While independent smaller-scale mobile operators do exist, they are often relegated to a secondary role: they largely serve niche markets, rely on roaming agreements with nation-scale mobile networks, or only provide private local networks.

As we'll discuss in §3.2, the current bias towards large-scale providers is not just an accident of history; rather, it is deeply ingrained in the design choices of the current cellular architecture. To reverse this, we propose a new cellular architecture, CellBricks, that is explicitly designed to accommodate providers of any scale. To achieve this, our architecture departs from current cellular designs in two important aspects. First, it removes the traditional requirement that users have a trusted relationship with the cellular network they are attached to, and instead enables users to consume (and pay for) service on-demand from any infrastructure operator. CellBricks achieves this by moving certain user management functions (*e.g.,* accounting, authentication) out of the cellular infrastructure and refactoring them between the user and an external "broker" service (§3.3). Secondly, it moves support for mobility from the network to the user device so that a user can experience seamless mobility even if she frequently switches between (potentially smaller-scale) providers, and so that she can do so without relying on complex network support and inter-provider roaming agreements.

Although CellBricks was originally motivated by the goal of enabling competition, we find that our design offers two additional benefits: simplification and efficient capacity scaling. By removing in-network support for user management and mobility, the cellular core in CellBricks is significantly simpler than in the current (notoriously complex) cellular architecture (§3.3). By allowing users to connect to *any* cellular network, CellBricks allows more efficient use of spectrum and infrastructure. This benefit is particularly valuable as 5G requires much denser deployments of base stations than previous generations, which amplifies existing coverage issues.

In summary, the benefits of CellBricks are threefold: (i) lowering the barrier to entry for new providers, (ii) simplifying cellular core infrastructure, and (iii) enabling more efficient use of cellular spectrum and infrastructure.

We design and implement CellBricks as an extension to open-source cellular platforms (Magma [99], srsLTE [278]). We evaluate CellBricks via a combination of experiments on a small-scale testbed and emulation over existing cellular and wide-area networks. We demonstrate that CellBricks is compatible with existing radios, introduces negligible overhead (between -1.61-3.06%) on application performance, and scales to a large number of users under different radio conditions.

In this paper, we focus on the technical feasibility of a cellular architecture that is more open to new entrants. We recognize that our proposal also gives rise to questions around incentives, spectrum, *etc.* We discuss these briefly in §3.3 but leave an in-depth exploration of such issues to future work.

Roadmap. In §3.2, we elaborate on why the current cellular architecture fails to accommodate small/mid-scale providers. We present the overall approach, design, implementation, and evaluation of CellBricks in §3.3, §3.4, §3.5, and §3.6 respectively. We discuss related work and conclude in §3.7.

Ethics Statement: This work does not raise any ethical issues.

3.2 Background and Motivation

3.2.1 The Current Cellular Architecture

Today's cellular networks comprise two main operational components: the Radio Access Networks (RAN) and the cellular "core" (called EPC in LTE, or 5GC in 5G). The RAN includes cell towers (called eNodeBs) that communicate over a radio interface with user equipment (UE). The RAN forwards traffic from UEs to the core which then forwards the traffic onward to the Internet.

The RAN defines how data is encoded and transmitted over the air between the cellular tower and a user device. Our architecture does not modify the RAN and hence we do not discuss it further. The cellular core implements a range of functions related to user authentication, mobility management, traffic classification and prioritization, usage accounting, and so forth. These functions are implemented as hardware or software appliances that may be deployed in a provider's Central Office, an edge data center, or (more recently) the cloud [226]. Importantly, the core serves as the *mobility anchor* for UEs: a UE's IP address, for example, is assigned by the core when the UE connects to the network, and this address remains the same as a UE moves between different towers. We do modify the cellular core in CellBricks and hence elaborate on it briefly (see [14] for details).

The cellular core includes: (i) *Control plane* functions that implement standardized signaling protocols for communication with UEs, (ii) *User plane* functions that implement packet forwarding, including classification and prioritization to enforce QoS levels, counters for accounting, *etc.*, and (iii) *Management plane* functions that maintain subscriber information and perform authentication and policies.

When a user connects to a mobile network it first goes through an "attachment" process which involves using standardized signalling protocols to communicate with the cellular core's control plane. This signalling triggers a series of management functions within the core including (i) authenticating the device, (ii) looking up its subscription plan, (iii) configuring the appropriate user plane functions based on

this subscription plan (e.g., configuring rate limits, packet classification rules and priorities), and (iv) creating one or more logical tunnels between the UE and the core to handle traffic. Once this attachment process is complete, the mobile network (radio and core) can process the user's traffic.

This attachment process is not repeated when a user moves from one cell tower to another within the same provider. Instead, the relevant components in the RAN and cellular core will coordinate to ensure that the communication state for that UE – *e.g.,* its tunnel state, QoS rules – are correctly applied to traffic arriving to/from the UE's new tower. This "handover process" is implemented by migrating the tunnels that carry the UE's traffic such that traffic continues to flow through the same elements in the cellular core, including the IP gateway connecting the core to the Internet.

Participants. Traditionally, the two main participants in a cellular network are the user with her UE and the Mobile Network Operator (MNO). The MNO owns and operates cellular infrastructure and also provides user support services such as sales, billing, customer care, marketing, *etc.* The user typically enters into a contractual agreement with one MNO which serves as her default or "home" provider. To provide broad coverage, an MNO may enter into contractual agreements with other MNOs and when a UE "roams" outside the coverage area of its home network, it can consume service from one of these other MNOs, with the roaming UE's traffic typically routed back through its home network.

Mobile *Virtual* Network Operators (MVNOs) are service providers that do not own RAN infrastructure, but instead provide user-facing services (sales, billing, *etc.*) while relying on business agreements with some number of MNOs to provide use of their RAN. Two well-known MVNOs in the US are Google Fi [117] (which uses the T-Mobile and US Cellular networks) and Cricket [75] (which uses the network of its owner, AT&T). In this scenario, the user contracts with an MVNO, and the MVNO in turn contracts with MNOs.

3.2.2 Limits of Today's Cellular Architecture

We argue that the above cellular architecture is fundamentally at odds with empowering smaller scale providers. There are two key reasons for this which we elaborate on below.

(1) Scaling coverage requires pre-established agreements. A user U can only obtain service from an MNO M with which it has a pre-established contractual agreement. This agreement may be direct (*i.e.,* between U and M) or indirect (*i.e.,* U has an agreement with N and N has an agreement with M that authorizes M to serve U). These agreements are how an MNO provides service to its users outside

its own footprint, and how MVNOs establish their coverage. Similar to peering in wide-area routing, these agreements establish *trust* between two entities based on which they cooperate in authenticating and billing users; *e.g.,* via inter-provider roaming protocols. Unfortunately, this approach scales poorly when we have many smaller providers because these inter-provider agreements are manually established and carry high transaction costs.

Today, the overhead of establishing such agreements is acceptable because each MNO has a large deployment and hence one only needs a small number of agreements to ensure broad coverage – *e.g.,* Google partners with two MNOs for its Fi service. But in an environment with many smaller-scale providers, the number of agreements required to ensure broad coverage would quickly become untenable. [1]

(2) Seamless mobility requires coordination between cell towers. A handover is the process of migrating a UE from one tower to another within one provider's network. Today, this involves cooperation between the towers and cellular core to ensure that a UE maintains its IP address and its active sessions are not disrupted. In current networks, because an MNO has a large deployment, handovers are the norm while crossing provider boundaries is rare and hence users mostly enjoy "seamless" mobility.

However, ensuring seamless mobility in a network of many smaller-scale providers is more challenging: in this case, switching towers will more frequently imply switching providers and preserving a UE's IP address when it crosses provider boundaries would be incredibly complex. Hence, simply carrying over today's cellular design to our context would lead to frequent IP address changes, thereby disrupting TCP connections and degrading the user experience.

In summary, the essential properties of a cellular network – seamless mobility and broad coverage – are difficult to achieve if we simply apply today's design to an infrastructure made up of many providers of any scale. This motivates us to revisit existing designs to eliminate the above problems.

3.3 Overview

We propose a new cellular architecture called CellBricks that starts with the MVNO architecture but systematically alters it to avoid the problems discussed in §3.2.2. CellBricks involves three entities: (i) users and their associated UEs, (ii) brokers, and (iii) cellular access providers of any scale, which we refer to as brick-Telcos (bTel-

[1] With ≥300,000 cell towers in the US [70], if all MNOs deployed 100 towers complete coverage would require 3,000 contracts per MNO *vs.* the few today. This is clearly impractical or at least, raises the barrier to entry.

cos).[2] Similar to MNOs, bTelcos own and operate cellular infrastructure (towers, core appliances, etc). Brokers act as intermediaries between users and bTelcos: a user enters into a contractual agreement with a broker and the broker is responsible for representing the user to bTelcos. From a user's perspective, she subscribes to cellular services from her broker and need not be aware of the specific bTelco her device is attached to, which will vary over time.

Up to this point, our architecture might appear identical to that of MVNO services. The key point of departure is that our architecture does *not* require a pre-established agreement between brokers and bTelcos. Thus, bTelcos have no pre-established agreements with users or brokers, and unlike MVNOs, brokers can provide service to their users over any available bTelco infrastructure. To our knowledge, CellBricks is the first architecture that allows both users *and* brokers to dynamically leverage untrusted access providers.

At a high level, we envisage that operation in such a network proceeds as follows.

(1) On-demand authentication and authorization. A UE (denoted U) may request service from a bTelco (T1) when it comes within range of T1. The request identifies the user's broker (B) and T1 forwards the request to B together with parameters describing the terms of service (*e.g.,* QoS and billing options) that T1 can provide. B authenticates both U and T1 (using a protocol outlined in §3.4.1). If B decides to authorize the request, it informs T1 of this and T1 can start providing cellular access to U. As part of this process, B and T1 might also negotiate additional features such as the need for lawful intercept (as defined in [96, 5, 4]).

(2) Billing. Periodically, U and T1 independently send verifiable and tamper-proof *usage reports* to B. These reports might summarize both the bandwidth used and connection quality that U received. At some later time, T1 bills B based on the usage reports. Compensation is realized in the same manner as other online financial transactions, building on standard techniques for online authentication and payments. Note that we mediate the *process* of payments but do not dictate the actual pricing scheme which is left open to innovation.

(3) Mobility. Later, U may come in the range of bTelco T2 and may wish to switch from T1 to T2. To do so, U simply repeats the same authentication and authorization steps with T2 as it did with T1 and then switches to T2. As a result of the switch, U's IP address may change. To handle this situation without requiring coordination between bTelcos, we employ a host-based mobility procedure (§3.4.2) that does not disrupt users' application-level sessions.

[2]Brokers are similar to current MVNOs and bTelcos to MNOs. However, we introduce some fundamental differences in their role and functions and hence introduce new terminology to avoid confusion.

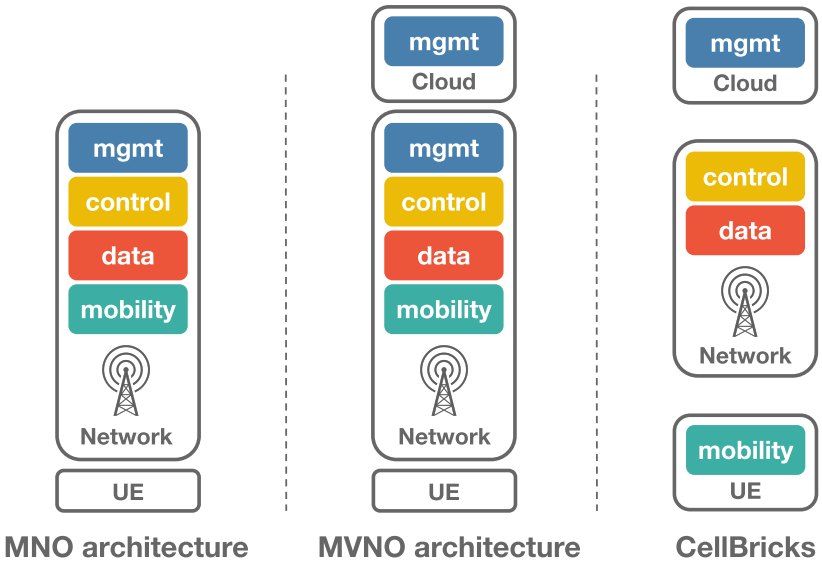

Figure 3.1: The network here refers to both RAN and cellular core infrastructure. The Cloud contains those portions of the cellular service typically run in a datacenter (e.g., subscriber database). The MVNO arch. requires in-network support for management because they still rely on usage accounting and authorization implemented in the core.

(i) No "scale bias": a bTelco can generate revenue by providing service to users within its radio range, irrespective of the scale of its deployment.
(ii) Few pre-established contractual agreements: a bTelco can begin providing service without requiring contractual agreements with users, other bTelcos, or brokers. In our example, neither U nor B have a pre-established relationship with T1. Instead, B and T1 authenticate each other on-demand and (as we'll describe) do so using standard public key cryptography techniques. As we explain later, a bTelco only needs a certified public key and an ability to settle payments; these requirements are standard for online merchants.
(iii) Simplification: CellBricks is a simpler cellular infrastructure to implement and operate. All mobility in CellBricks is host-driven, with bTelcos implementing no particular in-network support for mobility. CellBricks makes no distinction between switching between towers (handovers) or providers (roaming). This eliminates the need for coordination between bTelcos, and even among towers within the same bTelco, while users move about. Further, user authentication is managed by brokers using standard, widely-deployed public key cryptographic techniques. Figure 3.1 summarizes the "division of labor" in each architecture. From an operational per-

spective, by not requiring a pre-defined trust relationship between brokers and bTelcos, CellBricks removes costly integration and testing procedures commonly required today for establishing roaming and network sharing arrangements among operators.
(iv) Infrastructure efficiency: Rising demands on existing cellular infrastructure are driving *network densification*, with nascent 5G networks requiring larger numbers of smaller cell sites to deliver their promised network capacity. Deploying dedicated radio infrastructure for each provider is capital-intensive and inefficient. In contrast, CellBricks facilitates low-friction infrastructure sharing, allowing any number of brokers to take advantage of a bTelco's deployment. More generally, CellBricks gives greater power of choice to users, brokers, *and* bTelcos: users and brokers can use any bTelco while bTelcos can simultaneously serve multiple brokers. Moreover, this choice can be exerted in a fine-grained manner allowing for a range of policies (*e.g.,* selecting bTelcos based on their historical performance).
(v) Seamless integration of private networks: A growing number of *private* cellular networks serve specific populations or use-cases – *e.g.,* enterprise campus or industrial IoT contexts [95, 33] – and there is interest in integrating these private networks with public cellular networks in a controlled manner [294, 122]. E.g., allowing an employee to seamlessly transition from her MNO to the enterprise's private network. This is not easily achieved with today's cellular architecture but is naturally accommodated in CellBricks.

3.3.1 Discussion

Although our primary goal is to evaluate the *technical feasibility* of CellBricks, we briefly address a few questions regarding adoption that a reader may have at this point. That said, there are many open questions regarding the market structure and business incentives surrounding CellBricks that are beyond the scope of this paper.
1) What about spectrum? CellBricks requires no changes to the Radio Access Network (RAN) and bTelcos can use any spectrum available to them. Trends in the spectrum regulatory environment are favorable to new entrants, providing them several options for obtaining spectrum. *E.g.,* in the US, the Citizen's Broadband Radio Service (CBRS) [23] provides 150MHz of spectrum in the 3.5GHz band on a dynamically shared basis, allowing wireless operators to deploy networks without costly exclusive spectrum licenses; many commercial deployments of CBRS-based LTE and 5G mobile networks are already underway [120]. Other countries have adopted regulatory constructs that allow new entrants to operate in licensed, but unused, cellular spectrum [68, 16].

It is also feasible that new providers can simply license spectrum from incumbent providers, where this is mutually beneficial [15]; *e.g.,* where the incumbent has no

existing or planned infrastructure. Hence existing providers can use new entrants in a franchise-like model, leasing the right to operate in the incumbent's spectrum in certain areas. Our proposal provides a technical foundation for businesses to take advantage of these innovative licensing schemes, while remaining compatible with existing licensing frameworks.

For *new bTelcos*, building and operating a cellular network represents an opportunity to participate in a profitable and growing market [258]. CellBricks merely makes this opportunity more accessible to new entrants. Further, because bTelcos are inherently multi-tenant (that is, a single bTelco cell site can support multiple brokers), bTelcos can serve more customers with the same infrastructure, enabling financially profitable operation in a wider range of contexts.

Brokers in CellBricks are equivalent to today's MVNOs or the consumer-facing side of an MNOs, and share similar incentives: the business opportunity of participating in a growing market and the broader benefits of improved user access [246, 99, 286]. As long as demand for cellular service exists, mobile operators will compete to meet that demand. CellBricks simply removes architectural barriers that currently limit this competition. Further, unlike MVNOs today who are at the mercy of their underlying MNOs, CellBricks brokers could easily switch between bTelcos, if necessary, to seek favorable commercial terms.

What about incumbent providers? While it might appear that they have little incentive to embrace our architecture, we speculate that this may not be universally true. Building and operating a radio network is the most capital intensive portion of a mobile network's operation which is exacerbated by 5G's need for dense deployments [45]. With our architecture, existing MNOs can leverage bTelco infrastructure without massive financial investments while still benefiting from their ownership of spectrum (akin to a franchise model). MNOs today already embrace sharing passive infrastructure (*e.g.,* towers) to solve densification and rural expansion [13]; our architecture simply allows them to do so more extensively. Despite these potential benefits, it is still quite likely that incumbent providers would find CellBricks more of a threat to their dominant positions than an opportunity for more efficient infrastructure. Fortunately, CellBricks can be incrementally deployed with no change to, or cooperation from, legacy operators. Specifically, a CellBricks broker could have contractual agreements with some (legacy) MNOs while also leveraging new CellBricks-compatible bTelcos. In this incremental deployment model, MNOs continue to run their legacy protocols and UEs run both legacy and SAP authentication protocols in a dual-stack mode.

Finally, users benefit from bTelcos in the short term through improved coverage and reap the benefits of a more competitive market in the long term.

3) Won't brokers be the new monopoly? We believe this is unlikely to be a concern. First, the barrier to entry for starting a broker is low, requiring no investments in cellular infrastructure or long-term agreements with bTelcos. Instead, the main requirement for a successful broker is the ability to attract users and provide customer support. Many players meet this requirement: content providers, online retailers (*e.g.,* Amazon), traditional retailers (*e.g.,* Costco), credit institutions (*e.g.,* Visa), and non-profit entities such as governments. Second, the broker market is likely to remain competitive because it is easy for users to switch brokers or even sign up with multiple brokers.

3.4 Design of CellBricks

To realize CellBricks we must address three questions: (i) How do we ensure secure attachments in the absence of mutual trust between bTelcos and users/brokers?, (ii) How do we minimize disruption to users' connections when switching between bTelcos? and (iii) How do we ensure secure billing and QoS enforcement in the absence of mutual trust between bTelcos and users/brokers? Next, we describe our solution for each of these. For ease of exposition, we use U to represent the UE, B the broker, and T the bTelco.

3.4.1 Secure Attachments

The foremost challenge is secure attachments, *i.e.,* to ensure secure authentication and authorization in the absence of mutual trust between bTelcos and users/brokers.
Design Rationale. We begin by noting that the process by which a UE attaches to a cellular network can be decomposed into three steps [98]. The first is to establish radio-layer connectivity with the tower, for which we simply reuse existing techniques. The second is authentication which, today, means mutual authentication between a UE and MNO, and is implemented using a shared secret key that is pre-established between the UE (via its SIM card) and the home MNO [9]. The last step to set up the parameters of the service (*e.g.,* QoS settings). For CellBricks, we must revisit the last two steps as we cannot build on the assumption of a trusted relationship between the UE(U)/broker(B) and the bTelco (T). Instead, our requirements for CellBricks are: (i) mutual authentication between U and its B, (ii) mutual authentication between T and B,[3] and (iii) authorization, by which we mean that T must obtain irrefutable proof that B authorized it to service this U; this is required as T need not trust B.

We propose an approach that moves away from shared secrets and instead relies on public-private key cryptography, as is common in online services today, to achieve

[3]Since the U trusts B, it is sufficient that the broker authenticates the T and we do not need additional direct authentication between the U and T.

SAP: UE Procedures

Invoked by the UE upon attaching to a new bTelco.

Arguments:

id_x	identifier of entity x
authVec	authentication vector
authReqU	auth. request UE sends to bTelco
authRespU	auth. response received from bTelco
pk_B	broker's public key
sk_U	UE's secret key
n	random nonce generated at UE

Procedures:

1. **Set** authVec = (id_U, id_B, id_T, n)
2. **Encrypt** authVec with pk_B → **authVec***
3. **Sign** authVec* with sk_U → **$sig_{authvec}$**
4. **Send** authReqU = ($sig_{authvec}$, authVec*, id_B) to bTelco

Upon receiving authRespU:

5. **Verify** $sig_{authrespU}$ with pk_B; do 6. if succeed
6. **Decrypt** authRespU with sk_U; use ss, defined in broker procedures, to configure NAS security context

Figure 3.2: A summary of the steps run at the UE, as part of the secure attachment protocol.

these goals. We assume all entities – Us, Bs, and Ts – have an associated public key and that B and T keys are signed by a Certificate Authority (CA). Under these assumptions, we design a *secure attachment protocol (SAP)* that achieves our security goals using standard public-key authentication techniques. Our SAP protocol is efficient, requiring only a single round-trip from the U to T to B, and back, compared to two round-trips between U and MNO in the current architecture.

SAP protocol. Briefly, the SAP protocol is invoked when U moves to a different T and involves the following procedures and message exchanges (detailed procedures can be found in the code blocks in Fig.3.2 and Fig.3.3):

(1) Message from U to T: U crafts a message requesting service from T. The message contains an *authentication vector*, which includes the identifiers of the T, B, and U itself; plus a nonce. An identifier could be the digest of the owner's public key; or the IMSI [322] (if U), IP address, or domain names (if B and T). The nonce is generated as a random string at U and serves to protect against replay attacks. U encrypts this

message with B's public key, signs it, and sends it to T. Because T never observes a cleartext identifier for U, it cannot act as an "IMSI catcher" [283].
(2) Message from T to B: T augments the request received from U with the service parameters related to QoS (described later). T signs the augmented request, and forwards it to B.
(3) Message from B to T: When B receives a request, it authenticates both U and T and decides whether to approve the request based on U and T's profiles. If approved, B returns a message that contains two signed and encrypted (sub-)responses: (i) *authRespT* that includes identifiers of U and T, a shared secret *ss*, and QoS parameters (the last two are described next); (ii) *authRespU* that includes identifiers of U and T, *ss*, and the U-generated nonce. On receiving B's message, T authenticates B by validating B's signature in *authRespT*; this response serves as the authorization for T to serve U. Then, T replies to U with *authRespU*.
(4) Message from T to U: On receiving *authRespU*, U authenticates B by validating B's signature in *authRespU*; this response helps U confirm that its access to T is now authorized.

As a summary, U is responsible for identifying itself to T and B; T forwards authentication messages between U and B, and B authenticates and authorizes both U and T. Finally, both U and T will use *ss* in the responses to set up their security contexts following the existing security procedures. Note that SAP's security context is identical to that used in EPS which includes keys for protecting AS and NAS messages, as well as NAS counters and identifiers. One could refer to [224] for details. Briefly, the shared secret *ss* is used as the master key (also known as `KASME` [219]) in the security mode control (SMC) [11] procedures to derive keys for ciphering and integrity protection of their AS and NAS messages [10], which we reuse otherwise unmodified from today's standard. After the security context is established, we reuse unmodified session establishment procedures to provide U with access to public networks, during which T assigns an IP address to U.

In addition, SAP is also used to communicate various service parameters such as *e.g.,* QoS settings and whether lawful interception is to be invoked [6, 4]. While today's network implements both the policy and mechanism for these features, CellBricks decouples these, with policy decisions made by B and communicated to T which implements them. This is done by augmenting the authentication protocol to include QoS parameters (*i.e.,* qosCap and qosInfo in Fig.3.3, we omit the other possible policy parameters for brevity) and other service parameters, and the set of parameters can also be dynamically updated. Specifically, we have T inform B what QoS options (qosCap) it can enforce and that B can then send specific parameter values (qosInfo). Doing so requires a standardized approach to expressing these pa-

rameters, *e.g.,* for QoS, we propose to adopt the existing 3GPP definitions of QoS parameters [6]. We will describe how a broker ensures that bTelcos correctly enforce the QoS in §3.4.3.

SAP is designed such that U only requires *a small set of static parameters* for attachment; specifically, U's key pairs and B's public key. This state can be embedded in the U's SIM card, in exactly the same way that the shared secret used for authentication today is embedded and distributed to users. Note that U's public keys are used only for interactions with B, who issues U's key pair in the first place, hence no certificates are needed for U's public keys. Moreover, B can revoke U's public key by simply invalidating the key in its database. For T and B, we assume their public keys and corresponding certificates are distributed and maintained using standard PKI techniques, akin to existing Internet services.

Lastly, in an effort to understand what security properties SAP can offer and what it cannot, we discuss the security of SAP in the context of several common attacks in our technical report [53].

3.4.2 Seamless Mobility

Since CellBricks enables a potentially large number of smaller-scale bTelcos, switching between towers often implies switching bTelcos: How do we minimize the disruption to the user's connections in the face of these frequent switches?

Design rationale. In today's networks, most mobility events involve *handovers* in which a user switches from one tower to another within the same provider's network and support for such handovers is embedded in the network; *i.e.,* towers coordinate using signaling protocols to ensure that a user's traffic is routed through a consistent gateway in the cellular core. This ensures that the device retains its IP address as it moves between towers and hence that its active TCP connections are not disrupted.

This network-driven coordination is difficult to implement in CellBricks as it requires cooperation and interoperability between bTelcos, adding both operational and technical complexity which in turn raises the barriers to entry for a new bTelco. Instead, we propose a host-driven approach that essentially eliminates the concept of a handover: a user simply detaches from one cell tower and independently attaches to a new tower (run by the same or different bTelco) via the SAP protocol. This approach is simple as it requires no network support or coordination between towers.[4]Similar to today's network-driven UE-assisted handover, where UEs conduct performance measurements to help the network make handover decisions, our UE-

[4]While we do not preclude coordination across towers in a *single* provider, we're intrigued by the possibility of removing this complexity from the network entirely and hence evaluate this extreme design point in §3.6.

driven handover can benefit from network assistance too. For instance, UE-driven handover can perform smarter cell selection based on the list of neighbor cells learned from the network. We believe such UE-driven, network-assisted handover is feasible and promising as demonstrated in today's Wi-Fi roaming [30, 288, 264] and Wi-Fi-cellular handover [289, 326, 234].

Handling IP changes. As a result of our host-driven approach to mobility, U's IP address may change as it switches towers which raises an important question: how do we avoid disrupting U's connections? We observe that the reason the current handover process must retain U's IP address is to avoid breaking U's open transport-layer (typically TCP) connections. But is Layer 3 (L3) the right network layer at which to address this problem? For CellBricks, we argue not. A host's IP address today reflects its location in the Internet's routing topology and the administrative domain to which it belongs; we want to preserve this property. Instead, we want higher layers – specifically, the transport layer – to be capable of adapting to changes in the endpoint's IP address.

Fortunately, new transport protocols such as MPTCP [233] and QUIC [177] already provide such support, though motivated by different use-cases than ours.[5] These protocols have explicit connection identifiers within their L4 header and use IP addresses only for packet delivery. This separation allows the use of multiple underlying IP addresses for the same connection. Simply adopting these protocols, which are already standardized and widely deployed [103, 150], solves our problem with no additional change and no network support.

For instance, MPTCP introduces the notion of a subflow – a flow of TCP segments operating over an individual path. A single MPTCP connection can operate with multiple subflows that can be dynamically added and removed over the lifetime of the connection [233]. Fig.3.4 illustrates how a MPTCP connection with a single active subflow reacts to IP changes in the context of bTelco detachment and attachment. In brief: (i) At the end of the detachment procedure, the baseband processor at U deletes the radio bearer (used at bTelco A) and informs the OS kernel that the IP address of the UE's network interface is no longer valid (as a result, the interface's IP is typically set to `0.0.0.0`); then (ii) the MPTCP stack at the UE is notified about the address invalidation and will watch for a new address until reaching a predefined timeout (default to 60s) while the existing subflow (IP_{1-S}) stays inactive. If the timeout is reached, the MPTCP connection will be torn down. (iii) Once the UE securely attaches to the bTelco B, a new "bearer" (a tunnel connecting the UE to a gateway to the Internet), is created using the UE's new IP address.

[5] MPTCP's original goal was to improve the performance of a single connection by leveraging the multiple paths between a source and destination.

Once the network interface regains a new address, the UE's MPTCP stack uses its new source IP and initiates a three-way handshake to create a new subflow (IP_{2-S}); the UE also informs the server side of the connection to remove the previous subflow (IP_{1-S}) via the REMOVE_ADDR option. Once IP_{2-S} is established, the UE and server can resume exchanging packets over the MPTCP connection.

We believe this *host-based* approach is the right long-term solution: architecturally, it respects Internet design principles and layering, can be deployed with no support from the network, is supported by major operating systems (*e.g.,* Windows, Linux, X, Android, iOS) and is seeing growing deployment [220, 308] including for multi-access in 5G [85]. There are solutions besides MPTCP and QUIC – *e.g.,* HIP [217] and SCTP [281] – that can also handle IP changes. We leave an exploration of these options to future work.

Finally, although host-driven mobility is our preferred approach, it requires support at both endpoints. To support incremental deployment while these protocols are ubiquitously deployed, our strategy (used in our prototype) is simply to fallback to TCP when MPTCP is unavailable and rely on the application and/or L7 protocols (*e.g.,* SIP re-invite [260]; HTTP range headers [147]) to efficiently restart failed connections.

3.4.3 Verifiable Billing and QoS

The last piece of the puzzle is: how do we ensure secure and verifiable billing and QoS enforcement in the absence of mutual trust among the UE, broker, and bTelco?

Design rationale. We focus on ensuring accurate *accounting*, by which we mean the ability to obtain an accurate record of the network resources a UE consumed at a bTelco. Such accounting is the foundation on which billing between the various parties – T-to-B, and B-to-U – can be implemented and we leave the question of pricing open to innovation. This approach matches today's architecture where accounting within the cellular core supports a range of service plans; *e.g.,* based on flat-rate pricing, usage caps, *etc.*

In current networks, accounting is based on measuring traffic statistics in the "packet gateway" of the cellular core (PGW in 4G, UPF in 5G). Even though operators could miscount or over-bill, users generally trust their results because of their reputations as large carriers and their contractual agreements. With untrusted bTelcos, we instead need an accounting protocol that is *tamper-resistant* and *verifiable*.

We assume that bTelcos are not malicious in the sense of wanting to disrupt a user's service but that they could be motivated to lie about resource consumption if it increases their revenue *and* if they believe they can lie without being detected. This latter seems reasonable given the capital costs of setting up a cellular tower: to

see a profit, a bTelco will need to remain operational for some period of time but if it is suspected of cheating then a broker might simply choose not to use the bTelco. The same may be said of UEs and their Brokers. We call this a "dishonest but not malicious" threat model. These assumptions are analogous to many customers and retail businesses - large and small - in the real world.

Verifiable billing and QoS. At a high level, our approach is to have T and U independently measure the traffic volume and QoS for U's session. Then, we have them periodically send encrypted and signed traffic reports that contain those measurements to U's broker B. A traffic report includes the following information: *(i) Session identifier* which uniquely identifies a session between U and T; *(ii) Relative timestamp* within the session, which is used for B to align U's and T's reports; *(iii) Usage metrics* that accounts for traffic volume consumed at the uplink (UL) and downlink (DL) in bytes; *(iv) Duration* for phone call and events such as SMS messages; *(v) QoS metrics*, as defined by the 3GPP standard, including the average bit rates, packet loss, and packet delay etc., reported separately for both the DL and UL [1].

The challenge is that T's traffic reports are untrusted, and T might have an incentive to inflate the usage values. Therefore, U will independently measure its own traffic statistics and periodically sends a traffic report R_u to B. Since U may have an incentive to *deflate* the usage values, we ensure that R_u cannot be tampered with by U and hence B can trust R_u. We discuss this further below.

This simple approach sets up the right incentive structure: dishonest reporting by either Us or Ts will manifest as a discrepancy between their reports and, while small discrepancies are expected and tolerated, a large or persistent discrepancy will be viewed as anomalous. We assume B and T store a history of report summaries and anomalies and hence, over time, build up a *reputation system* based on which either party can decline to cooperate - U/B can switch to using a different T, while T can decline to serve U/B. We elaborate on design considerations for this reputation system in what follows.

Reputation System We focus on addressing the use of a reputation system to enforce correct resource accounting.[6] To do so, B maintains: (i) a per-bTelco aggregate *reputation score* and (ii) a list of its own users that are suspected to have tampered with their device. We expect the latter to be a small number, because implementations at the UE side are embedded into baseband firmware and hard to tamper afterwards, which allows the broker to carefully review the firmware implementation and ensure its correctness. Likewise, T maintains an aggregate reputation score per

[6]We believe that one could extend the reputation system to enforce QoS but leave a design of this to future work.

broker.

Reputation scores can be derived from the UE's and the bTelco's traffic reports in a manner that is left open to innovation. Here we present one design based on simple heuristics but imagine that in practice brokers can implement more sophisticated strategies. Fig.3.5 describes this design where B compares the discrepancy of reported DL usage against a *threshold value* that is calculated based on the UE's reported DL loss rate and a fixed tolerance ratio ϵ. (*e.g.,* derived from the acceptable link loss rate). When the discrepancy is greater than the threshold, B considers this as an anomaly and records this incident (a "mismatch"). B then derives the reputation score based on the number of mismatches, weighted by the degree of mismatch. We leave an exploration of exactly how to do this weighting to future work. Note that for T, it discovers discrepancies with U's reports either indirectly from the broker's final settlement or directly by requesting U to also send it a copy of the traffic report.

Given the reputation scores, B can decide whether to authorize an attachment according to the reputation score of the bTelco as well as whether the user is on the suspect list. Likewise, T can decide not to service any users belonging to a broker with a poor aggregate score. The exact policy that each broker and bTelco will adopt is open to innovation.

In terms of security properties, CellBricks's reputation system is generally vulnerable to the same failure modes as any reputation based systems, and at the same time benefits from existing countermeasures. Interested readers could refer to our technical report for more discussion on this issue [53].

Our final requirement is to ensure that the traffic report from the UE cannot be tampered with by the user (since the user may have an incentive to undercount, just as the bTelco has an incentive to overcount). A tamper-resistant accounting protocol at the UE has two components: a secure measurement function that accurately records a user's traffic statistics, and a protocol that safely communicates these statistics from the UE to the broker.

We propose to embed the measurement function in the UE's baseband, which today implements all cellular functions and is assumed to be tamper-resilient [186].[7] To ensure these measurements cannot be tampered with once extracted from the baseband, we propose to sign and encrypt the measurement report on the baseband.

The above changes can be implemented as just a *firmware* upgrade to the existing UE baseband and introduces little overhead because: (i) usage and QoS metrics

[7]Some tools [251, 187] can read (but not write) the modem's internal state, and some attacks can steal security credentials and sensitive information [329, 145]. However, none of these can overwrite modem statistics.

are already available in today's baseband processor (*e.g.,* PDCP counters for bytes sent/received and RLC metrics for packet loss [7]), (ii) today's baseband processor already implements encryption. Moreover, these operations are only performed once every reporting cycle, which we anticipate will be on the order of many seconds or even minutes.

3.5 Prototype Implementation

We prototype CellBricks using existing open-source cellular platforms, given changing baseband firmware requires certificates from baseband and/or device manufacturers. As described in Fig.3.6, the prototype includes four components: UE(s), the base station (eNodeB), the cellular core, and our broker implementation (brokerd). Our testbed has two x86 machines: one acts as UE and the other as bTelco (eNodeB + EPC). We connect each machine to an SDR device (USRP B205-mini [97]) which provides radio connectivity between the two machines. On each machine, we run an open-source LTE suite (srsLTE [278]) with the UE machine runs the srsUE stack and the eNodeB machine runs the srsENB stack. We extend srsUE with the UE-side changes mentioned in §3.4. We install an MPTCP-enabled Linux kernel (v4.19) on the machines and run apps in docker containers. The containers use the network stack of the host machine, allowing applications to run unmodified because MPTCP is largely backward compatible with the existing socket API.

For the cellular core and broker, we build on Facebook's Magma [99][8], an open-source software platform that serves as an extensible mobile core network solution. The two main components we extend are the access gateway (AGW) and the orchestrator (Orc8r). The AGW implements the core network (EPC), and the Orc8r implements a cloud service that configures and monitors the AGWs. We extend AGW to support our secure attachment protocol: we define new NAS messages [10] and handlers and implement these as extensions to Magma's AGW and srsUE. Finally, we implement the broker service (called *brokerd*) as part of Magma's Orc8r component deployed on AWS. Brokerd maintains a database of subscriber profiles (called SubscriberDB) and implements the secure attachment protocol, processing authentication requests from bTelcos. Our prototype does not include the reputation system. We defer its implementation and evaluation to larger-scale CellBricks deployment in future.

In summary, we introduce two new protocols to the existing 3GPP standard (the SAP and our accounting protocol), as well as a new dependency on end host transport protocols (*e.g.,* MPTCP) to support mobility. We modify only the UE and core network to achieve this, which importantly allows reuse of unmodified commercially

[8]CellBricks's implementations and evaluation results can be found in [52].

available cellular base station equipment. In total, our extensions to Magma includes 2,493 LoC in the AGW (in C) and 263 LoC in Orc8r (in Python); we add 940 LoC (in C) to the srsUE.

3.6 Evaluation

We evaluate CellBricks using a combination of two approaches: (i) benchmarks of our prototype testbed (§3.5), and (ii) emulation over existing cellular and wide-area networks. The former is limited in scale but validates end-to-end correctness and demonstrates compatibility with existing radios, user devices, and base stations. The latter allows us to answer what-if questions regarding application performance under real-world conditions with real applications.

CellBricks raises two main performance questions: (i) how much overhead does our attachment protocol (that includes three parties) introduce compared to the existing cellular protocol (that includes two parties)? and (ii) does CellBricks's host-based approach to mobility impact application performance relative to today's in-network approach? We evaluate these questions in what follows.

3.6.1 Prototype Performance

Methodology: We measure the end-to-end latency due to our attachment protocol, measured from when the UE issues an attachment request to when attachment completes. Note that we do not take into account the potential cell selection time, as the target cell information is usually known prior to attachments. From the E2E latency, we first remove the time spent in the RRC (radio) and lower layers since this value depends largely on the choice of radio hardware and protocol implementation and these components remain unmodified in CellBricks. Moreover, because we use software-based implementations of the RRC and and lower layers, the latency through the radio stack is higher than it would be in a typical hardware-based RRC ($\approx 130ms$) which could mask other system overheads that we introduce. To understand where the remaining time is spent, we instrumented the relevant components of our prototype – Access Gateway (AGW), SubscriberDB, Brokerd, eNodeB and UE – to measure the processing delay at each.

In our experiments, the UE, eNodeB, and AGW are always located in our local testbed and we run experiments with the subscriber database (SubscriberDB) and Brokerd either hosted on Amazon EC2 [26] or our local testbed. Running in the cloud matches current deployment practice in which certain core network components are run in the carrier's datacenter or on public clouds. For each setup, we repeat the same attachment request using both unmodified Magma and Magma with our modifications to implement CellBricks. We repeat each test 100 times and report

average performance.

Results: Fig.3.7 shows the attachment latency (after removing the time spent in the RRC and lower layers) for three different placements of the SubscriberDB and Brokerd for both unmodified Magma (our baseline, denoted BL in the figure) and Magma with our modifications to implement CellBricks (denoted CB). In each case, we also show the breakdown of latency at each module. The portion labeled "Other" is simply the leftover latency once we remove the time spent in each of the above modules from the attachment latency; this leftover time is dominated by the latency between the AGW and the SubscriberDB/Brokerd.

Looking first at the overall latency, we see that in all cases, the attachment time with Magma-CellBricks is comparable to Magma-unmodified. In fact, the attachment latency with Magma-CellBricks is 14.0% smaller than the unmodified Magma when we run the SubscriberDB and Brokerd in the us-west-1 region (31.68ms vs. 36.85ms) and 40.8% smaller when in us-east-1 (98.62ms vs. 166.48ms). This is because the standard S6A attachment procedure [284] in our baseline involves two round-trips between the AGW and the SubscriberDB (the standard involves two requests, an Authentication Information Request and an Update Location Request (ULR) made to the Subscriber DB) whereas in Magma-CellBricks a bTelco does not send the second (ULR) request (see §3.4).

Looking now at the breakdown in latency, we see that in the local setup, the attachment request processing at the AGW and Brokerd accounts for about 70% of the total request latency ($\approx 20ms$), before and after modifying Magma alike. This confirms that our changes to Magma such as adding brokerd and crypto operations introduce negligible performance overhead ($\approx 2ms$) across the modules. When the SubscriberDB and Brokered are in the cloud, the total request latency is dominated by the network latency between the AGW and cloud as the "Other" bars indicate. Our latencies for us-west-1 are lower than us-east-1 simply because the former is geographically closer to our local testbed.

In summary, CellBricks adds little overhead to the attachment process and – by eliminating one round-trip between the AGW and cloud – can even improve attachment latencies compared to existing cellular implementations. However, as mentioned earlier, CellBricks undergoes attachments more frequently than in current cellular networks; we evaluate the impact of these more frequent attachments next.

3.6.2 Emulation over the Internet

With CellBricks's host-driven approach to mobility, switching towers can involve re-authenticating and initiating new transport-layer "subflows," each of which could impact end-to-end performance. This impact depends on multiple factors such as

the frequency of handovers, packet loss, *etc.* To capture these in a realistic manner, we *emulate* CellBricks over the T-Mobile network in our urban region. This allows us to capture real-world conditions such as the density of tower deployment, devices on the move, real-time background traffic, handover patterns, and application behavior. Prior work has shown that MPTCP performs well in the face of soft handovers between Wi-Fi and cellular providers [234, 83, 76, 255, 60]. In this paper, we instead evaluate MPTCP when migrating across cellular access providers (bTelcos) and do so under the extreme scenario in which each provider operates only a single tower. To our knowledge, this is the first comprehensive evaluation of host-driven mobility in wide-area cellular networks.

Overview. In today's cellular infrastructure, a UE typically retains its IP address when it switches towers. Hence, the crux of our approach is that we will *emulate* an IP address change each time a handover occurs. At a high level, our emulation works as follows. First, we use a real UE (running Linux, MPTCP, and applications) that connects to a real cellular network (T-Mobile) and we exploit low-level APIs on the UE's Qualcomm chipset to detect when a handover occurs. Whenever a handover is detected, we emulate an IP address change to the container running our test applications. This involves invalidating the old address and creating a new one. We introduce a delay d between invalidation and when the new address is available - d is a parameter that we can tune to model the overhead of authentication/attachment (as measured in the previous section). This change in IP address will in turn trigger MPTCP to take appropriate action (*e.g.,* creating a new subflow). Finally, we use GRE tunneling to carry packets with the emulated IP address between the client and server. Tunneling is used only for emulating IP changes in today's infrastructure, and will not be needed in a real CellBricks deployment. Throughout the above process, we run an application and measure its performance. We leave the evaluation of CellBricks on iOS/Andoird apps, on protocols other than MPTCP, and on soft (make-before-break) handovers to future work.

Methodology. We now describe this emulation process.

(i) *Equipment.* Our UE consists of a ZTE MF820B LTE USB Modem (with Qualcomm chipset) [347] and a laptop (Ubuntu 18.04 with 4.19 MPTCP-enabled kernel) that runs application client code. The modem uses an unlimited prepaid SIM card and connects to the laptop via a USB port. We run the server side of the applications on Amazon EC2 (region: `us-west-1` with `c5.xlarge` instances). We use two UEs and two EC2 server VMs for each run: one UE-VM pair runs MPTCP while the other runs regular TCP. The TCP pair is our baseline that represents current infrastructure (see iv), while the MPTCP pair is subject to our emulation of CellBricks.

Application		**MTTHO**		**Ping: p50**		**iPerf: Avg. Throughput**		**VoIP: MOS**		**Video: Avg. Quality Level**		**Web: Avg. Load Time**	
Unit		second		millisecond		mbps		1-5/excellent		level (0-5)		second	
Route / Time of Run		D	N	D	N	D	N	D	N	D	N	D	N
Suburb	MNO	73.50	65.60	45.95	46.71	1.25	17.27	4.38	4.38	1.96	4.91	4.78	1.81
	CellBricks					1.20	16.85	4.35	4.33	1.98	4.91	4.96	1.76
Downtown	MNO	68.16	50.60	49.60	48.53	1.14	16.54	4.30	4.33	2.03	4.94	5.12	1.89
	CellBricks					1.11	15.41	4.25	4.32	1.97	4.94	5.22	1.89
Highway	MNO	44.72	25.50	49.48	48.38	1.10	11.38	4.34	4.34	1.95	4.89	5.05	1.86
	CellBricks					1.11	12.42	4.27	4.30	1.97	4.90	5.18	1.80
Overall Perf. Slowdown		-	-	-	-	2.06%	3.06%	1.15%	0.92%	0.51%	-0.20%	2.60%	-1.61%

Table 3.1: Comparisons of application performance in CellBricks vs. today's cellular networks (MNO).

(ii) *Detecting handover.* Qualcomm chipsets expose a diagnostics interface via which we read baseband messages, *e.g.,* using the tool QCSuper [267]. We pass the RRC messages to Wireshark's in-memory capture to extract handover events. On detecting the start of a handover, we emulate an UE IP change as described below.
(iii) *Emulating IP change.* We first note that we only emulate IP changes as below for the UE-VM pair running MPTCP; our baseline UE-VM pair run unmodified TCP and we do not change their IP address across handovers. Applications run inside docker containers on the UE and VM. To emulate IP changes, on detecting a handover at the UE, a proxy program invokes `ifconfig` to set the container's (virtual) network interface to `0.0.0.0` – this emulates address invalidation when switching bTelcos (§3.4.2). The proxy then waits for a time d before re-assigning the interface a new IP address. The interval d represents the overhead of attachments in CellBricks. Unless noted otherwise, we set $d = 31.68ms$, based on our prototype benchmarks in §3.6.1 (us-west-1 test).

Interestingly, our early experiments revealed that the mainline MPTCP implementation limits how fast the MPTCP stack can react to an IP address change. It does so by introducing a wait period between when it first detects an address change and when it takes any corrective action (*e.g.,* starting a new subflow). This wait period is hard-coded to 500 ms (see `address_worker` [221]) which effectively masks the overhead d that CellBricks introduces. This wait period is an optimization to avoid flapping and can be adjusted. For our default test setup, we choose to *retain* this 500 ms wait period so as to reflect the performance one can expect with MPTCP as deployed today and hence our default results represent the pessimistic case for CellBricks. Later, we modify MPTCP to remove this default value and repeat key tests to show the effect of varying d.

Finally, to carry packets between our MPTCP-based UE and VM, we set up a software switch (OVS [306]) on each side of the UE and VM. The client-side OVS switch tunnels the packet to the OVS switch at the server, which strips off the packet's outer header such that the server sees packets with the UE's new IP address. For parity, we run the same OVS setup in our baseline TCP scenario but in this case, OVS simply pass packets through without any tunneling.
(iv) *Applications and their metrics* We run four classes of applications: standard network benchmarks (ping, iperf [149]), voice calls (pjsua [242]), video streaming (HLS [225, 136]), and web browsing (page downloading). Since VoIP does not use TCP (and hence MPTCP), we need a different approach to handle IP address changes in CellBricks. For this, we modify the pjsua client to use SIP's re-invite mechanism where a host sends a SIP re-Invite message to its peer upon IP changes allowing both endpoints to set up new RTP sessions [266],

Table 3.1 lists the performance metrics we track for each application. To measure the quality of voice calls, we used an industry standard quantitative call quality metric, the Mean Opinion Score (MOS), which can be numerically derived from the packet loss, latency, and jitter measured during the call [257]. The MOS varies from 1 to 5.0 where a score of 2 indicates poor quality and 4 indicates good quality. For video streaming, we measure the average quality level of the video playout, a key metric that is used to estimate the quality of experience for HLS/DASH-based video streaming [192]. Each quality level maps to a pre-defined video quality (*e.g.,* bitrate settings) and the higher the level the better the video quality but also the greater the network bandwidth consumption. Our HLS server streams 6 different quality levels (0-5) varying from 144p to 720p, transcoded using ffmpeg [101] from the same video file. We play the video stream with the hls.js [136] player at the UE and add instrumentation to collect metrics.

(v) *Mobility trajectory.* We pick three representative routes in the downtown, suburban, and highway areas of our geographic region. We repeatedly drive along each route with two UEs, each independently running the same application. We run tests during the day and the midnight-to-dawn period because, as we discovered during our drives, T-Mobile enforces different rate limiting policies at different times.

Main results: Table 3.1 summarizes our key results. We show the performance for each of our four applications, separated by whether the tests were run in the day (D) *vs.* night (N). We compare the performance of CellBricks to our baseline which is the current MNO architecture running TCP, for each route (suburb, highway, *etc.*) and in summary (the last two rows). The 2nd and 3rd column report some basic statistics for calibration: the mean-time-to-handover (MTTHO) measures the average time between handovers and the ping results measure the network latency from our UE to our server VM in EC2 (us-west). As expected, we see lower MTTHO when driving faster (*e.g.,* at night *vs.* day).

Overall, our main finding is that CellBricks with MPTCP achieves performance comparable to the TCP baseline for all four applications, with a slowdown of at most 3.06% (last row). Surprisingly, we even observe cases where CellBricks outperforms the MNO baseline, *e.g.,* web downloads at night are 1.61% faster with CellBricks (we explore why shortly).

In secondary observations, we see that: (i) video streaming is least sensitive to the choice of handover schemes due to its use of segment buffering that helps tolerate throughput fluctuations during handovers, (ii) most applications perform better at night when T-Mobile relaxes its rate limiting, allowing applications to achieve higher throughput (an average of 15.46 Mbps at night compared to 1.16 Mbps during the day). VoIP is less bandwidth intensive (requiring $\approx$ 30 kbps) and hence is less

sensitive to this effect.

Understanding CellBricks's performance. We dig deeper to understand CellBricks's competitive performance.

One factor is simply the frequency of handovers – even in our worst-case (driving along the highway at night), we observe a handover only every 25.5 seconds and hence any overheads of re-attachment are averaged out. (Recall from earlier in this section, that MPTCP by default introduces a wait time of 500ms before initiating a new subflow.) However, as we'll show shortly, even when we zoom into the periods around handovers, CellBricks performs well.

We find that the reason for this has to do with the dynamics of slow-start. On a handover, CellBricks initiates a new MPTCP subflow, which enters slow-start and quickly catches up with its TCP counterpart in the MNO baseline (recall that the TCP connection undergoes no change during handovers beyond reacting to any loss that might occur). In fact, for short periods, the MPTCP subflow achieves *higher* throughput than the TCP flow. *E.g.,* Fig.3.8 zooms in on the iPerf throughput at 1-second intervals for a 50s period in four of our traces. A handover event occurs at around second 23 and we see that MPTCP's performance drops close to zero (reflecting MPTCP's 500ms wait period) but then quickly ramps back up and temporarily even overshoots the TCP flow (we see this in the "spike" that appears in the few seconds right after the handover). We were also able to reproduce this phenomenon in controlled experiments.

Factor analysis: varying attachment latency. Next, we evaluate the impact of attachment latency on the performance of CellBricks. For this, we reconfigure MPTCP to remove the 500ms default wait period and rerun the iperf experiments for different attachment latencies: d=32, 64, 128 ms. (Recall from our prototype benchmarks that we expect attachment latencies in the 30-80ms range depending on the location of our broker.) We measure performance at night so that performance is less constrained by T-Mobile's rate limits. Fig.3.9 shows our results. The Y-axis shows the average throughput that CellBricks achieves normalized by that of our baseline's TCP throughput. To show the performance impact at different timescales, we measure normalized throughput in the n seconds after a handover and plot performance for different n on the X-axis. I.e., a data point corresponding to 2s on the X-axis shows the average throughput MPTCP achieved in the 2s window after handover, normalized by the average throughput that TCP did in the same period.

As expected, we see that performance degrades with higher attachment latencies. *e.g.,*, at second 2, CellBricks with an attachment latency of 32ms has 7.7% higher performance than with an attachment latency of 64 ms. We also see that

removing the 500ms wait time in MPTCP improves our performance and, although MPTCP and TCP eventually converge to equal throughput, CellBricks now routinely outperforms our MNO baseline during handovers! In general we find that, without MPTCP's 500ms wait time, CellBricks achieves 10%-30% higher throughput than TCP in the first few seconds after handovers due to the impact of slow-start.

In summary, CellBricks's approach to mobility does not degrade application performance.

3.7 Related Work and Conclusion

Various efforts seek to improve implementations of the *existing* cellular architecture; *e.g.,* via disaggregation [105], improved modularity [286, 39, 250], or leveraging software [245, 156]. We instead propose a new architecture that redesigns and reorganizes functions across different players so as to improve competition amongst providers and reduce architectural complexity. We see these goals as complementary.

CellBricks shares similarities to network infrastructure sharing approaches like MORAN [121] and open-access networks [67] which enable MNOs to operate on the same RAN infrastructure. These require trust and tight coupling between the shared RAN and the MNOs that use it, incurring high transaction costs and limiting scale. CellBricks alieviates these constraints by supporting lightweight, many-to-many relationships between bTelcos and brokers. In the Wi-Fi domain, there are proposals like eduroam [321] and OpenRoaming [317] that allow users to receive access from Wi-Fi hotspots operated by a diverse range of organizations. Compared with these proposals, CellBricks allows untrusted access providers with its secure authentication and billing protocols, and supports seamless mobility across bTelcos with its host-driven design.

CellBricks shares many high level goals with the personal router project [65]: *e.g.,* enabling an open, competitive market in which small access providers can offer services. However, the personal router project relies on Mobile IP [240] for mobility and implicitly assumes that access providers are trusted, both of which are different from CellBricks, as mentioned above.

Finally, there is a long history [102] of researchers and industry proposing novel architectures aimed at simplifying the deployment of mobile networks, including truly micro-scale networks [109, 259, 133, 271], particularly in rural areas. *E.g.,* Magma [99] and CCM [128] do so via an orchestrator service that acts as an intermediary in establishing trust relationships with both bTelcos and larger providers. More generally, these efforts aim to enable small-scale networks where MNOs provide no service, while ours is to potentially enable the *replacement* of these MNOs by bTelcos.

We believe the time is right to explore new cellular designs such as CellBricks since we are seeing a proliferation of low-cost software defined radio base stations (which makes it easier to explore new designs) at the same time 5G has brought a pressing need for denser deployments (which makes finding new and efficient designs important). CellBricks can be incrementally deployed, initially complementing existing networks just as current cellular networks do generational upgrades. Moreover, all of the changes required by CellBricks are quite feasible: no change to the radio/RAN, straightforward changes to the software functions in the cellular core, minor changes to UE firmware, and only configuration changes (to enable MPTCP) to the network software stack on clients and servers. An incremental deployment of these relatively minor changes would enable a transformation in how cellular service is delivered.

SAP: bTelco Procedures

Invoked by the bTelco upon getting a SAP request.

Arguments:

authReqU, authRespU, pk_B

authReqT	auth. request bTelco sends to broker
authRespT	auth. response from broker to bTelco
qosCap	QoS capability definitions
sk_T	bTelco's secret key

Procedures:

Upon receiving authReqU from UE:

1. **Lookup** broker (e.g. its IP) with authReq.id_B
2. **Set** authReqT = (authReqU, qosCap)
3. **Sign** authReqT with sk_T → **$sig_{authreqT}$**
4. **Send** ($sig_{authreqT}$, authReqT) to broker

Upon receiving ($sig_{...}$, authRespU, $sig_{...}$, authRespT) from broker:

5. **Verify** $sig_{authrespT}$ with pk_B; do 6. if succeed
6. **Reply** ($sig_{authrespU}$, authRespU) to UE; **Decrypt** authRespT; set UE profile and NAS security context

SAP: Broker Procedures

Invoked by the broker upon serving a SAP request.

Arguments:

authReqT, authRespT, authVec*

sk_B	broker's secret key
ss	shared secret used by UE and bTelco
qosInfo	QoS information for the UE session

Procedures:

Upon receiving ($sig_{authreqT}$, authReqT) from bTelco:

1. **Verify** $sig_{authreqT}$ with pk_T and do 2. if succeed
2. **Decrypt** authVec* with sk_B
3. **Lookup** id_U and id_T; **Verify** $sig_{authvec}$ with **pk_U** and id_T matches **pk_T**; do 4. if both succeed
4. **Encrypt** (id_B, id_T, ss, n) with pk_U → authRespU**;** (id_B, ss, qosInfo) with pk_T → authRespT
5. **Sign** authResp → **$sig_{...}$**, authRespT → **$sig_{...}$** with sk_B
6. **Reply** ($sig_{...}$, authRespU, $sig_{...}$, authRespT) to bTelco

Figure 3.3: A summary of the SAP procedures that run at the bTelcos (top) and the brokers (bottom).

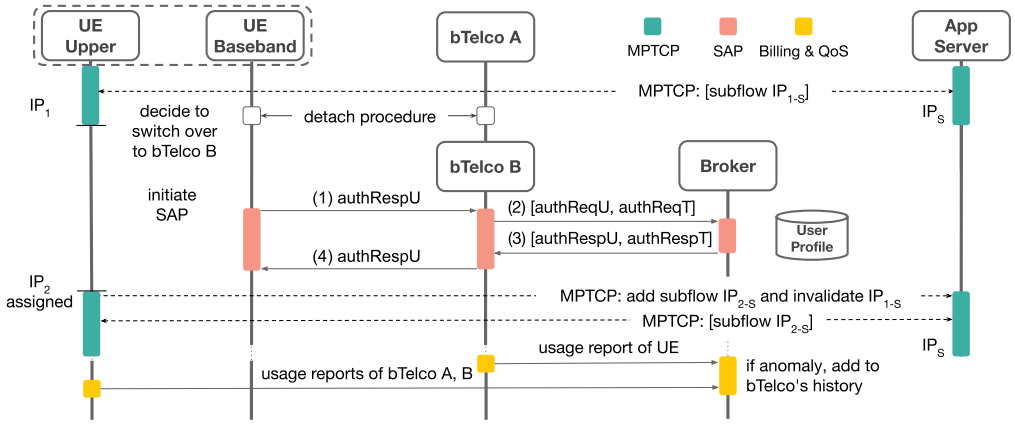

Figure 3.4: An overview of the attachment, mobility, and billing/QoS process in CellBricks depicting the key events and message exchanges happen during the SAP, MPTCP, and billing protocol over time. Note that authReqU/T and authRespU/T are defined in Fig.3.2 and Fig.3.3.

Verifiable Billing and QoS

Invoked by the broker at the end of each billing cycle.

Arguments:

$tr_{\{U,T\}}$	traffic report from UE or bTelco; it includes usage (e.g. dl_bytes) and QoS (e.g. dl_loss) info
thresh	anomaly threshold
dsp	discrepancy of reported usage

Procedures:

Once received both tr_U and tr_T:

1. **Set** thresh = (tr_U.qos.dl_loss + ϵ) × tr_T.usage.dl_bytes
2. **Set** dsp = tr_T.usage.dl_bytes - tr_U.usage.dl_bytes
3. **If** dsp > thresh, **increment** the # of anomaly at the bTelco

Upon receiving a reputation query for bTelco T:

4. **Report** the # of anomaly occurred at T

Figure 3.5: A summary of the steps run at the broker to enable verifiable billing and QoS.

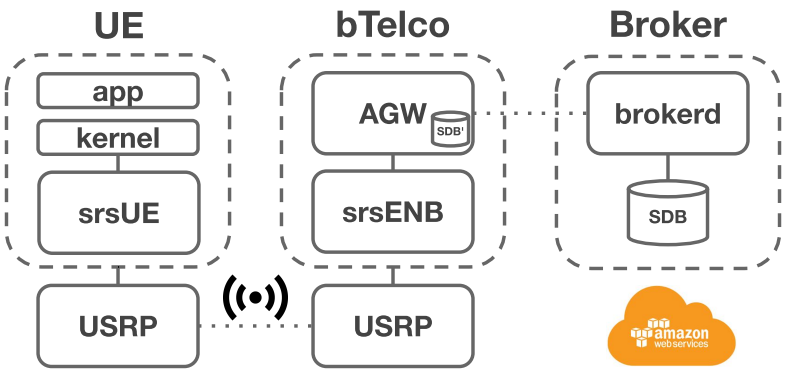

Figure 3.6: An overview of our testbed. AGW: access gateway; SDB: SubscriberDB.

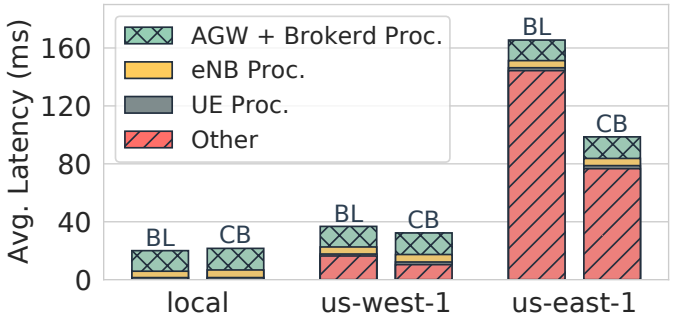

Figure 3.7: Latency breakdown by module in the Magma baseline (BL) and CellBricks (CB) during an attachment request.

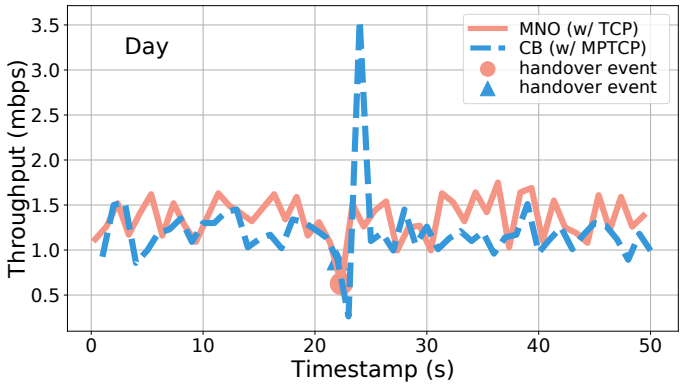

Figure 3.8: Comparison of the network throughput (iperf) achieved by MNO and emulated CellBricks over time.

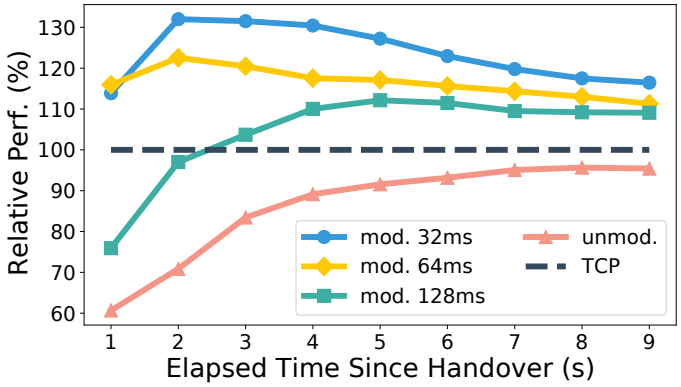

Figure 3.9: Impact of varying attachment latency on the iperf throughput. We report the relative performance using the TCP results from the same run as the baseline.

Chapter 4

Privacy-preserving Cellular Architecture

4.1 Introduction

Providing users with *location* privacy is an important part of the larger challenge of online privacy. Unfortunately, today's cellular architecture offers little location privacy: network operators know the identity of a user and the geographic location of the access point to which that user connects and hence can trivially track a user's location in time. There is mounting concern over this situation as cellular providers are reported to routinely share their users' location profiles [172, 319, 73, 74, 163]. Moreover, 5G is likely to require smaller cell sizes [45] thus exposing much finer-grained location information and exacerbating the privacy problem.

Hiding a user's location from their network operator is challenging because connecting to an access point fundamentally reveals the user's location. One approach to improving privacy is to hide the user's *identity* from the network operator using so-called "blindly signed tokens" [265, 55, 205]. However, as discussed in §4.3, this approach comes at the cost of preventing network operators from providing *identity-based services*. These are services whose correct execution depends on the user's identity, such as post-pay [54], QoS prioritization [2] and lawful interception [4]. Such services are an essential part of today's networks and hence it is unlikely that operators can/will abandon them in exchange for improved user privacy. Thus, our question is whether we can enable location privacy (*i.e.,* ensuring that network operators cannot easily track or infer a user's location) *without* compromising on identity-based services.

Privacy and identity-based services might seem to be fundamentally at odds. However, we see a way forward via mobile *virtual* network operators (MVNOs) such as Google Fi and Cricket [117, 75]. MVNOs are service providers that do not own radio infrastructure but instead provide user-facing services (sales, billing, *etc.*) while relying on business agreements with some number of traditional mobile network

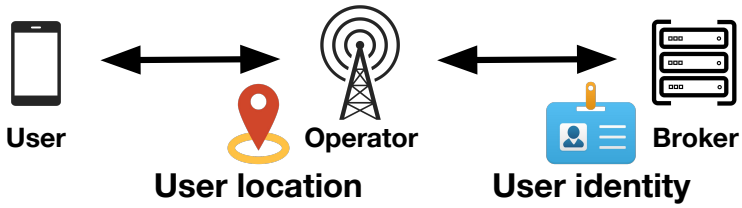

Figure 4.1: LOCA's overall architecture.

operators (MNOs) to provide the radio infrastructure. In this scenario, users pay MVNOs for service and MVNOs settle with MNOs on behalf of users. In other words, with MVNOs in the picture we can decouple infrastructure operation from user management and the MVNO acts as a broker between the user and the infrastructure operator.[1]

As shown in Fig. 4.1, our insight is that the existence of a broker between the user and operator enables us to reconcile privacy with identity-based services by strategically hiding different pieces of information from each party: the broker (*i.e.,* MVNO) knows the user's identity but not her location, while the operator (*i.e.,* MNO) knows the user's location but not her identity. With this arrangement, the broker can still tell the operator what identity-based services are to be applied to the user without revealing the user's identity, and the operator can implement the required services without knowing the identity of the user on whose behalf they are implemented.

However, hiding information in this manner is challenging for four reasons. First, in order to hide the user's identity from the operator, we must hide not just her identity but also her *trajectory* across multiple cell towers. This is because the operator could still infer the user's identity based on the sequence of towers she has visited, a form of privacy loss we refer to as *trajectory leakage* (§4.3.3).

Second, in order to hide the user's location from the broker, we must also hide the *identity of its operator* from the broker. This is because the locations of an operator's cell tower deployments are public knowledge and hence can reveal a user's location [229]. The emergence of operators with small footprints, such as private and enterprise 5G networks, underscores the importance of this [95, 33, 294, 122].

The last two challenges arise because of this need to hide the identity of the operator from the broker. Brokers will always want to ensure that only authorized operators service their users. Since our approach hides the operator's identity from

[1]In this paper, we use the terms MVNO and broker interchangeably; we do the same with the terms MNO and operator.

the broker, we now need a solution that allows the broker to verify the legitimacy of an operator *without revealing the operator's identity.* Lastly, when it comes time to settle payments, the operator should be able to claim payment from the broker *without* revealing what users it has served (since doing so would otherwise reveal user locations).

We design a privacy-preserving protocol that addresses the challenges above. Our contribution lies in developing new techniques (*e.g.,* aggregate claims) and synthesizing them with existing ones (*e.g.,* blind signatures, zero-knowledge proofs) into an end-to-end **Location-Oblivious Cellular Architecture (LOCA)**. To our knowledge, LOCA is the first system to enable location privacy for users while also supporting a provider's operational goals such as usage-based billing, QoS and service levels, lawful intercept, and so forth.

We evaluate the privacy and scalability of our protocol through formal analysis, simulation, prototype implementation, and wide-area experiments. We recognize that LOCA does introduce certain complexity and system overheads. However, our evaluation shows that these overheads are modest and within reach of what can be practically supported today. An important part of our contribution is thus in exposing the architectural complexity and performance tradeoffs that might be necessary to achieve our privacy goals.

Our work is based on certain assumptions about user and operator incentives. We assume that privacy concerns will influence some users in their selection of providers which will incentivize some operators to adopt the proposed techniques.[2] In addition, a growing number of jurisdictions have enacted policies that require providers to protect user privacy and, as discussed in §4.3, our architecture makes it easier for a provider to ensure compliance with these legal requirements. We do not assume that this motivation will apply to all users or operators: since our architecture can co-exist with the existing cellular infrastructure, we envision it will be applied to (by) the subset of users (providers) that are motivated by location privacy.

Finally, we recognize that there are many ways in which a user's location may be revealed through their online activities (*e.g.,* posting timestamped photos). We do not claim to prevent all forms of location leakage. Our focus is only on preventing the leakage of location information that today occurs every time a user connects to the cellular network.

In summary, the contributions of this paper are: (1) a new approach to preserve user location privacy while providing identity-based services; (2) the detailed design

[2]Such market dynamics are already emerging in other contexts such as the smartphone market [29, 263, 141].

and implementation of a protocol (LOCA) based on this approach, and an evaluation of its performance and scalability; and (3) a formal analysis of the privacy provided by LOCA. Looking forward, we view LOCA as a first step towards privacy-preserving cellular infrastructure with room for improvement along multiple dimensions. We discuss these limitations extensively in the paper to motivate efforts on addressing these issues.

4.2 Background

The cellular ecosystem: MNOs and MVNOs Traditionally, the two main participants in a cellular network are the user with her device (called User Equipment, or UE) and the Mobile Network Operator (MNO). The MNO owns and operates cellular infrastructure and also provides user support services such as sales, billing and customer care. The user typically enters into a contractual agreement with one MNO which serves as her "home" provider. The user then consumes cellular services from her home provider or visited MNOs that her home provider has roaming agreements with.

In recent years, we've seen the rise of Mobile *Virtual* Network Operators (MVNOs). MVNOs are service providers that do not own radio infrastructure, but instead provide user-facing services (sales, billing, *etc.*), often focusing on serving specific underserved market segments [276, 179], while relying on business agreements with some number of MNOs to provide use of their radio infrastructure. In other words, the MVNO acts as a *broker* between the user and the infrastructure operator. In this scenario, the user contracts with an MVNO, and the MVNO in turn contracts with MNOs. Two well-known MVNOs in the US are Google Fi [117] and Cricket [75]. MVNOs can be involved in cellular operations to varying degrees, ranging from fully offloading to MNOs to operating their own core networks.

Identity-based services: These are services whose correct execution depends on the user's identity. An example of such services is lawful interception, a function that allows law enforcement agencies to selectively wiretap individual users [4, 96, 5]. In most countries, operators are legally required to support lawful interception. Additional examples of identity-based services include: (i) post-pay, which relies on identity-based accounting to charge a user based on her service consumption; (ii) QoS prioritization, where the network's treatment of a user's traffic depends on details of the user's subscription plan and past usage; (iii) deep packet inspection (DPI), where traffic is filtered based on the user's identity for purposes such as parental controls.

Location privacy in cellular networks: Location privacy, as defined in [42], is "the ability to prevent other parties from learning one's current or past locations". In the cellular context, this means that neither MVNOs nor MNOs should be able

to learn a particular user's current or past locations. The exception is when location information must be revealed for legal purposes like emergency services and forensics.

4.3 Approach and Design Rationale

In this section, we briefly discuss the goals and assumptions that motivate LOCA's approach.

4.3.1 System Assumptions and Threat Model

System model: LOCA assumes a broker-centric architecture like today's MVNOs. This architecture involves three entities: (i) users, (ii) brokers, and (iii) operators. Operators own and operate cellular infrastructure. Brokers act as intermediaries between users and operators: a user subscribes to services from her broker, and the broker represents the user to operators, including handling settlements with each. LOCA requires brokers to authenticate their users.[3] The user need not be aware of the specific operator her device is attached to.

Threat model: We adopt a common threat model among privacy preserving systems that seek to prevent inadvertent information leakage between participants [126, 82, 159, 215, 63]. We assume brokers and operators are *semi-honest* (*i.e.,* honest-but-curious) and *non-colluding*: they follow the protocol but will attempt to extract user location information from the protocol execution, and that brokers and operators do not collude. We also assume that operators may attempt to *overbill* brokers by lying about session usages or what users they serve[4]. Attacks based on out-of-protocol information or collusion are out of scope but discussed in §4.5.

Incentives: One might ask why brokers and operators would implement the changes we propose. We believe that adopting our system is beneficial to them for both financial and legal reasons: as users are becoming more privacy-conscious [318, 118, 195], brokers that offer an opt-in location-oblivious service will be more attractive to customers. Second, doing so may soon become mandatory: regulations like GDPR recommend the privacy-by-design approach, which continues to place increasingly strong requirements on manipulating PII [40, 324, 296, 323]. By implementing a design such as ours, brokers and operators reduce their risk of inadvertently in-

[3]For MVNOs who by default offload all cellular operations, they can still support LOCA users by deploying their own authentication servers.

[4]One might ask whether we need to protect against over-billing if the operator is semi-honest. The reason we do so is because, as we'll see, *once we have privacy*, it becomes much easier for an operator to overbill since the broker cannot tell which users were serviced by the operator and hence cannot check the operator's billing claims. Hence, an operator can follow the protocol and yet overbill with impunity. To avoid this, we assume operators may overbill and design our protocol to prevent this.

Arch	Operator (O)	Broker (B)	ID-based SVC
Today	UID, Location, Trajectory	UID, OID	Full
PGPP	Location, Trajectory	OID	Partial
LOCA	Location	UID	Full

Table 4.1: Comparison of today's MVNO architecture, PGPP and LOCA in terms of information revealed to participants and support for identity-based services (ID-based SVC); U/OID: U/O's identity.

fringing privacy regulations. We explicitly assume that these benefits will outweigh the benefits of selling location data or implementing ad-hoc approaches to enforcing regulations, and thus we focus on the technical feasibility of a location-oblivious cellular architecture that also supports operational goals like usage-based billing and customized service levels.

4.3.2 Goals

Consider a user U, operator O, and broker B. We say that U's location privacy is violated when O and/or B know both U's identity and location. Today's cellular protocol trivially reveals both U's identity and her location. By protocol we mean the messages – their syntax and semantics – exchanged between U, B, and O as defined by the standard. Today, protocol messages carry U's identity, and the identity of the tower that U attaches to reveals U's location. Hence simply implementing the protocol allows an operator to track U's location with no special effort. In contrast, we are interested in modifying the existing cellular protocol standard to protect user privacy.

4.3.3 Approach

In research, the state of the art is the recently proposed PGPP protocol [265] which tries to provide location privacy by *hiding* U's identity from O and B. In PGPP, users are identified by a "blindly signed token" [55, 205] which they obtain during a registration phase prior to consuming service.[5] I.e., a user prepays for a certain quota of service (*e.g.,* some number of minutes of connectivity at a specified data rate) and in return obtains a blindly-signed token. When connecting to the network, the user presents this token via which the broker can authenticate the user without learning her identity.

To our knowledge, PGPP is the first system that tries to provide location privacy for cellular users. However, as we detail in §4.8, PGPP faces two drawbacks. First,

[5]Such a token is *blindly* signed by the broker who can later verify the signature without being able to link it back to the original signing request.

PGPP does not easily allow operators to support identity-based services, which are widely deployed in today's networks. Second, a user's *trajectory* across towers is still visible to operators and hence the protocol is vulnerable to "trajectory-based location leakage" in which the operator can learn the user's identity by correlating her trajectory with other out-of-band information.[6] In designing LOCA, we wished to avoid these limitations which, as we will see, leads to an altogether different approach.

In summary, our goal in LOCA is to design a cellular protocol that protects the location privacy of users by achieving the following properties: no party in the protocol (broker, or operator) should simultaneously know both the identity and the location of a user; the protocol should also not reveal the user's trajectory to either broker or operator. Finally, the protocol should support identity-based services including post-pay and lawful intercept. In this work, we propose LOCA, a new cellular protocol that achieves these stronger privacy guarantees while supporting identity-based services.

We briefly comment on the scope and limitations of LOCA as presented in this paper. Our goal is to safeguard users' location privacy at the *protocol* layer. This raises the bar relative to today's protocols but isn't sufficient to safeguard against violations that might occur outside the protocol, at other layers. For example: at the application layer, a user's identity might be revealed by inspecting their packets [34, 273], or physical-layer characteristics (*e.g.,* signal patterns) might be exploited to track a specific device [80, 114]. Such attacks are possible but (to our knowledge) not exploited today. However, if cellular protocols evolve to protect privacy, such app/physical layer leakages could become a more important issue. Fortunately, the research literature provides solutions to such attacks [314, 327, 158, 100, 346, 140] that we believe can coexist with protocol-layer solutions like LOCA. We elaborate on this in §4.5.2 but leave an in-depth exploration to future work.

There is an obvious tension between guaranteeing location privacy and offering identity-based services: connecting to a cellular tower fundamentally reveals a user's location, while customizing service to a user requires knowing the user's identity. Our insight is that we can extend broker-centric architectures to create a situation in which the broker knows the user's identity but not their location, while the operator knows the user's location but not their identity; neither broker nor operator knows the user's trajectory.

How do we achieve this? First, to hide U's location from B, we hide the identity

[6]For example, consider a user that regularly travels between their home and office location: the operator could narrow down the identity of the user by correlating this trajectory with residential information in billing records.

and location of the *operator* O from B. Recall that U attaches to the network (and hence to B) via O's infrastructure and hence, if B cannot tell where O is located, then it cannot tell where U is located either. Hiding O's location is not sufficient: we must also hide O's identity from B, as knowing O's identity might be sufficient to narrow down O's location (and hence U's location). An operator's tower locations are public knowledge and, moreover, we're seeing an increasing deployment of small-scale cellular networks due to the emergence of private and enterprise 5G networks, as well as various forms of community networks [95, 33, 294, 122].

As we will describe in §4.4, we hide O's identity from B by having O obtain an unlinkable token from B during an offline registration process.[7] O later uses this token (denoted $\hat{O}$) as its identifier when interacting with B. By the properties of blind signatures, B can verify that $\hat{O}$ is a pre-authorized operator but cannot link $\hat{O}$ to O. In addition, O hides its IP address from B by using anonymous communication solutions.

The above suffices to hide U's location from B. The other half of our arrangement is to hide U's identity from O. This is easily achieved since O does not need to know U's identity to service U; since B knows U's identity, B can tell O what services are required (rate limits, filtering rules, *etc.*) thus enabling identity-based services without revealing U's identity. Thus, U simply uses a temporary pseudonym (denoted $\hat{U}$) in her interactions with O. Finally, by periodically changing U's temporary pseudonym and randomizing attachment timing, we limit O's ability to track a particular user's trajectory.

As summarized in Table 4.1, the above approach offers U location privacy while still supporting identity-based services. However it gives rise to a new challenge: how does O receive payment for its services to U? In today's architecture, B directly settles with O based on the service that U received. We wish to preserve this direct billing system between O and B. Yet, our protocol intentionally hides O's identity from B. To address this issue, we devise a solution that allows O to reveal its true identity *only* when claiming payment from B. Our solution leverages zk-proof techniques to design a novel *aggregate claiming* procedure via which (i) O claims payment for an aggregate of the user sessions it has serviced, and (ii) B can verify the correctness of O's claim without revealing the identity of the users that O serviced.

4.4 Design

At a high level, the process of obtaining cellular services can be broken down into four phases or steps: (i) *registration*, during which the various parties (U, B, O)

[7] The use of such a token is similar to PGPP but used by O instead of U which we will see leads to a very different set of considerations.

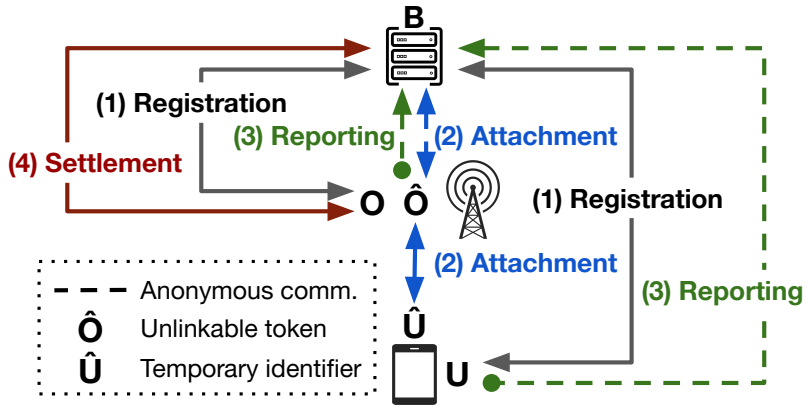

Figure 4.2: An overview of LOCA's protocols.

enter into pairwise contractual relationships: U signs up with B for service, and B with O as an operator for its users; (ii) *attachment* involves the protocol by which U discovers and connects to a tower in O's infrastructure, (iii) *mobility* involves the handover protocols via which U is migrated from one tower to another as needed, and (iv) *settlement* refers to the norms and processes via which B pays O for the service that O has provided B's users.

Of the above, *attachment* and *mobility* are defined by today's 3GPP standard while *registration* and *settlement* are out-of-band processes. Our goal is to implement LOCA with minimal disruptions to today's protocols, and without involving any new entities in the registration or settlements process.

Next, we describe LOCA's operation in these phases, an overview of which is given in Fig. 4.2.We briefly summarize how each phase is typically implemented in today's networks and then present the changes that LOCA introduces. Finally, we elaborate on how identity-based services work in LOCA.

4.4.1 Registration

Today: In today's networks, when U signs up with a broker B, they exchange shared secret keys (SSKs) that will be used for mutual authentication during the attachment process. In 5G, B also shares its public key (PK_B) with U so that U can encrypt her identity in later attachment requests.

LOCA: With LOCA, B and U continue to exchange PK_B and SSK. Like today, these keys will be used for mutual authentication between U and B (§4.4.2) and to hide U's identity from O. The main change LOCA introduces is in the registration process between B and O. When B and O sign up with each other, LOCA requires that they participate in a blind signature protocol [55, 205] as a result of which O

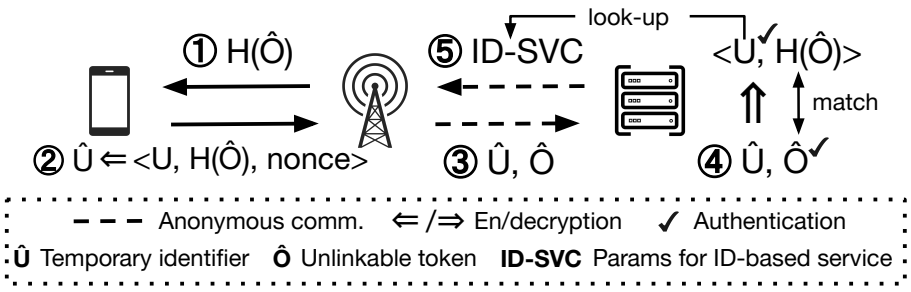

Figure 4.3: LOCA's attachment procedure.

obtains unlinkable tokens (denoted as $\hat{O}$) that are blind-signed by B. When $\hat{O}$ is later presented to B, the blinding process ensures that B can verify the signature but cannot link $\hat{O}$ to O. Thus blind tokens allow B to authenticate O without learning O's identity. LOCA uses a standard blind-signing protocol [55]. In addition to blind tokens, B and O also exchange a shared hash function H that will be used in our attachment and settlement processes as described later.

4.4.2 Attachment

Today: Attachment today involves three main steps. First, O broadcasts its identity on the radio control channel that U listens on to discover O. Next, after discovering O, U sends an *attachment request* to O who forwards the request to B for authentication. In 5G, U uses an encrypted identifier (termed SUCI [3]) in this attachment request. Finally, once U has been authenticated, B responds to O authorizing service. B's response includes U's permanent identifier (termed SUPI [8]).

Thus today's attachment process reveals O's identity to U in the first step. In the second step, B learns O's identity (and hence U's location) from both the contents of the attachment request and the act of receiving it from O (which reveals O's IP address). Finally, O learns U's identity via the authorization response it receives from B.[8] Thus today's attachment reveals U's identity and location to both B and O.

LOCA: We describe LOCA's attachment process with an emphasis on how we prevent (i) B from learning O's identity and (ii) O from learning U's identity. As mentioned in §4.3.3, we achieve the former by having O interact with B as $\hat{O}$ (O's unlinkable tokens) via anonymous communication channels. LOCA achieves (ii) by

[8]Prior to 5G, U's permanent identifier (IMSI) was included in the initial attachment request, allowing O to directly discover U's identity. Since 5G, U's attachment request uses an encrypted temporary identifier over the air to defend against IMSI catchers [283]. Nonetheless, O still learns U's permanent (SUPI) identifier from B's authorization response (step 3).

encrypting U's identity (with PK_B) and never exposing it outside of B. As shown in Fig. 4.3, LOCA's attachment process consists of the following five steps.
(i) Operator discovery. Instead of its actual identity, O broadcasts the hash of its token (*i.e.,* H($\hat{O}$)) on the control channel.
(ii) User preparation. U sends an attachment request to O (formatted as a NAS message [10]). This request includes B's identity, U's identity (IMSI) plus a nonce, and H($\hat{O}$). The last two – (IMSI+nonce) and H($\hat{O}$) – are encrypted by B's public key and serve as a temporary identifier for U which we denote as $\hat{U}$. We assume that B has a large user group so that its identity leaks little information on U's identity. The nonce ensures that $\hat{U}$ is different every time U attaches to the same O which helps prevent O from tracking U's trajectory (§4.4.3).
(iii) Operator preparation. On receiving U's attachment request, O forwards the request to B over an anonymous communication channel and uses its unlinkable token $\hat{O}$ to identify itself. Typical solutions for anonymous communication are Tor [298] and VPN [228] with different performance/security trade-offs, which we will discuss in §4.6.3. This anonymous channel can be set up offline, prior to attachment, whenever O changes token $\hat{O}$. Thus B does not see O's true identity nor the IP address from which $\hat{O}$ sends the request. The latter is necessary as several studies have shown that IP addresses can often be geo-located with high accuracy [69].
(iv) Broker authorization. On receiving the attachment request, B first verifies the $\hat{O}$ token thus ensuring that the request comes from an operator that B has previously authorized during the registration phase. Next B decrypts the request, and authenticates U via today's challenge-response protocol based on the shared secret key SSK [9]. In addition, B verifies that $\hat{O}$ is indeed the operator to which U wants to attach; B can verify this by validating H($\hat{O}$) (using the shared hash function established when O registered with B) and thus prevents replay or hijacking attacks. Once B has authenticated and verified the request, it looks up the parameters associated with U's service plan (as today): *e.g.,* rate limits, QoS parameters, whether to intercept U's traffic, and so forth. B then crafts a response authorizing the attachment (including the proper service parameters, security parameters that allow U to authenticate the network, etc), signs it, and returns it to $\hat{O}$.
(v) Access attachment. B's response authorizes $\hat{O}$ to service U as per the parameters from B. Beyond this point, $\hat{O}$ (*i.e.,* O) serves U as in today's networks. We elaborate on how O provides identity-based services to U in §4.4.4. Note that O can still perform functions like establishing radio bearers that require binding U's identifier to temporary identifiers like GUTI and RNTI; O simply uses $\hat{U}$ instead of U.

4.4.3 Mobility

Today: In current networks, mobility is implemented via a handover process, where O initiates U's migrations by directing U to switch from a tower T1 to another T2. This approach ensures a seamless mobility experience for U because U's IP address remains unchanged after the migration. However, as O initiates U's migrations, O trivially observes U's trajectory across handovers, jeopardizing U's location privacy.
LOCA: Trajectory leakages are inevitable if O fully controls U's mobility like today: although LOCA already hides U's identity from O during attachments, O can still track $\hat{U}$'s trajectory and use that to infer U's identity, making LOCA vulnerable to trajectory analysis. To mitigate this fundamental issue, we leverage a user-driven mobility approach proposed in [202]. In this approach, U initiates migrations across towers by simply detaching from T1 and then attaching to T2. U then relies on modern transport protocols like MPTCP [233] and QUIC [177] to maintain connections despite changing IP addresses. Prior work has shown that this user-driven approach does not degrade service even when reattaching on a *per-tower* basis [202]. LOCA adopts and extends this approach to minimize trajectory leakages with two techniques: (i) periodic reattachment and (ii) randomized attachment timings.

First, U will detach and reattach periodically (not at every tower) with a new temporary identifier. Thus, O cannot trivially track U across new sessions based on U's identifiers. The reattachment frequency is a configurable parameter that bounds the length of U's trajectory that is visible to O where length might be measured in time (*e.g.,* valuable for a mostly stationary user), in towers, or some combination thereof.

Even with periodic reattachment, O may still attempt to infer U's trajectory by doing a timing analysis over her detach and attach events. In particular, such analysis would be effective in a naive implementation that uses a fixed interval between when U detaches from T1 and subsequently attaches to T2. To address this issue, we have U wait for a randomized but bounded duration of time before issuing her attachment. When possible, we can also leverage make-before-break attachments[9] in which U may attach to T2 *before* detaching from T1 thus increasing the time window over which U can randomize their attach/detach events which makes inference harder. Together with periodic reattachment, this randomization of U's attachment times limits O's ability to correctly infer U's trajectory, because U's (re)attachments are obfuscated by the periodic (re)attachments from other nearby users.

We recognize that user-driven mobility introduces some complexity as well as

[9]The support for make-before-break, so-called dual active protocol stack (DAPS) handovers has been introduced in 5G 3GPP specifications [12, 110, 292].

dependencies on newer transport stacks, however this tradeoff is fundamentally necessary if we are to prevent trajectory leakages, and supporting these techniques incurs a minimal impact on the user's performance (§4.6.3). As we will detail in §4.5.1.3, the obfuscation effect of our approach depends on the specific configurations, *i.e.,* reattachment frequency and attachment time window; as well as the deployment scenarios, *i.e.,* the number of nearby users and the length of U's trajectory. Overall, under realistic deployment scenarios and configurations, the probability that O can correctly infer U's trajectory is negligible.

4.4.4 Identity-based Services

LOCA ensures that operators and brokers can continue to provide critical identity-based services, including allowing law enforcement agencies to locate specific users when required.

The key reason LOCA can support identity-based services is that brokers continue to know the identity of their users. This enables B and O to collaborate on identity-based services. For instance, during attachment, B can select the service level associated with U's plan and indicate that to O in its authorization response – *e.g.,* via the QoS Class Identifier (QCI) parameter [325]. O then simply enforces the QCI for the duration of its session with $\hat{U}$ without knowing U's true identity.

To realize services such as lawful interception, law enforcement agencies work with B and O. As today, O runs a lawful interception (LI) system — *e.g.,* installing an interception gateway [293]. A law enforcement agency notifies B of the user whose communication it wants to intercept. B passes on this notification to O during the attachment process, and then O's LI systems report the required information to the agency.

Emergency services (*e.g.,* 911 calls) work in a similar manner. A law enforcement agency knows U's identity and needs to learn U's location. The agency reaches out to B; B looks up U's current temporary identifier $\hat{U}$, and asks $\hat{O}$ (via their anonymous communication channel) to reveal $\hat{U}$'s location to law enforcement. Thus, the agency can collect U's current location without violating LOCA's privacy guarantees (*i.e.,* O does not know U's identity while B does not know O's identity or location). The same approach can be used to recover U's past locations based on the records logged at B and O.

4.4.5 Settlements

Today: In today's MVNO networks, B pays O based on U's service parameters and the resources consumed, as reported by O to B.While differing in the details, existing settlement processes all require that B knows which users/sessions were serviced by O, thus potentially violating user location privacy.

LOCA: To settle O's payments while preserving U's location privacy, LOCA's settlement process contains two phases: a *reporting* phase, where U and O report session usage to B; and a *claiming* phase, where O claims settlement from B.

Reporting phase: In LOCA, we define a *session* as the user-operator association that starts when U completes the attachment process with $\hat{O}$ and ends when U detaches from the same. At some point after a session ends, U and $\hat{O}$ independently send traffic reports to B. Note that O continues to hide its identity and location when sending its report to B. U reveals its identity to B but also sends its reports over an anonymous channel because its IP address can reveal its whereabouts. The traffic report from U lists the sessions in which U participated; $\hat{O}$ does the same for its sessions. Each entry in the list contains a session identifier (SID), usage metrics (*e.g.,* bytes, duration), and QoS metrics (*e.g.,* packet loss rate). In addition, O appends a nonce to each session in its report. These nonces are generated from the shared hash function H known to both O and B, and taking secret inputs that are only known to O. We call these inputs "embedded secrets", and as we will see, O later uses these secrets to claim its settlement from B.

B then compares the reports from U and $\hat{O}$, generates bills for U and publishes a *session table* to start the claiming phase. The table includes the usage calculated based on the reports from $\hat{O}$ and U, for all sessions during the last billing cycle. When generating statistics in the session table, B can consider factors other than reported usages such as QoS metrics.

Claiming phase: Every billing cycle, O reveals its identity and claims settlement from B but does so without revealing which sessions O has serviced. To achieve this, we must solve three problems: (i) ***No over-claims.*** How does B verify that O is claiming only the sessions O actually serviced? (ii) ***No mis-claims.*** How do we ensure that O can claim the sessions but no one else? (iii) ***Session oblivious.*** How does O claim settlement without revealing to B which sessions it is claiming? We combine zero-knowledge proofs with the above mentioned "embedded secrets" to address (i) and (ii); and "aggregate" claims to address (iii).

Embedded secrets serve as the basis for O proving its session ownership to B. However, naively having O reveal its secrets fails the session oblivious requirement because B now knows what users O has serviced. This leads to our aggregate claiming protocol that fulfills all three requirements:

Aggregate claiming with ZK-proof: First, we observe that in order to generate O's payments, B does not need to know *individual* session ownership; instead, it only needs to know the session ownership in *aggregation*, *i.e.,* the aggregate usages for payments for a specific O. Based on this insight, our *aggregate claiming* mechanism works as follows: the claiming begins with B publishing a session table readable to

LOCA: Aggregate Claiming Protocol

O: operator, **B**: broker; F: claiming function

Procedures:

Setup:

1. **B** performs (PK, VK) = VerifierSetup(F)
2. **B** keeps verifier key VK and sends the prover key PK to **O**

Per billing cycle:

3. **B** publishes a session table ST
4. **O** generates (z, π) = VCProve(F, ST, Secrets, PK)
5. **B** verify the claim (z, π) with VCVerify(ST, z, π)

ZK-Proof-based Verifiable Computation:

VerifierSetup(F)

- **B** compiles F into an arithmetic circuit C
- **B** performs zk-SNARK preprocessing on C, generates PK and VK, and returns (PK, VK)

VCProve(F, x, w, PK)

- **O** computes z = F (x, w)
- **O** uses zk-SNARK to generate a ZK-proof π based on (x, z) as the primary input, w as the witness, using PK, and returns (z, π)

VCVerify(x, z, proof, VK)

- **B** uses zk-SNARK to verify the proof with (x, z) as the primary input, using VK

Figure 4.4: A summary of the aggregate claiming protocol.

all Os. O then reveals its identity and claims its payment from B:

Intuitively, O's claim takes the form: "I have sessions that add up to X bytes." Because the number of different sessions that could add up to X is large, it is difficult for B to infer whether an individual session is part of O's claim or not, thus obfuscating the session ownership. In §4.5.1.2, we show that the expected number of session combinations that add up to the same X grows *exponentially* w.r.t. the total number of sessions in the table via both theoretical and empirical analysis.

Note that this naive aggregate claiming suffices if we assume O will not overbill B. However, it is important to realize that without additional mechanisms (like the zk-proof that follows), O can more easily overbill B without being detected in LOCA than in today's (non-privacy preserving) architecture simply because B does not know what users O serves.

Hence, since naive aggregate claiming allows O to overbill, we extend our solution such that O can *prove* its claim by showing that O knows the embedded secrets corresponding to its claim. For this, we leverage *proof-based verifiable computation* [310], a cryptographic tool that uses zero-knowledge proof to enable one party to prove to another that it has run a computation $z = f(x, w)$, where f is the function, x is the public input, w is the prover's private input and z is the output, without revealing any information about w. Proof-based verifiable computation systems have two components: (i) a zk-SNARK backend [254] that proves and verifies satisfiability of *arithmetic circuits*, and (ii) a compiler frontend that translates program executions to arithmetic circuits. Such an arithmetic circuit is also referred to as "a set of *constraints*".

Fig. 4.4 describes LOCA's aggregate claiming protocol. First, B performs *VerifierSetup*, where B compiles a claiming function F into an arithmetic circuit, and uses zk-SNARK to preprocess the circuit and generate prover key PK and verifier key VK. This verifier setup step needs to be performed only once, after which B keeps VK and sends PK to each participating O. The claiming function F takes two inputs: a session table with at most N sessions as the public input, and a set of (at most K) secrets as the private inputs. F computes the hashes of the provided secrets, iterates all the sessions in the session table, adds a session's usage to the aggregate usage if one of the precomputed hashes matches the nonce of that session, and finally returns the aggregate usage.

Next, once per billing cycle, each O performs *VCProve*, which involves two steps: (i) O executes the claiming function F with the session table and its embedded secrets, which returns the aggregate usage z for O's sessions. (ii) O passes to zk-SNARK the session table, its secrets, the computed aggregate z and the prover key PK, to generate a zero-knowledge proof π, which allows O to prove to B that it has secrets for sessions that add up to z, without leaking any information about individual session ownership. O then sends a *claim* including the aggregate usage z and proof π to B.

For each O's claim, B performs *VCVerify*, where B uses zk-SNARK to verify the proof π with the session table, the claimed aggregate z and the verifier key VK. If the verification passes, given the soundness property of the zk-SNARK proof system [254], B can confidently approve O's claim and generate O's payment according to the claimed aggregate usage and other factors such as O's reputation. The duration of a billing cycle is configurable: longer cycles lead to larger session tables, which in turn indicates stronger privacy protections (§4.5.1.2) at the cost of more expensive operations (§4.6.2).

Session group: The design presented above assumes a single session per token,

which may not scale to large deployments: O generates a proof every billing cycle, and proving with zk-SNARK is expensive [328]. In our setup, as we will show in §4.6.2, the time complexity to prove a circuit for the claiming function F is $O(K*N)$, where K is the maximum number of sessions O can claim and N the total number of sessions in the session table. Such proving time would be prohibitively long when there are a large number of sessions to claim.

To address this scalability challenge, we introduce the notion of a *session group*, which includes all the sessions that are associated with the same unlinkable token. By grouping multiple sessions into a single session group, we can reduce the number of entries in the session table. To support session groups, we made the following extensions to our protocol:

- **Attachment:** We allow O to use a single token and the corresponding anonymous communication channel for multiple sessions as the same session group.
- **Reporting:** We allow O to send a traffic report containing all the sessions of the session group.
- **Claiming:** We allow B to publish a session group table with one session group for each row. O claims session groups the same way as it claims sessions before.

The size of the session group is tunable in LOCA and determines how many sessions each token is used for. Tuning the group size allows LOCA to explicitly trade off between privacy and scalability: (i) a smaller session group is better for privacy, because it minimizes indirect location leakages (detailed in §4.5.3), which occur when a user of a session within a session group has her locations leaked, in which case users of other sessions within the group also suffer a privacy loss; (ii) larger session groups are desirable in terms of scalability of zk-SNARK, as it takes longer to actually generate a session group (with users' attachment), while proving cost remains the same, as N is the same, so zk-SNARK proving becomes relatively faster. Fortunately, modern zk-SNARK is fast enough that a balance between privacy and scalability can be achieved: as we will show in our evaluation (§4.6.2), LOCA can scale to large deployments with sufficiently small session groups and thus introduces only minimal privacy loss.

4.5 Privacy Analysis

Safeguarding location privacy requires fulfilling three properties: (i) O does not knows U's identity, (ii) B does not know U's location, and (iii) neither B nor O knows U's trajectories. To our knowledge, LOCA is the first protocol to meet these requirements. In this section, we analyze the conditions and assumptions under which LOCA meets these requirements. We show that LOCA achieves all three properties under the assumptions of our threat model which are that participants

are semi-honest and do not collude (§4.5.1). We then briefly consider attacks beyond our threat model and show that LOCA offers substantial protection even when participants use out-of-protocol information (§4.5.2) or collude (§4.5.3).

4.5.1 Semi-honest and Non-colluding

We first analyze LOCA's privacy properties under our threat model of semi-honest and non-colluding participants (§4.3.1).

4.5.1.1 Hiding U's identity from O

LOCA hides U's identity from O. Specifically, U's identity is encrypted using B's public key. B is thus the only party that can decrypt and observe U's identity in plaintext. B also never exposes U's identity to O, even after U successfully attaches.

4.5.1.2 Hiding U's location from B

LOCA hides U's location by (i) hiding O's identity and location when O interacts with B on behalf of U and (ii) hiding which users were serviced by O when O reveals its identity to claim its settlement. Next, we show how LOCA achieves (i) via the security properties of existing cryptographic constructs (*i.e.,* anonymous communication and blind signature) and achieves (ii) via aggregate claiming; we establish the latter property via formal analysis and empirical simulations.

For (i), LOCA leverages anonymous communication such that two parties can communicate without revealing their identities to one another. Similarly, LOCA builds on a blind signature scheme that allows a participant to authenticate another without learning its identity. Taken together, these existing cryptographic constructs allow operators to register and report sessions to brokers without revealing their identities.

Discussing the security of aggregate claiming requires more care. We break this process down into two halves (i) the security of the claiming mechanism itself, (ii) the information leaked by revealing the aggregate value to B. The former follows directly from the security of our zero-knowledge proof construction. We do not discuss this further. Instead, we focus on the impact of B learning the aggregate value of the claimed sessions. Specifically, we show that B has an exponentially small likelihood to correctly infer what sessions/users O has serviced based on the aggregate value.[10]Our core intuition is simple. Let us assume that for N session groups with a uniform distribution of session group usage from 1 to m, operators will claim the aggregate usage of K session groups, which sum to aggregate value S. The total number of possible session group combinations grows exponentially as a

[10]The general reasoning extends to when B analyzes multiple claims from different operators but we don't get into the details in this paper.

function of N. In contrast, the number of possible claimed values only grows linearly $(m * N)$. In expectation, there will consequently be exponentially many possible session group combinations that could have summed to S. We formally prove that this result holds as long as the ratio between K (the number of session groups belonging to an operator) and the total number of session groups N falls within a specific range. We identify this range formally below, and show through simulation that these bounds can be further improved and are wide enough to support realistic deployment scenarios.

Theoretical proof: We formulate the aforementioned problem as follows. Consider arrays X and Y, one of size $N - K$, and one of size K, where each cell contains a value from 1 to m drawn from the discrete uniform distribution. Let S be the sum of all elements in Y. We derive a bound on the expected number of possible subsets of elements in X that sum to S.

Theorem 4.5.1. *Considering two independent arrays X and Y, consisting of $N-K$ and K iid random variables from $U\{1, m\}$, there exists $L(m), U(m)$ such that the expected number of subsets in X, whose sums are equal to the sum of Y, is exponential w.r.t N, if $L(m) \leq \frac{K}{N} \leq U(m)$. Note that $L(m), U(m)$ depend on m, and $0 < L(m) \leq U(m) < 1$, $\forall m \in Z_{>0}$*

The proof, at a high level, works by (i) deriving the closed-form distributions of the sum and the subset sum of an array of discrete uniform variables similar to prior theoretical work [48], (ii) expressing the expected number of matched subsets with these two closed-form distributions, and finally (iii) reducing to an exponential lower bound for the expression. More details of the proof can be found in the appendix.

The reductions in step (iii) are highly conservative. Hence the proven feasible range of ratio $[L(m), U(m)]$ is narrow, and the exponential bound is small. We confirm through simulation that this bound holds analytically for a significantly wider range and encompasses many real-world scenarios:

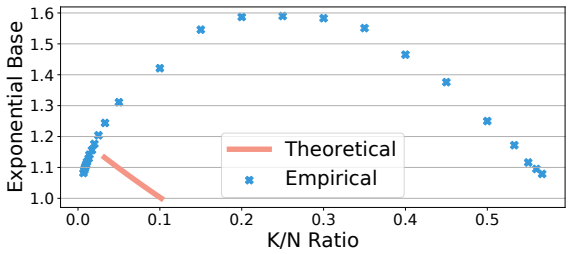

Figure 4.5: Exponential bounds for different K/N ratios with $m = 5$.

Empirical simulations: In these experiments, our goal is to understand within what range of ratio, the number of matched subsets grows exponentially w.r.t N. For each ratio K/N, we scale N while increasing K proportionally according to the ratio and estimate the expected number of matched subsets for the (K, N). More details about our simulation setup can be found in the appendix of the full report [201]. Now that we have estimates for multiple (K, N)'s of the ratio K/N, we fit the results with an exponential curve of N by performing linear fittings on the logs of the estimates:

$$R = a * b^N \rightarrow \log(R) = \log(b) * N + \log(a)$$

The slope of the fitted linear curve is thus the log of the exponential base. Our fitted linear curves closely match the logs of estimates with an adjusted R-squared value of over 0.99, which suggests a significant exponential relation between our estimates and N. Fig.4.5 shows the exponential bounds of different K/N ratios for uniform distribution with m=5. Compared with the theoretical results, the empirical results suggest much larger exponential bounds over a wide range of ratios: exponential base over 1.1 for ratios from 1/150 to over 1/2. We observe similar behavior with other values of m and with other non-uniform session group usage distributions.

4.5.1.3 Hiding U's trajectory

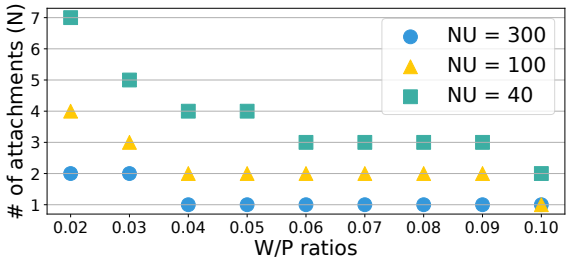

Figure 4.6: The longest trajectories beyond which the likelihood of correct inference is less than 1% for different NUs and W/Ps.

LOCA hides U's trajectory from O via (i) periodic reattachment and (ii) randomized attachment timing. The former prevents O from directly observing U's trajectory, and the later makes accurate timing-based trajectory inference infeasible:

With periodic reattachments, O is unaware of which attachments belong to U and hence O can only *infer* U's trajectory by correlating between detachment and subsequent attachment. By randomizing attachment timing, any detachment that arises within a time window before and after (with make-before-break handovers)

an attachment is equally likely to correlate with that attachment. We call this set of detachments "candidate detachments", and since all users periodically reattach, there is a *lower bound* on the number of candidate detachments. Lastly, to recover U's trajectory, O has to select the correct detachments for all of U's attachments along the trajectory, which becomes *exponentially* harder for longer trajectories. Modelling all these factors, we can analyze the difficulty of trajectory inference in LOCA: denoting time window as W, the reattachment period as P, the number of nearby users as NU, the number of candidate detachments as ND, the number of attachments in U's trajectory as N, we can derive the likelihood of O correctly inferring the trajectory $Prob$:

$$ND \geq 1 + NU * \frac{W}{P}, \quad Prob \approx (\frac{1}{ND})^N$$

This formulation tells us why accurate trajectory inference is infeasible: (i) since $Prob$ decays exponentially w.r.t N, even with a ND of 2 (only one alternative candidate detachment), O has a less than 1% likelihood of inferring a trajectory with more than 6 attachments; (ii) The ratio between the time window and re-attachment period (*i.e.,* $\frac{W}{P}$) is configurable, and a larger ratio increases ND and thus the inference difficulty. Fig.4.6 shows the longest trajectories that O can infer with a likelihood larger than 1% for different NUs and $\frac{W}{P}$s. For $\frac{W}{P}$ larger than 0.03, O is unable to infer long trajectories ($N > 4$), even if the number of nearby users is small ($NU = 40$).

We've shown that LOCA safeguards user location privacy at the protocol layer and under the assumptions of our threat model. We believe this raises the bar relative to the status quo however, as discussed earlier, LOCA would still be vulnerable to attacks that exploit either: (i) out-of-protocol information or (ii) information from other participants via collusion. We will next discuss such attacks, their impact, and potential mitigation strategies, but leave an in-depth study to future work.

4.5.2 With out-of-protocol information

Next, we show that (i) as a protocol-layer solution, LOCA does not prevent attacks based on out-of-protocol information, (ii) the impact of these attacks on LOCA is minimal and, (iii) mitigation strategies for these attacks can coexist with LOCA.
Attacks: B and O can compromise U's location privacy by exploiting out-of-protocol information. Here we enumerate some attacks that violate each of the three privacy properties: (i) If O has access to a resident directory near its cell towers, it can nail U down to a smaller user group. O might also learn U's identity by inspecting U's data traffic. (ii) If B is capable of network monitoring, it might learn O's identity by conducting a traffic analysis, where it observes traffic at each operator and correlates

that with incoming traffic it receives. (iii) O might track U's trajectory by profiling U's physical-layer characteristics, such as its signal patterns and strengths.
Impact: LOCA's design limits the impacts of out-of-protocol attacks on user's location privacy: (i) Attacks that allow O to uncover $\hat{U}$'s identity only incur *per-hop* leakages: U's identity remains unknown to O when she reattaches with a different $\hat{U}$. (ii) Attacks that allow B to uncover $\hat{O}$'s identity only incur *per-token* leakages: O's identity remains unknown to B when O switches tokens. This means that locations of users who are served by O with a different token from the revealed one remain unknown to B. (iii) Lastly, inter-operator attachments can minimize impacts of attacks that allow O to track U's trajectory. Firstly, instead of having her entire trajectories leaked, U suffers only *per-operator* leakages. Secondly, as U moves in and out of O, it is challenging for O to link all of U's trajectories within its footprint, because O is unaware of U's locations when U connects to other operators.
Mitigation: LOCA can coexist with countermeasures designed for different out-of-protocol information. For instance, for attacks based on traffic characteristics, end-to-end encryptions of U's traffic can help counter packet inspections by O; and communication systems that are robust to traffic analysis like Vuvuzela [304] could be adopted for communications between O and B. For attacks based on physical-layer signals, one could use defense mechanisms such as randomizing transmission coefficients [274] and injecting artificial noises [140].

4.5.3 With collusion

In the following, we show that (i) there are forms of collusion that lead to violations of user location privacy, and (ii) except for direct collusion between brokers and operators that serve the user, other forms of collusion only incur minimal leakages.
Attacks: Collusion between B and O reveals both U's identity and location. Note that this is the case for any MVNO-based architecture where B knows U's identity (for offering identity-based services) and O knows U's location (as it provides connectivity). Therefore, we focus on showing what other forms of collusion also impair user location privacy. For O, colluding with participants other than B does not provide it with extra information on U's identity or trajectories. For B, however, it can gain additional knowledge regarding U's location by colluding with (i) other users or (ii) other operators. The former is due to the use of session groups. Specifically, B knows that sessions in a session group belong to the same O, hence that users of these sessions have visited the same location at a similar time. Therefore, if some users who share session groups with U reveal their locations to B, then B knows U's location via such collusion. We call these "indirect location leakages". The latter is due to operators sharing the session table in the claiming phase. Specifi-

cally, B now effectively has a "smaller" session table consisting of only sessions from non-colluding operators, which is detrimental to the privacy guarantee provided by aggregate claiming.

Impact & Mitigation: While brokers gain extra user location information via collusion with other users or operators, the actual impacts are minimal and can be further reduced with different mitigation strategies. First, the impact of indirect location leakages is bounded by the size of session groups, which in turn depends on how fast the zk-SNARK backend is. Fortunately, even with a single-core backend, aggregate claiming can scale to large deployments with a session group that lasts as little as 20 s (§4.6.2). One could adopt faster backends like distributed zk-SNARK [328] to further reduce the size of session groups and hence leakages. Secondly, since the obfuscation effect of aggregate claiming is *exponential* w.r.t. the size of the session table (§4.5.1.2), a smaller table still grants sufficient protections. One could use a longer billing period to ensure a large enough session table even with collusion.

4.6 Implementation and Evaluation

In this section, we present the implementation of our LOCA prototype (§4.6.1) and investigate the two key questions regarding the feasibility of LOCA: (i) can LOCA scale to realistic deployment sizes? and (ii) how much overhead does LOCA introduce compared to existing cellular protocols? We answer the first question by performing a scalability analysis of the privacy building blocks (§4.6.2); and the second by conducting a performance analysis with wide-area experiments (§4.6.3).

4.6.1 Implementation

We prototyped LOCA as an extension to the CellBricks system [51] which is itself built from open source cellular platforms (Magma [99] and srsLTE [278]). We extended the operator and broker modules with the following: (i) the token generation and verification procedures implemented with rsablind [78]; (ii) the anonymous communication channel between the operator and broker implemented with Torsocks [298, 119] and NordVPN [228] and (iii) the claiming procedure implemented with Pequin [310, 247] that has a single-core libsnark [175] as the zk-SNARK backend. In total, our extension includes 478 LoC in C (for claiming), 144 LoC in Go (for unlinkable token), and 16 LoC shell scripts (for anonymous communication and various setup). We prototyped LOCA with these languages as they were used in the original implementations that we extended. We built a testbed with two x86 machines: one as the user's device and the other as the operator's cell and core. We connect each machine to an SDR device (USRP B205-mini [97]) for radio connectiv-

ity. Lastly, the broker's service is deployed on AWS instances [35].

As an opt-in service, LOCA can be incrementally deployed and adopted starting with a small number of LOCA-compatible users, brokers, and operators: users can have partial privacy by signing up with brokers that support LOCA and by using LOCA-based operators when available and falling back to legacy ones otherwise. We leave an evaluation of the privacy benefits under incremental adoption to future work.

4.6.2 Scaling analysis

LOCA must be able to scale to a large number of operators serving many users. Therefore, we evaluate whether the three privacy building blocks that we adopt can scale to large deployments, on the order of today's large MVNOs.

4.6.2.1 Blind signature

Blind signatures are used for generating and verifying unlinkable tokens. We measure a blind signature generation throughput of 522/sec and a verification throughput of 17202/sec on a 2.6GHz Intel I7-8850H CPU. These single-core throughputs are significant: generating 50 tokens for 10 operators per second. Moreover, brokers can easily achieve higher throughput with more cores or machines, hence we conclude that scaling blind signature operations will not be a problem.

4.6.2.2 Anonymous communication

For anonymous communication schemes in LOCA, an operator must have sufficient network capacity to send attachment requests to brokers. We measure the average network throughput of a Tor circuit to be 4.2 Mbps uplink and 6.1 Mbps downlink (consistent with Tor's reports [194]). Such throughput can support ≈ 400 attachment requests per second. Operators can easily scale up the throughput by establishing multiple Tor circuits with the same token. Alternatively, operators can use other anonymous communication schemes that have higher network throughput, such as VPNs (§4.6.3).

4.6.2.3 Aggregate Claiming with zk-SNARK

zk-SNARK has a long setup and proving time [328]. Given our aggregate claiming protocol is based on zk-SNARK, we evaluate whether the protocol can scale to large deployments. Since the generated keys are reused across billing cycles, zk-SNARK setup is performed offline only once, which excludes the setup time from the performance critical path. Hence we focus on the zk-SNARK proving time, which is invoked by each operator at every billing cycle to claim its session groups.

As noted in §4.4.5, LOCA allows claiming sessions in groups with a configurable size: smaller session groups offer stronger privacy guarantees as they minimize indirect location leakages. However, due to the slow zk-SNARK proving, operators may need to use large session groups so that they can *claim session groups faster than the rate of session group creation* and not develop a backlog of unclaimed sessions, at the cost of some privacy loss. To evaluate the amount of such privacy loss, we answer the following question: how small can session groups be while allowing operators to claim them fast enough? Specifically, we would like to obtain a *lower bound* for the average duration of a session group T[11]. As we will show next, even a single-core zk-SNARK implementation is fast enough to support session groups of small T, hence aggregate claiming will not be a scalability bottleneck.

As noted, we let K represent the maximum number of session groups an operator can claim and N represent the maximum number of session groups in the broker's session group table. If we denote $P(K, N)$ as the time it takes for zk-SNARK to prove the circuit of the claiming function parameterized by K and N, we have the following lower bound for T:

$$T \geq \frac{P(K, N)}{K}$$

To obtain the lower bound, we evaluate the proving time of our implementation for the claiming procedure $P(K, N)$. As mentioned in §4.4.5, proof-based verifiable computation has a compiler frontend and a zk-SNARK backend. Therefore, to evaluate $P(K, N)$, we need to answer two questions: (i) for a given K and N, how many constraints will the claiming function be compiled into? and (ii) how long will zk-SNARK take to prove these circuits of different sizes?

To answer the first question, we compile claiming functions with different Ks and Ns, and find the following formula that closely matches the numbers of constraints:

$$\#\ of\ constraints = K * (128 * N + 35394)$$

Terms in this formula are tied to the logic of the claiming function. As mentioned in §4.4.5, the claiming function contains two steps: (i) calculating hashes of the K provided secrets, and (ii) iterating through the N rows in the session table, checking whether the hash matches with one of the K precomputed hashes and adding it to the aggregate if so. Therefore, step (i) generates $35394 * K$ constraints, where 35394

[11] One can calculate the average number of sessions in a group as $T*r$, where r is the deployment-dependent rate of attachments for an operator.

is the number of constraints for computing a single SHA256 hash, consistent with prior work [171]; step (ii) contains an outer loop of N and an inner loop of K, which gets unrolled by the compiler into $128 * K * N$ constraints. Therefore, for a large enough N, the number of constraints scale almost linearly with $K * N$.

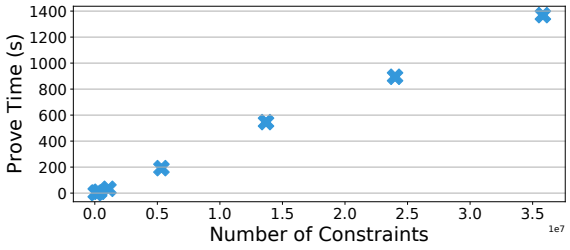

Figure 4.7: Proving time under varied number of constraints.

To answer the second question, we evaluate the proving time of compiled circuits with different numbers of constraints with a single-core libsnark backend on a 2.5GHz Intel 8259CL CPU. As shown in Fig 4.7, consistent with prior work [254, 328], the proving time increases linearly with the number of constraints: about 38 seconds per 1 million constraints.

Since we have shown that (i) the number of constraints of the claiming circuit increases linearly w.r.t $K * N$, and (ii) the proving time is linear w.r.t the number of constraints, we know that the *zk-SNARK proving time increases linearly w.r.t* $K * N$, *i.e.,* $P(K, N) = O(K * N)$. The constant factor c depends on the specific compiler frontends and zk-SNARK backends. For our implementations, $c \approx 128 * 38\ us = 4.894\ ms$. Therefore,

$$T \geq \frac{P(K,N)}{K} \approx \frac{c * K * N}{K} = c * N$$

This means that the lower bound on the duration of the session group grows *linearly w.r.t.* N. As stated earlier, we are mostly interested in cases of large Ns (*i.e.,* larger numbers of smaller session groups) as these lead to stronger privacy guarantees (§4.5.1.2). Fortunately, even with only the single-core libsnark backend, the lower bound of T for large N is reasonably small. As an example, the largest circuit that we evaluated (K=64, N=4096) has proving time $P(64, 4096)$=1369 s; this translates to a lower bound of T=$P(64, 4096)/64$=21.4 s. The asymptotic expression of $T = c * N = 4.864\ ms * 4096 \approx 20\ s$ matches with the measurement. The gap is due to ignoring the $35394 * K$ term, which will reduce as N goes even bigger.

Therefore, with $N = 4096$, the smallest session group that a single-core zk-SNARK can support has a duration of 20 s. This means that users who attach more

Figure 4.8: Average attachment latency of Magma baseline (BL), CellBricks (CB), LOCA-VPN and LOCA-Tor.

than 20 s apart cannot reveal any information about each other's location, even if one user's location were leaked to the broker. We do not evaluate circuits with more than $35M$ constraints due to the scaling limit of the libsnark implementation. Recent work [328] on distributed zk-SNARK allows faster proving of much larger circuits, the evaluation of which is left to future work.

4.6.3 Performance analysis

Lastly, we would like to understand the performance that users receive with LOCA. Procedures like token generation and aggregate claiming happen off the critical path of users receiving services, thus do not affect user experience. Instead, we focus on the attachment procedure, since LOCA's attachment is both more complex and more frequent than today's protocols. We thus measure the additional latency overhead that LOCA adds to the attachment procedure.

We replicate the wide-area test setup from CellBricks [202]: the user equipment and the operator's cell and cellular core are always located in our local testbed, and we run experiments with the subscriber database (in the case of Magma) and the broker hosted on AWS EC2 [26]. This matches deployment practice where certain core network components are run in the carrier's datacenter. For each setup, we repeat the same attachment request using different cellular implementations 100 times and report the average performance.

Fig.4.8 shows the attachment latency after removing the time spent in lower radio layers (*i.e.,* RRC layer and below) for different placements of the subscriber database and broker. We compare four schemes: (i) unmodified Magma (baseline, denoted BL, that captures today's cellular architecture), (ii) CellBricks (denoted CB), LOCA's attachment protocol with (iii) VPN (denoted LOCA-VPN) and (iv) Tor (denoted LOCA-Tor) as the anonymous communication channel.

We make two observations from these results. First, the choice of anonymous communication scheme introduces a tradeoff between trust assumptions and attachment latency: LOCA-VPN requires trusting the VPN provider but achieves faster attachments than LOCA-Tor. In fact, LOCA-VPN is only 5 to 15 ms slower than CellBricks and still faster than today's attachment (*i.e.,* Magma). The reason we outperform Magma's attachment latency is because today's attachment procedure requires two round trips to the cloud, while CellBricks optimized this process to a single round-trip; since we build on CellBricks, we inherit this performance gain.

Our second observation is that even the slower LOCA-Tor is sufficiently fast for periodic reattachments: prior work [202] shows that attachment latencies of up to 500 ms have a minimal impact on application performance, even when users reattach on a per-tower basis. Hence LOCA-Tor, with a constant 400 ms latency due to the overhead of Tor [194], can support frequent reattachments with minimal disruptions.

4.7 Discussion

Viewing LOCA as a first step towards privacy-preserving cellular infrastructure, we next discuss two notable areas for improvement and potential directions to achieving them: (i) supporting beyond semi-honest and non-colluding participants, and (ii) improving non-privacy-related aspects of LOCA.

4.7.1 Beyond semi-honest and non-colluding

As stated in §4.3.1, there are both financial and legal reasons for brokers and operators to be semi-honest and not collude. However, relaxing these assumptions can certainly facilitate adoption. We next discuss directions towards such relaxation.

Semi-honest: LOCA suffers from privacy leakages in the face of various active attacks, *e.g.,* those based on out-of-protocol information (§4.5.2), which restricts it to semi-honest participants. We see two orthogonal directions towards supporting more aggressive participants. First, one could adopt specific defense mechanisms for different attacks (*e.g.,* traffic analysis, device fingerprinting) that have been proposed in prior work [314, 327, 158, 100, 346, 140]. LOCA, as a protocol-layer solution, can coexist with these mechanisms. Second, instead of averting attacks, one can detect these attacks and punish the misbehaving participants. The detection mechanism can involve multiple parties. For instance, operator over-reporting usage can be detected by brokers cross comparing the operator's reports with the ones from users. For the punishment mechanism, a promising approach is to build up a reputation system [202], where misbehaviors are factored into participant's reputation scores. Participants with poor reputation then receive degraded treatments: *e.g.,* a broker can decline to authorize an operator in the registration phase (§4.4.1). Such an approach is appealing in the cellular context, where brokers and operators need to remain operational for long enough to see a profit, allowing their track records to be built up.

Non-colluding: As elaborated in §4.5.3, except for direct collusion between brokers and operators that serve the user, other forms of collusion only incur minimal leakages in LOCA. An interesting question is then whether we could relax this requirement of no broker-operator collusion. Intuitively, preserving location privacy

with *arbitrary* collusion seems unattainable: if a broker colludes with all the operators, it easily knows both the user's identity and all of her locations. Instead, we believe it is both feasible and interesting to investigate whether one could provide *partial* privacy guarantee if only a *subset* of operators collude with brokers. Under such a scenario, the coverage of non-colluding operators forms a region where little location information is revealed. Such a region is referred to as a *mix zone* and widely studied for location privacy in non-cellular contexts [42, 43, 124], and future work could leverage the insights of these work for cellular privacy.

4.7.2 Beyond privacy

Another area for improvement is the design and evaluation on non-privacy-related aspects of LOCA, such as performance and operational support. For performance, in §4.6.3, we measure LOCA's attachment latency to be less than 500 ms even with slower anonymous communication channel (*i.e.,* Tor), which was evaluated in [202] to have minimal performance impacts to applications like voice calls, video streaming and web browsing. It would be interesting to evaluate on more challenging applications such as video conferencing. Moreover, besides reducing trajectory leakages (§4.5.1.3), make-before-break handovers are expected to have better performance as well, the evaluation of which in LOCA is left to future work.

For operational support, LOCA supports tasks like identity-based services by having brokers offload these tasks to authorized but identity unknown operators (§4.4.4). However, there might be tasks that require knowledge of the operator's identity, such as recording misbehaving operators (for the aforementioned reputation system) and performing on-site inspections. To support these tasks, one potential approach is to involve a trusted third party when generating unlinkable tokens (§4.4.1). The goal is that upon legitimate requests, this third party can later assist in revealing the operator's identity for a token. One promising direction towards achieving this goal is to extend the registration phase with cryptographic constructs like secure multi-party computation (MPC) [115, 88, 344].

4.8 Related Work

Cellular: There has been extensive prior work on mitigating privacy violations by third parties other than network operators [283, 144, 174, 272, 31, 165, 287, 303, 111]. Our work instead focuses on protecting a user's location privacy from the network operator itself. To our knowledge, PGPP [265] is the only prior work that systematically studies this issue. As discussed earlier, PGPP adopts a different approach based on hiding users' identities from the network operator, which however compromises the network's ability to provide identity-based services and does not

address the issue of trajectory-related leakages. One advantage of PGPP is higher tolerance for collusions, as it hides user's identity from both operators and brokers. However, it also assumes semi-honest participants who will not actively thwart its privacy mechanisms.

CellBricks [202] is a new cellular architecture that aims to democratize cellular access by enabling users to easily leverage small-scale operators. LOCA borrows the idea of user-driven mobility, although we use it for privacy reasons while CellBricks requires it to give users the ability to dynamically select an operator of their choice. CellBricks does not address the issue of location privacy and hence is similar to 3GPP protocols in this regard. In fact, we note that the importance of hiding O's identity from B is greater under the CellBricks vision of larger numbers of smaller-scale operators.

General location privacy: There is extensive prior work on location privacy in non-cellular contexts [173, 197, 320, 89, 339, 275, 42]. These reveal four general methods for protecting location privacy: (i) regulatory strategies – government rules to regulate the use of personal information; (ii) privacy policies – trust-based agreements between individuals and whoever is receiving their location data; (iii) anonymity – use a pseudonym and create ambiguity by grouping with other people. (iv) obfuscation – temporal or spatial degradation of the location data. Regulatory strategies and privacy policies are orthogonal to computational countermeasures like techniques adopted in LOCA. In the cellular context, neither obfuscation nor anonymity is desirable: obfuscation is not feasible, because a user's location data is generated by the infrastructure, the temporal or spatial resolution of which is not determined by the user; anonymity is the approach adopted by PGPP [265] which, as discussed earlier, compromises on identity-based services. LOCA exploits the unique role of brokers and adopts a novel approach to preserving location privacy while supporting identity-based services. LOCA's approach of strategically hiding different pieces of information from each party has been investigated for preserving privacy in other contexts as well, such as Apple's private relay [77].

Applications of LOCA's privacy building blocks: Blind signatures have been applied for e-voting [146, 196, 166]. Anonymous communication has been used in social networking and web browsing [298, 137, 108]. Proof-based verifiable computation has been used in outsourced computing [176, 72, 62]. LOCA synthesizes these building blocks to support cellular procedures like attachment and aggregate claiming.

4.9 Conclusion

We presented LOCA, a novel cellular architecture that provides location privacy while supporting identity-based services such as usage-based billing, QoS, and lawful intercept.

We view our work as a first step towards enabling privacy-preserving communication infrastructure and hope that future work will extend our design to address additional threat models and reduced overheads, as well as explore the applicability of LOCA's design to other access technologies.

Bibliography

[1] 3GPP. LTE;Telecommunication management; Performance Management (PM); Performance measurements Evolved Universal Terrestrial Radio Access Network (E-UTRAN). Technical Specification (TS) 32.425, 3rd Generation Partnership Project (3GPP), 08 2016. Version 13.5.0.

[2] 3GPP. Lte;telecommunication management; performance management (pm); performance measurements evolved universal terrestrial radio access network (e-utran). Technical Specification (TS) 32.425, 3rd Generation Partnership Project (3GPP), 08 2016. Version 13.5.0.

[3] 3GPP. 5g; security architecture and procedures for 5g system. Technical Specification (TS) 33.501, 3rd Generation Partnership Project (3GPP), 10 2018. Version 15.2.0.

[4] 3GPP. Lawful Interception (LI);Handover interface for the lawful interception of telecommunications traffic. `https://www.etsi.org/deliver/etsi_es/201600_201699/201671/03.02.01_50/es_201671v030201m.pdf`, 2018.

[5] 3GPP. Lawful interception architecture and functions. Technical Specification (TS) 33.107, 3rd Generation Partnership Project (3GPP), 07 2019. Version 15.6.0.

[6] 3GPP. Policy and charging control architecture (3GPP TS 23.203 version 15.5.0 Release 15). `https://www.etsi.org/deliver/etsi_ts/123200_123299/123203/15.05.00_60/ts_123203v150500p.pdf`, 2019.

[7] 3GPP. Radio Link Control (RLC) protocol specification (3GPP TS 36.322 version 14.1.0 Release 14). `https://www.etsi.org/deliver/etsi_ts/136300_136399/136322/14.01.00_60/ts_136322v140100p.pdf`, 2019.

[8] 3GPP. 5g; security architecture and procedures for 5g system. Technical Specification (TS) 23.501, 3rd Generation Partnership Project (3GPP), 10 2020. Version 16.6.0.

[9] 3GPP. Lte; 3gpp system architecture evolution (sae); security architecture. Technical Specification (TS) 33.401, 3rd Generation Partnership Project (3GPP), 03 2020. Version 15.11.0.

[10] 3GPP. Non-Access Stratum. `https://www.3gpp.org/technologies/keywords-acronyms/96-nas`, 2020.

[11] 3GPP. Non-Access Stratum (NAS) protocpol for 5G System (5GS);. Technical Specification (TS) 24.501, 3rd Generation Partnership Project (3GPP), 01 2020. Version 15.6.0.

[12] 3GPP. Nr and ng-ran overall description. Technical Specification (TS) 38.300, 3rd Generation Partnership Project (3GPP), 07 2020. Version 16.2.0.

[13] 3GPP. 3GPP RAN Sharing. `https://www.3gpp.org/news-events/1592-gush`, 2021.

[14] 3GPP. 3GPP Specification series. `https://www.3gpp.org/DynaReport/36-series.htm`, 2021.

[15] Africa Mobile Networks. `http://www.africamobilenetworks.com`. Retrieved 6/2020.

[16] Agentschap Telecom. Regeling gebruik van frequentieruimte zonder vergunning 2008. `http://wetten.overheid.nl/BWBR0023553/volledig/geldigheidsdatum\_23-04-2013`, April 2013.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] CBRS Alliance. CBRS Alliance. `https://www.cbrsalliance.org`, 2021.

[24] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[25] Emmanuel Amaro, Stephanie Wang, Aurojit Panda, and Marcos K Aguilera. Logical memory pools: Flexible and local disaggregated memory. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 25–32, 2023.

[26] Amazon Web Service. Aws ec2 regions. `https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.RegionsAndAvailabilityZones.html`, 2021.

[27] 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.

[28] 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.

[29] Apple. Privacy - apple. `https://www.apple.com/privacy/`, 2021.

[30] Apple. Wi-Fi network roaming with 802.11k, 802.11r, and 802.11v on iOS. `https://support.apple.com/en-us/HT202628`, 2021.

[31] Myrto Arapinis, Loretta Mancini, Eike Ritter, Mark Ryan, Nico Golde, Kevin Redon, and Ravishankar Borgaonkar. New privacy issues in mobile telephony: fix and verification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 205–216, 2012.

[32] 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.

[33] AT&T. AT&T private cellular networks. `https://www.business.att.com/products/att-private-cellular-networks.html`, 2020.

[34] AT&T. Deep packet inspection explained. `https://cybersecurity.att.com/blogs/security-essentials/what-is-deep-packet-inspection`, 2021.

[35] AWS. Amazon ec2 instance types. `https://aws.amazon.com/ec2/instance-types/`.

[36] 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.

[37] 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.

[38] 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.

[39] Arijit Banerjee, Rajesh Mahindra, Karthik Sundaresan, Sneha Kasera, Kobus Van der Merwe, and Sampath Rangarajan. Scaling the LTE Control-Plane

for Future Mobile Access. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 19. ACM, 2015.

[40] Leda Bargiotti, Inge Gielis, Bram Verdegem, Pieter Breyne, Francesco Pignatelli, Paul Smits, Ray Boguslawski, et al. Guidelines for public administrations on location privacy: European union location framework. Technical report, Joint Research Centre (Seville site), 2016.

[41] 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.

[42] Alastair R Beresford and Frank Stajano. Location privacy in pervasive computing. *IEEE Pervasive computing*, 2(1):46–55, 2003.

[43] Alastair R Beresford and Frank Stajano. Mix zones: User privacy in location-aware services. In *IEEE Annual conference on pervasive computing and communications workshops, 2004. Proceedings of the Second*, pages 127–131. IEEE, 2004.

[44] 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.

[45] Naga Bhushan, Junyi Li, Durga Malladi, Rob Gilmore, Dean Brenner, Aleksandar Damnjanovic, Ravi Teja Sukhavasi, Chirag Patel, and Stefan Geirhofer. Network densification: the dominant theme for wireless evolution into 5g. *IEEE Communications Magazine*, 52(2):82–89, 2014.

[46] 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.

[47] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, volume 1, pages 126–134. IEEE, 1999.

[48] Camila CS Caiado and Pushpa N Rathie. Polynomial coefficients and distribution of the sum of discrete uniform variables. In *Eighth Annual Conference of the Society of Special Functions and their Applications, Pala, India, Society for Special Functions and their Applications*, 2007.

[49] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 79–92, 2021.

[50] 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.

[51] CellBricks. Cellbricks. `https://cellbricks.github.io/`, 2021.

[52] CellBricks. Cellbricks's artifacts. `https://cellbricks.github.io/artifact-sigcomm21/`, 2021.

[53] CellBricks. Cellbricks's technical report. `https://cellbricks.github.io/cellular_sigcomm_extended.pdf`, 2021.

[54] Mobile Internet Resource Center. Top pickfeatured overview: postpaid consumer plans by verizon (cellular data plans). `https://www.rvmobileinternet.com/gear/the-verizon-plan/`, 2021.

[55] David Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer, 1983.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] Yung-Chih Chen, Yeon-sup Lim, Richard J Gibbens, Erich M Nahum, Ramin Khalili, and Don Towsley. A measurement-based study of multipath tcp performance over wireless networks. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 455–468, 2013.

[61] 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.

[62] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Carlos Cid. Multi-client non-interactive verifiable computation. In *Theory of Cryptography Conference*, pages 499–518. Springer, 2013.

[63] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[64] Cisco. Cisco Annual Internet Report (2018–2023) White Paper. `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html`, 2020.

[65] David D Clark and John T Wroclawski. The personal router whitepaper. Technical report, MIT Technical Report, 2000.

[66] 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.

[67] Red Compartida. Red Compartida. `http://www.sct.gob.mx/red-compartida/index-eng.html`, 2021.

[68] Congreso de la Republica del Peru. Ley No. 30083. `http://www.leyes.congreso.gob.pe/Documentos/Leyes/30083.pdf`, 2013.

[69] Gene Connolly, Anatoly Sachenko, and George Markowsky. Distributed traceroute approach to geographically locating ip devices. In *Second IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2003. Proceedings*, pages 128–131. IEEE, 2003.

[70] Vertical Consultants. Cell tower industry facts & figures 2020. `https://www.celltowerleaseexperts.com/cell-tower-lease-news/cell-tower-industry-facts-figures-2016/`, 2020.

[71] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[72] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy*, pages 253–270. IEEE, 2015.

[73] Joseph Cox. I gave a bounty hunter $300. then he located our phone. `https://www.vice.com/en/article/nepxbz/i-gave-a-bounty-hunter-300-dollars-located-phone-microbilt-zumigo-tmobile`, 2019.

[74] Joseph Cox. Stalkers and debt collectors impersonate cops to trick big telecom into giving them cell phone location data. `https://www.vice.com/en/article/panvkz/stalkers-debt-collectors-bounty-hunters-impersonate-cops-phone-location-data`, 2019.

[75] Cricket. Cricket wireless. `https://www.cricketwireless.com/`, 2021.

[76] Andrei Croitoru, Dragos Niculescu, and Costin Raiciu. Towards wifi mobility without fast handover. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 219–234, 2015.

[77] Jason Cross. icloud+ private relay faq: Everything you need to know. `https://www.macworld.com/article/348965/icloud-plus-private-relay-safari-vpn-encryption-privacy.html`, 2021.

[78] CryptoBallot. Rsa blind signing using a full domain hash. `https://github.com/cryptoballot/rsablind`, 2021.

[79] 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.

[80] Boris Danev, Davide Zanetti, and Srdjan Capkun. On physical-layer identification of wireless devices. *ACM Computing Surveys (CSUR)*, 45(1):1–29, 2012.

[81] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment*, 4(8):494–505, 2011.

[82] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. {DORY}: An encrypted search system with distributed trust. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1101–1119, 2020.

[83] Quentin De Coninck and Olivier Bonaventure. Multipathtester: Comparing mptcp and mpquic in mobile environments. In *2019 Network Traffic Measurement and Analysis Conference (TMA)*, pages 221–226. IEEE, 2019.

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

[85] Omkar Dharmadhikari. 5G Link Aggregation with Multipath TCP (MPTCP). `https://www.cablelabs.com/5g-link-aggregation-mptcp`, 2019.

[86] 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.

[87] Paul Drongowski, Lei Yu, Frank Swehosky, Suravee Suthikulpanit, and Robert Richter. Incorporating instruction-based sampling into amd codeanalyst. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 119–120. IEEE, 2010.

[88] Wenliang Du and Mikhail J Atallah. Secure multi-party computation problems and their applications: a review and open problems. In *Proceedings of the 2001 workshop on New security paradigms*, pages 13–22, 2001.

[89] Matt Duckham and Lars Kulik. Location privacy and location-aware computing. *Dynamic & mobile GIS: investigating change in space and time*, 3:35–51, 2006.

[90] 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.

[91] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 1–14, 2019.

[92] 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.

[93] Ehcache. Ehcache — ehcache.org. `https://www.ehcache.org/`. [Accessed 09-12-2024].

[94] 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.

[95] Ericsson. Evolving cellular IoT for industry digitalization. `https://www.ericsson.com/en/internet-of-tRFWirelessWorldhings/iot-connectivity/cellular-iot`, 2020.

[96] ETSI. Lawful intercept ETSI. `https://www.etsi.org/technologies/lawful-interception`, 2020.

[97] Ettus. Usrp b205mini. `https://www.ettus.com/all-products/usrp-b205mini-i/`, 2020.

[98] EventHelix. Attachment Call Flow. `https://www.eventhelix.com/lte/attach/lte-attach.pdf`, 2019. Accessed: 2020-04-29.

[99] Facebook. Magma. `https://www.magmacore.org/`, 2021.

[100] Kassem Fawaz, Kyu-Han Kim, and Kang G Shin. Protecting privacy of {BLE} device users. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1205–1221, 2016.

[101] FFmpeg. A complete, cross-platform solution to record, convert and stream audio and video. `https://ffmpeg.org/`, 2020.

[102] Claude Fischer. *America Calling: A Social History of the Telephone to 1940*. University of California Press, Berkeley, CA, 1992.

[103] Alan Ford, Costin Raiciu, Mark Handley, Olivier Bonaventure, and C Paasch. Rfc 6824: Tcp extensions for multipath operation with multiple addresses. *Internet Engineering Task Force*, 2013.

[104] R.D. Foreman. Scale economies in cellular telephony: Size matters. Journal of Regulatory Economics 16, 297–306 (1999), `https://doi.org/10.1023/A:1008131223498`, 1999.

[105] Xenofon Foukas, Georgios Patounas, Ahmed Elmokashfi, and Mahesh K Marina. Network slicing in 5g: Survey and challenges. *IEEE Communications Magazine*, 55(5):94–100, 2017.

[106] Lorenzo Franceschi-Bicchierai. Why Monopolistic Telecoms Threaten Internet Equality. `https://mashable.com/2012/09/17/telecom-monopoly-internet-equality/`, 2012.

[107] 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.

[108] Eran Gabber, Phillip B Gibbons, Yossi Matias, and Alain Mayer. How to make personalized web browsing simple, secure, and anonymous. In *International Conference on Financial Cryptography*, pages 17–31. Springer, 1997.

[109] Hernan Galperin and François Bar. The Microtelco Opportunity: Evidence from Latin America. In *Information Technologies and International Development*, volume 3, 2006.

[110] Ruchi Garg. Dual active protocol stack handover (daps ho). `https://www.linkedin.com/pulse/dual-active-protocol-stack-handover-daps-ho-ruchi-garg/`, 2021.

[111] M Køien Geir et al. Privacy enhanced mutual authentication in lte. In *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 614–621. IEEE, 2013.

[112] Torsten J Gerpott, Wolfgang Rams, and Andreas Schindler. Customer retention, loyalty, and satisfaction in the german mobile cellular telecommunications market. *Telecommunications Policy*, 25(4):249 – 269, 2001.

[113] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. Evendb: Optimizing key-value storage for spatial locality. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[114] Hadi Givehchian, Nishant Bhaskar, Eliana Rodriguez Herrera, Héctor Rodrigo López Soto, Christian Dameff, Dinesh Bharadia, and Aaron Schulman. Evaluating physical-layer ble location tracking attacks on mobile devices. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1690–1704. IEEE, 2022.

[115] Oded Goldreich. Secure multi-party computation. *Manuscript. Preliminary version*, 78(110), 1998.

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

[117] Google. Google-Fi. `https://fi.google.com/about/`, 2021.

[118] Swish Goswami. The rising concern around consumer data and privacy. `https://www.forbes.com/sites/forbestechcouncil/2020/12/14/the-rising-concern-around-consumer-data-and-privacy/?sh=6e6200a6487e`, 2020.

[119] David Goulet. Torsocks. `https://github.com/dgoulet/torsocks`, 2021.

[120] GSA. LTE in Unlicensed and Shared Spectrum: Trials, Deployments and Devices. `https://gsacom.com/paper/lte-unlicensed-shared-spectrum-2/`, 2019.

[121] GSM Association. Unlocking Rural Coverage: Enablers for commercially sustainable mobile network expansion, 7 2016.

[122] GSMA. Enabling neutral host: CCS case study. `https://www.gsma.com/futurenetworks/wp-content/uploads/2018/09/180920-CCS_GSMA_Case_Study-FINAL_NE-Modelling-removed.pdf`, 2020.

[123] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[124] Nan Guo, Linya Ma, and Tianhan Gao. Independent mix zone for location privacy in vehicular networks. *IEEE Access*, 6:16842–16850, 2018.

[125] 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.

[126] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 91–107, 2016.

[127] 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.

[128] Shaddi Hasan, Mary Claire Barela, Matthew Johnson, Eric Brewer, and Kurtis Heimerl. Scaling Community Cellular Networks with CommunityCellularManager. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'19, pages 735–750, 2019.

[129] 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.

[130] Red Hat. Huge pages and transparent Huge Pages — Red Hat Product Documentation — docs.redhat.com. `https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-memory-transhuge`. [Accessed 03-12-2024].

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

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

[133] Kurtis Heimerl, Shaddi Hasan, Kashif Ali, Eric Brewer, and Tapan Parikh. Local, Sustainable, Small-Scale Cellular Networks. In *Proceedings of the Sixth International Conference on Information and Communication Technologies and Development*, ICTD '13, pages 2–12, Cape Town, South Africa, 2013. ACM.

[134] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[135] 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.

[136] HLS.js. Javascript hls client using media source extension. `https://github.com/video-dev/hls.js/`, 2021.

[137] Nguyen Phong Hoang and Davar Pishva. Anonymous communication and its importance in social networking. In *16th International Conference on Advanced Communication Technology*, pages 34–39. IEEE, 2014.

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

[139] https://community.intel.com/t5/user/viewprofilepage/user id/243808. Breaking the Memory Wall with Compute Express Link (CXL) — community.intel.com. `https://community.intel.com/t5/Blogs/Tech-Innovation/Data-Center/Breaking-the-Memory-Wall-with-Compute-Express-Link-CXL/post/1594848`. [Accessed 03-12-2024].

[140] Jinsong Hu, Shihao Yan, Feng Shu, Jiangzhou Wang, Jun Li, and Yijin Zhang. Artificial-noise-aided secure transmission with directional modulation based on random frequency diverse arrays. *IEEE Access*, 5:1658–1667, 2017.

[141] Huawei. Huawei privacy. `https://consumer.huawei.com/en/privacy/`, 2021.

[142] 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.

[143] 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.

[144] Syed Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. Lteinspector: A systematic approach for adversarial testing of 4g lte. In *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018.

[145] Syed Rafiul Hussain, Mitziu Echeverria, Omar Chowdhury, Ninghui Li, and Elisa Bertino. Privacy attacks to the 4g and 5g cellular paging protocols using side channel information. In *NDSS*, 2019.

[146] Subariah Ibrahim, Maznah Kamat, Mazleena Salleh, and Shah Rizan Abdul Aziz. Secure e-voting with blind signature. In *4th National Conference of Telecommunication Technology, 2003. NCTT 2003 Proceedings.*, pages 193–197. IEEE, 2003.

[147] Roy IFielding, Yves Lafon, and JulianETF Reschke. Rfc7233. `https://tools.ietf.org/html/rfc7233`, 2014.

[148] Intel. Timed Process Event-Based Sampling (TPEBS) — intel.com. `https://www.intel.com/content/www/us/en/developer/articles/technical/timed-process-event-based-sampling-tpebs.html`. [Accessed 03-12-2024].

[149] Iperf. iperf network benchmarks. `https://iperf.fr/`, 2021.

[150] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[151] 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.

[152] 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.

[153] 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.

[154] 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.

[155] JGraphT. JGraphT — jgrapht.org. `https://jgrapht.org/`. [Accessed 10-12-2024].

[156] Xin Jin, Li Erran Li, Laurent Vanbever, and Jennifer Rexford. SoftCell: Scalable and Flexible Cellular Core Network Architecture. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, pages 163–174. ACM, 2013.

[157] 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.

[158] Marc Juarez, Mohsen Imani, Mike Perry, Claudia Diaz, and Matthew Wright. Toward an efficient website fingerprinting defense. In *European Symposium on Research in Computer Security*, pages 27–46. Springer, 2016.

[159] Seny Kamara, Payman Mohassel, and Mariana Raykova. Outsourcing multiparty computation. *IACR Cryptol. Eprint Arch.*, 2011:272, 2011.

[160] 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.

[161] Stefan Karlsson. JEP 439: Generational ZGC — openjdk.org. `https://openjdk.org/jeps/439`. [Accessed 10-12-2024].

[162] 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.

[163] Kate Kaye. The $24 billion data business that telcos don't want to talk about. `https://adage.com/article/datadriven-marketing/24-billion-data-business-telcos-discuss/301058?mod=article_inline`, 2019.

[164] Meghan Keneally. How the T-Mobile and Sprint merger could impact consumers. `https://abcnews.go.com/Business/mobile-sprint-merger-impact-consumers/story?id=54826385`, 2018.

[165] Mohammed Shafiul Alam Khan and Chris J Mitchell. Trashing imsi catchers in mobile networks. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 207–218, 2017.

[166] Malik Sikandar Hayat Khiyal, Aihab Khan, Saba Bashir, Farhan Hassan Khan, and Shaista Aman. Dynamic blind group digital signature scheme in e-banking. *International Journal of Computer and Electrical Engineering*, 3(4):514–519, 2011.

[167] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for {Multi-Tiered} memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 715–728, 2021.

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

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

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

[171] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 944–961. IEEE, 2018.

[172] KrebsOnSecurity. Tracking firm locationsmart leaked location data for customers of all major u.s. mobile carriers without consent in real time via its web site. `https://krebsonsecurity.com/2018/05/tracking-firm-locationsmart-leaked-location-data-for-customers-of-all-major-u-s-mobile-carriers-in-real-time-via-its-web-site/`, 2018.

[173] John Krumm. A survey of computational location privacy. *Personal and Ubiquitous Computing*, 13(6):391–399, 2009.

[174] Denis Foo Kune, John Koelndorfer, Nicholas Hopper, and Yongdae Kim. Location leaks on the gsm air interface. *ISOC NDSS (Feb 2012)*, 2012.

[175] SCIPR Lab. Libsnark. `https://github.com/scipr-lab/libsnark`, 2021.

[176] Junzuo Lai, Robert H Deng, HweeHwa Pang, and Jian Weng. Verifiable computation on outsourced encrypted data. In *European Symposium on Research in Computer Security*, pages 273–291. Springer, 2014.

[177] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 183–196, 2017.

[178] 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.

[179] Sangwon Lee, Sylvia M Chan-Olmsted, and Hsiao-Hui Ho. The emergence of mobile virtual network operators (mvnos): An examination of the business strategy in the global mvno market. *The International Journal on Media Management*, 10(1):10–21, 2008.

[180] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 17–34, 2023.

[181] Baptiste Lepers and Willy Zwaenepoel. Johnny cache: the end of {DRAM} cache conflicts (in tiered main memory systems). In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 519–534, 2023.

[182] Jure Leskovec. Stanford Large Network Dataset Collection — snap.stanford.edu. `https://snap.stanford.edu/data/`. [Accessed 11-04-2025].

[183] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[184] 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.

[185] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[186] Yuanjie Li, Kyu-Han Kim, Christina Vlachou, and Junqing Xie. Bridging the data charging gap in the cellular edge. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 15–28, 2019.

[187] Yuanjie Li, Chunyi Peng, Zengwen Yuan, Jiayao Li, Haotian Deng, and Tao Wang. Mobileinsight: Extracting and analyzing cellular network information on smartphones. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 202–215, 2016.

[188] 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.

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

[190] 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.

[191] Jinshu Liu, Hamid Hadian, Hanchen Xu, Daniel S Berger, and Huaicheng Li. Dissecting cxl memory performance at scale: Analysis, modeling, and optimization. *arXiv preprint arXiv:2409.14317*, 2024.

[192] Yao Liu, Sujit Dey, Don Gillies, Faith Ulupinar, and Michael Luby. User experience modeling for dash video. In *2013 20th International Packet Video Workshop*. IEEE, 2013.

[193] 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.

[194] Karsten Loesing, Steven J. Murdoch, and Roger Dingledine. A case study on measuring statistical data in the Tor anonymity network. In *Proceedings of the Workshop on Ethics in Computer Security Research (WECSR 2010)*, LNCS. Springer, January 2010.

[195] Natasha Lomas. Uh oh! european carriers are trying to get into 'personalized' ad targeting. `https://techcrunch.com/2022/06/24/trustpid/`, 2022.

[196] Lourdes López-García, Luis J Dominguez Perez, and Francisco Rodríguez-Henríquez. A pairing-based blind signature e-voting scheme. *The Computer Journal*, 57(10):1460–1471, 2014.

[197] Zhaojun Lu, Gang Qu, and Zhenglin Liu. A survey on recent advances in vehicular network security, trust, and privacy. *IEEE Transactions on Intelligent Transportation Systems*, 20(2):760–776, 2018.

[198] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[199] 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.

[200] 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.

[201] Zhihong Luo, Silvery Fu, Natacha Crooks, Shaddi Hasan, Christian Maciocco, Sylvia Ratnasamy, and Scott Shenker. {LOCA}: A {Location-Oblivious} cellular architecture. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1621–1646, 2023.

[202] Zhihong Luo, Silvery Fu, Mark Theis, Shaddi Hasan, Sylvia Ratnasamy, and Scott Shenker. Democratizing cellular access with cellbricks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 626–640, 2021.

[203] 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.

[204] Zhihong Luo, Sam Son, Sylvia Ratnasamy, and Scott Shenker. Harvesting memory-bound {CPU} stall cycles in software with {MSH}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 57–75, 2024.

[205] Anna Lysyanskaya, Ronald L Rivest, Amit Sahai, and Stefan Wolf. Pseudonym systems. In *International Workshop on Selected Areas in Cryptography*, pages 184–199. Springer, 1999.

[206] 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.

[207] 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.

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

[209] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. Multi-clock: Dynamic tiering for hybrid memory systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA'22)*, 2022.

[210] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for

cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 742–755, 2023.

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

[212] MEMTIS. GitHub - cosmoss-jigu/memtis: Tiered memory management — github.com. `https://github.com/cosmoss-jigu/memtis`. [Accessed 09-12-2024].

[213] Christopher Mitchell. A Major Telecom Monopoly Fails America. `https://ilsr.org/a-major-telecom-monopoly-fails-america/`, 2020. Institute for Local Self-Reliance.

[214] Christopher Mitchell and H Trostle. Profiles of Monopoly: Big Cable & Telecom. `https://ilsr.org/monopoly-networks/`, 2018.

[215] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*, pages 19–38. IEEE, 2017.

[216] 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.

[217] Robert Moskowitz, Pekka Nikander, Petri Jokela, and Thomas Henderson. Host identity protocol. Technical report, RFC 5201, April, 2008.

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

[219] Mpirical. Key Access Security Management Entries. `https://www.mpirical.com/glossary/kasme-key-access-security-management-entries`, 2020.

[220] MPTCP. MPTCP Deployment. `http://blog.multipath-tcp.org/blog/html/index.html`, 2019.

[221] MPTCP. Mptcp/mptcp_fullmesh.c. `https://github.com/multipath-tcp/mptcp/blob/5b127fba5f34e8acbb5067d5940bd13678c7b7dc/net/mptcp/mptcp\_fullmesh.c\#L1070`, 2020.

[222] 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.

[223] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[224] Netmanias. Lte security ii: Nas and as security. `https://www.netmanias.com/en/?m=view&id=techdocs&no=5903`, 2013.

[225] Nginx. Nginx hls module. `http://nginx.org/en/docs/http/ngx_http_hls_module.html`, 2021.

[226] Binh Nguyen, Tian Zhang, Bozidar Radunovic, Ryan Stutsman, Thomas Karagiannis, Jakub Kocur, and Jacobus Van der Merwe. ECHO: A Reliable Distributed Cellular Core Network for Hyper-Scale Public Clouds. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 163–178. ACM, 2018.

[227] Dat Nguyen and Khanh Nguyen. Polar: A managed runtime with hotness-segregated heap for far memory. In *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 15–22, 2024.

[228] NordVPN. NordVPN. `https://nordvpn.com`, 2021.

[229] OpenCellID. The world's largest open database of cell towers. `https://www.opencellid.org/`, 2021.

[230] Oracle. HotSpot Virtual Machine Garbage Collection Tuning Guide — docs.oracle.com. `https://docs.oracle.com/en/java/javase/17/gctuning/garbage-first-g1-garbage-collector1.html`, 2024. [Accessed 10-12-2024].

[231] Oracle. Java support for large memory pages. `https://www.oracle.com/java/technologies/javase/largememory-pages.html`, 2024. [Accessed 03-12-2024].

[232] 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.

[233] Christoph Paasch and Sebastien Barre. Multipath TCP. `https://www.multipath-tcp.org`, 2021. Accessed: 2020-04-29.

[234] Christoph Paasch, Gregory Detal, Fabien Duchene, Costin Raiciu, and Olivier Bonaventure. Exploring mobile/wifi handover with multipath tcp. In *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design*, pages 31–36, 2012.

[235] Michael Paleczny, Christopher Vick, and Cliff Click. The java {HotSpot™} server compiler. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*, 2001.

[236] 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.

[237] 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.

[238] 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.

[239] Steven Pearlstein. Washington Post Article. `https://www.washingtonpost.com/business/2019/06/19/looming-t-mobile-sprint-merger-is-wakeup-call-free-markets-failures/`.

[240] Charles E Perkins. Mobile ip. *IEEE communications Magazine*, 35(5):84–99, 1997.

[241] 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.

[242] PJSIP. Pjsip project. `https://github.com/pjsip/pjproject`, 2021.

[243] Aaron Pressman. Fortune. T-Mobile claims largest 5G network. `https://fortune.com/2020/11/05/t-mobile-5g-coverage-verizon-best/`, 2020. Accessed: 2020-12-26.

[244] 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.

[245] Telecom Infra Project. OpenRAN. `https://telecominfraproject.com/openran/`, 2021.

[246] Telecom Infra Project. Telecom Infra Project. `https://telecominfraproject.com/`, 2021.

[247] The Pepper Project. Pequin: An end-to-end toolchain for verifiable computation, snarks, and probabilistic proofs. `https://github.com/pepper-project/pequin`, 2021.

[248] 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.

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

[250] Zafar Ayyub Qazi, Melvin Walls, Aurojit Panda, Vyas Sekar, Sylvia Ratnasamy, and Scott Shenker. A High Performance Packet Core for Next Generation Cellular Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 348–361. ACM, 2017.

[251] Qualcomm. Qualcomm qxdm. `https://bit.ly/2YrOVw1`, 2020.

[252] Moin Qureshi and Gabriel H Loh. Fundamental latency trade-offs in architecturing dram caches: Outperforming impractical sram-tags with a simple and practical design. In *Proc. of the 45th Intl. Symp. on Microarchitecture, Vancouver, Canada*, volume 10, 2012.

[253] 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.

[254] Charles Rackoff and Daniel R Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Annual International Cryptology Conference*, pages 433–444. Springer, 1991.

[255] Costin Raiciu, Dragos Niculescu, Marcelo Bagnulo, and Mark James Handley. Opportunistic mobility with multipath tcp. In *Proceedings of the sixth international workshop on MobiArch*, pages 7–12, 2011.

[256] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 392–407, 2021.

[257] ITU-T Recommendation. Perceptual evaluation of speech quality (pesq): An objective method for end-to-end speech quality assessment of narrow-band telephone networks and speech codecs. *Rec. ITU-T P. 862*, 2001.

[258] ABI Research. Top 10 Mobile Carriers Generate US 202B in Gross Profit. `https://www.abiresearch.com/press/top-10-mobile-carriers-generate-us-202-billion-in-/`, 2013. Accessed: 2020-12-26.

[259] Rhizomatica. `http://rhizomatica.org/`. Retrieved 4/2013.

[260] Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, Jon Peterson, Robert Sparks, Mark Handley, and Eve Schooler. Rfc3261: Sip: session initiation protocol, 2002.

[261] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.

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

[263] Samsung. Samsung's approach to privacy. `https://www.samsung.com/us/account/our-approach-to-privacy/`, 2021.

[264] M Isabel Sanchez and Azzedine Boukerche. On ieee 802.11 k/r/v amendments: Do they have a real impact? *IEEE Wireless Communications*, 23(1):48–55, 2016.

[265] Paul Schmitt and Barath Raghavan. Pretty good phone privacy. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1737–1754, 2021.

[266] Henning Schulzrinne, Stephen Casner, Ron Frederick, and Van Jacobson. RTP: A Transport Protocol for Real-Time Applications. `https://tools.ietf.org/html/rfc3550`, 2020.

[267] P1 Security. P1sec/qcsuper. `https://github.com/P1sec/QCSuper`, 2020.

[268] 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.

[269] Lyndon Seitz. Mobile vs. Desktop Internet Usage. `https://bit.ly/2MPNi5V`, 2020.

[270] 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.

[271] Spencer Sevilla, Matthew Johnson, Pat Kosakanchit, Jenny Liang, and Kurtis Heimerl. Experiences: Design, Implementation, and Deployment of CoLTE, a Community LTE Solution. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[272] Altaf Shaik, Ravishankar Borgaonkar, N Asokan, Valtteri Niemi, and Jean-Pierre Seifert. Practical attacks against privacy and availability in 4g/lte mobile communication systems. *arXiv preprint arXiv:1510.07563*, 2015.

[273] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM conference on special interest group on data communication*, pages 213–226, 2015.

[274] Yi-Sheng Shiu, Shih Yu Chang, Hsiao-Chun Wu, Scott C-H Huang, and Hsiao-Hwa Chen. Physical layer security in wireless networks: A tutorial. *IEEE wireless Communications*, 18(2):66–74, 2011.

[275] Reza Shokri, George Theodorakopoulos, Jean-Yves Le Boudec, and Jean-Pierre Hubaux. Quantifying location privacy. In *2011 IEEE symposium on security and privacy*, pages 247–262. IEEE, 2011.

[276] Pham Hai Son, Sudan Jha, Raghvendra Kumar, Jyotir Moy Chatterjee, et al. Governing mobile virtual network operators in developing countries. *Utilities Policy*, 56:169–180, 2019.

[277] 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.

[278] srsRAN. srsLTE: Your own mobile network. `https://www.srslte.com/`, 2020.

[279] 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.

[280] Statista. US Wireless Subscriptions Market Share. `https://bit.ly/2Yxsbuz`, 2020.

[281] Randall Stewart and Christopher Metz. Sctp: new transport protocol for tcp/ip. *IEEE Internet Computing*, 5(6):64–69, 2001.

[282] 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.

[283] Daehyun Strobel. Imsi catcher. *Chair for Communication Security, Ruhr-Universität Bochum*, 14, 2007.

[284] Tech Sujhav. Diameter protocol explained: S6a/s6d. `https://diameter-protocol.blogspot.com/2012/07/s6as6d.html`, 2012.

[285] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–121, 2023.

[286] Oğuz Sunay, Shad Ansari, Woojoong Kim, Pingping Lin, Hyunsun Moon, Badhrinath Padmanabhan, Guru Parulkar, Larry Peterson, and Ajay Thakur. Aether: Enterprise-5G/LTE-Edge-Cloud-as-a-Service. Technical report, Open Networking Foundation, 2020.

[287] Keen Sung, Brian Neil Levine, and Marc Liberatore. Location privacy without carrier cooperation. In *IEEE Workshop on Mobile Security Technologies, MOST*, page 148. Citeseer, 2014.

[288] Mist Systems. 802.11k, 802.11r, and 802.11v. `https://www.mist.com/documentation/802-11k-802-11r-802-11v/`, 2021.

[289] Hughes Systique. LTE Wifi Data Offload - A Brief Survey. `https://hsc.com/DesktopModules/DigArticle/Print.aspx?PortalId=0&ModuleId=1215&Article=224`, 2014.

[290] 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.

[291] Brian R Tauro, Brian Suchy, Simone Campanoni, Peter Dinda, and Kyle C Hale. Trackfm: Far-out compiler support for a far memory world. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 401–419, 2024.

[292] Techplayon. 5g nr dual active protocol stack (daps) handover – 3gpp release 16. `https://www.techplayon.com/5g-nr-dual-active-protocol-stack-daps-handover-3gpp-release-16/`, 2020.

[293] TelcoBridges. Lawful intercept solutions. `https://www.telcobridges.com/solutions/operators/lawful-intercept`, 2021.

[294] Telecoms. Neutral host networks and how to support them. `https://telecoms.com/opinion/neutral-host-networks-and-how-to-support-them/`, 2020.

[295] 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.

[296] Tessian. 22 biggest gdpr fines of 2019, 2020, and 2021 (so far). `https://www.tessian.com/blog/biggest-gdpr-fines-2020/`, 2021.

[297] TIOBE. TIOBE Index - TIOBE — tiobe.com. `https://www.tiobe.com/tiobe-index/java/`. [Accessed 03-12-2024].

[298] Tor. Tor. `https://www.torproject.org/`, 2021.

[299] 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.

[300] Twitter. GitHub - twitter/cache-trace: A collection of Twitter's anonymized production cache traces. — github.com. `https://github.com/twitter/cache-trace`. [Accessed 11-04-2025].

[301] 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.

[302] Tommaso M. Valletti. Is mobile telephony a natural oligopoly? Review of Industrial Organization 22, 47–65 (2003), `https://doi.org/10.1023/A:1022191701357`.

[303] Fabian Van Den Broek, Roel Verdult, and Joeri de Ruiter. Defeating imsi catchers. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, pages 340–351, 2015.

[304] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 137–152, 2015.

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

[306] Open vSwitch. Open vswitch. `https://www.openvswitch.org/`, 2021.

[307] Midhul Vuppalapati and Rachit Agarwal. Tiered memory management: Access latency is the key! In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 79–94, 2024.

[308] W3Techs. Usage statistics of QUIC for websites. `https://w3techs.com/technologies/details/ce-quic`, 2021.

[309] 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.

[310] Riad S Wahby, Srinath TV Setty, Zuocheng Ren, Andrew J Blumberg, and Michael Walfish. Efficient ram and control flow in verifiable outsourced computation. In *NDSS*, 2015.

[311] 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.

[312] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A {Memory-Disaggregated} managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280, 2020.

[313] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. {MemLiner}: Lining up tracing and application for a {Far-Memory-Friendly} runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 35–53, 2022.

[314] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective attacks and provable defenses for website fingerprinting. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 143–157, 2014.

[315] 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.

[316] Nick Wanninger, Tommy McMichen, Simone Campanoni, and Peter Dinda. Getting a handle on unmanaged memory. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 448–463, 2024.

[317] WBA. OpenRoaming. `https://wballiance.com/openroaming/`, 2021.

[318] Lance Whitney. Data privacy is a growing concern for more consumers. `https://www.techrepublic.com/article/data-privacy-is-a-growing-concern-for-more-consumers/`, 2021.

[319] Zack Whittaker. Us cell carriers are selling access to your real-time phone location data. `https://www.zdnet.com/article/us-cell-carriers-selling-access-to-real-time-location-data/`, 2018.

[320] Björn Wiedersheim, Zhendong Ma, Frank Kargl, and Panos Papadimitratos. Privacy in inter-vehicular networks: Why simple pseudonym change is not enough. In *2010 Seventh international conference on wireless on-demand network systems and services (WONS)*, pages 176–183. IEEE, 2010.

[321] Klaas Wierenga and Licia Florio. Eduroam: past, present and future. *Computational methods in science and technology*, 11(2):169–173, 2005.

[322] Wikipedia. International mobile subscriber identity. `https://en.wikipedia.org/wiki/International_mobile_subscriber_identity`, 2020.

[323] Josephine Wolff and Nicole Atallah. Early gdpr penalties: Analysis of implementation and fines through may 2020. *Journal of Information Policy*, 11:63–103, 2021.

[324] Ben Wolford. What are the gdpr fines? `https://gdpr.eu/fines/`, 2021.

[325] RF Wireless World. LTE QoS quality of service, class identifier(QCI), QoS in LTE. `https://www.rfwireless-world.com/Tutorials/LTE-QoS.html`, 2021.

[326] RF Wireless World. LTE to WLAN(wifi) Handover. `https://www.rfwireless-world.com/Terminology/LTE-WLAN-handover.html`, 2021.

[327] Charles V Wright, Scott E Coull, and Fabian Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *NDSS*, volume 9. Citeseer, 2009.

[328] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. {DIZK}: A distributed zero knowledge proof system. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 675–692, 2018.

[329] Christos Xenakis and Christoforos Ntantogian. Attacking the baseband modem of mobile phones to breach the users' privacy and network security. In *2015 7th International Conference on Cyber Conflict: Architectures in Cyberspace*, pages 231–244. IEEE, 2015.

[330] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad:{Non-Exclusive} memory tiering via transactional page migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 19–35, 2024.

[331] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 331–345, 2019.

[332] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. Improving program locality in the gc using hotness. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 301–313, 2020.

[333] Albert Mingkun Yang and Tobias Wrigstad. Deep dive into zgc: A modern garbage collector in openjdk. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(4):1–34, 2022.

[334] Juncheng Yang, Yao Yue, and KV Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)*, 17(3):1–35, 2021.

[335] 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.

[336] 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.

[337] 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.

[338] Fangxi Yin, Denghui Dong, Sanhong Li, Jianmei Guo, and Kingsum Chow. Java performance troubleshooting and optimization at alibaba. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 11–12, 2018.

[339] Hui Zang and Jean Bolot. Anonymization of location data does not work: A large-scale measurement study. In *Proceedings of the 17th annual international conference on Mobile computing and networking*, pages 145–156, 2011.

[340] 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.

[341] 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.

[342] 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.

[343] 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.

[344] Chuan Zhao, Shengnan Zhao, Minghao Zhao, Zhenxiang Chen, Chong-Zhi Gao, Hongwei Li, and Yu-an Tan. Secure multi-party computation: theory, practice and applications. *Information Sciences*, 476:357–372, 2019.

[345] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with cxl in virtualized environments. In *Symposium on Operating Systems Design and Implementation*, 2024.

[346] Yulong Zou, Jia Zhu, Xianbin Wang, and Victor CM Leung. Improving physical-layer security in wireless communications using diversity techniques. *IEEE Network*, 29(1):42–48, 2015.

[347] ZTE. Mf820b lte usb modem quick guide. `https://usermanual.wiki/ZTE/MF820B/pdf`, 2012.