Karlsruher Institut für Technologie

Stage-aware Scheduling in a Library OS

Bachelorarbeit
von

cand. inform. Christian Schwarz

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	M. Sc. Mathias Gottschlag

Bearbeitungszeit: 01. Dezember 2017 – 27. März 2018

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, den 27. März 2018

Abstract

Scalable high-performance network servers are a requirement in today's distributed infrastructure. Event-driven concurrency models often provide better scalability properties than multi-threaded servers but many legacy applications still follow the multi-threaded model where each request is handled by a dedicated operating system thread. Recent profiling at Google suggests that the instruction working set of many server applications does not fit into the private i-caches of contemporary processors, causing underutilization of their super-scalar out-of-order pipeline. In a multi-threaded server with an oversized instruction working set, context switches between two request-handler threads are thus likely to cause i-cache misses and subsequent pipeline stalls.

We start by analyzing existing approaches to optimize the cache behavior of network servers. One technique applicable to multi-core systems is executing different parts of an application's code on different cores. By migrating threads to those cores whose caches contain the threads' current instruction working set, the application's code is effectively spread over the system's private i-caches and code misses are greatly reduced. Proof-of-concept work at the KIT OS group shows the potential of this technique, but the implementation does not scale to multiple clients and cores. In this thesis, we therefore propose that the spreading technique described above must be tightly integrated with the OS thread scheduler. We present an unintrusive user-space API that allows partitioning a multi-threaded server's request handler code path into *stages*. Our scheduler then dynamically assigns cores to stages and dispatches threads on their current stages' cores.

We evaluate our design and its implementation in the OSv library operating system by adapting the MySQL database management system to our solution. We achieve up to 22% higher throughput caused by a 65% reduction of L2 i-cache misses without having to sacrifice request latency for this improvement.

Contents

Abstract **v**

Contents **1**

1 Introduction **3**

2 Related Work **7**
2.1 Network Servers & Concurrency 8
2.2 Cohort Scheduling & Staged Computation 9
2.3 Staged Event-Driven Architecture 10
2.4 STEPS 11
2.5 Top-Down Performance Analysis 13
2.6 Profiling Datacenter Applications 15
2.7 Interim: Applicability to SMP Systems 16
2.7.1 Private Caches 16
2.7.2 Real Parallelism 16
2.8 Computation Spreading 17
2.9 Prototype in Linux at the KIT OS Group 18

3 Analysis **21**

4 Design & Implementation **25**
4.1 The OSv Library Operating System 28
4.1.1 The OSv Scheduler 29
4.2 The Stage API 30
4.3 Fast Thread Migration 32
4.4 Thread Migration on Wake-Up 36
4.5 Core Allocation Policy 40
4.5.1 OSv's Page Access Scanner 44

5 Evaluation **47**

5.1 Evaluation Setup 48
5.1.1 Configurable Idle Strategy in Upstream OSv 51
5.2 Adopting the Stage API in MySQL 51
5.3 Whole-System Benchmark with MySQL 55
5.4 Validation of Design Assumptions 57
5.4.1 Instruction Working Set Spreading 57
5.4.2 Changes to Thread Migration 59
5.4.3 Core Allocation: Validation of Implementation 64
5.4.4 Core Allocation: Performance Evaluation 66
5.5 Discussion 68

6 Conclusion **69**
6.1 Future Work 70

Appendix **73**

Bibliography **75**

Chapter 1

Introduction

Network servers are of ever-increasing importance in today's infrastructure, handling requests from clients at high degrees of concurrency. A common implementation pattern is a *sequential* handler routine that processes each request, consisting of logical stages such as protocol handling, decoding/encoding and actual business logic. Concurrency is achieved by executing the handler code per connection in separate operating system thread – either by spawning a thread per request or by using a thread pool. The sequential handler implementation is simple to follow for programmers and abstractions for the aforementioned threading models are available in virtually any programming language and operating system. [PDZ99; Keg99]

Given the increasing CPU-memory-performance gap, it is paramount that the request handler's working set fits into the private CPU caches. The code-part of the working set is of particular importance to avoid stalling the execution pipeline of contemporary super-scalar out-of-order processors. The dominant CPUs in the server market are `x86-64` based Intel processors featuring separate 32KiB L1-d and L1-i caches and a unified 256KiB L2 cache per core, as well as a shared inclusive L3 cache that is significantly larger. [NehAnand] Recent profiling at Google shows that this memory hierarchy fails to meet the demands of contemporary server applications: Request handling code exhibits little spacial locality and thrashes the private CPU caches, leading to frequent pipeline stalls and sub-optimal application performance.[Kan+15]

Previous work in this field includes techniques such as *cohort scheduling*: Threads that execute the same working set are grouped into *cohorts* and dispatched in series, thereby amortizing instruction cache misses among all threads in the cohort. *Staged Computation* expands this concept to general

software architecture and *staged event-driven architectures* (SEDA) generalizes it further: Request processing state is encapsulated into an object which is passed through a pipeline of stages interconnected by queues. Each stage controls the concurrency model used to perform its work, allowing cohort scheduling to be employed. A derivative of cohort scheduling has been implemented in a research DBMS under them term *STEPS*, yielding an overall speedup of 40% in the TPC-C benchmark.[LP01; WCB01; HA04; Har05]

Despite the promising results of the research presented above, the concepts have found little practical adoption. For example, the popular database management systems MySQL still uses thread-based concurrency (one handler per request) with a pre-spawned pool of handler threads. [MySQL]. Apart from the required software-engineering effort do adapt an existing project to SEDA, we must also consider that the solutions mentioned so far were designed for uniprocessors. However, given the ubiquity of SMP systems, techniques such as *computation spreading* propose a different approach to mitigate i-cache thrashing: With computation spreading, OS code is executed on different cores than user-level code, thereby effectively splitting the application's working set at the user-kernel-boundary, resulting in better use of the private caches per core. Experimental work at the KIT OS group generalizes computation spreading to support migration points at arbitrary positions in the application code: Application developers manually partition their code into *stages* and insert one-line stage switching calls into the request handling code path. By dedicating CPU cores to stages and migrating threads between the cores on stage switch, the request handler's instruction working set is spread over the private caches of these cores. A proof-of-concept implementation in Linux and the MySQL database management system shows that for a single concurrent client, L2 cache misses can be reduced by up to 40%, correlating with up to 17% increased throughput. However, for more than one client and for increased core counts, the achieved throughput is up to 40% lower compared to running on standard Linux. The proof-of-concept is thus clearly not work-conserving, but shows the potential gains achievable by the technique.

In this thesis, we present a stage-aware thread scheduler that sustains the performance improvements achieved by instruction working set spreading at a high number of concurrent clients and multiple CPU cores. Our solution requires minimal customization effort for adoption in multi-threaded servers, making it particularly attractive for large legacy codebases that cannot be easily refactored to support more efficient concurrency models. Specifically, our contributions are as follows:

- We analyze the design aspects that make the early 2000's approaches of cohort scheduling and STEPS inapplicable to SMP systems.
- We analyze the design and implementation of the aforementioned proof-of-concept implementation at the KIT OS group and point out why it is not work-conserving.
- We present a user-space C++ API for applications to manually define stages and stage switching points in the request-handling code path.
- We present the design and implementation of a work-conserving scheduler that allocates CPUs to stages and migrates threads as necessary to preserve warm instruction-caches.
- We show the practical applicability of our solution by adopting it in MySQL, resulting in 22% increased throughput and a 65% reduction in L2 i-cache misses.

The remainder of this thesis is structured accordingly. In Chapter 2, we examine preceding work in the area of high-performance network server design. Chapter 3 describes the design of the KIT OS group's proof-of-concept, pointing out why it is not work-conserving. Subsequently, Chapter 4 presents the design and implementation of our solution in the OSv library operating system. Finally, we evaluate our implementation in Chapter 5 by applying it to MySQL and showing that our design decisions have their intended effects.

Chapter 2

Related Work

High-performance network servers must handle requests at a degree of concurrency that exceeds the parallelism provided by hardware in the form of CPU cores. Much research has gone into techniques to handle this problem, further constrained by additional requirements such as predictable response times and fairness among clients. This chapter starts with an overview of the prevalent concurrency models in the early 2000s' software landscape. We proceed with an introduction to *cohort scheduling*, *staged computation* and *staged event-driven architecture* which stem from the same time period and present *STEPS*, which applies above concepts to a research database management system, showing that these software-only solutions lead to performance improvements due to reduced i-cache misses and better branch prediction. Despite these in research systems, large-scale profiling at Google from 2015 shows that the memory hierarchy in contemporary processors is still suboptimal for typical datacenter applications. Given that the approaches above targeted single-core machines from the early 2000s, we take a step back and analyze their applicability to today's multi-core systems and memory hierarchies, coming to the conclusion that major redesigns are required to provide work-conserving solutions. Motivated by the ongoing relevance of the topic and a changed hardware landscape, we explore *computation spreading*, a technique that explicitly targets SMP systems and uses thread migration to spread the instruction working set of a thread over multiple CPU cores. We conclude with a proof-of-concept implementation at the KIT OS group which extends computation spreading to arbitrary split points within user-level code, forming the basis for this thesis.

2.1 Network Servers & Concurrency

Network server software is generally concerned with receiving requests from clients over a network connection, acting upon them and returning a response. In addition to the network I/O, disk access is very common, for example within file servers. The request handlers are thus commonly I/O bound. [WCB01]

I/O *hardware* interfaces are asynchronous, allowing the CPU to perform useful work while the slow I/O operation completes. However, the traditional *software* abstractions exposed by UNIX-like operating systems are synchronous: Processes or threads that *block* on I/O operations are preempted from the CPU and only resume execution once the result of the operation is available. Out of this situation, the following type of software architecture emerged: *Multi-process* and *multi-threaded* servers handle requests by following a sequential description of the steps involved to fulfill the requested task. Concurrency is then achieved by having multiple threads (either spawned on demand or from a pre-spawned pool) execute the same request-handling function for different connections and to interleave these control flows by time-multiplexing the CPU. This interleaving happens each time an I/O activity blocks or at the end of the scheduler-assigned time slice. Canonical scalability problems with this approach are the context switching overhead and the minimal amount of memory each thread requires, e.g., for its TCB and stack. Furthermore, synchronizing request handlers on shared resources without busy waiting requires kernel-supported synchronization primitives, implying syscall overhead. [PDZ99; Keg99; ALL89; WCB01]

One alternative to the above are *event-driven architectures* where the server consists of a loop that allocates a state object per request and implements a finite state machine driven by those objects. Each thread of the server acts on one state object at a time, but does not perform blocking operations. Instead, the operations are merely initiated and the state machine immediately switches to the next state object. The server loop picks up the completion events of these asynchronous I/O operations and changes the corresponding state object accordingly. This change in turn eventually triggers the state machine to perform the next logical request-handling step for the request represented by the state object. [PDZ99; WCB01; Keg99]

For this thesis, the most relevant difference between multi-threaded and event-driven architecture lies in the representation of request-handling state: Multi-threaded servers encode it implicitly in the thread's execution state, consisting mostly of its stack, registers and instruction-pointer. In contrast,

event-driven servers explicitly define the finite-state machine and have a dedicated representation of the processing state in the state objects.

Although the above description focuses on the role of I/O, nothing stops application developers from partitioning their CPU-bound or memory-intensive request-handling steps using the same mechanism. The next sections give an introduction to *staged computation* and *staged event-driven architectures* which build on top of the event-driven model.

2.2 Cohort Scheduling & Staged Computation

In the early 2000s, Larus et al. investigated the cache behavior of I/O intensive online-transaction processing workloads (OLTP) in servers implementing the multi-process or multi-threaded architecture. They observe high cache miss rates and instruction stalls, attributing it to the large amount of system calls made by the request handler threads: System calls themselves have a large working set disjoint from the application code and might bring an entirely different working set into the cache when blocking and switching to another thread. [LP01]

Cohort scheduling is then proposed as a scheduling policy to dispatch threads that are currently executing the same code segment in batches (*cohorts*): The first thread in a cohort may incur instruction / data cache misses but all successors of the same cohort benefit from a warm cache. Naturally, the threads must yield the CPU before leaving this shared code segment and the segment size must not exceed the i-cache size to avoid thrashing the cache. The number of threads per cohort represents the central trade-off in the scheduling policy: Large cohorts yield fewer amortized cache misses but cause higher response time due to progress only being made once enough threads have reached the synchronisation point required for batch dispatch. Furthermore, since the amortization only reduces the execution time but does not eliminate it, the minimum response time is necessarily increased. [LP01]

For immediate application, the authors suggest systems calls as pre-existing synchronisation points because they do not require modifying user-space code. However, to increase cache locality in arbitrary parts of an application, the authors propose *staged computation*: In this programming model, instead of synchronous calls to subroutines, the request-handling stage posts asynchronous operation requests to other stages, each encapsulating a particular functionality and state. A stage has *scheduling autonomy*, which means

it is free to choose the concurrency model that suits the type of provided operation best — cohort scheduling being just one of several options. [LP01].

Cohort scheduling at the syscall boundary is non-invasive with regards to application code bases and will we revisited by *computation spreading* (Section 2.8). In contrast, staged computation requires non-trivial refactorings in existing applications, in particular if they already follow the multi-threaded model: Synchronous, blocking operations must be converted to asynchronous operations and continuations, and the scheduling autonomy granted to each stage must actually be addressed in code.

2.3 Staged Event-Driven Architecture

Staged event-driven architecture (SEDA) is a software architecture generalizing the idea of staged computation. The primary motivation for its inception was the construction of network servers whose performance should degrade linearly under an increasing number of concurrent requests. SEDA provides a framework for application developers to implement an event-driven server by only defining event handlers. Each event handler represents a *stage* and is invoked by its *stage controller*, which executes the event handler on the stage's thread pool with input from the stage's *incoming event queue*. The event handler can enqueue additional work into other stages' queues, but cannot call into their code directly, enforcing an explicit boundary in the application's control flow. The stage controller works in a feedback loop to ensure a stage's performance requirements are met, for example by adjusting the thread pool / *cohort size* to meet a certain response time goal.[1] Stage controllers can be further customized to the application's unique requirements, providing an opportunity for developers to take full control of each stage's concurrency-model. [WCB01]

The authors evaluate SEDA by implementing an HTTP server and measuring the performance metrics *total concurrent throughput* and *response time* over a varying number of concurrent clients. Additionally, they measure the amount of requests each client completes, defining an equal distribution of service among clients as ideally *fair*. The results show 16–20% higher throughput at little-varying response times and high fairness among clients in comparison to the the multi-threaded Apache web server. The authors'

[1] SEDA terminology for a resource controller implementing cohort scheduling is "batching controller".

explanation for these results is that the SEDA server queues all requests *inside* the application whereas the multi-threaded Apache web server will not accept new client connections when reaching the maximum size of its thread pool, causing exponentially increasing response times for non-accepted clients due to exponential back-off algorithm employed in TCP. [WCB01]

SEDA shows the software-engineering and performance benefits of staged computation but requires adapting the application to use the provided framework of event handlers and stage controllers. In contrast, our solution requires minimal effort for adoption in multi-threaded servers, for which we expect non-trivial amounts of refactoring work to be required for SEDA. Notably, the SEDA authors suggest operating system support for their model, emphasized by their need to implement asynchronous I/O abstractions for their resource controllers. Although async I/O may facilitate the implementation of SEDA itself, it does not address the hurdle of refactoring existing code bases.

2.4 STEPS

The *Synchronized Transactions through Explicit Processor Scheduling* project (STEPS) implements a derivative of cohort scheduling in the SHORE research database system. [Reg12]

The authors reproduce the observations already made by the authors of cohort scheduling [LP01] for database systems: The execution flow in databases exhibits low spatial locality and the instruction working set does not fit into the CPU caches, leading to pipeline stalls and a high amount of mispredicted branches. In contrast to cohort scheduling, STEPS groups threads into *team lists* organized by the high-level application-specific operation they currently execute, for example `INSERT` or `UPDATE` queries. The code path of the operation then contains context-switching points (*`CTX` calls*) at which the currently executing thread yields control to the next member of its team list, which amounts to user-mode round-robin scheduling within a team. Context-switching points are placed such that the working set between two of them fits into the CPU's L1 i-cache. The schema described above leads to a step-by-step progression of the entire team that is depicted in Figure 2.1. As an example, let us imagine three context-switching points A, B and C and a team of ten threads executing between A and B. After the last thread reaches B, the team moves forward to the next step *B to C*. Although the first thread

executing this step will incur capacity i-cache misses, it pre-warms i-cache and branch predictors for the remaining nine threads. [HA04]

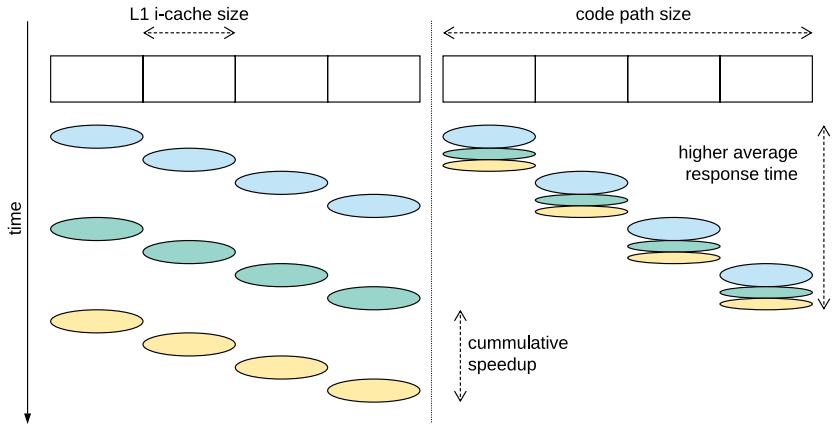


Figure 2.1: Idealized scenario of three threads (different colors) executing the same code path that does not fit into the L1 cache. **Left**: Each thread passes through the code completely before yielding CPU to the next thread. **Right**: A team of threads trickles down the code path in code segments that do not exceed the L1 i-cache size.

Blocking threads or those that abort the operation would fall behind in this trickle-down schema and thus no longer share their ex-team members' code. The solution is to remove blocking or aborting threads from the team list and track them as *stray threads*. The publications on STEPS describe that the thread package in SHORE required modification to support this schema, but falls short on implementation details. For this thesis, though, blocking is a highly relevant topic because it is a major problem in the proof of concept implementation at the KIT OS group (see Section 2.9 and Chapter 3). By examining the 6.0 release of SHORE which was published after STEPS, we can identify a base class for all DBMS threads called `sthread_t` which provides wrappers for basic blocking I/O functions such as the `read` or `write` syscalls. Under the assumption that these abstractions existed in the version of SHORE that STEPS was based on, we are confident that the STEPS authors used these wrappers to get notified about potentially blocking I/O requests by a thread to remove it from its team list. [Reg10; Reg12]

Regardless of the exact implementation details, we are not convinced that the adoption of STEPS in arbitrary applications requires as few code modifications as claimed by the authors: SHORE already comes with a custom thread abstraction around the *POSIX threads API* (pthreads) and centralized wrappers around blocking I/O. In contrast, arbitrary applications will

use the pthreads and C standard libraries directly, in which case either substantial modifications to the code base are necessary to establish a situation as found in SHORE or runtime-patching via `LD_PRELOAD` would be required.

Lastly, STEPS's handling of stray threads should be considered: Recall that stray threads are removed from their team when performing potentially blocking syscalls, and only re-join a team when re-starting a new high-level operation. [HA04] This means that STEPS does not account for the instruction footprint of the operating system at all. However, Agarwal et al. show that the operating system has a significant cache footprint and should thus be considered when trying to optimize the i-cache footprint. [AHH88; CWS06] Stray thread will further continue to compete for CPU time for the time until they finally block. Since the OS scheduler is unaware of the team lists, it may schedule a stray thread in the midst of a team, evicting its cache state inadvertently. STEPS tries to prevent such situations by giving a "hint" to the scheduler to prioritize the next thread that remained in the team, but does not elaborate on the additional syscall overhead. In this thesis, we propose an implementation that is implemented in the OS scheduler, thus avoid the information loss and implementation complexities associated with STEPS's user-space approach.

2.5 Top-Down Performance Analysis

For the upcoming Section 2.6 and the evaluation in Chapter 5, we require some basic understanding of modern out-of-order processors and the *top-down performance analysis method*.

Modern high-performance microprocessors employ various techniques to exploit instruction-level parallelism (ILP) and to keep all functional units of the system busy. *Pipelining* is a very basic solution that splits the execution of an instruction into multiple phases. An instruction passes through an N-staged pipeline in $N * C_{max}$ clock cycles where C_{max} is the number of clock cycles required for the slowest pipeline stage. More sophisticated *superscalar* processors replicate the functional units of individual stages, enabling multiple instructions to be completed (*retired*) per clock cycle. The state-of-the-art technique to manage this pipeline is called *dynamic pipeline scheduling* where it is the job of the hardware instead of the software and compiler to optimally supply functional units with work. Further techniques to exploit ILP are branch prediction and speculative execution. [PH05].

In any way, the above architecture is barely comparable to the classical model of a single-issue pipeline. Thus, Intel describe their contemporary micro-architectures as being split into 3 main parts: An *instruction fetch unit* that decodes the incoming instruction stream into micro-operations (uops), an *out-of-order execution engine* that executes those uops on functional units and an *in-order retirement* unit that ensures that the effects of the issued instructions appear in program order. [Cor16].

Yasin [Yas14] further simplify this view by only differentiating between *frontend* and *backend*: The frontend is responsible for fetching instructions, decoding them to uops and supplying them to the backend, which in turn is in charge of fetching operands, scheduling uops on the functional units and retiring instructions in program-order. The authors present the **top-down** performance analysis method, which helps to identify micro-architectural bottlenecks by classifying CPU time into categories that are organized in a nested tree. When applying the method, one traverses the tree top-down, usually following those categories marked with the highest percentage of spent CPU time. The data source for the classification are performance counters which are built into to the CPU and are accessible by software.

For this thesis, we will focus on the *top-down top-level breakdown*, i.e., the highest level of the category tree. The categories at this level are *retiring*, *bad speculation*, *backend-bound* and *frontend-bound*. The first two categories indicate that actually or at least potentially useful work was done whereas the latter two categories classify those pipeline slots lost due to microarchitectural bottlenecks: Backend-bound cycles do not perform useful work because operations in the backend take exceptionally long, thereby introducing *bubbles* into the execution pipeline. Examples for this category may be a sub-optimal instruction mix or d-cache misses on uop-operands. In contrast, frontend-bound cycles do not perform useful work because the backend is undersupplied with uops. For example, i-cache misses will cause no instructions to be decoded and thus no uops to be issued for execution. [Yas14; Cor17; Cor16]

In this thesis, we optimize i-cache behavior by partitioning the instruction working set. Reducing the percentage of frontend-bound cycles is thus used as a success metric in our evaluation.

2.6 Profiling Datacenter Applications

Although the micro-architectural performance problems caused by large i-cache footprints were already observed in [LP01] and [HA04] in the early 2000s, recent profiling work at Google shows that computer architecture is still not able to satisfy the requirements of contemporary scale-out applications. However, the potential performance gains and cost savings due to improved cache behavior become relevant at scale, motivating a re-assessment of the situation.

Kanev et al. identify a significantly higher amount of pipeline stalls in real-world datacenter applications when compared to the SPEC CPU2006 benchmark. Apart from the overhead associated with distributed software architecture (*datacenter tax*), a large cache footprint (both code and data) reveals the major performance bottleneck on today's processors for typical datacenter applications: The authors observe 60% of μop slots to be backend-bound, and 15 – 30% to be frontend-bound. In particular, the authors find that more than 5% of cycles are wasted due to empty front-end buffers, which is attributed to instruction read misses in the private L2 cache, resulting in slow accesses to the CPU's shared last level cache. Additionally, the observations confirm that applications under active development grow in their instruction working set, worsening the situation.[2] The authors do not explicitly investigate why datacenter applications have such large instruction working sets but mention *lukewarm* code and static linking as possible contributors. [Kan+15]

In comparison to the work published a decade earlier (see Section 2.2 and 2.4), Kanev et al. focus on computer-architecture and only pay brief attention to the operating system: Approximately 20% of CPU time is spent in the kernel with more than 5% in the scheduler alone. Additionally, it is noteworthy that the 90-th percentile of the observed machines handles more than 4500 concurrent threads. Given that at least a portion of these threads will be part of network servers, the question of the concurrency model thus stays very relevant although not explicitly stated as a source of performance-optimizations. [Kan+15]

Looking at the related work presented so far, we conclude that for more than 15 years, typical datacenter applications have been observed to exhibit high i-cache-miss and branch-misprediction rates, resulting in sub-optimal use of the available CPU resources and motivating continued research in this field.

[2] The authors present exemplary results of up to 27% per year.

2.7 Interim: Applicability to SMP Systems

Cohort scheduling, SEDA and STEPS were all evaluated on single-core machines. Adaption to multi-core systems is not mentioned at all or stays theoretical. [HA04; HA03] Given today's ubiquitous multi-core systems and the ongoing relevance of high i-cache footprints, re-examining the applicability of the above approaches seems appropriate.

2.7.1 Private Caches

The uniprocessor machines used to evaluate cohort scheduling, SEDA and STEPS feature a small L1 cache and a larger, higher-latency L2 cache. However, contemporary symmetric multi-core systems typically have a *private* L1 and L2 caches per core and a *shared* L3 cache. The capacities of the per-core L1 and L2 are comparable to the early 2000s' uniprocessors' caches. The access latency between L1, L2 and L3 increases by a factor of $3 - 4$ between each level. Beyond the L2 level, instruction cache misses can typically not be compensated by out-of-order processing. [HP02; 7cp]

Given these hardware constraints, an adaption of cohort scheduling and STEPS to multiprocessors must target the private caches of each core to benefit from warm i-caches. More importantly, the existence of private caches per hardware thread adds the problem and opportunity to allocate this resource: In terms of STEPS, multiple steps can now be spread over different cores instead of time-multiplexing only one cache. We will come back to this observation when assessing *computation spreading* in Section 2.8.

2.7.2 Real Parallelism

We recall from Section 2.2 that the scheduling trade-off in both SEDA and STEPS consists of reduced cache misses vs. increased response time due to *batching*. However, the real parallelism available on SMP systems brings with it another scheduling dimension that STEPS and cohort scheduling do not address: To utilize the available hardware threads, load must be balanced among them to maximize resource utilization, which generally surpasses response time as the primary scheduling goal. All relevant operating systems abstract CPU cores through threads, and the OS scheduler implements load balancing by migrating threads between cores, often directed by metrics such as the length of each core's the ready queue (*load*). [MN04]

Cohort scheduling and STEPS were not designed for this situation because **batching is inherently not work-conserving on SMP systems**: All threads in a cohort or team must run on the same core to benefit from the warm i-cache, implying that only one thread per team can run at any given time. Furthermore, STEPS and the OS scheduler work actively against each other because the latter will see all threads in a cohort as runnable and actively spread the cohort over all (presumably idle) cores.

An easy fix would be a mechanism for STEPS to influence the load balancing to favor threads of the same team to stay on the same core, e.g., through thread pinning. Thread-pinning is obviously not work-conserving on multi-core systems and requires additional mechanisms and communication between STEPS and the OS scheduler, thus complicating the implementation. On a busy system, one could further argue that enough runnable threads are available to simply employ cohort-scheduling per core. However, in contrast to the approach of *computation spreading* presented in the next section, per-core cohort scheduling replicates to each core the repeated capacity misses when a team moves to the next step.

The above assessment shows that cohort scheduling, SEDA and STEPS cannot be directly applied to today's multi-core systems. The availability of private caches per core must be accounted for and new requirements such as load balancing between cores must be addressed. However, given the ongoing relevance of the problems identified more than 15 years ago, we continue this chapter by investigating publications that explicitly target multi-core systems.

2.8 Computation Spreading

Computation spreading (CSP) in its most general form is a technique to spread the instruction working set of a process more effectively on SMP systems which commonly feature private caches per core. Chakraborty et al. [CWS06] show that this cache hierarchy leads to redundantly stored code. For example, in a traditional thread-based concurrency model, each request-handling thread may interact with the file system through system calls which leads to redundantly stored file system code in each core's private cache. The authors present a solution that dedicates CPU cores to either OS code or user-level code. Threads entering or leaving the OS synchronously through syscalls, exceptions or page faults are then migrated between compatible cores. [CWS06]

A central aspect of the implementation is the thread migration mechanism, which is based on hardware virtualization features: All synchronous kernel entries are intercepted and used to trigger the thread migration, allowing adoption of CSP without any modifications to *guest* OS and user-space software. [CWS06] However, since hardware-based thread migration is only available in hypervisor mode, cooperation from the software executing in this mode is required. Deployment in public clouds based on hardware-virtualization is thus only possible with explicit support by the cloud provider and would require further work in the hypervisor scheduler. Our solution does require hardware virtualization only due to limited driver support but implements thread migration in software and can thus be deployed to existing infrastructure.

The authors evaluate their solution using full-system simulation of an eight-core machine with private L1 and L2 cache and a shared L3 cache. Server applications, which are of primary interest for this thesis, exhibit 25 – 58% fewer L2 instruction misses and 9 – 25% improved branch misprediction rates. Data read misses are less affected, with only 13 – 19% decrease for Apache web server and an *online transaction processing* (OLTP) application. The approach leads to a speedup of up to 20% in Apache and 9.4% for OLTP, measured by comparing total runtime of a full-system simulation. [CWS06]

Despite the very specific implementation, it should be noted that the concept presented by the authors is more general and not limited to the syscall interface: They suggest migration points at arbitrary positions in the application code and corresponding core allocation policies. [CWS06] However, we cannot find a concrete proposal ready for implementation.

2.9 Prototype in Linux at the KIT OS Group

At the KIT OS group, a Linux-based proof-of-concept implementation combines the idea of staged execution, cohort scheduling and computation spreading into a C API that allows for intuitive conversion of existing code bases to staged execution: The application developer manually identifies stages and inserts library calls into application code for switching the current stage. Each CPU core is assigned one or more stages and each time a thread switches to a new stage, it is migrated to a core assigned to that stage, benefiting from a warm i-cache. This approach targets applications that employ thread-based concurrency, i.e., the multi-threaded server architecture outline in Section 2.1, since the stage-switching is embedded into the control flow

of the request handler and thus not explicitly modeled like in SEDA (see Section 2.3).

Fast thread migration mechanism is required for this technique to succeed. Otherwise, the performance benefits of always-warm caches per stage are destroyed by the migration time. The Linux built-in facility for this purpose, `sched_setaffinity`, is impractical because it uses expensive inter-processor-interrupts to implement this syscall, resulting in $9\mu s$ – $14\mu s$ of migration time. As a consequence, thread migration was implemented in user-space: For each user-level thread (ULT), there still exists exists a kernel-level thread (KLT), but KLTs are pinned once to a specific core and stage. ULTs run on a KLT of the stage they are currently in and migrate to a different KLT when switching stages. [KST10]

When a ULT makes a call to switch stages, its context is saved and enqueued into the next stage's incoming migrations queue. The source KLT now waits for new ULTs on its own stage's incoming migrations queue. If it is empty, the KLT makes a blocking syscall `sys_dequeue` to a kernel component to wait for incoming migrations. The kernel component must ensure that there is always one KLT per core either doing work or actively dequeuing ULTs to utilize the CPU. To accomplish the availability of a dequeuing KLT per core, a callback from the Linux scheduler informs the kernel component of task state changes. For example, if the currently dequeuing KLT K_1 executes a ULT, and that ULT blocks on a mutex, K_1 transitions from `running` to `blocked`. The kernel component must then wake up another KLT K_2 that is currently waiting for incoming migrations of that stage on that core. Otherwise, the core does not perform any work (for the application) until K_1 acquires the mutex and becomes `runnable`. Figure 2.2 visualizes this situation.

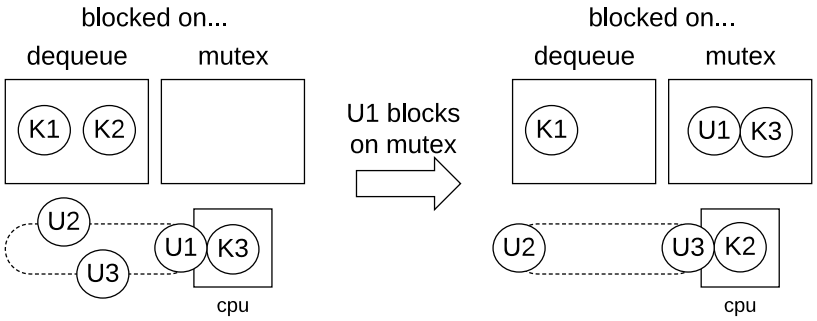


Figure 2.2: When a ULT issues a blocking syscall, the current KLT blocks. Another KLT must take over executing ULTs from the core's queue. Note that although there is a KLT for each ULT, the relationship is not fixed.

The authors evaluate their system using the TPC-C benchmark, comparing throughput and cache behavior of a modified version of MySQL 5.6.38 against an unmodified version as a baseline. For a single client and low core counts, they observe the expected 40% reduction in L2 cache misses, correlating with a 17% speedup in throughput. However, for both an increased number of concurrent clients and a higher number of CPU cores, throughput is $10 - 40\%$ below baseline. Additionally, L2 miss rates and achieved throughput do not correlate with more than one concurrent client. In the next chapter, we present our analysis of the reason behind this behavior.

Chapter 3

Analysis

Related work has shown that the instruction working set of datacenter-class server applications often exceeds the size of modern processors' private i-caches. Computation spreading is proposed as a technique to spread the instruction working set of such servers over the available private caches through thread migration. The authors provide an implementation based on hardware virtualization instructions to separate OS kernel code from the application (see Section 2.8).

In contrast to computation spreading, the proof-of-concept implementation at the KIT OS group proposes a software-only solution that allows splitting the instruction working set of an application at arbitrary positions in its code. Multi-threaded servers can be adapted to use the technique without significant refactoring. The MySQL DBMS shows a speedup of up to 17% in throughput in the TPC-C benchmark with a single client. However, with multiple clients and cores, the solution performs worse than unmodified MySQL. This chapter provides an explanation of the inherent design problems in the proof-of-concept which lead to the behavior described above and motive the work presented in this thesis.

Let us imagine a system as depicted in Figure 3.1, starting with a ULT U_1 in stage S executing on a KLT K_1. As described in Section 2.9, K_1 is pinned to core C_1. When U_1 performs a blocking syscall, for example when trying to acquire a lock, K_1 blocks. The Linux scheduler now dispatches another task T on C_1 to maximize CPU utilization, where T is not a K_i of our application. In fact, all K_i are blocked in `sys_dequeue`. When K_1 finally acquires the mutex and is `ready` again, it is still pinned to C_1 because the solution needs pinning to perform thread migration in user-space. However,

C_1 is still executing T, not K_1, and thus K_1 must be put into C_1's ready queue.

On a single-core system, the situation described above would be acceptable because T is also performing useful work. However, we have *another* CPU C_2 where a KLT K_2 is dequeuing ULTs for the *same stage* S: If K_2 does not have any ULTs to execute, K_1 should be migrated to C_2 immediately when it is woken up and continue execution there, benefiting from the warm on-core caches. But the implementation only performs thread migration when a ULT calls the stage switching API. There is no mechanism in place to save K_1's state and enqueue it to K_2's incoming migration queue on wake-up. One might assume it is possible to enqueue U_1 to K_2 since we saved its register state on kernel entry via `pthread_mutex_lock`: This is not possible because there might still be kernel code that needs to run after the mutex is acquired, before returning to U_1.

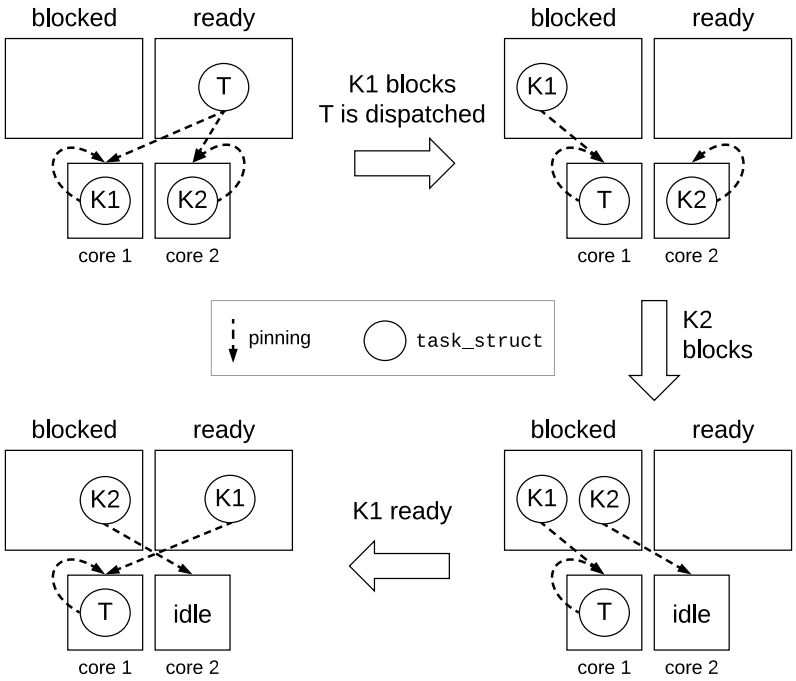


Figure 3.1: The proof-of-concept implementation is not work-conserving. When K1 becomes ready it should be dispatched to C2 immediately. But K1 is pinned to C1 due to the implementation of thread migration.

We identify several fundamental problems in the approach taken by the proof-of-concept implementation: The requirement for fast thread migration drove

the design toward a user-space solution which decouples the threads known by the application (ULTs) from the threads known by the kernel (KLTs). However, the kernel scheduler still only handles KLTs and assumes a 1:1 threading model, which leads to an **ambiguous role of KLTs** in the proof-of-concept: When switching between stages, the user space thread migration code views KLTs as the CPUs they are pinned to. But when a ULT running on a KLT interacts with the Linux kernel, the kernel sees a normal `task_struct` and continues to assume the 1:1 threading model where tasks can just block. The proof-of-concept works around this schizophreny by introducing a callback from the scheduler to react to blocking KLTs, but fails to handle asynchronous events like wake-ups, which we expect to require further integration into the scheduler.

We conclude that the proof-of-concept does not model the situation correctly: The association of stages and CPU cores is piggybacked onto the KLTs using `sched_setaffinity`, leading to an ambiguous role of KLTs. We thus propose the following guidelines for a proper design:

- Stages must be modeled as kernel objects, separate from CPUs and threads.
- The association of stages and CPU cores must be represented explicitly.
- Threads must carry the information in which stage they are executing.
- The scheduler must honor this information by scheduling threads onto cores that are associated with their respective stage.
- The scheduler must trade off the potential gains of always-warm caches against existing scheduling goals such as resource utilization, fairness and response time.

These guidelines revert the complex situation of ULTs and KLTs to a simple 1:1 threading model and remove the special-case of blocking kernel activity. The remainder of this thesis presents our design which follows the proposal above.

Chapter 4

Design & Implementation

Certain classes of datacenter applications exhibit large instruction working sets that do not fit into the private cache of commodity processors. On a micro-architectural level, the large instruction working set leads to i-cache thrashing and thus underutilization of the cores' functional units, which in turn degrades application performance. This chapter presents our operating-system-based solution to mitigate these micro-architectural problems in multi-threaded servers using a software-only solution.

Multi-threaded servers handle client connections in threads, each following the same, sequential, potentially blocking, code path. For example, in a database management system that is accessed over TCP, a request-handling thread will

(a) read the request data from a socket, using the network stack,

(b) deserialize & parse it,

(c) perform the requested operation,

(d) serialize the response and

(e) send it back to the client.

Database systems exhibit a significant i-cache footprint in step (c) per high-level SQL operation (see Section 2.4). The code footprint of the network stack is also significant, as is the filesystem and potentially the serialization library.

All these steps combined will execute more code than fits in the private i-cache, leading to capacity i-cache misses when executed on a single core.

Additionally, some code will be executed twice, but with another i-cache sized working set being used inbetween: For example, steps (a) and (e) share parts of the network stack but steps (b), (c) and (d) will execute before (e). Step (e) will thus likely incur i-cache misses for code that step (a) had brought into the cache at the beginning of the procedure. Furthermore, we must consider the situation where a request-handler thread A blocks, for example on I/O in steps (a), (c) or (e): The scheduler will dispatch another request-handler B on the same core, which may be in a different step than A. At worst, B will evict all of A's cache state. Looking at multi-core systems, the above behavior can be observed on each core because contemporary OS schedulers balance threads based on load of the cores, not on cache state. [MN04]

We solve above problems on modern multi-core systems by implementing a suitable scheduling policy in the OS scheduler. **We define a *stage* as a span in the request-handling code path that has an instruction working set smaller than the private i-cache size.** Ideally, different stages' instruction and data working sets are fully disjoint. Given the example database system, one would define the following stages: S_n for network stack, S_{ser} for serialization and one per SQL operation (S_{INS} for `INSERT`, S_{SEL} for `SELECT`, ...). It is important to emphasize that stages do not necessarily form a pipeline: Reading from and writing to the network socket form one stage because these operations share the network stack code, as do request deserialization and response serialization because they both use the serialization library.

Once the stages are defined, a request-handler thread must switch to the appropriate stage before performing the next step of request-handling. In concrete terms, this means that developers must call a system API in the request-handling code path. In the example above, a SELECT request would result in the following switch series: S_n, S_{ser}, S_{SEL}, S_{ser}, S_n. For the rest of this thesis, we will refer to an application that uses the stage API as described above as a *stagified* application.

Our scheduler tracks a thread's association to its current stage in the thread's TCB. Additionally, it tracks per stage the number of runnable threads associated with it. We then use this information to periodically compute the *current core allocation*, which assigns CPU cores to stages proportional to the stages' load. A core is thus always allocated to exactly one stage. The scheduler then implements the following guiding principle: **A thread is only dispatched on cores that are allocated to the thread's current stage**. When a thread uses the system API to switch to a stage S, we choose the least-loaded core C among the cores allocated to S and enqueue

the switching thread into C's runqueue. When a thread blocks, it no longer counts toward the load of stage but does not lose the association to it. Thus, when a blocked thread is woken up again, the *waker* simply uses the current core allocation to choose a suitable core for the woken thread and enqueue the still-sleeping thread into that core's runqueue.

The promise behind the approach described above is the same as in computation spreading and the proof-of-concept implementation at the KIT OS group (Chapter 2): **By dedicating CPU cores and hence their private caches to a stage and by scheduling threads exclusively on the cores of their current stages, the i-cache always contains the instruction working set of all threads in the CPU's runqueue.** This reduces instruction cache misses, pipeline stalls and mispredicted branches, resulting in faster execution of the stage's code and cheaper context switches on the CPU.

Our solution is specific to certain types of server applications and thus not suitable as a general-purpose scheduling policy. Ideally, we would thus extend an existing scheduler such as the Linux *completely fair scheduler*. However, extending the Linux scheduler is out of scope of this thesis due to the expected design and implementation complexity. Instead, we implement our solution in the OSv library operating system, a small kernel that is bundled with an application into an appliance-like virtual machine image.

The remainder of this chapter presents our design and its implementation in detail: Section 4.1 gives an introduction to the OSv library operating system, providing the knowledge required to understand the implementation constraints we faced. We proceed with details on the user-space API illustrated with examples (Section 4.2), followed by the design of our thread migration mechanism (Section 4.3). In Section 4.4, we explain the details involved in implementing thread-migration on wake-up, which required significant refactoring of OSv's thread state tracking and timer implementation. Section 4.5 then presents the CPU core allocation policy. Where appropriate, we refer to the relevant commits in the Git repository of our modified version of OSv which can be obtained from the the appendix.

4.1 The OSv Library Operating System

We implement our stage-aware scheduling solution in the OSv *library operating system*, which breaks up the traditional divide between the kernel and applications: Traditionally, an operating system provides abstractions from the physical hardware and multiplexes it among *multiple* untrusting users and applications, providing protection and isolation at various levels. With the rise of hardware virtualization, it has become common practice to deploy a traditional OS running a *single* application in a virtual machine running on top of a hypervisor. In this situation, both the hypervisor and the guest kernel implement protection mechanisms, but because each VM only runs a single application, the protection efforts of the guest kernel may be considered duplicate. [Mad+13; KCE14]

OSv addresses this situation by delegating all resource abstraction and protection responsibilities to the hypervisor. It confines itself to providing a familiar execution environment for a *single trust domain*, providing facilities for multi-threading, scheduling, a network stack and filesystems. To run an application on OSv, it is bundled with the OSv kernel into a virtual machine image. When the VM starts, OSv establishes a *single virtual address space* and dynamically links the application against OSv's own C standard library, in which system calls are merely calls to OSv-internal functions. This technique allows dynamically linked applications built on Linux to execute unmodified on OSv, unless they depend on facilities intentionally not provided by OSv such as forking new processes. The single trust domain property furthermore allows omitting all privilege level switching between OSv and the application. In summary, the removed user-kernel-boundary and syscall overhead improves application performance significantly without any application code changes required. However, if application developers choose to specifically target OSv, more efficient system APIs provide further optimization, e.g., zero-copy networking as opposed to the traditional socket API. [KCE14]

Although OSv incorporates pre-existing components for ACPI, the filesystem and originally the network stack, the core system is implemented in C++11. The codebase of ca. 400000 lines of code is very small compared to the 20 million lines of Linux 4.13, which can be attributed to both the use of a more expressive programming language and the comfort of only providing device drivers for typical hypervisor-emulated devices and paravirtualization. The OSv scheduler in particular amounts to less than 3000 lines, making OSv particularly attractive for this thesis. [KCE14]

4.1.1 The OSv Scheduler

The solution presented in this chapter must be implemented in the scheduler. To understand the design and implementation constraints we faced, this section gives a summary of the upstream OSv scheduler implementation.

The OSv developers state that the "thread scheduler [...] should be lock-free, preemptive, tick-less, fair, scalable and efficient." As such, it features CPU-local runqueues containing runnable threads, which are sorted in ascending order by the threads' recent average runtimes. Load-balancing is implemented by a periodically-invoked per-core load-balancer thread that migrates threads from its runqueue to other cores if these are less loaded. [KCE14]

OSv implements thread migration through a set of lock-free single-producer-single-consumer queues: Given N CPUs, each CPU has N `incoming_wakeup` queues, one per source CPU, containing pointers to the TCBs of the threads being migrated. At every reschedule, a CPU then drains all incoming queues into its runqueue. Each CPU is guaranteed to have an always runnable idle thread that spins for a short time and halts the CPU if no other runnable threads are in the runqueue. Thus, a source CPU may need to send an inter-processor-interrupt (IPIs) to the target CPU after enqueuing the migrated thread. For this purpose, each CPU has a bitmask, each bit representing one source CPU / wake-up queue: The source CPU will atomically set its bit in the target CPU's bitmask and only send the IPI if the bit was not set before, avoiding unnecessary flooding the target CPU with IPIs. [Sys17; KCE14]

Migrating runnable but not currently running threads is easy in this model: The source CPU removes the runnable thread from its runqueue and puts it into the incoming queue of the target CPU. However, threads may also migrate themselves to another CPU while still running, for example through the pthread_set_affinity API. Let us call this case *synchronous thread migration*: In OSv, the migrating thread spawns a short-lived helper thread and immediately schedules out. The helper thread will then put the migrating thread's TCB into the correct incoming_wakeup queue. [Sys17]

One complication of the above procedure are timers: Apart from CPU-local timers such as the preemption timer, a thread can program timers as well, for example for use with `SIGALARM`. Per CPU, a list of active timers is kept, sorted in ascending order by their expiration time. The OSv timer implementation then programs a single hardware-timer (clock_event driver) such that it fires at the nearest expiration time. On x86-64, the CPU-local Advanced Programmable Interrupt Controller (LAPIC) timer is used for this

purpose. [SDM105] In the timer interrupt handler, the expired timers are removed from the sorted list and the LAPIC timer is reprogrammed. Threads whose timer fired are then woken, i.e., marked as runnable and put into the CPU's ready queue. It is crucial to observe that the upstream OSv timer abstraction does not allow a thread to have timers set on multiple CPUs: All data structures related to timer management are only protected by masking interrupts because they are only manipulated by the thread itself or from the timer interrupt context. For thread migration, this timer-locality means that timers must be removed from the source CPU's timer list before enqueuing the thread into the target CPU's incoming_wakeup queue. On the target CPU, after adding the migrated timers to its timer list, the LAPIC may need to be reprogrammed if any of the migrated timers' expiration dates are earlier than the previous head of the list. [Sys17]

In addition to synchronous thread migration, upstream OSv also supports *remote asynchronous thread migration*, i.e., the situation where a thread T_i executing on CPU C_1 initiates the migration of a thread T_m currently executing on a CPU C_2 to CPU C_3. As outlined in the previous paragraph, T_m's timers must be migrated, which can only be initiated from the source CPU itself. Upstream OSv only requires this feature for implementing pthread_attr_setaffinity_np and uses a helper thread T_h that chases T_m until T_h can successfully mask all interrupts on the same CPU as T_m and suspend T_i's timers. The remaining steps are the same as in synchronous thread migration. [Sys17]

4.2 The Stage API

Our scheduling policy spreads an application's working set over the private caches of all available CPU cores by scheduling threads only on those cores that have its current stage's code in their private cache. We require developers to manually *stagify* their applications by defining stages and inserting switching calls into the request-handling code path. To facilitate adoption, our primary design goal for the stage API is to be non-invasive, allowing maintenance of private patches to an existing open source code base.

In OSv, we implement stages as a C++ class with public member functions for stage definition and switching. Without the syscall boundary, we can simply export the corresponding headers to the application, which receives pointers to the kernel objects representing stages and invokes methods on them. However, applications for OSv must be built on the Linux host where

```
#include <osv/stagesched.h>

stage *s_net, *s_ser, *s_db;

int main() {
  s_net = stage::define("net");
  s_ser = stage::define("serialization");
  s_db = stage::define("db");
  // start threads ...
}

void request_handler(database db, net::conn c) {
  s_net->enqueue();
  buf b = c.recv();
  s_ser->enqueue();
  req req = sql::parse(b);
  s_db->enqueue();
  response resp = db.handle_request(req);
  s_ser->enqueue();
  buf b = sql::respond(resp);
  s_net->enqueue();
  c.send(b);
}
```

Figure 4.1: Example for a stagified application using the stage API in pseudo C++. Because we implement our solution in OSv, we can simply use pointers to the kernel objects representing stages.

the stage API header and implementation are not available. To avoid build and linker failures, we provide a no-op implementation of the API as a shared library, enabling us to produce application binaries that work on both Linux and OSv.[1] Figure 4.1 gives an impression of how the stage API could be used in an existing application.

In addition to the direct invocation of the switching API, we provide a data-structure similar to C++11 lock guards: Following the *resource acquisition is initialization* (RAII) paradigm, the structure allows switching to a stage for the lifetime of a block and to automatically switch back to the previous stage when the guard object goes out of scope. The guard objects can also be used to shift the responsibility of switching stages from the function caller to the callee, which is particularly useful when a piece of code is large enough to

[1]The OSv linker ignores missing libraries, and the app does not crash because the OSv implementation of the stage class provides the required functions.

have its own stage but is called from more than one other stage. Figure 4.2 shows how the above example looks like with the RAII-style API.

The commits in the Git repository corresponding to the changes presented in this section are `439b5501` and `9eaf3a8b`.

4.3 Fast Thread Migration

Once an application has been stagified by the application developer, our solution must act on the stage switching calls made during request handling. Specifically, we must find a core whose private caches contain the next stage's code and migrate the calling thread to that core. We defer discussion of the core allocation policy to Section 4.5 and focus on the migration mechanism for now.

For a net benefit from spreading the instruction working set, the performance gained through always warm i-caches must outweigh the cost of thread migration. One significant contributor to the latter is migration latency, which is the reason why we require a low-latency mechanism. Upstream OSv's uses short-lived helper-threads for synchronous thread migration and inter-processor interrupts (IPIs) to notify the target CPU about incoming migrations. The IPIs are necessary because the target CPU may be temporarily halted in the idle thread. **We state that the upstream mechanism is unsuitable for our use-case of frequent stage-switching:** At first, recent measurements on AMD systems have shown that IPI propagation on bare-metal systems is 1700ns. For OSv, we assume significant overhead due to virtualization, which we will further investigate in Section 5.4.2. The IPI propagation time puts a lower bound on the achievable migration latency but additionally, the impact of IPIs on the target CPU's pipeline must be considered: An arriving interrupt will flush the pipeline or at least obsolete the instructions it currently executes. [She13] At last, on the source CPU, the use of helper threads implies memory allocation for the new TCB and an invocation of the OSv scheduler, which does not run in constant time.

As a consequence of the above, we choose to implement a lower-latency alternative for thread migration that neither uses IPIs nor helper threads: When a thread switches to another stage, the switching thread queries the core allocation policy for a target CPU that has the target stage's code in its private caches. The thread then puts its TCB into the *incoming migrations queue* of the target CPU *while running* and subsequently

```
#include <osv/stagesched.h>

class thread {
    stage_stack stagestack;
    ...
};

stage *s_net, *s_ser, *s_db;

int main() {
  s_net = stage::define("net");
  s_ser = stage::define("serialization");
  s_db = stage::define("db");
  // start threads ...
}

void thread::request_handler(database db, net::conn c) {
  buf b = c.recv();
  req rq = sql::parse(b);
  response rs = db.handle_request(rq);
  buf b = sql::respond(rs);
  c.send(b);
}

// Use stage_stack::guard in all functions
req sql::parse(buf& b) {
  stage_stack::guard g(thread::current()->stagestack, s_ser);
  ...
}
```

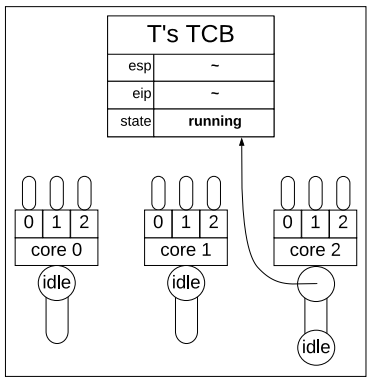
Figure 4.2: stage_stack keeps previous stages in a stack. The stage_stack::guard constructor pushes the stage pointer onto the stage_stack and switches to the stage. The corresponding destructor pops the current stage's pointer from the stack and switches back to the previous stage.

schedules out. The target CPU's reschedule routine and its idle thread eventually dequeue the TCB pointer from the incoming migrations queue into the runqueue, thereby completing the migration.

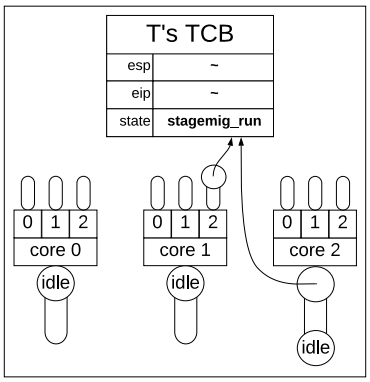
Although the above paragraph gives a good overview of our mechanism, we omitted two important details: First, there is a race condition between the point where a migrating thread T puts its TCB pointer into the incoming migrations queue and the point at which it schedules out: If the target CPU dequeues T's TCB before T has scheduled out, the TCB contains will invalid register state or the state from the last time T scheduled out. If T now starts running on the target CPU, we have the hazardous situation of T running on both CPUs simultaneously, primarily visible through a corrupted stack. We solve this problem by representing the stage-switch as a *two-step* thread state transition, visualized in Figure 4.3: Before the migrating thread puts its TCB into the target CPU's incoming migrations queue, it sets its state to stagemig_run. The target CPU will not put threads in that state into its runqueue but instead enqueue it back into the incoming migrations queue because the migration is not yet finished. On the source CPU, after T has scheduled out, we then set T's state to stagemig_sto. Eventually, the target CPU will dequeue T again, observe from its state that it is not running anymore and enqueue it into its runqueue.

The other issue we face with our migration mechanism is that in upstream OSv, the idle thread halts CPUs using the `HLT` instruction. This works in upstream due to the use of IPIs which wake the CPU up again, but we want to avoid IPIs in our solution and therefore provide two boot-time configurable alternatives: First, the idle thread can busy-wait for changes to the incoming migrations queue. Alternatively, it can use the `MONITOR` and `MWAIT` instructions to wait for changes to the queue while keeping the core in a low power state. Both options have benefits and downsides: Busy-waiting offers 16% lower migration latency (see Section 5.4.2) but consumes more power, which is undesirable for technical (dark silicon), economical (increased cost) and ecological reasons. `MWAIT` in combination with *simultaneous multithreading* (SMT) in contrast allows the core to switch to another hardware thread, which also opens a window for increased performance. [SDM810; Har+11] Neither of the alternatives cause an exit to the hypervisor, to which the VM will appear 100% busy.

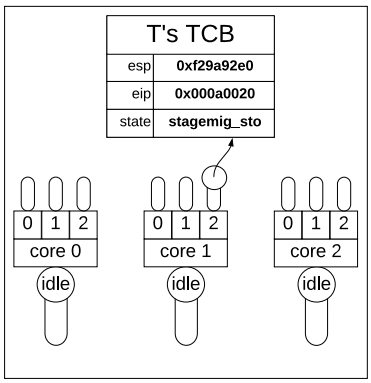
At last, we want to provide some implementation details that we omitted for easier understanding of the design: We implemented above mechanism in the early course of this thesis with a limited understanding of the upstream OSv migration mechanism. Therefore, instead of replacing the upstream



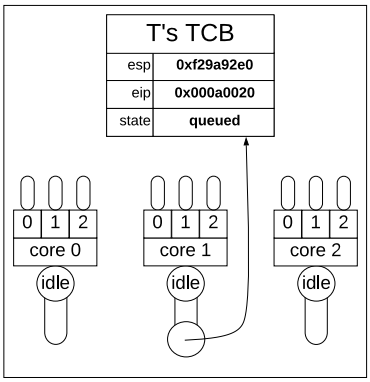
(a) T is running on core2 and wants to migrate to core1.



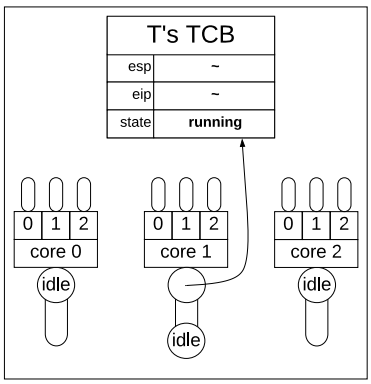
(b) T switches to stagemig_run and puts a pointer to its TCB into the incoming migrations queue at core1. Core1's idle thread does not dequeue TCBs with state $\neq$ stagemig_sto.



(c) T schedules out. The context switching routine sets its state to stagemig_sto after T has stopped running on core2.



(d) The idle thread on core1 observes T in state stagemig_sto and admits it to the ready queue.



(e) T starts executing on the core1.

Figure 4.3: Fast thread migration without helper threads.

mechanism and extending the OSv scheduler's incoming_wakeups dequeuing routine, we implemented above solution as a supplement. As such, in our implementation, the *incoming migrations queue* referred to above is different from the incoming_wakeups queues from Section 4.1.1: We currently use a lock-free multi-producer-single-consumer queue which we originally used to poll for changes in the idle thread. The addition of `MWAIT` however required waiting for changes to the upstream incoming wake-ups queue and our queue simultaneously. We solved this problem by also setting upstream mechanism's bitmask from the stage-switching code path and by using `MWAIT` on its address. Combined with the changes we present in the next Section, we want to merge the two mechanisms in the future, thereby de-duplicating the dequeuing code in the idle thread.

The commits in the Git repository corresponding to this section are `439b5501` and `f00b5872`.

4.4 Thread Migration on Wake-Up

In Chapter 3, we analyzed that the proof-of-concept implementation in Linux at the KIT OS group is not work conserving: It only allows thread migration to be initiated from user-space and thus prohibits migrating threads to idle cores at the time the thread unblocks. In contrast, our solution models stages explicitly and tracks a thread's association with its current stage in the thread's TCB. Utilizing all cores of a multi-core system is thus trivial in theory: When a blocked thread is woken up, we find a CPU that has that thread's current stage's code in its cache and enqueue the woken-up thread's TCB into that CPU's runqueue.

However, in practice, this simple concept is complicated by the realities of the OSv scheduler implementation. Let us first look at the wake-up-related state transitions that a thread might go through in upstream OSv, as depicted in Figure 4.4: The most common case will be that a thread goes to sleep, waiting for some predicate to become true, for example that a timer has expired or that it acquired a mutex. The *running* thread will then transition synchronously to the *waiting* state and schedule out. Let us now imagine that another thread makes the predicate become true. That *waker* knows the threads waiting for this condition and wakes them using the wake() function, which puts the woken thread's TCBs into their last core's incoming_wakeups, which is in turn eventually drained into the core's runqueue.

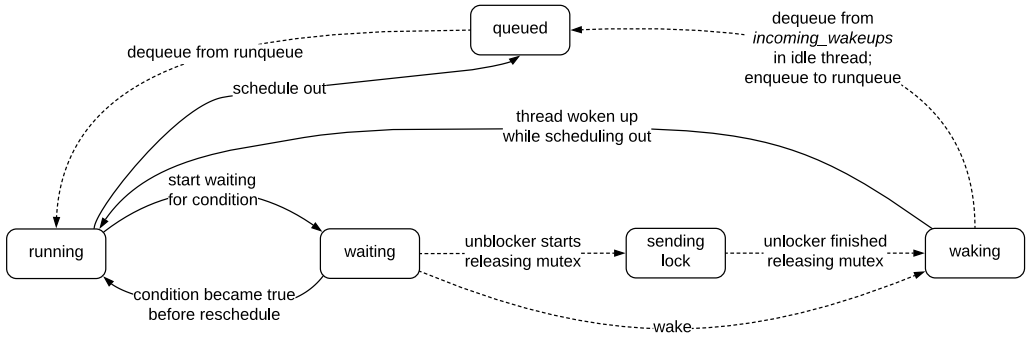


Figure 4.4: Upstream OSv wake-up-related thread state transitions: States *waiting*, *sending_lock* and *waking* are ambiguous with regard to whether the thread is still scheduling out whether it has fully stopped executing. Dashed lines represent events asynchronous to thread execution, solid lines represent synchronous events.

The crucial observation in this section is that the corresponding **OSv thread states do not precisely encode whether the a thread has stopped executing**. In fact, only threads in state *queued* are guaranteed to be stopped and only threads in state *running* are guaranteed to be running. The remaining *waiting*, *sending_lock* and *waking* states are ambiguous: In each of these states, the thread can either have completed scheduling out or it can still be on the way to do that. In the latter case, the thread may even observe that it has already been woken up again by some other thread and move itself from *waking* back to *running*.

Upstream OSv can afford the ambiguous state representation because its wake-up mechanism is limited to ensuring that the woken thread is made runnable again on its current CPU. **Our solution however requires precise knowledge of whether the woken thread is still running or whether it has actually stopped executing on its current CPU.** In the former case, we do not want to migrate it because its current CPU still has a hot cache. However, in the far more common case of waking up a stopped thread, we want to use the opportunity for load balancing, thereby maximizing CPU resource usage: Among the cores that have the to-be-woken thread's stage's instruction working set in their private cache, the least loaded core is chosen as a migration target. And because the woken-thread is not running at the time of wake-up, we can implement remote asynchronous thread migration that works without a chasing helper thread.

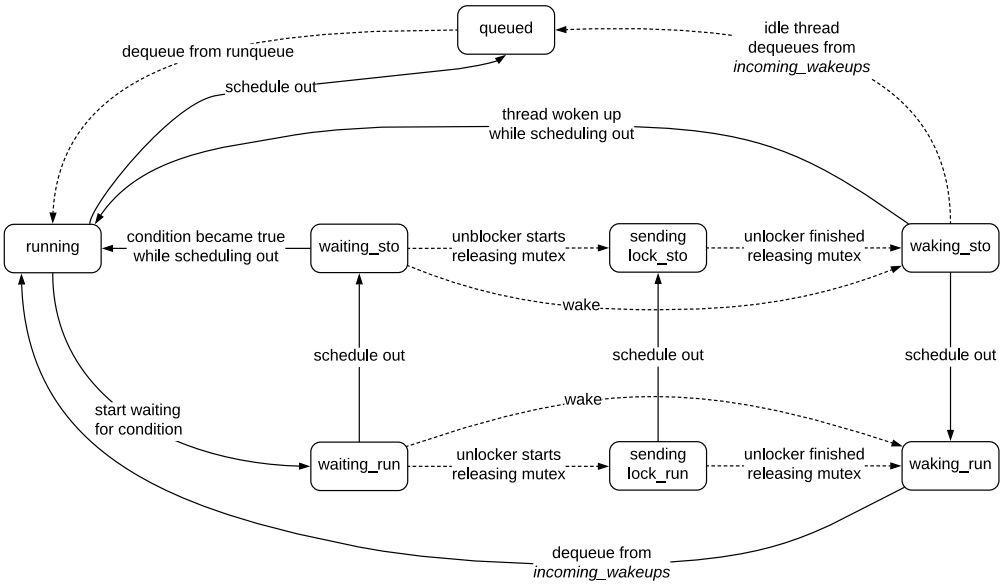


Figure 4.5: Modified OSv wake-up-related thread state transitions: All states are unambiguous about whether the thread is still running or completed scheduling out. Dashed lines represent events asynchronous to thread execution, solid lines represent synchronous events.

The first pillar of our work-conserving scheduler implementation is therefore a significant refactoring of the OSv thread states, the result being depicted in Figure 4.5. As we already did with the *stagemig_run* and *stagemig_sto* states in Section 4.3, we spread the three *waiting*, *sending_lock* and *waking* states into $\star$*_run* and $\star$*_sto* sub-states; the the former suffix represents a still-running thread whereas the latter represents a stopped thread. **The context switching routine then implements the switch from $\star$*_run* to $\star$*_sto* after saving the register state of the thread to its TCB.**[2] An asynchronous waker can thus always distinguish between running and stopped threads.

With the refactoring of the thread states done, we can now discuss our faster alternative for remote asynchronous thread migration. Let us recall some details on upstream OSv from Section 4.1.1:

(a) The waker may run on another CPU than the woken-up thread.

[2]The implementation in the context switching routine is a generalization of the two-step migration mechanism presented in Section 4.3.

(b) The timer implementation requires all timers of a thread to be in the timer list of one CPU.

(c) Remote asynchronous thread migration relies on a short-lived helper thread that chases the to-be-migrated thread and eventually performs the same steps required for synchronous thread migration in the name of the to-be-migrated thread.

The helper thread is necessary because the timer implementation only allows suspending timers on the core they would fire on. Looking at the version control history of upstream OSv, we see that the current timer implementation (commit 786a576e0, 2013) predates asynchronous thread migration support (commit 6bb95442c, 2016). Furthermore, upstream only requires the latter for the POSIX pthread_attr_setaffinity_np syscall. Therefore, we assume that the helper-thread-based solution was considered good enough by the upstream developers. However, we require remote asynchronous thread migration to be fast and thus do not want to pay the price of the helper thread. Also, we cannot afford a remote function call to reprogram the source CPU's LAPIC timer.

Let us therefore reconsider the minimal requirements to make remote asynchronous migration work: We need a waker T_w on CPU C_1 be able to migrate a stopped thread T_s from a remote source CPU C_2 to a remote destination CPU C_3. T_w must

1. remove T_s's timers from C_2's timer list,
2. put T_s's TCB into the incoming migrations queue of C_3 that corresponds to CPU C_1 and
3. set the corresponding bit in C_3's incoming migrations bitmask.

We observe that the timers in step 1 only need to be *removed* remotely and that *resuming* them on CPU C_3 is the job of the CPU-local dequeue operation. We recall that the CPU-local timer lists are sorted in ascending order by expiration date and that the underlying hardware timer is programmed to the expiration date closest in the future. However, it is not a problem if the hardware timer fires and the corresponding timer object is no longer in the list: The implementation will simply re-program the hardware timer to the closest timer in the future, i.e., the list head timer's expiration date. Our solution for remotely suspending timers exploits this property: We protect each CPU's timer list with a recursive spinlock and extend the timer implementation accordingly. The waker can then follow the steps described above.

The source CPU may incur a spurious timer interrupt, but our technique avoids all overhead associated with the original helper thread.

After these substantial refactorings, we finally have an efficient way to perform remote asynchronous thread migration in the context of the waker and thus immediately dispatch a woken thread on a suitable CPU core according to the current core allocation.

The relevant commits for this section are `80d619a8` and `7df1b22f` (state spreading), `13fbe674` (remotely suspend timers) and `ed043e94` (thread migration on wake-up).

4.5 Core Allocation Policy

Up to this point, we have discussed the two main mechanisms of our design: We use synchronous thread migration for stage switching and asynchronous remote thread migration for load-balancing thread migration on wake-up. Also remember that the principal rule of our scheduler is to only dispatch threads on cores that are allocated to the thread's current stage. Both mechanisms therefore migrate a thread to the least-loaded core in the set of cores allocated to the thread's current stage. This section now presents the policy used to periodically compute the core allocation.

The basic concept of our allocation policy is simple: **We define *stage load* as the number of *runnable* threads in a given stage and dedicate CPU cores to stages proportional to the stages' load.** After computing a core allocation, we use it to select a target CPU on stage switch and wake-ups (see Sections 4.3 and 4.4). If a stage has been assigned multiple CPUs, the CPU with the shortest runqueue is chosen. Each CPU schedules its runqueue using non-preemptive round robin. After a given core allocation has exceeded the boot-time-configurable *maximum assignment age* (default: 20ms), we re-evaluate the core-allocation policy using the then up-to-date stage load distribution. It is crucial to observe that this setup creates a feedback-loop: The number of cores available to a stage increases the CPU time available to its runnable threads, thereby avoiding that a single stage becomes a point of congestion. Among the cores of a stage, the load is evenly distributed by always choosing the CPU with the shortest runqueue for migrations.

For a more detailed description of the implementation, let us first summarize the requirements of the rest of the system: When a thread synchronously

switches stages, we need to find a target CPU that is assigned to the target stage. When waking up a stopped thread, we need to find a CPU that is assigned to that thread's current stage. We expect wake-ups and thread switches to be very frequent in a loaded system, mandating that *using* the current core allocation should be in $O(1)$. Additionally, all cores will use the same core allocation concurrently, but at the same time, our design also involves periodic updating of the assignment. The update mechanism must therefore be designed to allow all cores to make progress during an update. Lastly, an update should be minimal in the number of cores that switch to another stage to minimize the sum of capacity misses incurred due to the update.

Our implementation is based around the *assignment* vector which represents a concrete mapping of CPU cores to stages, and the *requirement vector*, which only specifies the number of cores to be assigned to each stage. On a system with C cores and S stages, a requirement is a vector

$$r = \begin{pmatrix} r_1 & r_2 & \dots & r_S \end{pmatrix} \text{ where } r_i \in \mathbb{N}_0 \text{ and } \sum_{i=1}^{S} r_i = C.$$

It is important to emphasize that the components of r must be integers, because CPU cores can only be assigned in integer quantities. A valid assignment a_r derived from r is then a functional mapping of each of the C cores to exactly one stage:

$$a_r = \begin{pmatrix} a_1 & a_2 & \dots & a_C \end{pmatrix}, a_i \in 1 \dots C \wedge \forall s \in 1 \dots S : |\{\, i : a_i = s \,\}| = r_s$$

When requirements change to r' and we need to update the assignment to $a'_{r'}$, our goal must then be to minimize $|\{\, i : a'_i \neq a_i \,\}|$ because this is the number of cores that incurs capacity i-cache misses due to change of the executed instruction working set partition.

We compute the requirements vector following the algorithm in Figure 4.6: At first, we collect the current stage load of every stage in the stage-load-vector. To smoothen out spikes due to frequent switching of threads between blocked and runnable state, we then feed this vector through an exponentially-decaying moving average filter. We compute the requirements vector proportional to the load distribution of the smoothened stage-load-vector. By normalizing the filter result vector to the number of available CPU cores, we get an ideal floating point count of cores to be assigned to each stage. However, since cores can only be assigned at integer quantities, we perform element-wise integer division on the normalized vector, the divisor being the number of assignable cores. The integer quotient is then a

vector that contains the numbers of directly assignable cores, which we add to the new requirements vector. The remainders vector in turn contains the left-over fractions of CPU per stage. We distribute the still unassigned CPU cores by repeating the above procedure with the remainders vector until all cores are assigned. To ensure that this loop terminates, we prioritize cores with higher fractions of CPU cores.

The *minimal transition* from a_r to $a'_{r'}$ is then implemented by building a delta vector $\Delta r = r' - r$: Negative components represent stages with excess cores whereas positive components represent need for cores. A simple nested loop then re-assigns cores between components of Δr and updates the assignment accordingly. Note that the constraints on requirement and assignment vectors imply that $\sum_{i=1}^{S} \Delta r_i = 0$, guaranteeing that a full transfer is always possible. The core re-assignment loop prefers to re-assign cores with a lower index in the global list of cores over those with a higher index, which implies that cores with a higher-index switch stages less often given a stable system load. The current asymptotic runtime complexity of the loop currently is in $O(S^2 * C^2)$. However, we did not spend time on searching lower-runtime algorithms and we do expect most applications to have $S \leq 8$ and $C \leq 32$, which interpolates to $\approx 200us$ runtime, based on the results of the evaluation (see Figure 5.7c, Section 5.4.3). More importantly, the time spent on the computation is amortized because the result is re-used for the time specified via the max_assignment_age boot-time parameter. For future work, it might be worth to investigating more efficient assignment transition algorithms or to consider auto-tuning the max_assignment_age based on detected system configuration.

The last problem of the updating procedure is to support concurrent use of the old assignment while computing a new one. We use OSv's version of read-copy-update [MS98] (RCU) for this purpose: When accessing the current assignment through its public getter, we compare the timestamp of the assignment's creation against the current time. If the max_assignment_age is exceeded and there is no other thread already computing a new assignment, the calling thread becomes the updater. RCU allows all other threads to safely use the old assignment until the new one has been computed.

An important detail of the above algorithm is that a core assignment vector that assigns 0 cores to a given stage is valid. For example, another stage may be assigned all available cores. However, our stage switching implementation requires a target CPU at the time the thread calls the stage switching API. In other words: We do not have a mechanism to queue up threads whose stage has not been assigned a dedicated core. Our solution is to choose the core with the highest index in the global list of cores as the victim core C_v

Require: $l = \begin{pmatrix} l_1 & l_2 & \dots & l_C \end{pmatrix}$, current stage load vector
$\tilde{l} = \begin{pmatrix} \tilde{l}_1 & \tilde{l}_2 & \dots & \tilde{l}_C \end{pmatrix}$, exp.-dec. avg. stage load vector
Ensure: r, valid requirements vector

$\tilde{l} \leftarrow \alpha * l + (1 - \alpha) * \tilde{l}$ ▷ exp.-dec. average
$p \leftarrow \tilde{l} / \|\tilde{l}\|_1$ ▷ norm. load, becomes first priority vector p
$c \leftarrow C$ ▷ number of cores left to assign
$r \leftarrow \begin{pmatrix} 0 & 0 & \dots & 0 \end{pmatrix}$
while c > 0 **do**
    assert $\|p\|_1 == 1.0$
    $f \leftarrow p * c$ ▷ floating-point core count
    if $\forall i \in 1 \dots S : f_i < 1.0$ **then** ▷ priority shifting
        $i_{min} \leftarrow \arg\min_{s \in 1 \dots S} f_s$
        $i_{max} \leftarrow \arg\max_{s \in 1 \dots S} f_s$
        $p_{i_{max}} \leftarrow p_{i_{max}} + p_{i_{min}}$
        $p_{i_{min}} \leftarrow 0$
        continue
    end if
    $r \leftarrow r + \lfloor f \rfloor$ ▷ assign integer core counts
    $c \leftarrow c - \|\lfloor f \rfloor\|_1$ ▷ account for assigned cores above
    $p \leftarrow (f - \lfloor f \rfloor) / \|f - \lfloor f \rfloor\|_1$ ▷ priority based on remainders
end while

Figure 4.6: The algorithm to compute the requirement vector proportional to stage load. From the second iteration on, we distribute cores proportional to the remainders of the previous iteration. Note that because $l_i \geq 0$, $\|\cdot\|_1 = \sum_{i=1}^{S} |l_i| = \sum_{i=1}^{S} l_i$.

which must incur the i-cache miss for the underloaded stages' threads. The reasoning behind this decision is that the described situation should be rare in the first place: If a stage is constantly underloaded, the stage-switching points were likely not chosen for a representative workload. The idea behind choosing C_v and not, for example, C_0, is that the core with the highest index is already advantaged in the *minimal transition* algorithm: C_v will least frequently switch stages and is therefore best-suited to be penalized with cache-thrashing threads from the underloaded stage. Validating and improving this workaround is subject for future work.

The initial implementation of the allocation algorithm was committed in revision `2f2a8628` and refined in `cf16063b`. The requirements vector is computed as described above since revision `1d3b2595`.

4.5.1 OSv's Page Access Scanner

During the implementation of the core allocation policy we found that an OSv system thread called *page access scanner* (PAS) consumes significant CPU time. OSv's main filesystem is a port of the Zettabyte File System (ZFS) which includes its own Adaptive Replacement Cache (ARC). The OSv page cache uses the PAS thread to inform the ARC about recent usage of file-backed pages by checking and subsequently clearing the page table entries' accessed bits. The PAS consists of an infinite loop that self-tunes the CPU time it consumes through a target value $t \in [0.1, 20]$: It scans pages for $t\%$ and then sleeps for $100 - t\%$ of the time it runs, where t depends on a decaying average of the page access rate. The frequency at which the PAS activates is hard-coded to 1000Hz.

The PAS is not pinned to a specific core and would thus be subject to load-balancing in upstream OSv. However, we removed the upstream load-balancer in our solution, which implies that the PAS always executes on cpu0 because it does not use the stage API. As such, it is also invisible in the stage load vector because we track stage load in the stage switching and the reschedule routines, not by summing up the length of a stage's cores' runqueues. Thus, whenever the core allocation policy assigns cpu0 to a stage, that stage's threads will compete with the PAS for CPU-time and cache-residency. Given that a stage's threads' working set is (a) disjoint from the PAS and (b) chosen to barely fit the private cache of a core, increased instruction cache misses are to be expected.

Build PAS freq	throughput Our Solution	Upstream OSv	L2 code miss rate Our Solution	Upstream OSv
10	643.69	549.50	0.39	0.75
1000	552.06	498.22	0.42	0.74

Figure 4.7: The effect of 10Hz vs. 1000Hz PAS frequency on TPC-C throughput and L2 code misses. Reducing the PAS frequency reduces code misses for our solution, resulting in increased throughput. The results were produced with enabled `MWAIT` in the idle thread and 12 TPC-C terminals.

We can quantify the problem outlined above by measuring the effect of reducing the frequency at which the PAS runs from 1000Hz to 10Hz. The results for this experiment are shown in Figure 4.7: With 10Hz, our solution achieves 17% higher throughput compared to 1000Hz. Upstream in contrast is much less affected with only 10% higher throughput at 10Hz PAS frequency. The differences in throughput are visible in the i-cache miss ratios: Whereas our solution incurs 3 percent points fewer code misses with 10Hz, upstream OSv's code misses increase by 1 percent point.

It is important to emphasize that the change to 10Hz causes the PAS to perform less work in both upstream and our solution: By adding additional tracepoints, we found that at 10Hz, the 100x longer time slice available to the PAS is sufficient to scan the entire page table, causing the PAS to go to sleep early. However, as seen in the results above, the 100x less frequent but longer-running invocations of the PAS mitigate the impact on i-cache and throughput for our solution.

We have considered several approaches to handle the default 1000Hz PAS frequency because implementation details of the page cache are technically out of scope for this thesis. At first, one could model cpu0 as a *50 – 80% CPU*. However, this solution is not very general since we lack a precise model for the impact of the cache trashing caused by the PAS. Furthermore, the current implementation would require many special cases to support CPUs in non-integer quantities.

Another idea is to declare one core as *the thrashing core*, exclude it from the core allocation and use it to run the PAS. The natural downside of this approach is that, if the PAS does not run 100% of the time, we waste resources on that core. A mitigation would be to change the dispatch policy for underloaded stages, i.e., those stages that do not have enough relative load to win a single dedicated CPU core: In contrast to distributing threads

of underloaded stages equally over all cores, one could dispatch them all on the thrashing core.

Finally, we considered defining a dedicated stage for the PAS. The main problem with this approach is that our concept is designed for many requests per stage. The fictional PAS-stage however would only ever have zero or one runnable thread, thus likely never win a dedicated core. Instead, it would be frequently dispatched on a random core and evict that core's stage's working set. Experiments showed that this option is not worth pursuing without the concept of a *thrashing core*, at which point one could fully fall back to the proposal in the previous paragraph.

Reducing the PAS Frequency to 10Hz

The primary goal of this thesis is to show that it is possible to implement work-conserving, stage-aware scheduling policy by choosing the right abstractions and mechanisms. Ideally, the corresponding scheduling policy should be competitive without further modifications to the remaining operating system. However, our approach and time constraints did not allow us to solve the problems that the page access scanner poses to our scheduling policy in a general way. **Therefore, for the entire evaluation, we set the PAS frequency to 10Hz in both upstream OSv and our modified version.** Despite the modifications to upstream OSv being technically unnecessary, we consider the comparison between the 10Hz-versions of upstream and our version of OSv to be fair with regards to available CPU time, since both configurations benefit equally from the removed impact of the PAS. The design of a general-purpose solution for the PAS is thus left open to future work.

Chapter 5

Evaluation

In Chapter 4, we presented our operating-system-based solution to mitigate the sub-optimal cache behavior of multi-threaded server applications: We provide a system API to define *stages* which represent spans in the request-handling code path whose instruction working set is smaller than the size of a CPU core's private cache. The OS scheduling policy is aware of these stages and migrates threads to different cores when they switch stages using fast thread migration mechanisms. By dedicating CPU cores to stages, threads always execute code that is in the private cache, thus avoiding expensive off-core cache or memory accesses.

In this chapter, we evaluate our implementation in the OSv library operating system to validate the following theses:

1. Existing applications can adopt our solution with minimal changes required to the existing codebase. We show this by documenting the adoption of the stage API in MySQL 5.6.
2. Our solution improves relevant performance metrics for these applications compared to upstream OSv and virtualized Linux. We show this with the industry-standard TPC-C benchmark.
3. Above performance improvements are due to the stage-aware scheduling technique presented in this thesis:
   (a) Our solution spreads an application's instruction working set over multiple cores. We show this using the top-down performance analysis method and other performance-counter-based metrics.

(b) Our fast thread migration mechanism and migration on wake-up are required for improving net application performance. We show this by implementing the stage API in upstream OSv based on the upstream thread migration mechanisms and by comparing performance of both sides using fixed core allocation.

(c) Our core allocation policy allocates CPUs proportional to stage load. We show this by instrumenting the corresponding code with OSv tracepoints.

(d) Our core allocation policy improves application performance compared to fixed core allocation. We show this using TPC-C with a varying the number of concurrent client connections.

The remainder of this chapter is structured accordingly: Section 5.1 describes the hardware and software setup common to all measurements presented in this thesis. Subsequently, we show the adoption of the stage API in MySQL 5.6 in Section 5.2. Section 5.3 then presents the MySQL performance comparison using TPC-C. The remaining sections cover the validation of our design assumptions: In Section 5.4.1, we show that we successfully spread MySQL's instruction working set over multiple cores. We show the advantages of our thread migration mechanisms in Section 5.4.2 isolated from the influence of the core allocation policy. The core allocation algorithm and its influence on performance is then validated separately in Section 5.4.3 and 5.4.4. We conclude with a discussion of the results in Section 5.5.

5.1 Evaluation Setup

We evaluate our solution on a single-socket Intel Xeon E5-2618L v3 CPU, a Simultaneous multithreading (SMT) system with with 8 cores, 2 hardware threads each. The system is equipped with 32 GiB of memory consisting of $4 * 8$ GiB DDR4 DIMMs clocked at 2133Mhz. The CPU's cache hierarchy consists of a 20MiB shared, inclusive, 20-way set associative L3 cache, private unified 256KiB L2 caches and private separate 32KiB L1-d and L1-i caches. All private caches are 8-way set-associative. Via the BIOS, we disable SMT because the core allocation policy does not support it (see Section 6.1). We also disable automatic switching to lower C-states to reduce latency jitter due to increased wake-up time from low-power states. A more detailed dump of the hardware configuration is available in the appendix (Chapter 6.1).

We use the Fedora 27 Linux distribution with Linux kernel version 4.13.9 (Fedora package 4.13.9−300.fc27.x86_64) as the machine operating system. We remove the measurement noise caused by dynamic voltage and frequency scaling by setting the scaling governor to `performance` mode and disabling the `turbo` frequency range.

Due to the nature of OSv targeting only virtual machines, our evaluation is done using VMs. For all experiments, we use the Fedora-packaged QEMU 2.10.0-1 software, which in turn uses Linux KVM and thus Intel VT-x for hardware virtualization. We use the virtio network and disk paravirtualization drivers in OSv. Disk images are in raw format, located in the host's tmpfs and emulated without caching using pthread-based disk I/O. We observed heavy cache thrashing when allowing QEMU helper threads on the same cores as the vCPU threads. To be able to show the changes in cache behavior that can be achieved with stage-aware scheduling, we thus use the following CPU configuration: We isolate physical cores 2 – 7 from the Linux scheduler using the isolcpus=2−7 kernel command line parameter. The VM is then started with 6 virtual CPU cores (vCPUs) and we map each of the QEMU vCPU threads to one of the isolated cores in range 2 – 7. The remaining QEMU threads which are used for timers, async-I/O, etc., continue to be dispatched by the Linux scheduler on cores 0 and 1.

The goal of the above setup is to give the guest operating system control of the cache state of its vCPUs. Our solution requires dedicated cores and caches because all performance benefits are achieved by optimizing cache behavior. However, we are also unaware how dedicated vCPUs could disadvantage Linux or upstream OSv in the comparisons made in this chapter.

When using Linux as a baseline, we refer to an installation of Debian 9 in a virtual machine that starts the MySQL 5.6.38 Docker image on boot. The MySQL data directory is bind-mounted from the ext4-formatted root partition on the emulated HDD to avoid the overhead of unionfs. For all other aspects, the QEMU configuration described above applies.

With regards to OSv, we emphasize again we had to reduce the frequency of the *page access scanner* (PAS) system thread from 1000Hz to 10Hz (see Chapter 4.5.1). To maintain a fair comparison between our solution and upstream OSv with regards the metrics presented in this chapter, we use the 10Hz frequency on both sides. For all other aspects of the VM configuration, the QEMU configuration described above applies.

For evaluating MySQL, we use the industry-standard TPC-C online transaction processing benchmark. TPC-C simulates an online transaction pro-

cessing workload by modelling a retail company with warehouses and distribution districts that handles sales transactions at sales terminals. The number of warehouses controls the amount of data stored in the database and the number of terminals controls the the number of concurrent clients to the database system. [Cou] We use the TPC-C implementation provided by the open source *OLTP Benchmark* project at Git revision `019d9cf` from February 2018 [OLTP]. The process of provisioning the database with test data is very time-consuming and not relevant to the benchmarks in this thesis. Therefore, we re-use a one-time-generated pre-seed with two warehouses. For the weights of the different TPC-C request types,, we use the default settings in OLTP.

We run the TPC-C benchmark driver from a separate *benchmark machine* connected to the *main machine* using direct 10 Gbit/s ethernet. A dedicated benchmark machine allows us to use high numbers of concurrent benchmark clients without influencing the CPU time available to the QEMU helper threads on the main machine's core 0 and 1. We further use a Linux layer 2 bridge device to enable direct connections from the benchmark machine to the virtual machine, thus avoiding most of the overhead of packet filtering on the main machine.

For capturing performance counter data, we use the Linux `perf` facility in global-monitoring mode on the main machine for CPUs 2-7. We use the `ukHG` event modifiers to capture all events that happen on these cores, regardless of whether they execute code of the main-machine's user-space, kernel or in the guest. We found that without these modifiers, the OSv caused by the application in OSv are not visible because unlike the Linux VM, OSv does not need to change the privilege level when executing application code. For recording more event types than physical PMUs available on the CPU, we use perf's time-multiplexing feature, which re-programs the PMUs at the host kernel's compile-time configurable `CONFIG_HZ` frequency (`CONFIG_HZ=1000` on our system). [Aut]

For orchestrating the execution of VMs, the OLTP benchmark tool and perf, and for collecting the benchmark results, we use a self-written wrapper tool called `doltp` which we make available in the appendix.

We process the result data using Python 3.6, using the the pandas 0.22 package in combination with numpy 1.14.0, matplotlib 2.1.2 and seaborn 0.8.1. All graphs and figures in this chapter are rendered reproducibly from the raw data using code in an IPython notebook, which we make available in the appendix.

5.1.1 Configurable Idle Strategy in Upstream OSv

For the upcoming sections, we require further customizations to the idle thread implementation of upstream OSv. Specifically, we add a kernel command line parameter that allows to select the action taken by the idle thread when there are no other runnable threads in the runqueue. We provide the following alternatives and will refer to them by name for the rest of this chapter:

`busy` Busy-waiting for incoming thread migrations in the incoming_wakeups queue as part of the upstream thread migration mechanism (see Section 4.1.1).

`halt` The default behavior in upstream OSv: Spin for a short time, then issue the `HLT` instruction.

`mwait` Instead of using `HLT`, use `MONITOR` and `MWAIT` to wait for changes to the incoming wake-ups bitmask. Do not spin before `MWAIT`.

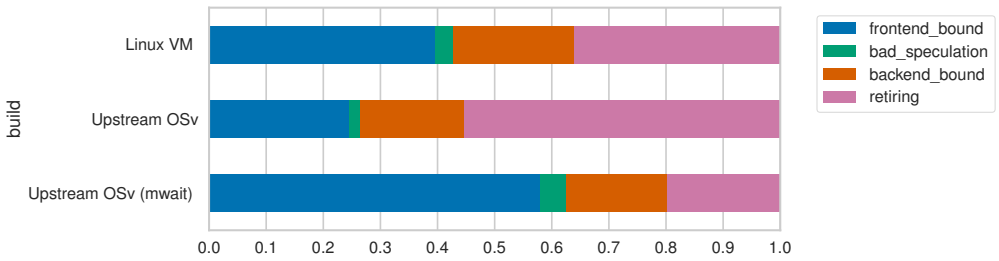
The difference between `mwait` and `halt` is that the latter causes an exit to the hypervisor whereas the former does not. To the hypervisor, `mwait` will thus appear to consume 100% of CPU time.

Furthermore, only `mwait` can be used to measure an undistorted top-down top-level breakdown: Spinning in a tight loop is neither frontend nor backend-bound and will thus inevitably blow up the retired fraction in the breakdown. An example for the impact of spinning can be seen in Figure 5.1a.

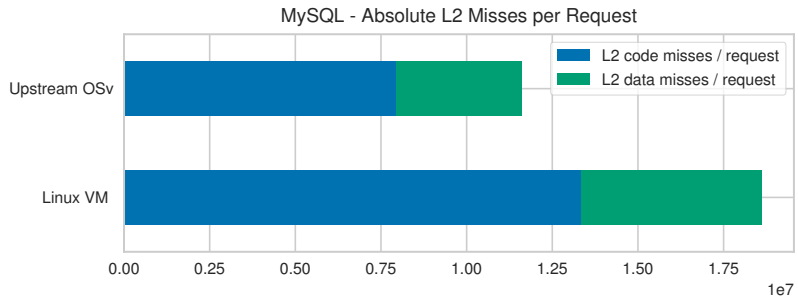
When referring to *upstream OSv* without further qualifiers, we mean upstream OSv with the `halt` configuration. For top-down comparison, we use `mwait` in both our solution and upstream.

5.2 Adopting the Stage API in MySQL

MySQL is a widely-used open source relational database management system written primarily in C/C++. The project started in 1994 but continues to evolve until today. The MySQL server provides its services over a custom TCP protocol and supports concurrent clients through thread-based concurrency: A pool of pre-spawned threads is used to handle incoming connections and additional threads are spawned on demand if the number of concurrent clients exceeds the pool size. For this evaluation, we use MySQL 5.6.38 which consists of more than 1.5 million lines of code. [IW; MySQL; Gmb]



(a) Top-down analysis top-level breakdown. The true frontend-boundedness of MySQL on upstream OSv is only visible when removing spinning in the idle thread, which we did with the `mwait` configuration. The break-down corresponds to the observations made by Kanev et al. (Section 2.6).



(b) L2 cache misses. Although MySQL on OSv incurs fewer L2 misses, the break-down between code and data misses is similar to Linux: More than 66% of all L2 cache misses are code misses, confirming the assumption that MySQL's instruction working set does not fit into the private cache of contemporary processor cores.

Figure 5.1: MySQL on Linux and upstream OSv during a TPC-C benchmark run.

Furthermore, MySQL exhibits the micro-architectural profile of data-center applications observed by Kanev et al. at Google (see Section 2.6): As shown in Figure 5.1a, 58% of pipeline slots during a TPC-C benchmark run are frontend-bound, 18% backend-bound and 5% lost to bad speculation, leaving merely 20% of pipeline slots to be actually retired. Since Kanev et al. attribute the frontend-boundedness to i-cache thrashing, we also measured L2 cache misses (Figure 5.1b): More than 66% of private L2 cache misses in both Linux and Upstream OSv are due to code reads.

All of the above makes MySQL a prime example application that could benefit from our solution: We expose stages to applications via a C++ interface, the existing MySQL codebase is large and thus not trivially refactorable

to the application architectures presented in Chapter 2, and the threading model is compatible with our solution. The L2 metrics further support the assumption that MySQL can benefit from spreading its instruction working set over multiple cores. At last, given the recent trend of deploying applications as virtual appliances, we expect that OSv's strong requirement for this type of deployment is acceptable due to the performance improvements that our solution delivers.

As described in Section 4.2, the stage API requires developers to insert stage switching calls into the request-handling code path such that the underlying thread migration spreads the instruction working set over the available CPU cores. Our initial approach for this task was to read the MySQL source code, starting at the connection handler and figuring out which of the called code modules are large enough to become separate stages. As a result, we used the RAII-style switching to separate

- the code interfacing with the network stack (function do_handle_one_connection),
- the SQL parser (function dispatch_command) and
- the code path handling *insert* queries (function mysql_insert).

The above process was very time-consuming because we were not familiar with the MySQL code base and the resulting throughput measured by TPC-C was too unstable to make it worth pursuing the approach any further.

In parallel to this thesis, staff at the KIT OS group worked on a simulation-based approach to automatically find optimal stage switching points. At first, a memory access trace is recorded using a full-system simulator while running a representative workload. In addition to the access trace, a log of the function calls made during each incoming request is made. The second step is then responsible for finding the optimal position of N stage switching points: For every combination of N different functions, it replays the memory access trace in a cache simulator and counts the cache misses. The combination of functions with the minimal number of misses is then chosen as stage switching points under the assumption that they represent an optimal working set partitioning. A developer then manually inserts the switching calls directly before the function call. [Got+18]

For this thesis, we used the above approach to find an optimal partitioning for 3 stages. A larger number of stages was impractical due to the runtime of the above algorithm. The resulting diff statistics are printed in Figure 5.2: The changes amount to 54 insertions and two deletions throughout nine files,

```
git diff --stat 38e2b4d1fe7564b8e1a0d0f0eeb7f542d0d534b6
 CMakeLists.txt              |  5 ++++-
 sql/CMakeLists.txt          |  5 ++++-
 sql/my_osv_stagesched.cc    | 18 ++++++++++++++++++
 sql/my_osv_stagesched.h     | 14 ++++++++++++++
 sql/mysqld.cc               |  5 +++++
 sql/sql_class.h             |  1 +
 sql/sql_lex.cc              |  3 +++
 sql/sql_parse.cc            |  2 ++
 storage/perfschema/pfs.cc   |  3 +++
 9 files changed, 54 insertions(+), 2 deletions(-)
```

Figure 5.2: The Git diff statistics of all changes to upstream MySQL required to insert the stage-switching points computed by the simulation approach. They amount to only 0.0036% of the MySQL codebase.

and 32 lines of these insertions are isolated in two new files prefixed with `my_osv` that contain the declaration and definition of stage pointers as global variables. This change amounts to only 0.0036% of the lines of code present in MySQL and is simple to maintain across patch-level changes. The design goal of a slim stage API can therefore be considered fulfilled.

However, there are a number of obvious points for criticism in the above approach: First, the cache-simulator-based switching points are located at semantically unintuitive positions in the source code. Thus, although the patch is small in size, a human developer can only port it across patch-levels of a given version of MySQL, hoping that the instruction working set does not change significantly. We expect new major versions to require a one-time re-run of the simulation process. Second, the patches are specific to the simulated cache hierarchy, which implies that different patches and thus different builds may be required for deployment on a fragmented hardware infrastructure. However, it should be noted that the very popular Intel x86-64 micro-architectures since *Nehalem* (2008) have used the same cache sizes and associativity per core, although *Skylake* server processors have moved form 256KiB to 1MiB L2 cache in 2017 [NehAnand; SkyWch]. At last, one can argue that the promised ease-of-use is actually limited by the requirement to use the cache simulator to find switching points. Indeed, we can imagine a lot of improvements to the tooling but leave this subject to future work.

5.3 Whole-System Benchmark with MySQL

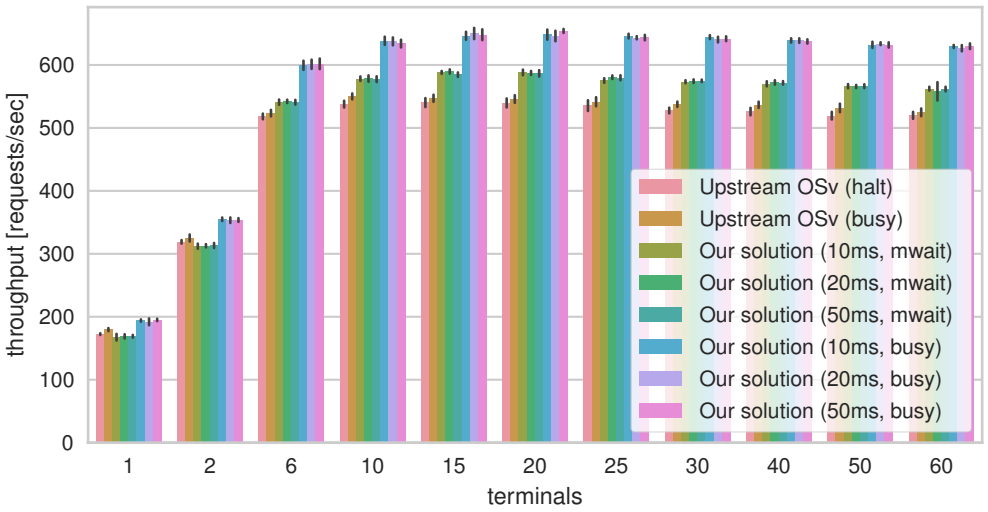
In this section, we show that stage-aware scheduling is able to improve performance of real-world applications by the example of MySQL, which we adapted to use the stage API in the previous section. We use the TPC-C benchmark setup as described in Section 5.1 to compare the performance of MySQL running on our solution vs. upstream OSv vs. the Linux VM.

For the purpose of this thesis, the aspects of multi-client and multi-core scalability are of particular interest: MySQL implements thread-based concurrency, i.e., it uses one operating system thread per client connection and relies on the OS scheduler to utilize multiple CPUs effectively. In Chapter 2, we have presented related work showing this model's negative effect on cache behavior: A request handler's instruction working set does not fit into the private cache of an individual core, thus causing cache thrashing and lost performance. As laid out in Chapter 4, our solution avoids this problem, expecting performance gains due to always-warm instruction caches.

The most important performance metrics for a database system under OLTP workloads are *throughput*, measured in completed requests/second, and the *request latency*. To show the effectiveness of our solution, we measure these metrics over a varying number concurrent client connections (terminals). For each data-point, we perform 8 40s TPC-C benchmark runs with identical parameters. We discard the result data of the first and last two seconds to exclude warm-up and tear-down effects of the benchmark implementation. We also vary the max_assignment_age parameter to experiment with its influence on application performance.

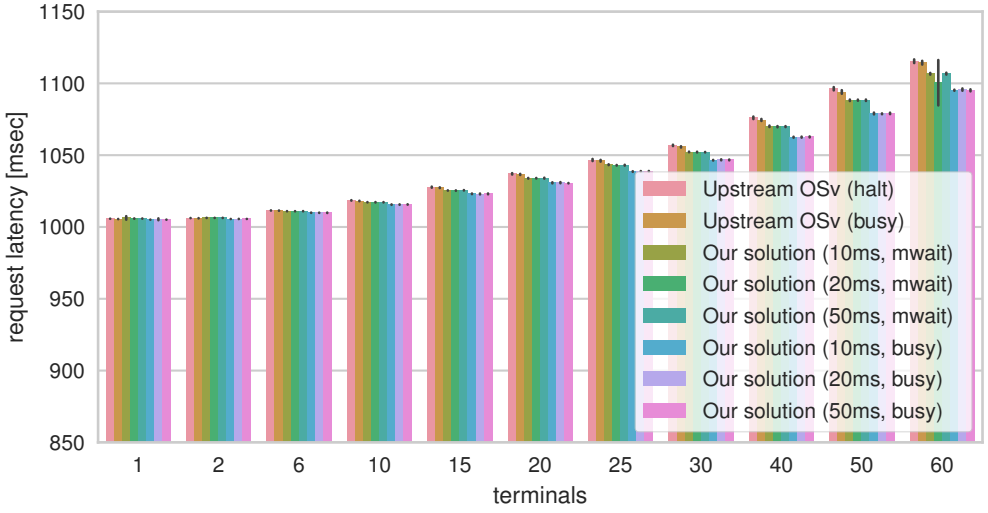
With regards to throughput (Figure 5.3a), our solution achieves a speedup of up to 22% at 50 TPC-Terminals. Whereas upstream OSv throughput is stagnant after the number of terminals exceeds the number of available hardware threads (6 vCPUs), our solution's maximum throughput is achieved at 20 terminals. The influence of the max_assignment_age is negligible. In contrast, the influence of `MWAIT` vs. busy-waiting is significant: We achieve more than 2x the speedup with busy-waiting compared compared to `MWAIT`.

Response time is much less affected by our solution than throughput, as is shown in Figure 5.3b. Only at very high numbers of concurrent clients do we observe lower response times with our solution compared to upstream OSv. However, even at 60 terminals, the difference amounts to merely 15ms at mean latencies of 1110ms vs. 1095ms.



terminals build	1	2	6	10	15	20	25	30	40	50	60
Our solution (20ms, busy)	1.11	1.11	1.16	1.19	1.20	1.20	1.20	1.21	1.21	1.22	1.21
Our solution (20ms, mwait)	0.98	0.98	1.05	1.08	1.09	1.09	1.08	1.09	1.09	1.09	1.07
Upstream OSv (busy)	1.04	1.02	1.01	1.02	1.01	1.01	1.01	1.02	1.02	1.02	1.01
Upstream OSv (halt)	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

(a) TPC-C throughput and corresponding speedup with upstream OSv as the baseline. Our solution is 22% faster than upstream with busy-waiting 9% faster when using `MWAIT` in the idle thread. The `max_assignment_age` parameter (left in parentheses) does not significantly affect the performance of our solution.



(b) TPC-C request latency. Our solution achieves up to 15ms lower request latency at 60 terminals. Note however that this is only 1.5% faster than upstream and that the y-axis in the diagram starts at 850msec.

Figure 5.3: Whole-system benchmark of MySQL on upstream OSv vs. our solution using the TPC-C benchmark. The error bars indicate the standard deviation over the 8 benchmark runs per configuration. The legend entries in top-to-bottom order correspond to the bars in left-to-right order.

5.4 Validation of Design Assumptions

Although the results above show that our solution achieves up to 22% higher throughput in the TPC-C benchmark, we must still validate that our design decisions are responsible for these improvements.

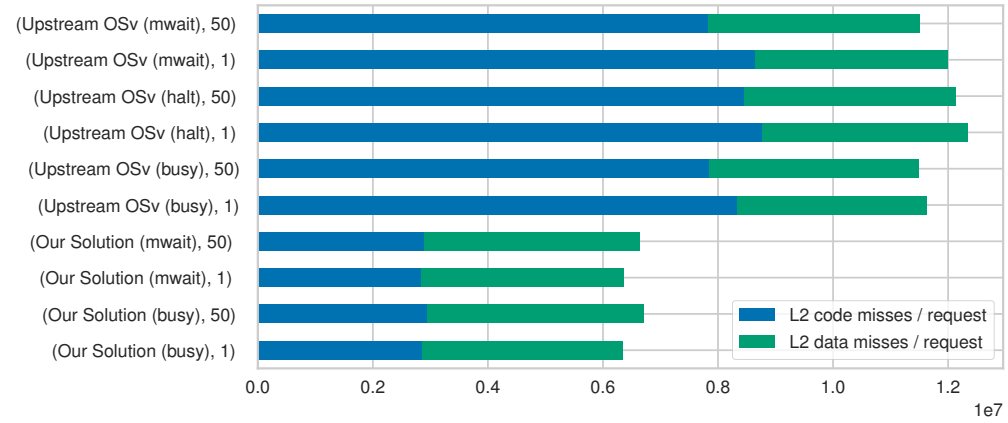
In the following subsections, we thus isolate individual aspects of our design and show that they have the intended effect. In Section 5.4.1, we show that our solution actually spreads MySQL's instruction working set over multiple cores. Subsequently, in Section 5.4.2, we show the influence of fast thread migration and thread migration on wake-up. Lastly, we validate the behavior of the core allocation policy and evaluate it's influence on application performance.

Given the observation that the max_assignment_age parameter has negligible impact on throughput, we use the default value of 20ms. However, we continue to perform measurements with both busy-waiting and `MWAIT` in the idle thread to find the reason for this parameter's significant influence on throughput.

5.4.1 Instruction Working Set Spreading

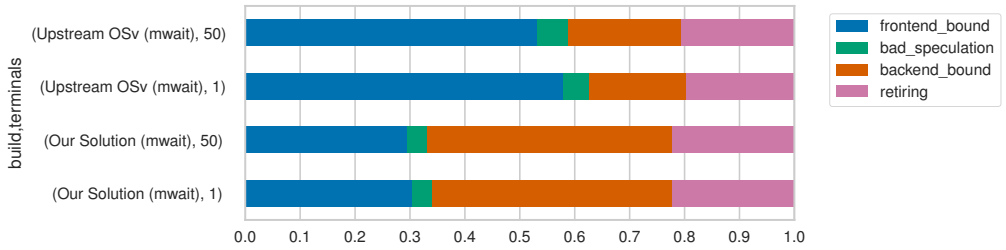
To validate our primary design goal of spreading the application's instruction working set over multiple cores, we use hardware performance counters and the Linux `perf` utility to capture events on the vCPUs while executing the TPC-C benchmark. Specifically, we record L2 cache events as well as those events required to compute the top-down analysis top-level breakdown (see Section 2.5).

Figure 5.4a and 5.4b visualize the results of the run described above. We observe a reduction of L2 code misses per request by up to 65% while increasing data misses by at most 2%. `MWAIT` vs. busy-waiting does not affect the cache behavior of our solution significantly. However, we see increased code misses in upstream when using `HLT` vs. `MWAIT` and busy-waiting. Our explanation for this observation is that calling `HLT` in the VM causes an exit to the hypervisor and thus additional code misses. With regards to single vs. multiple terminals, we see that our solution only shows increased data misses with unchanged code misses. Increased data misses are to be expected given that more requests are likely to touch more (different) data in the same time interval, thus inducing capacity misses. Upstream shows a reduction in code



build	L2 code misses / request	L2 data misses / request
Our Solution (busy)	0.35	1.02
Our Solution (mwait)	0.34	1.02
Upstream OSv (busy)	0.93	0.99
Upstream OSv (halt)	1.00	1.00
Upstream OSv (mwait)	0.93	1.00

(a) L2 cache misses with 1 and 50 TPC-C terminals. The table below shows the relative amount of code and data misses for 50 terminals with *Upstream OSv (halt)* as a baseline. The 65% reduction of L2 code misses shows the effectiveness of instruction working set spreading.



build	terminals	frontend_bound	bad_speculation	backend_bound	retiring
Our Solution (mwait)	1	0.31	0.04	0.44	0.22
	50	0.30	0.04	0.45	0.22
Upstream OSv (mwait)	1	0.58	0.05	0.18	0.20
	50	0.53	0.06	0.21	0.21

(b) Top-down analysis top-level breakdown at 15 terminals using `MWAIT` in the idle thread. Our solution shifts 27 percent points from *frontend-bound* to *backend-bound* and *retiring*.

Figure 5.4: Validation of instruction working set spreading using CPU performance counters during the TPC-C benchmark.

misses with multiple terminals, which is likely due to code sharing between threads scheduled on the same core.

For the top-down top-level breakdown, we compare the `MWAIT` versions of upstream and our solution, which is necessary because spinning blows up the *retiring* fraction because a tight loop is neither frontend nor backend-bound. The results are depicted in Figure 5.4b: Up to 27 percent points from *frontend-bound* and 1 percent point from *bad speculation* shift to the *backend-bound* and *retiring* category. As explained in Section 2.5, this shift is expected given the reduction in L2 code misses.

We interpret the above results as confirmation that we successfully spread the instruction working set of MySQL over multiple cores. The use of `MWAIT` does not correlate with the cache behavior of our solution, i.e., the quality of instruction working set spreading is independent of the sleeping strategy.

5.4.2 Changes to Thread Migration

The analysis in the previous section shows that we are successfully spreading the instruction working set of MySQL, which we accomplish through thread migration. In Section 4.3, we have presented a thread migration mechanism that we claim to be faster than the upstream OSv facility. Additionally, we refactored the wake-up mechanism to allow thread migration on wake-up, enabling immediate dispatch of a woken-up thread to any of its current stage's cores (see Section 4.4). In this section, we validate that we reduced migration latency with our migration mechanism, its effect on throughput and the additional speedup achieved through thread migration on wake-up.

As a baseline for comparison, we implement the stage API in upstream OSv using the upstream thread migration primitives: Specifically, we disable the load balancing mechanism and implement thread migration on stage switch using the thread::pin(thread*, cpu*) remote asynchronous thread migration API (see Section 4.1.1 and file `core/sched.cc` in [Sys17]). We cannot not use the synchronous API (thread::pin(cpu*)) for this purpose because it reproducibly crashes with failed assertions in the upstream timer implementation when invoked at high frequency [Sch]. We configure both upstream and our solution to use fixed core allocation with two dedicated cores per stage. Both implementations choose the core with the shorter runqueue length as migration target. In addition to the configurable idle strategies in Section 5.1.1, we also implement an option to disable pre-halt spinning in the idle thread to avoid distortions to the top-down top-level breakdown. For brevity, we

will refer to the pre-`HLT` or pre-`MWAIT` spinning as `prespin` and for direct `HLT` or `MWAIT` as `nospin`.

We measure stage switching latency using a self-written micro-benchmark (`hopper`). We configure the benchmark to define 3 stages and perform 100000 round-robin stage migrations. `hopper` discards the first 1000 iterations and collects the remaining results in a histogram with 200ns buckets. Figure 5.5 shows the results: Our solution achieves a mean latency of 1300ns with more than 97% of migrations under 1400ns. The influence of `MWAIT` is visible in form of a slightly wider-spread distribution and an increased mean value 1660ns, amounting to 360ns additional mean latency.

Upstream OSv achieves 6146ns mean latency in the `mwait-nospin` configuration and 6528ns in `halt-nospin` configuration. As can be seen in the histogram, `mwait-prespin` and `busy` achieve practically equivalent latencies at $\approx$ 4300ns whereas the `halt-prespin` configuration has approximately the same offset to `mwait` in the `nospin` case. For the micro-benchmark, the small difference between the `prespin` configurations is unsurprising: The benchmark does not perform any work in a stage and switches immediately to the next one. Therefore, it reaches the first stage again before both of its cores went to sleep.

We were unable to find an explanation for the different penalties of `MWAIT` for our solution vs. upstream (+360ns vs. +1800ns). However, the more important observation is that even with busy waiting, our solution achieves more than 70% lower latency: The mean value of 1660ns with `MWAIT` is comparable to the bare-metal IPI propagation of 1700ns measured in [She13], which is exceeded by more than 4800ns by upstream OSv. At last, since busy-waiting in upstream and our solution have the same propagation time through the cache coherence protocol, we can be certain that all additional migration latency with busy-waiting in upstream OSv is due to interrupt virtualization and the performance implications of helper threads in upstream OSv.

Despite the insights presented above, we must still investigate whether our improvements to thread migration have any practical effect on application performance. We use the TPC-C benchmark and our stagified version of MySQL to compare the throughput achieved with the stage API implementation on upstream OSv vs. our solution.

The results are presented in Figure 5.6. With a single terminal and busy-waiting, our solution achieves 8 – 9% higher throughput. At 15 terminals in contrast, we achieve 18% increased throughput, which is further sustained for higher terminal counts. We interpret the single-terminal speedup as the

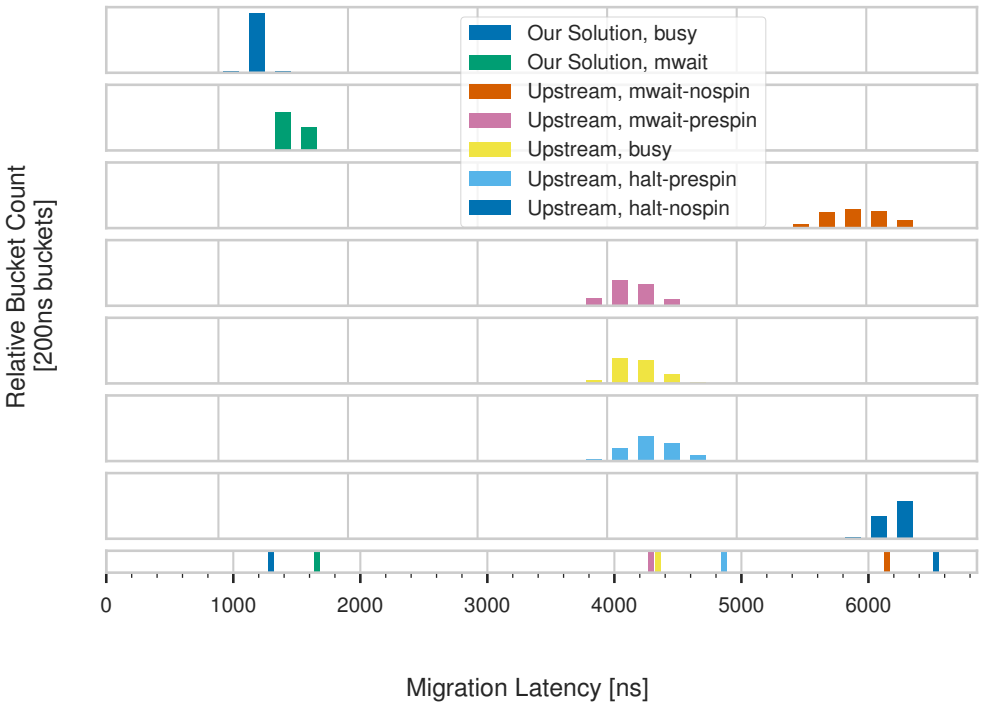
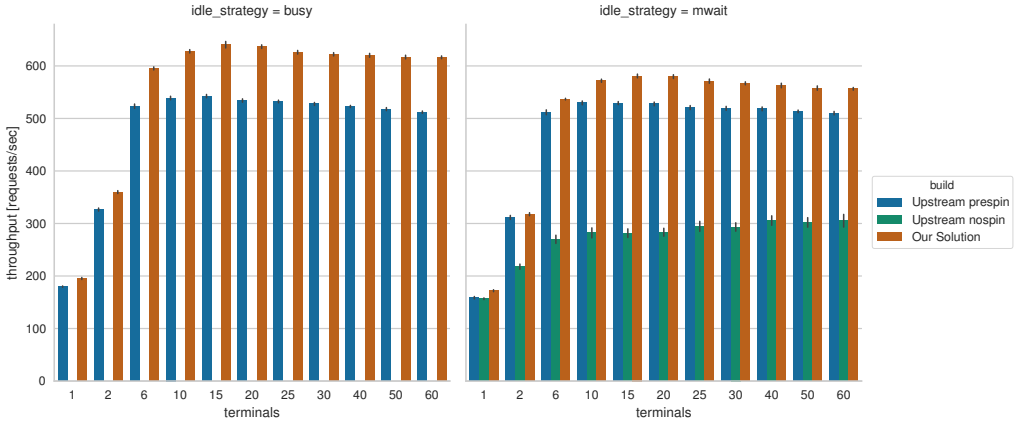


Figure 5.5: Histogram of `hopper` migration latencies with 200ns buckets, bars drawn at the center of each bucket. The last row shows the exact mean values. Our solution shows more than 70% lower migration latency and is less affected by `MWAIT` than upstream.

influence of reduced migration latency: With a single client, there is always at most one runnable request-handler thread in the system, hence no need for thread-migration on wake-up. For multiple terminals, the additional 10 percent points in throughput can consequently be attributed to thread-migration on wake-up.

Looking at the effect of `MWAIT` vs. busy-waiting, the 90% higher throughput of `upstream prespin` and `upstream busy-waiting` vs. `upstream nospin` is very prominent. In contrast, our solution is only 10 – 14% faster with busy-waiting vs. `MWAIT`. These effects correlate with the different migration latency increases observed with the micro-benchmark (+1800ns vs +360ns) for which we do not have a plausible explanation. Since the migration latency increases are not the same, we cannot clearly attribute the throughput differences between upstream and our solution to either thread-migration on wake-up or absolutely lower migration latency.

Given the results presented in this section, we can conclude that our changes to thread migration are the most relevant contributor to the total 22% increase of throughput achieved by our solution (see Section 5.3). Although both migration latency and thread-migration on wake-up have been shown to provide performance improvements in the busy-waiting case, the relationship of `MWAIT` to migration latency and throughput requires further investigation in future work.



build	Our Solution			Upstream nospin			Upstream prespin		
terminals	1	15	50	1	15	50	1	15	50
idle_strategy									
busy	195.0	640.0	617.0	N/A	N/A	N/A	180.0	543.0	518.0
mwait	172.0	581.0	558.0	157.0	282.0	302.0	159.0	529.0	513.0

Figure 5.6: TPC-C throughput achieved with fixed core allocation over varying number of terminals. The error bars represent standard deviation over 8 benchmark runs. Our solution increases single-terminal throughput by 8 – 9% and up to 18% for multi-terminal with busy-waiting. `upstream pre-spin` is very similar to `upstream busy-waiting`, whereas `upstream nospin` is almost 50% slower, showing that `upstream pre-spin` practically never actually halts the CPU.

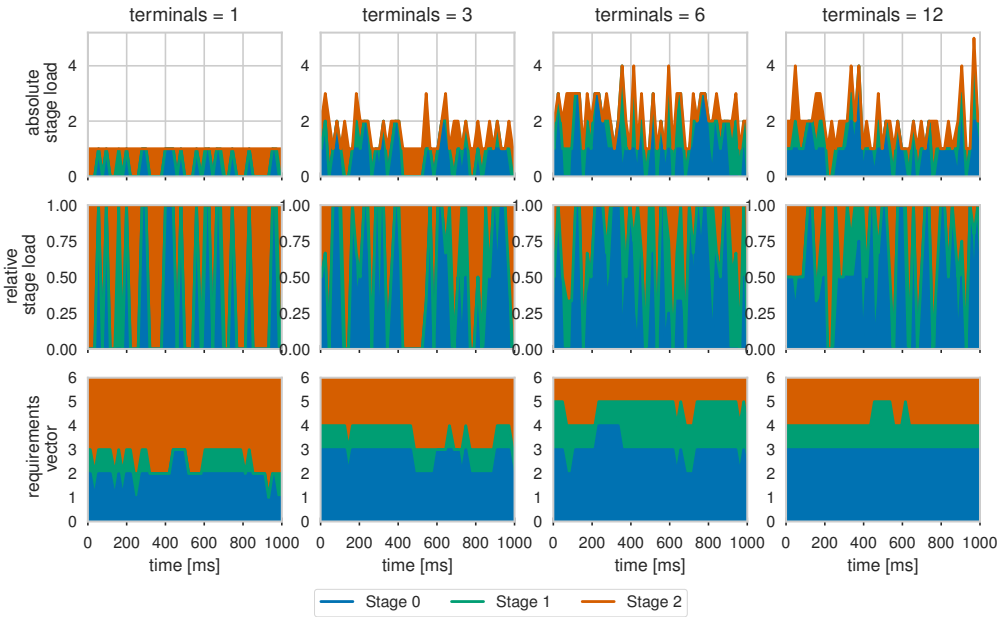
5.4.3 Core Allocation: Validation of Implementation

The last major component of our design is the core allocation policy. We validate that it works as designed by instrumenting the allocation code with OSv tracepoints, recording the momentary stage load vector, the new core assignment and the total execution time of the core allocation algorithm in nanoseconds. We use the default max_assignment_age of 20ms while executing the TPC-C benchmark with 1, 3, 6 and 12 terminals. The figures referred to in this paragraph present a 1000ms window of a manually captured trace, which amounts to 50 invocations of the core allocation routine. We check relevant data structure invariants at runtime and will thus rely on visualizations to validate that the overall idea works as intended.

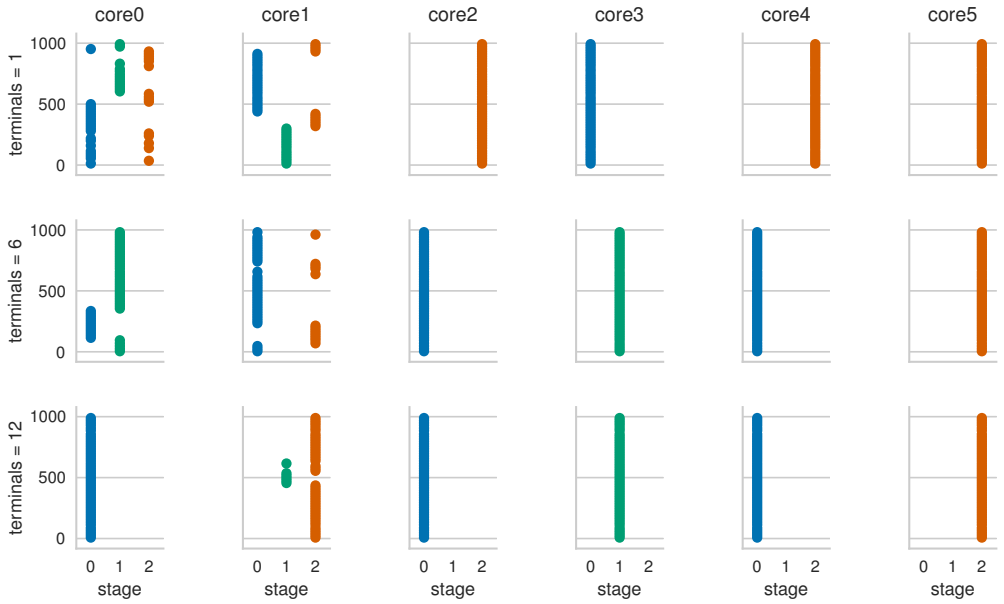
As can be seen in Figure 5.7a, the raw stage load (number of runnable threads per stage) is very spiky and only grows slowly with the number of TPC-C terminals. As explained in Section 2.1, this pattern is typical for I/O bound servers: We assume that MySQL's request handlers frequently switch between runnable and blocked state. The spiky load profile is smoothened by the exponentially-decaying average filter, which can can be well observed in the first column: Stage 0 load rises shortly before the 400ms mark but its corresponding entry in the requirements vector only rises from 2 to 3 cores at about 420ms. Also, we see that the number of cores assigned to a stage only rarely drops to 0, which — combined with the L2 metrics from Section 5.4.1 — confirms that our stage switching points were well chosen: All stages are used sufficiently to be assigned at least one core most of the time.

The decisions of the *minimal transition* algorithm are visualized in Figure 5.7b: As designed, we only see the cores with a lower index switch stages frequently whereas cores with a higher index clearly tend to keep the same stage. Since we do not have a precise model of the opportunity costs of thread migration vs. cache thrashing, we consider avoiding flapping of cores a positive result.

At last, we take a look at the runtime of the entire core allocation algorithm in Figure 5.7c: The runtime is obviously independent of the number of threads in the system (around $1\mu s$) but exhibits a very high standard deviation. Given the fact that the runtime is amortized over 20ms, less than 0.1% of CPU time will be spent on the scheduling algorithm. However, we want to emphasize that we did not show the critical $O(S^2 * C^2)$ asymptotic runtime complexity of *minimal transition* algorithm in this section, since the evaluation setup is fixed to $S = 3$ stages and $C = 6$ cores.



(a) Stage load and resulting requirements vector over time. The exponentially-decaying average filter smoothens the very spiky stage load profile.



(b) Core Assignment over time (time on y-axis). The minimal transition algorithm leads to stable allocations, in particular for higher core counts.

terminals	mean runtime [ns]	stddev runtime [ns]
1	872.825261	432.738347
6	1128.008584	992.392119
12	1093.613082	612.995302

(c) Runtime of the algorithm, independent of the terminal count.

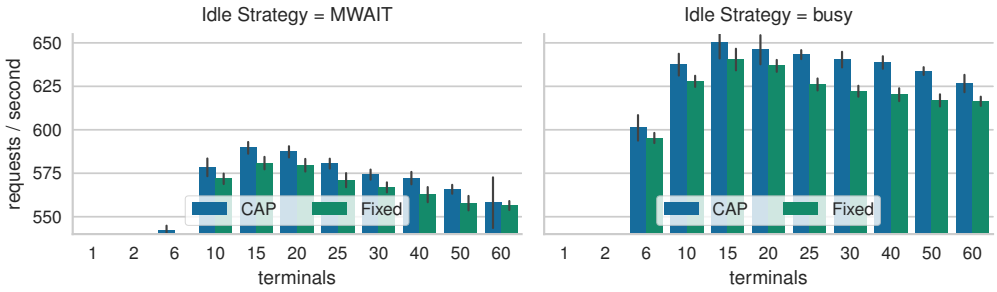
Figure 5.7: Visualization of the core allocation policy decisions.

5.4.4 Core Allocation: Performance Evaluation

After having validated that the core allocation policy works as designed in the previous section, we will now evaluate its effect on application performance. As a baseline, we use our version of OSv with fixed core allocation and two dedicated cores per stage. We continue to use the length of the runqueue to decide on which core within a stage a thread should be dispatched. We use the TPC-C benchmark with a varying number of terminals to evaluate MySQL performance and concurrently capture L2 metrics on the vCPUs. Additionally, we perform two separate runs with `MWAIT` vs. busy-waiting in the idle thread. The comparison between fixed core allocation and the core allocation policy is fair with regards to available CPU resources because we define exactly 3 stages for MySQL, resulting in two dedicated cores per stage.

As is visible in Figure 5.8b, the maximum increase in throughput achieved by the core allocation policy is only 3%, starting at 25 TPC-C terminals and higher. Additionally, these 3% are only reached when using busy-waiting — the speedup with `MWAIT` is at most 2%. Further, especially for `MWAIT`, the improvement lies within the standard deviation and is thus not very relevant. The L2 cache metrics show that the core allocation policy reduces codes misses per request by up to 5%. However, data misses are increased by 1 – 2%. The impact of dynamic core allocation on request latency is negligible (not shown in the figure).

Since these improvements are very minor compared to what we observed in Section 5.4.2, it would be rather easy to dismiss dynamic core allocation as unnecessary. However, the evaluation setup we used in this section is very beneficial to fixed core allocation: The fixed core allocation is equivalent to the core allocation policy if (a) the stage switching points are perfectly placed with regards to stage load and instruction working set and (b) the number of vCPUs is a multiple of the number of stages. If these two criteria are true, the core allocation proportional to stage load will always result in 2 cores per stage, which is equivalent to the fixed core allocation presented in this example. We have already shown in Section 5.4.3 that our stage switching points are very close to an ideal placement, which is likely due to the simulation-based approach that was used to find them (see Section 5.2). Because we use 6 vCPUs for 3 stages in MySQL, condition (b) is also true.



(a) TPC-C throughput. The y-axis is off-set. Error bars represent standard deviation.

	Terminals Idle Strategy	1	2	6	10	15	20	25	30	40	50	60
TPC-C Throughput	MWAIT	0.98	0.98	1.01	1.01	1.02	1.01	1.02	1.01	1.02	1.01	1.00
	busy	0.98	0.98	1.01	1.02	1.01	1.01	1.03	1.03	1.03	1.03	1.02
TPC-C Request Latency	MWAIT	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99
	busy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
L2 code misses / request	MWAIT	1.00	0.96	0.97	0.99	0.97	1.00	1.00	0.99	0.97	0.96	1.00
	busy	0.97	0.96	0.96	0.99	0.99	0.98	0.98	0.95	0.98	1.00	0.97
L2 data misses / request	MWAIT	1.03	1.01	1.01	1.00	1.01	1.02	1.01	1.01	1.02	1.01	1.01
	busy	1.05	1.01	1.01	0.99	0.98	1.01	1.01	1.01	1.02	1.02	1.00

(b) Speedups.

Figure 5.8: Comparison of the core allocation policy vs. fixed core allocation with varying number of terminals and idle thread strategy. The throughput improvement of dynamic core allocation is very minor (2 – 3%). However, the evaluation setup is optimal for the baseline, thus mandating future work to evaluate the true effectiveness of dynamic core allocation.

5.5 Discussion

As shown in in the previous section, our stage-aware scheduling solution is able to improve the throughput of the multi-threaded MySQL database server by up to 22%. The speedup is sustained at a high number of concurrent clients. In contrast to previous work in the field, we do not sacrifice request latency for improved throughput.

By stagifying MySQL, we have shown that our solution requires very little modifications to an existing legacy code base to reap the provided performance benefits. However, we have learned that a simulation-based approach [Got+18] for finding stage switching points is preferable over manual placement by a human developer.

Our thread migration mechanism achieves $\approx$ 1700ns migration latency which is more than more than 70% lower than the corresponding upstream OSv primitive. The lower overhead of our mechanism is responsible for an $8 - 10\%$ speedup in throughput with a single client compared to an implementation based on upstream thread migration. Additionally, we achieve another 10 percent points in throughput improvement by implementing thread migration on wake-up and using it in our scheduling policy, thereby overcoming the limitations of the KIT OS group's proof of concept implementation in Linux.

We have shown that our dynamic core allocation policy works as designed by analyzing traces of the allocation algorithm. The performance evaluation has shown that for our benchmark scenario the contribution of dynamic core allocation to the total speedup is only $2 - 3\%$ percent points.

Although this chapter has shown that our design works as intended and that it improves net application performance, the core allocation policy requires further evaluation. For future work, we plan to conduct experiments using different load profiles and less-optimal stages switching points. Additionally, a mathematical model of the opportunity cost of thread migration vs. incurring i-cache misses would help to define an optimal core allocation policy. The optimal policy would then serve as a scale for evaluating real policies. At last, further experiments with both lower and higher CPU core counts as well as different cache hierarchies are required to show the general applicability of our solution.

Chapter 6

Conclusion

In this thesis, we have presented a scheduler design that leverages fast thread migration to spread a multi-threaded server's instruction working set over the private caches of an SMP system. Instruction working set spreading improves application performance because datacenter applications have been shown to thrash the private i-caches of contemporary processors. Our implementation in the OSv library operating system and the evaluation with the MySQL database management system show that our solution increases throughput by up to 22% and reduces i-cache misses by 65%.

Previous work in the field from the early 2000s unnecessarily trades off request latency for throughput by batching requests to achieve the re-use of warm i-caches. We argue that these designs are outdated due to the ubiquitous availability of SMP systems which enable techniques such as computation spreading. We have presented the groundwork for our thesis done at the KIT OS group: A software-only approach for computation spreading implemented as a prototype in Linux that allows applications to partition a thread's instruction working set at arbitrary points in the application code. By migrating threads to different cores at these points, the instruction working set is spread over the private i-caches of all cores and i-cache misses are reduced.

Our contribution is an analysis of the root cause for the lack of multi-core and multi-client scalability in the Linux prototype, the definition of clean OS abstractions for the concept of computation spreading and a work-conserving thread scheduler design implemented in the OSv library operating system. We define stages as operating system objects that represent spans in the code path of a server application's request handler threads. Each stage is

smaller than the private i-cache of a single core. We further track a thread's current stage in its TCB, enabling the implementation of a scheduler that takes cache-residency of a thread's instruction working set into account. Our scheduling policy time-multiplexes cache-residency by dedicating cores to stages for certain time intervals. The scheduler dispatches a thread in a particular stage only onto those cores that are assigned to its current stage, which causes all threads to execute on pre-warmed i-caches.

The evaluation has shown that we achieve 22% increased throughput in an optimal configuration and 9% increased throughput when using the power-saving `MWAIT` instruction when waiting for incoming threads in the thread migration mechanism. Both configurations reduce L2 code misses by 65%. Request latency is also slightly reduced, confirming our thesis that an SMP-oriented design can avoid the penalties on request latency of the early 2000's proposals. We have shown that fast thread migration and the integration of our policy into the scheduler's wake-up mechanism are necessary to achieve the performance improvements described above. At last, we have confirmed that our core allocation policy works as designed and that its contribution to the overall throughput speedup is at most $2 - 3$ % compared to fixed core allocation. However, we provide arguments to show that fixed core allocation is not a suitable baseline for evaluating the contribution of dynamic core allocation to the observed performance improvements.

6.1 Future Work

We have already discussed the limitations of the evaluation in Section 5.5: Due to lack of a known optimal allocation policy, we chose fixed core allocation as a baseline for evaluating our dynamic core allocation policy. However, the choice of migration points and the fixed evaluation setup of 3 stages and 6 cores advantages fixed core allocation, leaving little room for improvement with dynamic core allocation. Future work should thus include a known optimal allocation policy and a re-evaluation of our core allocation policy with regards to sub-optimal stage-switching points, responsivity to varying load profiles as well as different core and stage counts.

Apart from the limitations of the evaluation, there is also room for improvement in the general design. First, the evaluation setup with pinned QEMU helper threads is not representative of the current offerings of public cloud hosting providers. Apart from evaluating other virtualization technologies,

we can imagine a hypercall interface to pin QEMU helper threads to specific dedicated vCPUs from within the guest VM.

Second, integration of our solution into other languages can be simplified by providing a C-version of our stage API. A particularly interesting candidate is Java because the standard library encourages traditional thread-based concurrency [Ora], to which our solution is applicable.

Third, our scheduler could be extended to program performance-counters during runtime to measure the net-effect of thread migration, e.g., by monitoring L2 i-cache misses. Additionally, server applications could inform the scheduler about request completions. Given a mathematical model of the opportunity cost of thread migration, the scheduler could then use the feedback from above data sources as input to the core allocation algorithm.

Finally, the core allocation policy should be extended to support simultaneous multithreading (SMT): The hardware threads of an SMT-enabled CPU are commonly exposed to the operating system as separate logical CPU cores. The corresponding topology information, i.e., which of these logical cores belong to which physical cores, can be obtained through instructions such as CPUID. [SDM87a] Logical cores share the private cache of the physical core. [SDM87b] The core allocation policy would need to assert that all logical cores of a physical core are assigned to the same stage to prevent i-cache thrashing. We expect the performance-degrading effect of `MWAIT` to be mitigated by using SMT because the logical cores that are not idling can continue to perform useful work. [SDM87b]

Appendix

This thesis was submitted with a digital copy of our implementation, benchmarking tools and supplemental utility scripts. Furthermore, we make this material available via GitHub: All repositories referenced below are published under the `github.com/problame` account. To facilitate orientation in the codebase, we provide the the following overview.

- **OSv versions**: Available as branches in the `ba-osv` repository.
  - **The implementation our our solution:** The version that we used for our evaluation is `b18d6af8` which is HEAD of the `stage`.
  - **Upstream OSv** For using upstream OSv as a baseline the evaluation, we made several changes.
    * The baseline with page access scanner frequency 10Hz and configurable idle strategy is `7b853491` which is HEAD of the `eval_upstream` branch.
    * The additional implementation of the stage API on top of the upstream primitives is `bf6ed9ac` which is HEAD of the `eval_upstream_threadmig` branch.
- **Micro-Benchmark and MySQL OSv ports.** They are used by all of the above versions and are thus located in the `ba-osv-apps` (HEAD of `master` branch). The OSv versions use submodules to include this repository.
- **The tool we wrote to coordinate the benchmark execution.** We call it `doltp`, it is responsible for starting the VM, starting perf, starting `oltpbenchmark` on the remote machine, etc. It is available in the `ba-doltp` repository and has all library dependencies in the `vendor` directory.

- **The LaTeX sources for this thesis** are available in the `ba-thesis` repository.
- **The scripts to perform the benchmark runs, the raw result data and the IPython notebook used to render the evaluation figures.** They are located in the `ba-thesis` repository, in the `thesis/evaluation` subdirectory.
- **Details on the hardware used for the evaluation.** We provide the output of some standard hardware information tools in the `ba-thesis` repository in the `thesis/evaluation/hw` subdirectory, along with the shell script to generate the output.

Bibliography

[7cp] 7-cpu.com. *7-Zip LZMA Benchmark — Haswell*. URL: http://www.7-cpu.com/cpu/Haswell.html (visited on 02/14/2018).

[AHH88] Anant Agarwal, John Hennessy, and Mark Horowitz. "Cache Performance of Operating System and Multiprogramming Workloads." In: *ACM Trans. Comput. Syst.* 6.4 (Nov. 1988), pp. 393–431. ISSN: 0734-2071. DOI: 10.1145/48012.48037. URL: http://doi.acm.org/10.1145/48012.48037.

[ALL89] T. E. Anderson, D. D. Lazowska, and H. M. Levy. "The Performance Implications of Thread Management Alternatives for Shared-memory Multiprocessors." In: *SIGMETRICS Perform. Eval. Rev.* 17.1 (Apr. 1989), pp. 49–60. ISSN: 0163-5999. DOI: 10.1145/75372.75378. URL: http://doi.acm.org/10.1145/75372.75378.

[Aut] Perf Wiki Authors. *Linux kernel profiling with perf*. URL: https://perf.wiki.kernel.org/index.php/Tutorial (visited on 03/09/2018).

[Cor16] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 248966-033. June 2016. Chap. 2.3: Intel Microarchitecture Code Name Sandy Bridge.

[Cor17] Intel Corporation. *Intel VTune Amplifier 2018 Help*. 2017. URL: https://software.intel.com/sites/default/files/managed/db/b7/intel-vtune-amplifier-help-2018.zip (visited on 03/20/2018).

[Cou] Transaction Processing Performance Council. *The TPC Benchmark C Standard Specification, Revision 5.11*. URL: http://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf (visited on 03/09/2018).

[CWS06] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. "Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly." In: *SIGARCH Comput. Ar-*

chit. News 34.5 (Oct. 2006), pp. 283–292. ISSN: 0163-5964. DOI: 10.1145/1168919.1168893. URL: http://doi.acm.org/10.1145/1168919.1168893.

[Gmb] Solid IT GmbH. *DB-Engines Ranking — popularity ranking of database management systems*. URL: https://db-engines.com/en/ranking (visited on 03/08/2018).

[Got+18] Mathias Gottschlag et al. "Towards Fully Automatic Staged Computation." Accepted to the 8th Workshop on Systems for Multi-core and Heterogeneous Architectures. 2018.

[HA03] Stavros Harizopoulos and Anastassia Ailamaki. "A case for staged database systems." In: *CIDR*. DIAS-CONF-2003-001. 2003.

[HA04] Stavros Harizopoulos and Anastassia Ailamaki. "Steps Towards Cache-resident Transaction Processing." In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB '04. Toronto, Canada: VLDB Endowment, 2004, pp. 660–671. ISBN: 0-12-088469-0. URL: http://dl.acm.org/citation.cfm?id=1316689.1316747.

[Har+11] N. Hardavellas et al. "Toward Dark Silicon in Servers." In: *IEEE Micro* 31.4 (July 2011), pp. 6–15. ISSN: 0272-1732. DOI: 10.1109/MM.2011.77.

[Har05] Stavros Harizopoulos. "Staged database systems." PhD thesis. 2005.

[HP02] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1-55860-596-7.

[IW] Unknown Interviewer and Michael Widenius. *Five Questions With Michael Widenius - Founder And Original Developer Of MySQL*. URL: https://web.archive.org/web/20090313160628/http://www.opensourcereleasefeed.com/interview/show/five-questions-with-michael-widenius-founder-and-original-developer-of-mysql (visited on 03/08/2018).

[Kan+15] Svilen Kanev et al. "Profiling a warehouse-scale computer." In: *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE. 2015, pp. 158–169.

[KCE14] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. "OSv—Optimizing the Operating System for Virtual Machines." In: *Proceedings of USENIX ATC'14: 2014 USENIX Annual Technical Conference*. 2014, pp. 61–72.

[Keg99] Daniel Kegel. *The C10K problem*. 1999. URL: http://www.kegel.com/c10k.html (visited on 02/13/2018).

[KST10] Md Kamruzzaman, Steven Swanson, and Dean M Tullsen. "Software data spreading: leveraging distributed caches to improve single thread performance." In: *ACM Sigplan Notices*. Vol. 45. 6. ACM. 2010, pp. 460–470.

[LP01] James R Larus and Michael Parkes. "Using cohort scheduling to enhance server performance." In: *ACM SIGPLAN Notices*. Vol. 36. 8. ACM. 2001, pp. 182–187.

[Mad+13] Anil Madhavapeddy et al. "Unikernels: Library operating systems for the cloud." In: *Acm Sigplan Notices*. Vol. 48. 4. ACM. 2013, pp. 461–472.

[MN04] Marshall Kirk McKusick and George V. Neville-Neil. "Thread Scheduling in FreeBSD 5.2." In: *Queue* 2.7 (Oct. 2004), pp. 58–64. ISSN: 1542-7730. DOI: 10.1145/1035594.1035622. URL: http://doi.acm.org/10.1145/1035594.1035622.

[MS98] Paul E McKenney and John D Slingwine. "Read-copy update: Using execution history to solve concurrency problems." In: *Parallel and Distributed Computing and Systems*. 1998, pp. 509–518.

[MySQL] *MySQL 5.6 Reference Manual :: 24.1.1 MySQL Threads*. URL: https://dev.mysql.com/doc/refman/5.6/en/mysql-threads.html (visited on 03/08/2018).

[NehAnand] Anand Lal Shimpi. *Cache Hierarchy Nehalem: Everything You Need to Know about Intel's New Architecture*. 2008. URL: https://www.anandtech.com/show/2594/9 (visited on 02/14/2018).

[OLTP] *OLTPBenchmark Project*. URL: https://github.com/oltpbenchmark/oltpbench (visited on 11/03/2017).

[Ora] Oracle. *The Java Tutorials: Thread Pools*. URL: https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html (visited on 03/21/2018).

[PDZ99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. "Flash: An Efficient and Portable Web Server." In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '99. Monterey, California: USENIX Association, 1999, pp. 15–15. URL: http://dl.acm.org/citation.cfm?id=1268708.1268723.

[PH05] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design, Third Edition: The Hardware/Software Interface*. The

Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2005. ISBN: 9780080550336.

[Reg10] The Board of Regents of the University of Wisconsin System. *SHORE 6.0.0 release tarball*. 2010. URL: `http://research.cs.wisc.edu/shore-mt/ftp/6.0/shore-sm-6.0.0.tar.gz` (visited on 02/09/2018).

[Reg12] The Board of Regents of the University of Wisconsin System. *SHORE Storage Manager: The Multi-Threaded Version*. 2012. URL: `http://research.cs.wisc.edu/shore-mt/` (visited on 02/09/2018).

[Sch] Christian Schwarz. *OSv issue #952: high-frequency thread pinning / sched_setaffinity causes failed assertion wrt timers*. URL: `https://github.com/cloudius-systems/osv/issues/952` (visited on 03/24/2018).

[SDM105] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol. 3B. 253669-033US. Oct. 2017. Chap. 10.5.4: APIC Timer.

[SDM810] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol. 3B. 253669-033US. Oct. 2017. Chap. 8.10: Management of Idle and Blocked Conditions.

[SDM87a] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol. 3A. 253669-033US. Oct. 2017. Chap. 8.9.2: Hierarchical Mapping of CPUID Extended Topology Leaf.

[SDM87b] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol. 3A. 253669-033US. Oct. 2017. Chap. 8.7: Intel Hyper-Threading Technology Architecture.

[She13] Benjamin H Shelton. *Popcorn Linux: enabling efficient inter-core communication in a Linux-based multikernel operating system*. 2013.

[SkyWch] The wikichip.org Users. *Skylake (server) - Microarchitectures - Intel*. URL: `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)#Memory_Hierarchy` (visited on 02/14/2018).

[Sys17] Cloudious Systems. *OSv Git Repository. Commit d52cb125*. 2017.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. "SEDA: an architecture for well-conditioned, scalable internet services." In: *ACM SIGOPS Operating Systems Review*. Vol. 35. 5. ACM. 2001, pp. 230–243.

[Yas14] A. Yasin. "A Top-Down method for performance analysis and counters architecture." In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2014, pp. 35–44. DOI: 10.1109/ISPASS.2014.6844459.