Towards Fully Automatic Staged Computation

Mathias Gottschlag
Karlsruhe Institute of Technology
os@itec.kit.edu

Christian Schwarz
Karlsruhe Institute of Technology
me@cschwarz.com

Marc Rittinghaus
Karlsruhe Institute of Technology
os@itec.kit.edu

Frank Bellosa
Karlsruhe Institute of Technology
os@itec.kit.edu

ABSTRACT

Server applications often experience many stall cycles because their working set for individual requests exceeds the size of fast private CPU caches. Existing solutions for this problem usually involve refactoring the application to split it into multiple parts with smaller working sets. Scheduling these parts on multiple cores reduces the cache miss rate and increases performance. However, such refactoring of existing applications is often too labor-intensive.

In this paper, we describe an automatic solution to partition existing server applications and to execute the parts on individual cores to improve cache locality. Our system records the memory accesses of the application running representative input data and uses the resulting memory access trace to repeatedly try out different partitioning schemes in a cache simulator. The best-performing solution is then used to generate code to automatically migrate the application between cores. Our solution is already able to improve the performance of the MySQL database by 8.6% and is able to reduce L2 cache misses by more than 50%, even though only minimal developer interaction is required.

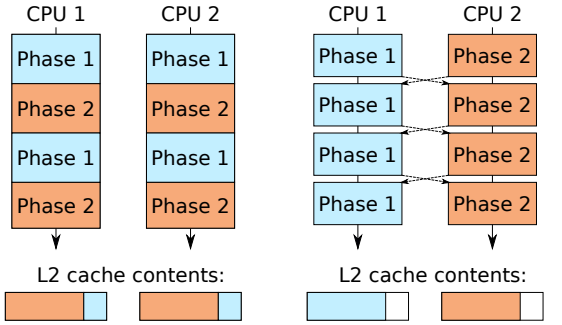
1 INTRODUCTION

Server and scale-out applications are often not able to efficiently utilize modern out-of-order CPUs. Their working set exceeds the size of fast private L1 and L2 caches, mostly due to the size of the code, so instructions and data have to be frequently fetched from slow L3 cache[7]. The resulting long-latency memory accesses cause frequent pipeline stalls.

Existing solutions for this problem include more complex hardware prefetchers to predict the memory access patterns produced by these applications[16] as well as increasing the size of the L2 cache[10]. Both approaches are able to reduce the cache miss rate of server applications. The approaches require hardware modifications, though, so deployment is expensive as existing CPUs have to be replaced.

However, the nature of server and scale-out applications allows for a different class of software-only solutions. To process incoming requests, these applications repeatedly execute similar code paths with a similar working set. Whereas the working set for each individual request does not fit into the L2 cache, the applications can be split and scheduled in a way that the system repeatedly executes the same portion of the application on the same core[14]. As a result, if each part's working set fits into the corresponding L2 cache, the L2 cache miss rate is significantly decreased.

SFMA'18, April 23, Porto, Portugal
2018.



(a) Execution on one core **(b) Migration between cores**

Figure 1: If different execution phases of applications are scheduled on different cores, the cores' private caches each hold parts of the working set. Therefore, such execution with phase-based affinity (b) has lower cache miss rates than if threads are always scheduled on the same core (a).

This technique is used by *cohort scheduling*[14], which allows the developer to describe the application as a pipeline of several processing steps. The system defers processing of requests and then executes each processing step on a group of requests, thereby achieving up to 13% higher throughput compared to a traditional server application which processes one request until completion before processing the next request. However, cohort scheduling[14] and similar approaches[8] require the developer to rewrite the application to use fine-grained parallelism models which give runtime libraries fine-grained control over scheduling. Applying these approaches to existing server applications is therefore excessively labor-intensive, as large codebases need to be rewritten.

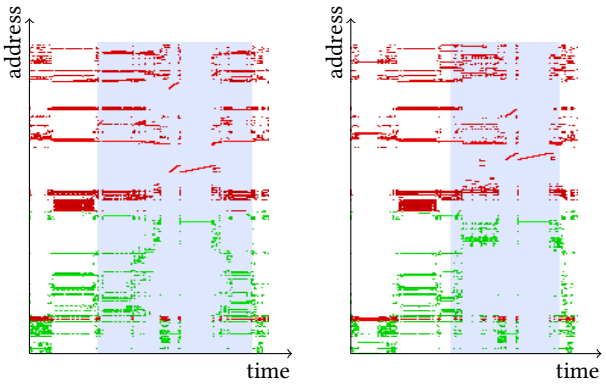
We present a system which automatically splits existing multi-threaded server applications into suitable parts so that the overall cache miss rate is significantly reduced when each part is executed on a different CPU. As shown in Figure 1, in this scenario, each core's private caches would only hold part of the application's working set, which reduces the number of cache misses in the private CPU caches.

We describe a method to analyze the memory accesses of an application to find such a split, and we present a solution which repeatedly migrates the threads of the application between cores according to the split, so that each part of the application is always executed on a suitable core. Our contributions are as follows:

- We show that the fine-grained execution phase behavior in database applications can be exploited to increase cache locality by scheduling different phases on different cores.
- We present a technique to automatically determine function calls in existing applications at which migrating the application to a different core can reduce cache misses, and we present a suitable efficient thread migration mechanism.

2 MOTIVATION

Existing database management systems show significant execution phase behaviour. To show that executing different phases on individual cores can significantly reduce cache misses, we recorded memory access traces for several database requests of the TPC-C benchmark executed on top of MySQL 5.6. To record the memory accesses, we ran the database in an instrumented version of the QEMU[3] emulator. QEMU was modified to record all memory accesses caused by instruction fetches and explicit load and store instructions, both from the application and from the operating system. As the slow emulation significantly changed the timing behavior of the code, we executed MySQL on the lightweight OSv library operating system[13] and reduced the frequency of all timer-based activity such as OSv's page access scanner or its preemption timer.



(a) Query 1 (b) Query 2

Figure 2: Plot of all recorded memory accesses for a simulation of the execution of two TPC-C database queries. Green pixels represent instruction fetches, whereas red pixels represent load and store instructions. Simulating parts with white background on one core and parts with blue background on a second core reduces L2 cache misses by 32%.

Figure 2 shows a plot of the memory accesses of two database queries from the TPC-C benchmark. The first query has a working set of 399 KiB, of which 225 KiB are code, whereas the working set of the second query is 382 KiB, of which 214 KiB are code. On modern CPUs with 256KiB of L2 cache[1], these working sets do not fit into the L2 cache, which significantly limits reuse of cache contents among different queries executed on the same core. However, our simulation shows that 228 KiB of memory is accessed by both queries, so such reuse could be beneficial.

Both working sets would fit into L2 cache if twice the amount of cache was available. To find out whether it is possible to utilize the combined capacity of the caches of two cores, we simulate the memory accesses of both database queries shown in Figure 2 in a cache simulator. If all memory accesses are simulated on the same core, the second query causes 6027 cache misses. We then repeat the simulation 160000 times, each time simulating a different part of each request on a second core instead. If the accesses marked in blue are executed on the second simulated core, the total cache misses of the second query are reduced to 4069 (32% less).

As the combined cache size is larger than the working set for each individual request and as more than half of the second request's working set has already been accessed by the first request, one could have expected the number of cache misses to be even lower. However, the cache miss count depends on several factors. First, the two cores access a significant amount of shared data, so the effective combined cache capacity is reduced. Second, whenever one core modifies a shared cache line, the line is evicted from other cores' L2 cache, thereby causing a cache miss on the next access. Therefore, we expect that distributing the working set among multiple caches only results in a net benefit if the overlap between the working sets of different cores is sufficiently low. Such is the case in particular if the application has different phases, where each phase performs a different set of tasks and therefore accesses different memory locations, but where multiple executions of the same phase have similar working sets. Figure 2 shows that this condition is true for database management systems, as the memory trace shows that the access pattern of the application changes over the time of a single network request.

Migrating threads between cores at execution phase boundaries also reduces the TLB miss rate, as less pages are accessed per core, and the branch misprediction rate, as each core only executes a subset of all branches.

3 DESIGN

The cache miss rate of existing server applications can be significantly reduced if each core only holds parts of the application's working set. Therefore, we expect a solution to perform well which dedicates cores to application execution phases, migrating the threads to a different core whenever the beginning of a different application phase is reached. However, automatic solutions such as the one we describe have limited information about the application's structure and its execution phases. Therefore, we employ the two-phase approach shown in Figure 3.

During the first phase, as described in Section 3.1, our solution determines an approximation of the execution phases. As we expect that execution phases are usually triggered by function calls in the program, we determine which functions are called periodically. We also record all memory accesses while the application processes a number of requests. In an off-line step, we then use the recorded memory accesses to simulate the effect of migrating between cores at different subsets of these function calls. We use the subset which yields the lowest amount of cache misses as an approximation of the application's execution phases. Our solution then automatically instruments the application binary so that the selected function calls trigger the appropriate migration to a different core.

In the second phase, this instrumented application is executed. Whenever execution hits the selected function calls, the thread

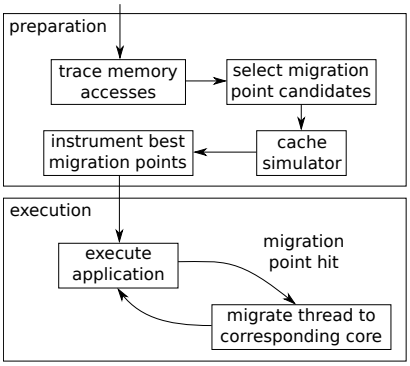


Figure 3: The two phases of our approach: First, memory traces are used to split the application into multiple parts. Second, during normal execution, the application migrates between cores at the boundary between different parts.

migrates to the corresponding core which already holds parts of the working set of the following code if that code has been executed before. Our thread migration mechanism is described in Section 3.2.

3.1 Estimating Execution Phases

Our solution treats most of the server application as a black box and does not have information about execution phases. Therefore, we analyze memory access traces to find good points in the application at which to migrate to a different core.

This memory access tracing step is implemented as described in Section 2: The application is placed in a virtual machine and is executed in the QEMU[3] emulator which was modified to record all the memory accesses triggered by load and store instructions in the virtual machine. Additionally, the emulator records a memory access whenever a basic block – a straight-line sequence of instructions which is always executed from the beginning and does not contain any branch instructions except at the end of the block – is executed. Recording executed code serves both to record instruction fetch accesses and to enable our analysis tool to recognize function calls in the resulting trace by matching the addresses with the function symbols of the application. For each of these accesses, the virtual target address, the size of the accessed data and the current value of the program counter are recorded. SimuTrace[17] is used for efficient storage of these potentially large memory access traces. Note that the emulator causes enough overhead to significantly change the timing behavior of the application. Therefore, as described in Section 2, we use OSv[13] in the virtual machine to reduce the amount of concurrent background work, and we reduce the frequency of timer-based background activity.

The application is modified by the developer to insert a marker into the trace whenever a network request is received, and memory accesses are recorded until a sufficient number of requests has been processed by the server application. To reduce the amount of data while still simulating representative behavior, we then filter the result to keep only the accesses caused by 10 randomly selected network requests. To find good points for thread migration, we repeatedly replay these accesses in a simple cache simulator, each time migrating at different function calls. If, for example, we want to split the working set of the application into 3 parts (suitable for execution on at least 3 cores), we select a set of 3 different periodically called functions from the application and, whenever the first instruction of one of these functions is executed, switch to the corresponding core and continue to simulate our memory accesses on that core. The cache simulator simulates multiple cores with a three-level inclusive cache hierarchy which uses the LRU replacement policy at all levels. As the large number of simulations makes simulation performance critical, we use a simple hand-written performance-optimized simulator. After every simulation, we count the cache misses. The set of migration points which produced the lowest number of cache misses is then used to instrument the application. At the beginning of each of the functions, the software automatically inserts a call to a runtime library into the existing application binary to migrate the thread to the corresponding core.

As this method is impractical for applications with a large number of functions, we have to limit the amount of functions which are considered as candidates for migration points. Therefore, we only consider those functions which are called at most once per network request, as functions which are called with high frequency are unlikely to signal major execution phase changes and would only cause excessive thread migration. Note, though, that even leaf functions can cause cache-thrashing working set changes, so we do not limit our search to functions near the root of the call tree. Also, we only consider functions which are called for at least 80% of all network requests. These restrictions cause the number of candidates for migration points to drop to approximately 100 functions for MySQL, which reduces computational complexity to acceptable levels for the scenario shown in our evaluation. In the future, we plan to explore random sampling to further reduce the number of cache simulations.

3.2 Thread Migration

The resulting instrumented program causes threads to migrate between different cores several times during the execution of every network request. Most operating systems provide system calls for such thread migration. For example, the `sched_setaffinity` function provides a mechanism on Linux to select the cores on which a thread is allowed to run. However, we have found that function to be too slow for our use case (9 μs - 14 μs[12]).

Instead, we have developed a fast thread migration mechanism in the OSv library operating system. To perform a thread migration, threads insert themselves into the migration queue of the target CPU and let the current CPU switch to a different thread. The target CPU's scheduler then removes the thread from the queue and resumes execution. Whenever a CPU becomes idle, it executes the `mwait` instruction[2] on the address of the migration queue to wait for incoming threads. This method does not require any costly inter-processor interrupts or context switches and, as OSv executes all code with kernel privileges, does not require any privilege changes.

Although our prototype is limited to OSv, most mechanisms could likely be applied to other operating systems such as Linux as these systems likely only lack optimization and do not have any relevant fundamental differences. For example, Linux could introduce similar per-cpu queues for incoming migrated threads and, if one thread does not fully utilize its time slice and the cpu's queue

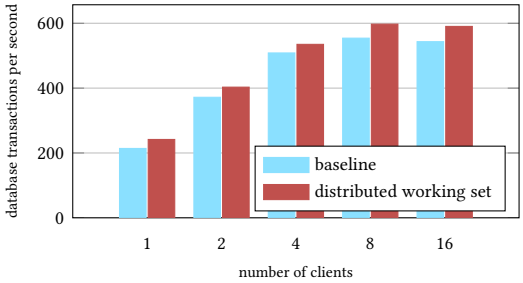


Figure 4: Performance of the TPC-C benchmark for various numbers of database clients

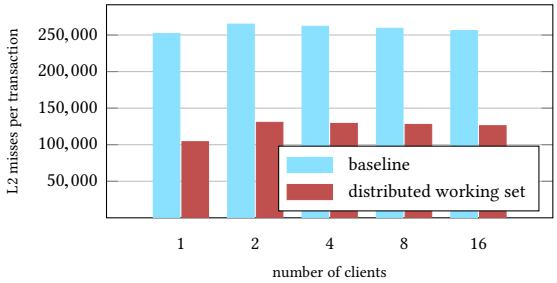


Figure 5: Cache miss counts per database transaction

holds another thread from the same process, could directly execute that thread with no scheduler invocations or context switches.

4 EVALUATION

We evaluated our approach with the TPC-C benchmark executed against the MySQL database management system, with the intent to show whether our approach results in improved application performance and whether such performance improvement can be attributed to the cache miss reduction caused by distributing the working set among multiple cores.

We conducted all experiments on a system with an Intel Xeon E5-2620 v4 CPU, which contains 8 cores with 256 KiB private L2 cache each. Both benchmark client and benchmarked application were executed on the same system. The virtual machine with OSv and the benchmarked MySQL instance was executed on 4 cores, and the OLTPBench[6] TPC-C benchmark client was executed on the other 4 cores. To reduce noise during the measurements, we significantly slowed down all periodic background tasks of OSv such as the page cache access scanner. For the experiments, all L2 prefetchers were disabled on the virtual machine's cores, as they are ineffective for many server workloads[7] and we observed them to cause significant performance degradation for unmodified MySQL.

4.1 Performance

To measure the performance improvement caused by our solution, we executed the TPC-C benchmark with varying numbers of database connections and measured the throughput of the system. The results of these measurements are shown in Figure 4. The line marked with *baseline* shows the results for an unmodified instance of the MySQL instance being executed on an unmodified version of the OSv operating system, and the line marked with *distributed working set* shows the results for a MySQL instance which was instrumented with calls to migrate the threads between cores. As our prototype of the analysis tool described in Section 3.1 does not yet support automatic instrumentation of existing binaries, we manually inserted the appropriate function calls exactly as suggested by the tool. The modified MySQL instance is executed on our modified version of OSv with support for fast thread migration.

The results show throughput to be improved by 8.6% on average, varying between 13.0% (1 database connection) and 5.2% (4 database connections). However, these results alone do not explain whether the performance improvement results from the increased cache locality or from other changes in the OS scheduler. For example, as all application threads use our migration mechanism and are therefore pinned to cores, we removed the load balancing mechanism of OSv's original scheduler. To determine the cause for the performance differences, we executed the benchmark with our modified version of OSv, but disabled thread migration in MySQL, so that all code is executed on a single core. With a single database connection[1], this system performed 4.5% better than unmodified OSv, but 7.5% worse than execution with thread migration. This result shows that, whereas some of the performance improvement has to be attributed to other changes to the OSv scheduler, most of the performance difference is actually caused by migrating the database application threads between cores.

4.2 Cache Misses

To demonstrate the effect of the thread migrations on cache locality, we repeated the experiments from the last section and used the CPU's performance monitoring counters to measure the number of L2 cache misses per executed database transaction. Figure 5 shows the results of this experiment. On average, our solution reduced the cache misses per transaction by 52.3%. Again, we repeated the experiment with disabled thread migration, which showed that simplifications in the scheduler in the case of a single database connection are responsible for a cache miss reduction of 6.7%. The vast majority of the cache miss reduction is caused by thread migration between cores, though, which indicates that the performance improvement is caused by improved cache locality.

4.3 Discussion

Our prototype is able to significantly increase the performance of MySQL, with only minimal manual modifications to the application. However, as the goal of our prototype was only to demonstrate the viability of automatic staged computation, the prototype has a number of limitations. We list these limitations in the following and discuss their impact on our evaluation results.

No automatic instrumentation. Due to a lack of time, our prototype does not yet support automatic insertion of migration code into the application. Instead, we manually inserted migration function calls at the locations in the code according to the output of the analysis tool described in Section 3.1. As our analysis, however,

[1] As restricting the application to one core prevents scaling to multiple network connection, we only give the results for a single connection.

currently only reports locations at the beginning of functions, automatic insertion of code is easy to implement, and we plan to extend our prototype accordingly. We also plan to reevaluate whether limiting potential locations for thread switching to the beginning of functions has significant impact on the quality of the results.

Overhead of memory access tracing. Currently, memory access traces are generated by executing the application in an instrumented full-system simulator. The resulting overhead is significant, and we have observed more than 50x slowdown due to emulation for single-threaded execution. In some cases, this slowdown can significantly change the behavior of the application. We plan to evaluate use of a deterministic trace and replay solution such as SimuBoost[18] to improve memory access recording performance. Such a technique is able to generate accurate memory access traces with negligible timing artifacts and provides enough performance to be used to temporarily trace live production workloads.

5 RELATED WORK

Several techniques have been developed to improve the cache locality of applications whose working set does not fit into the processor cache. Those techniques split the application into parts and change the scheduling of these parts, similarly to our approach. However, the techniques use different methods to split of the application and use different scheduling techniques.

Staged computation. Larus and Parkes[14] describe a programming paradigm where the application developer designs the application as a set of stages, each containing a collection of operations and the corresponding data. Multiple stages can be chained together by passing a continuation from one stage to the next. Through that mechanism, a server application can, for example, be implemented as a pipeline of multiple stages, where each stage performs some of the work required to process a request. The explicit separation into stages allows the underlying runtime to employ *cohort scheduling*[14] to delay or prioritize invocations of stages so that the same stage is consecutively invoked multiple times on the same core, thereby increasing cache locality. As a result, staged computation results in reduced cache, TLB, and branch predictor miss rates.

Most approaches to staged computation, however, require existing server applications to be refactored to make use of the pattern[8, 14], which requires a significant engineering effort for complex server applications. In contrast, and similarly to our approach, STEPS[9] uses memory access traces to automatically inserts context switches for cohort scheduling. However, STEPS requires manual instrumentation of the application to inform the runtime about the type of executed database operation at fine scale. We show that significant performance improvements are possible even without any manual analysis of the application.

In addition, the context switch locations produced by STEPS are limited to cohort scheduling on a single core and are less suited to be used in any approach similar to the one described in this paper which schedules different stages to individual cores. STEPS does not align context switches to execution phase boundaries, which increases the number of L2 cache misses (as described in Section 2).

O^2 Scheduling. Whereas staged computation enforces a fixed mapping of code sections to application stages, O^2 scheduling[4] mainly places objects on different cores so that the operations on these objects are executed on the core which has the required data in its private cache. Thereby, large data structures are distributed among multiple cores.

We argue that O^2 scheduling is likely not well suited for server and cloud applications, as these suffer from their large code working set[7] and therefore especially profit from approaches which try to distribute code among multiple cores. O^2 scheduling, however, executes operations wherever the data is, potentially causing the same code to be loaded into multiple cores' private caches.

Region Scheduling. Lee and Schwan[15] provide a method which tracks the working set of applications on a per-page basis and uses the information to assign each pages to a specific core. The pages are unmapped from all other cores, and the application is migrated to the appropriate core whenever a page fault occurs. In contrast, we analyze the working set of the application in a separate offline step and instrument the application accordingly to reduce the runtime overhead of our approach.

Hardware-Assisted Approaches. Chakraborty et al.[5] migrate applications between cores at finer timescales by executing user code on one core and operating system code on a second core. They implement thread migration in hardware in order to keep overhead low. Kallurkar and Sarangi[11], in contrast, implement a more flexible method to spread both application and operating system code to multiple cores, but require hardware support to determine changes to the working set. We show that similar results can be achieved on existing hardware with a software-only solution.

6 CONCLUSION

Server applications frequently suffer from very high numbers of CPU stall cycles because the working set does not fit into fast private L1 and L2 caches. In such situations, partitioning the code of the applications and distributing it among multiple cores can significantly improve the cache miss rate by loading parts of the working set into separate L2 caches. Existing techniques to do so, however, either require the developer to refactor the application or require custom hardware. In this paper, we demonstrate an automatic approach to split server applications into parts and distribute those parts across multiple cores to improve cache locality.

Our solution first generates a memory access trace of the server application for several network requests and uses that trace to simulate the effects of different splits of the application to multiple cores on the number of cache misses. The best split is then used to instrument the application with function calls to trigger migration between cores at the appropriate places in the program. We have evaluated our solution with MySQL and the TPC-C benchmark. In this scenario, the solution was able to reduce cache misses by up to 52.3%, which resulted in a 8.6% performance improvement.

In the future, we will evaluate our approach with more server and cloud applications. In addition, we will evaluate techniques such as deterministic trace and replay to reduce the impact of memory access tracing via full-system simulation on the application's performance.

REFERENCES

[1] *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Intel Corporation, June 2016.

[2] *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4.* Intel Corporation, December 2017.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[4] Silas Boyd-Wickizer, Robert Morris, M Frans Kaashoek, et al. Reinventing scheduling for multicore systems. In *12th Workshop on Hot Topics in Operating Systems (HotOS XII)*, 2009.

[5] Koushik Chakraborty, Philip M Wells, and Gurindar S Sohi. Computation spreading: employing hardware migration to specialize cmp cores on-the-fly. *ACM SIGPLAN Notices*, 41(11):283–292, 2006.

[6] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[7] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices*, volume 47, pages 37–48. ACM, 2012.

[8] Stavros Harizopoulos and Anastassia Ailamaki. *Affinity scheduling in staged server architectures.* School of Computer Science, Carnegie Mellon University, 2002.

[9] 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*, pages 660–671. VLDB Endowment, 2004.

[10] Aamer Jaleel, Joseph Nuzman, Adrian Moga, Simon C Steely, and Joel Emer. High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 343–353. IEEE, 2015.

[11] Prathmesh Kallurkar and Smruti R Sarangi. Schedtask: a hardware-assisted task scheduler. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 612–624. ACM, 2017.

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

[13] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. OS v—optimizing the operating system for virtual machines. In *Proceedings of USENIX ATC'14: 2014 USENIX Annual Technical Conference*, page 61, 2014.

[14] James R Larus and Michael Parkes. Using cohort scheduling to enhance server performance. In *ACM SIGPLAN Notices*, volume 36, pages 182–187. ACM, 2001.

[15] Min Lee and Karsten Schwan. Region scheduling: efficiently using the cache architectures via page-level affinity. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 451–462. ACM, 2012.

[16] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)*, 49(2):35, 2016.

[17] Marc Rittinghaus, Thorsten Groeninger, and Frank Bellosa. Simutrace: A toolkit for full system memory tracing. White paper, Karlsruhe Institute of Technology (KIT), Operating Systems Group, May 2015.

[18] Marc Rittinghaus, Konrad Miller, Marius Hillenbrand, and Frank Bellosa. Simuboost: Scalable parallelization of functional system simulation. *Simulation*, 1:1, 2013.