SchedTask: A Hardware-Assisted Task Scheduler

Prathmesh Kallurkar*
Microarchitecture Research Lab
Intel Corporation
prathmesh.kallurkar@intel.com

Smruti R. Sarangi
Department of Computer Science
Indian Institute of Technology Delhi
srsarangi@cse.iitd.ac.in

ABSTRACT

The execution of workloads such as web servers and database servers typically switches back and forth between different tasks such as user applications, system call handlers, and interrupt handlers. The combined size of the instruction footprints of such tasks typically exceeds that of the i-cache (16-32 KB). This causes a lot of i-cache misses and thereby reduces the application's performance. Hence, we propose SchedTask, a hardware-assisted task scheduler that improves the performance of such workloads by executing tasks with similar instruction footprints on the same core. We start by decomposing the combined execution of the OS and the applications into sequences of instructions called *SuperFunctions*. We propose a scheme to determine the amount of overlap between the instruction footprints of different *SuperFunctions* by using Bloom filters. We then use a hierarchical scheduler to execute *SuperFunctions* with similar instruction footprints on the same core. For a suite of 8 popular OS-intensive workloads, we report an increase in the application's performance of up to 29 percentage points (mean: 11.4 percentage points) over state of the art scheduling techniques.

CCS CONCEPTS

• **Software and its engineering** → **Scheduling**; Virtual memory; • **Computer systems organization** → *Multicore architectures*; *Cloud computing*;

KEYWORDS

scheduling, architectural support for operating system, cache pollution

ACM Reference format:
Prathmesh Kallurkar and Smruti R. Sarangi. 2017. SchedTask: A Hardware-Assisted Task Scheduler. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3123984

1 INTRODUCTION

The execution of OS-intensive applications such as web servers and database servers typically switches between different tasks such as application code, system call handlers, and interrupt handlers. Previous works [7, 12, 17, 32, 36] have shown that the combined size of the instruction footprints of these tasks typically exceeds that of the i-cache (16-32 KB). Since traditional OS schedulers typically execute these tasks on the same core, they evict each other's i-cache lines, and thereby reduce the overall performance of OS-intensive applications by up to 50% [12, 36].

Several papers [7, 16, 30, 32, 36] have proposed to tackle this problem through *core specialization*. Under this scheme, tasks with dissimilar instruction footprints are executed on different cores. *FlexSC* [36] and *Disaggregated OS Services* [30] execute user applications and system call handlers on separate cores. However, these techniques are agnostic to asynchronous events such as interrupts. Hence they do not perform well for IO intensive applications. *SLICC* [7] is another state of the art core specialization technique. It spreads the i-cache footprint of an application across different cores and uses special hardware to migrate a thread to a core that may contain the i-cache line, which it may access next. However, this technique does not allow an idle core to steal pending threads waiting at other cores. Hence, it suffers from high core-idleness when there is a significant imbalance of work across cores.

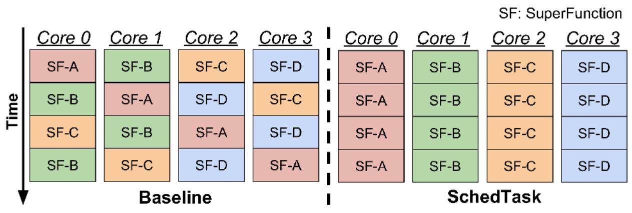


Figure 1: An overview of the *SchedTask* technique

This paper proposes *SchedTask*, a hardware-assisted fine-grained scheduler for OS-intensive applications. We start by decomposing the combined execution of the OS and the applications into sequences of instructions called SuperFunctions. We then propose a scheme that determines the amount of overlapping sequences of instructions between different *SuperFunctions* using a hardware based Bloom filter. Those *SuperFunctions* that are deemed similar are scheduled on the same core, resulting in reduced i-cache pollution.

We first characterize the execution of 8 popular OS-intensive applications and show that there is a high correlation between the type and the number of *SuperFunctions* that are executed in consecutive epochs (fixed intervals of time). Therefore, by profiling (at run-time) the *SuperFunctions* that are executed in an epoch, we can make better decisions regarding how to schedule *SuperFunctions* in the next epoch.

A core might become idle if all *SuperFunctions* that are currently being executed are scheduled to run on other cores. We show that it is possible to improve the performance by scheduling a *SuperFunction* that is already assigned to one of the other cores to run on the idle

*The author contributed to this work while he was a student at IIT Delhi.

core. This work stealing approach works as long as the selected *SuperFunction* executes instructions that are already in the i-cache of the idle core.

Specifically, this paper makes the following contributions:

(1) A fine-grained decomposition of a thread's execution into *SuperFunctions*.
(2) A technique to quantify the overlap between *SuperFunctions* at run-time and then appropriately schedule them using a two-pass technique.
(3) A novel work-stealing algorithm to increase the instruction execution throughput by scheduling a suitable *SuperFunction* on an idle core, thereby improving the performance.

We evaluate our approach for a suite of 8 popular OS-intensive applications. For these applications, we show that the performance benefit of *SchedTask* is better than that of the state of the art scheduling technique *SLICC* by up to 29 percentage points (mean: 11.4 percentage points).

We discuss the related work in Section 2 and the details of *SuperFunctions* in Section 3. We then discuss the benchmarks and their characterization in Section 4 and the details of our implementation in Section 5. Finally, we discuss the main results in Section 6. We discuss additional results pertaining to the compared techniques in the appendix [5].

2 RELATED WORK

2.1 Core Specialization

Futuristic operating systems such as Corey [14], Factored OS [39], and Barrelfish [9] follow the principle of core specialization. They model the operating system as a server, which runs on a selected set of cores. Applications use remote procedure calls (RPC) to submit system call requests. However, these operating systems still do not support the popular OS-intensive applications such as *Apache* web server or the *MySQL* database server. We find that the functionality of these operating systems is still very restrictive and it will take time for them to reach the maturity of a traditional operating system such as Linux.

Several papers [7, 8, 16, 22, 30–32, 36] have recently proposed core specialization solutions for traditional operating systems. *Selective offloading* [32] uses twice the number of cores as a normal system; half the cores are reserved to execute application code and the rest half are reserved to execute OS code. Threads execute the application code on application cores and are transferred to an OS core if they execute a system call instruction. The primary drawback of this technique is that it lacks a load balancing algorithm. Even if an application core is idle, it cannot execute applications that are waiting to execute on other application cores. Additionally, they do not specialize OS cores for specific OS tasks. Hence, we observe high i-cache pollution in the OS cores. In Section 6, we show that even while consuming half of the area (for the cores) of the *Selective offloading* technique [32], *SchedTask* outperforms their technique by around 12.5%.

FlexSC [36] executes user applications and system call handlers on separate cores. It executes application threads on top of a special user-level scheduler. The scheduler takes a system call request from the application thread and offloads it to special OS threads that execute on different cores. It then executes another *runnable* thread belonging to the same application. When a single-threaded application calls the user-level scheduler, it offloads the system call's execution to OS threads and then yields execution to the Linux scheduler. As we show in Section 6.1, executing the Linux scheduler for every system call can lead to a slowdown of up to 63%.

Disaggregated OS Services [30] improves upon the *FlexSC* technique in multiple ways. It divides the user applications, and groups the system call handlers into multiple regions; each region is executed on different cores. It then uses a scheduler to migrate a thread from one core to another based on the data region that it is accessing. While the authors propose a runtime region detection algorithm for application code, the regions accessed by the OS code are identified by the OS programmer. For example, all filesystem related system calls are treated as accessing the same data region. Like *FlexSC* this technique also ignores i-cache pollution due to OS tasks such as the scheduler or interrupt top-half and bottom-half [40] handlers.

SLICC [7] is a hardware technique that reduces the i-cache misses of OLTP workloads. *SLICC* spreads the i-cache footprint of an application across multiple cores, and uses a hardware unit to migrate threads between these cores. The hardware migration algorithm of *SLICC* is agnostic to OS events such as system call handlers. Hence, while the technique is able to group common portions of application+OS execution across threads of the same application, it fails to take advantage of common OS execution across different applications. Consequently, the performance of *SLICC* suffers when multiple OS-intensive applications are executing at the same time. *STREX* [8], a recently proposed technique also uses a hardware module to reduce i-cache misses of OLTP workloads. *STREX* time-multiplexes the execution of similar transactions on a single core such that the instructions fetched by one transaction are reused by subsequently executed transactions. However, as mentioned in the original paper, *SLICC* outperforms *STREX* for a 32-core system. We thus compare our technique against the *SLICC* technique (omit *STREX*) in Section 6.

2.2 Architectural Support

2.2.1 *Additional Caches.*

Nellans et al. [32], Chandran et al. [17], and Bhalla et al. [12] propose to reduce the i-cache pollution in OS-intensive applications by storing the lines belonging to the application and the OS in separate caches instead of the same cache. Bhalla et al. [12] have additionally considered a third cache to store hypervisor lines. The main drawback of this line of work is the 100% area overhead of the additional caches, along with the complexity of the logic to locate, and migrate lines between the caches.

2.2.2 *Instruction Prefetching.*

Instruction prefetching is an alternate approach to improve the performance of OS-intensive applications. PIF [19] and RDIP [27] are the most advanced instruction prefetchers that are implemented completely in hardware. However, these schemes require additional hardware structures of 64-200 KB per core, which makes it difficult to deploy such schemes in real hardware. Other notable instruction prefetchers such as pTask [26] and CGP [6] require recompilation of the user applications, which makes such schemes unsuitable for systems that rely on third party binaries.

Choice of instruction footprint as a scheduling parameter: For OS-intensive applications, giving attention to i-cache misses as opposed to d-cache misses is a standard design decision taken in almost all the related work [6–8, 12, 16, 19, 26, 27, 36]. This is because OS-intensive applications have low i-cache hit rates (80-90%). Additionally, optimizations in modern processors (OOO pipelines, load store queues, data prefetchers) already hide the latencies of d-cache misses. It can happen that collaterally d-cache misses reduce (as in our case); however, this is not any design's primary objective.

3 SUPERFUNCTION

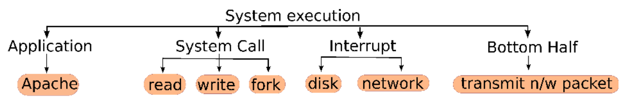


Figure 2: Decomposition of the system execution (Apache)

As shown in Figure 2, let us decompose the code running on a system into four categories: (1) applications, (2) system call handlers, (3) interrupt handlers, and (4) bottom-half handlers. For example, an *Apache* executable is an application. *read* and *write* are examples of system call handlers. This decomposition helps us in finding pieces of code that have a predictable pattern of execution. We shall use this high level notion to define a *SuperFunction*.

In formal terms, a *SuperFunction* is defined as an ordered list of triplets <`pc,t,c`>, where `pc` is the program counter of the instruction that was executed at time `t` on core `c`. It captures a sequence of retired instructions. In this paper, we define four types of *SuperFunctions* based on the type of the task (as shown in Figure 2). They are: (1) application, (2) system call handler, (3) interrupt handler, and (4) bottom half handler. A *SuperFunction* begins and ends on specific OS events. They are as follows: (1) start of a user process, (2) system call instruction, (3) hardware interrupt, and (4) invocation of a bottom half handler's routine. When a *SuperFunction* terminates, a new *SuperFunction* begins. Note that by our definition, if two instances of the *read* system call are executing concurrently, then each of them represents a different *SuperFunction*. Let us elaborate further.

An application *SuperFunction* is the entire user-mode execution of a process. It is created by the fork system call handler and continues till the process completes execution. In contrast, the OS *SuperFunctions* are merely event handlers that are executed in response to OS-specific events. These trigger events are: system call instructions for system call handlers, interrupts for interrupt handlers, and function calls to bottom-half handler routines for bottom half handlers. When a core receives a hardware interrupt, it pauses the currently executing *SuperFunction* and starts the interrupt *SuperFunction*. On completing the interrupt handler, the previously paused *SuperFunction* resumes execution.

Let us discuss the insights regarding why we define *SuperFunctions* this way. Consider two threads of the *Apache* program running separately. Both will execute the *read* system call, and their instruction and data footprints will be roughly similar. If the execution of these code sequences are scheduled on the same core, we can take advantage of locality effects. Moreover, it is possible that the execution of the *read* system call of *Apache* might actually be not that different from the execution of the *read* system call of *MySQL*. We would benefit by locality in this case as well. Stretching the argument further, in a heterogeneous ensemble of tasks, we wish to find all the similar sequences of execution, such that we can co-locate these execution segments, and take the fullest advantage of locality. For this purpose, we need to break the execution of a typical system intensive workload by inserting artificial boundaries (create *SuperFunctions*), and then try to find similarities. We shall discuss mechanisms to identify similarities across *SuperFunctions* in Section 3.1 and Section 3.2. Our scheduler uses these mechanisms to schedule the execution of similar *SuperFunctions* on the same core; we shall discuss it in Section 5.

3.1 Type of Task

SuperFunction	Category ID (2 bits)	Sub-category ID (62 bits)
System call handler	0	System call ID
Interrupt handler	1	Interrupt ID
Bottom half handler	2	Program counter of the bottom half handler's function
User application	3	Checksum of the code pages

Table 1: Category and subcategory of *SuperFunctions*

We encode the type of the task that a *SuperFunction* is performing in a 64-bit number called the *superFuncType*. Since *SuperFunctions* performing the same type of task typically have similar instruction footprints, *SchedTask* executes *SuperFunctions* with the same *superFuncType* on the same core.

superFuncType represents a task's category and subcategory. Table 1 shows the value that we assign to each category and subcategory of tasks. In a 64-bit *superFuncType*, first 2 bits represent the task's category, and the remaining 62 bits represent its subcategory.

An **OS SuperFunction** is executed in response to an OS event; we use the event's property to encode its *superFuncType*. Hence, the *superFuncType* of a *read* system call handler[1] will be *3* irrespective of the application that called it. Similarly the *superFuncType* of a *keyboard* generated interrupt[2] will be *0X4000000000000001* irrespective of the application that is consuming the keyboard's input.

Application SuperFunctions: We define an application's *superFuncType* as a hash of all code pages that it accesses at runtime. At the beginning of an application's execution, we set its *superFuncType* to 0 and disable the execute permission for all its code pages. When the OS receives a security exception for a valid code page of the application, it computes a hash of the code page contents and adds it to the application's *superFuncType*. Then the OS enables the execute permission of the code page and resumes the application's execution. All threads belonging to the same application have the same *superFuncType*. Since the hash computation is performed only once for each code page, the execution overhead of creating an application's *superFuncType* is miniscule ($< 0.0001\%$).

3.2 Similarity between Different Types of Tasks

Consider a scenario where three *SuperFunctions*: `read`, `pread`, and `fork` system call handlers are simultaneously created by different

[1]system call ID 3 for Linux 2.6
[2]interrupt ID 1 for Linux 2.6

threads. If *SchedTask* is forced to execute two of them on the same core, it will choose to execute `pread` and `read` on the same core because they mostly execute the same set of instructions. This decision will improve the instruction locality and thereby improve the application's performance. Notice that each of these *SuperFunctions* has a different *superFuncType*. Hence, we need a mechanism to identify the similarity between different *superFuncTypes*.

SchedTask quantifies the similarity between two *superFuncTypes* as the number of common physical pages (containing instructions) accessed. Since two applications sharing the same executable (e.g.: two instances of `scp` applications) or the same library (e.g.: `libc.so`) can use different virtual addresses to access the same i-cache lines, the overlap between different *superFuncTypes* must be detected in terms of physical page frames and not virtual pages. A summary of all the physical page frame numbers (PFN) of all instructions executed for a particular *superFuncType* is stored in a 512-bit vector called the *Page-heatmap*. This contains a hash of all the PFNs accessed, and this hash is produced by a Bloom filter [13].

We want to capture the PFN of all the instructions that *SuperFunctions* belonging to a particular *superFuncType* have accessed in the last time-epoch. Hence we do this: At the start of an epoch, the *Page-heatmap* associated with each *superFuncType* is set to all zeros. Before executing a *SuperFunction*, *SchedTask* loads its *superFuncType*'s *Page-heatmap* in a special register called the *Page-heatmap register*. When an instruction with PFN pf is committed in the pipeline, we set the $(hash(pf)\ mod\ 512)^{th}$ bit of the *Page-heatmap register* to $true$. We define: $hash(pf) = (pf) + (pf \gg 9) + (pf \gg 18) + (pf \gg 27) + (pf \gg 36) + (pf \gg 45)$. Note that we need only 9 bits to index the 512-bit *Page-heatmap register* and the PFN is 52 bits[3] long. Hence, we perform five right-shift operations (9,18,27,36,45) on the PFN to consider all of its 52 bits in the hash function of the Bloom filter.

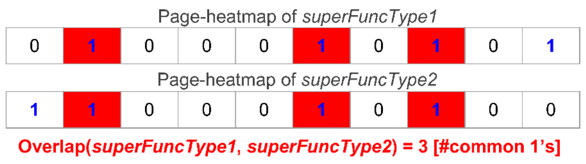

Page-heatmap of superFuncType1
0 1 0 0 0 1 0 1 0 1
Page-heatmap of superFuncType2
1 1 0 0 0 1 0 1 0 0
Overlap(superFuncType1, superFuncType2) = 3 [#common 1's]

Figure 3: Quantifying the similarity between two superFuncTypes

At the end of a time-epoch, we calculate the similarity between two *superFuncTypes* as the Hamming weight (number of 1s) of the bit-vector representing the bitwise-and of their respective *Page-heatmaps* (see Figure 3 for an example scenario). The extra **hardware** required to calculate this similarity is: (1) 512-bit *Page-heatmap register*, (2) hardware to implement the hash function, and (3) assembly instructions to load and store the *Page-heatmap register*. The *Page-heatmap* values of different *superFuncTypes* are maintained in the kernel's address space for security reasons. We calculate the similarity between *Page-heatmaps* vectors by breaking a single 512-bit bitwise-and operation into sixteen 32-bit operations (supported by existing hardware).

Keeping storage requirements and performance in mind, we chose to compute the similarity in instruction footprints at the granularity of pages. Furthermore, we limit the number of bits in our Bloom filter to 512. We shall discuss the consequences of such choices in Section 6.5. Note that in an alternate rendition of this idea, it is possible to give the capability to user applications and/or the kernel to modify the *Page-heatmap* register in software. However, since the OS can change the page mappings of an application at runtime, the software approach must map each instruction's virtual address to its PFN at runtime; the overhead of executing extra mapping instructions by accessing the TLB/page tables will cause a significant slowdown in the application's execution. Recent works [3, 33, 38] have shown that exposing the virtual to physical page mapping to user applications can cause security vulnerabilities. Hence, modern operating systems do not allow non-root applications to access the address mapping [2] and therefore a software-based approach would not be applicable in scenarios where non-root users are running OS-intensive applications on the same server. Keeping all these issues in mind, we did not follow the software-based approach.

3.3 Structure associated with a SuperFunction

Our scheduler does NOT change the original algorithm of any *SuperFunction*. It merely governs when and where should a *SuperFunction* run. To do so, we execute a special code snippet at the start of a *SuperFunction*'s execution. This code creates a structure describing the upcoming *SuperFunction*. It then calls scheduler routines that decide when and where should the upcoming *SuperFunction* run. Let us first discuss the information that we maintain for each *SuperFunction*. We shall discuss the scheduler routines in Section 5.

We maintain the following information for each *SuperFunction*:

(1) *superFuncType*: described in Section 3.1.
(2) *superFuncID*: unique 64-bit number that is assigned to each *SuperFunction*.
(3) *parentSuperFuncPtr*: address of the parent *SuperFunction*'s structure. We define a hierarchical relation between *SuperFunctions* so that we can transfer the execution of a thread from a *SuperFunction* to the one it was called from. We shall discuss the usage of this field in Section 5.1.
(4) *tid*: ID of the thread that created the *SuperFunction*.
(5) *coreID*: ID of the core that is currently handling the *SuperFunction*.

superFuncID: Assuming that the system has n cores, the i^{th} core assigns *superFuncIDs* sequentially in the range $[\frac{2^{64}*i}{n}, \frac{2^{64}*(i+1)}{n} - 1]$. If the range is exhausted, the *superFuncID* assignment wraps around. We do not maintain a global *superFuncID* counter because as pointed by Boyd-Wickizer et al. [15], such a counter can lead to a performance bottleneck when multiple cores are simultaneously creating a *SuperFunction*.

4 BENCHMARK CHARACTERIZATION

4.1 Experimental Setup

We use a modified version of the full system emulator, Qemu [10], to collect the execution trace of the entire system. The execution trace contains information that is sufficient to perform a detailed timing simulation: list of retired instructions, load/store addresses, branch outcomes, and OS-specific events such as interrupts and system calls. We subsequently feed these traces to a detailed cycle-accurate

[3]Assuming a 64 bit physical address and a page offset of 12 bits

simulator, Tejas [34]. Table 2 shows the details of our simulated system.

Parameter	Value	Parameter	Value
Cores	32	Technology	22nm
Pipeline			
Retire Width	4	Integer RF (phy)	160
ROB Size	168	Branch Predictor	TAGE
iTLB	128 entry	dTLB	128 entry
L1 i-cache, d-cache (Private caches)			
Associativity	4	Size	32 KB
Latency	3 cycles		
L2 cache (Private cache)			
Associativity	4	Size	256 KB
Latency	8 cycles		
Coherence	Directory based MOESI		
L3 cache (Shared NUCA cache)			
Associativity	8	Size	8 MB
Avg. Latency	18 cycles		
OS	Debian GNU/Linux 6.0.1 squeeze		

Table 2: Baseline System Details

4.2 Benchmarks

We evaluate our technique for a suite of 8 popular OS-intensive benchmarks. Our choice of applications is inspired by previous work [7, 12, 19, 27, 32, 36] that also improve the performance of OS-intensive benchmarks.

(1) *Find:* This benchmark simulates the execution of an application that browses the local filesystem. Specifically, we execute the Linux command `find` to search for a file in a large `ext3` file system, starting from the root directory.
(2) *Iscp:* This benchmark simulates the execution of a large network-copy over a secure connection. Specifically, we execute the Linux command `scp` to copy a 10 GB file from a remote machine to the local machine.
(3) *Oscp:* This benchmark is similar to the *Iscp* application, except that it copies a file from the local machine to a remote machine.
(4) *Apache:* This benchmark simulates the execution of a web server. Specifically, we execute the *Apache* web server on the local machine and use the `ApacheBench` utility to request web pages from a remote machine. We configure the `ApacheBench` utility to request 96 web pages simultaneously; this corresponds to 3 web pages for each core.
(5) *DSS:* This benchmark simulates the execution of a decision control system. Specifically, we execute the `minimal cost supplier` query of the TPC-H benchmark [4] for a database of 1 GB; we use a MySQL database server.
(6) *FileSrv:* This benchmark simulates the execution of a file server that serves concurrent filesystem requests such as read, write, create, and delete on the local filesystem. Specifically, we execute the `fileserver` workload of Filebench [1] with 400 threads.
(7) *MailSrvIO:* This benchmark simulates the execution of a filesystem related system calls for a hypothetical mail server. Specifically, we execute the `mailserver` workload of Filebench [1] with 96 threads.
(8) *OLTP:* This benchmark simulates the execution of a database server. Specifically, we execute the *OLTP* workload of *Sysbench* [28] with 96 threads.

The first 3 benchmarks (*Find*, *Iscp* and *Oscp*) are single-threaded, and the remaining 5 benchmarks are multi-threaded. For evaluating the impact of different core specialization techniques on single-threaded benchmarks, we simulate one instance of the application on each core of the system. We now discuss the characterization results of each benchmark for a representative block of 1 billion instructions per core (akin to [7, 8, 19]).

4.3 Instruction Breakup

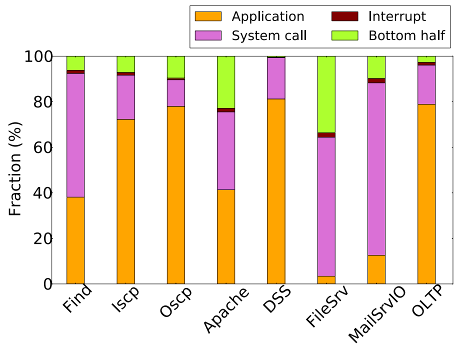


Figure 4: Instruction breakup

Figure 4 shows a breakup of instructions for a system using the Linux scheduler. We show the fraction of execution (in terms of instructions) of each *SuperFunction* category: (1) application, (2) system call handler, (3) interrupt handler, and (4) bottom-half handler. As we seek to replace the Linux scheduler with a new scheduler, we ignore the execution of the Linux scheduler routines in the instruction breakup. We use the binary of the Linux kernel to determine which instructions of the trace correspond to the start of the scheduler and the bottom-half routines. Now let us understand the instruction breakup of each benchmark in detail.

Find searches for a specific filename in the inode structures of the directories recursively; as the search operation is relatively simple, the fraction of execution of its application *SuperFunction* is low (around 35%). Majority of the system call *SuperFunctions* executed by *Find* are related to filesystem browsing. Since *Iscp* decrypts the entire data that it reads over the network-socket, the fraction of execution of its application *SuperFunction* is high. The instruction breakup of *Oscp* is similar to that of the *Iscp* benchmark primarily because the nature of both the benchmarks is similar. *Apache* executes a lot of system calls for handling web page requests; most of these system calls are related to socket-create, and network read/write operations. Consequently, the fraction of execution of its system call handlers is high (around 35%). Additionally, since a web server receives a lot of network interrupts, *Apache* executes a lot of interrupt and bottom-half handlers; this is reflected in the high fraction of execution of its bottom-half handlers (around 20%). Since *DSS* executes long search and aggregate operations on database records, the fraction of execution of its application *SuperFunctions* is high (around 80%). *FileSrv* executes a lot of filesystem related system calls. As it interacts heavily with the hard disks, it receives a lot of disk interrupts; hence, the fraction of execution of bottom half handlers is high (around 35%). *MailSrvIO*, like *FileSrv* also executes a lot of filesystem related system calls. Hence, the fraction of execution of its system call handlers is high (around 70%). *OLTP*, like

DSS also reads database records from the disk and then performs search operations on them; hence, the instruction breakups of both these benchmarks are similar.

4.4 Instruction Breakup: Similarity across Epochs

We now study the similarity between the instruction breakups of *SuperFunctions* across two consecutive epochs of execution. We consider time-epochs of 3 ms (inspired by previous works [12, 37]). We represent an instruction breakup as a vector of n elements, where each element denotes the execution fraction(%) of a particular type of SuperFunction. We measure the similarity between the instruction breakups of two epochs as the *cosine similarity* of the vectors representing their instruction breakups. The cosine similarity between two vectors A and B of length n is defined as:

$$Cosine\ similarity\ (A,B) = \frac{\sum_{i=1}^{n} A_i.B_i}{\sqrt{\sum_{i=1}^{n} {A_i}^2}.\ \sqrt{\sum_{i=1}^{n} {B_i}^2}} \qquad (1)$$

Its ranges from -1 meaning exactly opposite, to $+1$ meaning exactly the same, with 0 indicating no correlation. For all benchmarks, we observe the same pattern while computing the cosine similarity of instruction breakups across two consecutive epochs of execution: the similarity of instruction breakups between two consecutive epochs is low (0-0.3) when a benchmark begins execution, it increases as the benchmark executes more and more epochs, and finally, stabilizes at high similarity values (> 0.995). This behavior is expected as the code that an OS-intensive application executes at the beginning of its execution (*libC* initialization, allocating data structures) is typically not executed again. However, as the main loops in a benchmark begin to execute, we observe that similar *SuperFunctions* execute repeatedly.

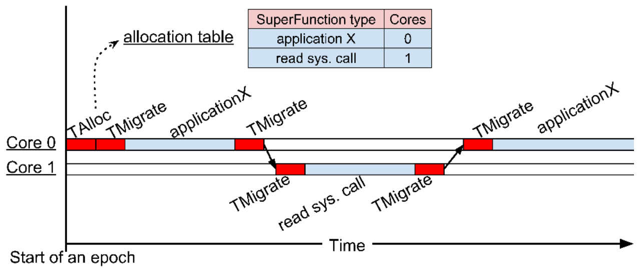
The **main takeaway** from this section is that the execution of OS-intensive applications is highly repetitive. Hence, we can use a simple scheduler that collects the running times of *SuperFunctions* in one epoch, and use the same to create a schedule for the next epoch. During an epoch's execution, *SchedTask* simply migrates *SuperFunctions* to the most appropriate cores when they start execution. We now discuss the details of our scheduler in the next section.

5 SCHEDTASK

5.1 The Timeline of a Thread's Execution

Figure 5 shows how an application thread is executed on a system with the proposed technique. At the beginning of the epoch, an OS function called *TAlloc* is executed on core 0. This function maintains an in-memory system-wide *allocation table* that was created at system initialization time. An *allocation table* stores information regarding which core should execute which type of *SuperFunction*. The scheduler uses it to migrate a thread between cores depending on the type of *SuperFunction* that it is going to execute next. In this example, we begin the execution on core 0.

When *TAlloc* gets executed, it updates the *allocation table* according to the profile that has been collected in the previous epoch. Then, another OS function called *TMigrate* gets executed. Based on the type of the *SuperFunction* that is going to be executed next, *TMigrate* decides on which core the thread should run, possibly migrating it



SuperFunction type	Cores
application X	0
read sys. call	1

Figure 5: Timeline of a thread's execution

to another core. We assume in this example that the *allocation table* indicates that the *application X SuperFunction* should run on core 0. Therefore, *TMigrate* schedules the thread to run on core 0. Note that because *TMigrate* has to schedule every *SuperFunction*, it has to run after every *SuperFunction* on all cores. Suppose that *application X* is going to perform the read system call, which is a different *SuperFunction*, the execution is trapped and the *TMigrate* function is invoked again to schedule it. According to the *allocation table*, the read system call handler should run on core 1. Consequently, *TMigrate* decides to migrate the thread to core 1 and the read system call handler will be executed there. Meanwhile, a *TMigrate* function running on another core might schedule a *SuperFunction* to run on core 0. If this was the case, then core 0 will execute the *SuperFunction* rather than being idle.

Once the read system call handler completes, the execution is trapped again and *TMigrate* gets executed. This time, however, *TMigrate* has to return to the *application X SuperFunction* rather than scheduling a new *SuperFunction*. *TMigrate* recognizes this relation through the *parentSuperFuncPtr* field and schedules the thread to run on core 0.

Figure 6 shows the data structures used by *TAlloc* and *TMigrate* for a scenario where a 4-core system is executing four types of of *SuperFunctions*. Kindly refer to this figure when we introduce the data structures in the subsequent text.

5.2 TAlloc

TAlloc is executed on core 0 at the start of each epoch. It maintains three in-memory data structures: *stats table*, *allocation table*, and *overlap table*. *stats table* stores the frequency, total execution time, and the *Page-heatmap* of each *superFuncType*. *TAlloc* first aggregates (see the aggregation operation in Figure 6) the per-core *stats table* of the last time-epoch and updates the system-wide *stats table*.

TAlloc then allocates cores to each *superFuncType* in direct proportion to its execution fraction in the last epoch; this information is maintained in the *allocation table*. In Figure 6, each *superFuncType* has an execution fraction of 25% in a 4-core system. Hence, we allocate one core for each *superFuncType* (consider a homogeneous system). Once the *allocation table* is created, we transfer each thread to the core that is mapped to its *SuperFunction*'s *superFuncType*. If an interrupt handler x is supposed to run on core y, then *TAlloc* programs the interrupt controller to route interrupts of ID x to core y. Interrupts whose IDs are not present in the *stats table* are mapped to core 0 by default. In order to minimize the cost of transferring threads from one core to another, we perform core allocation only

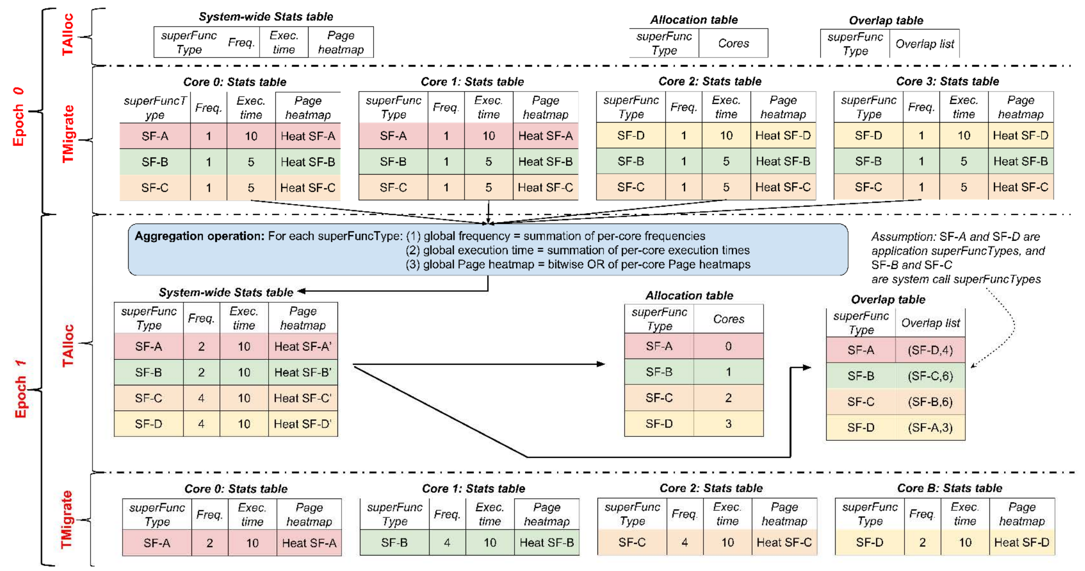

Epoch 0
TAlloc
System-wide Stats table
superFunc Type | Freq. | Exec. time | Page heatmap
Allocation table
superFunc Type | Cores
Overlap table
superFunc Type | Overlap list
TMigrate
Core 0: Stats table
superFuncType | Freq. | Exec. time | Page heatmap
SF-A | 1 | 10 | Heat SF-A
SF-B | 1 | 5 | Heat SF-B
SF-C | 1 | 5 | Heat SF-C
Core 1: Stats table
SF-A | 1 | 10 | Heat SF-A
SF-B | 1 | 5 | Heat SF-B
SF-C | 1 | 5 | Heat SF-C
Core 2: Stats table
SF-D | 1 | 10 | Heat SF-D
SF-B | 1 | 5 | Heat SF-B
SF-C | 1 | 5 | Heat SF-C
Core 3: Stats table
SF-D | 1 | 10 | Heat SF-D
SF-B | 1 | 5 | Heat SF-B
SF-C | 1 | 5 | Heat SF-C
Aggregation operation: For each superFuncType: (1) global frequency = summation of per-core frequencies
(2) global execution time = summation of per-core execution times
(3) global Page heatmap = bitwise OR of per-core Page heatmaps
Assumption: SF-A and SF-D are application superFuncTypes, and SF-B and SF-C are system call superFuncTypes
Epoch 1
TAlloc
System-wide Stats table
SF-A | 2 | 10 | Heat SF-A'
SF-B | 2 | 10 | Heat SF-B'
SF-C | 4 | 10 | Heat SF-C'
SF-D | 4 | 10 | Heat SF-D'
Allocation table
SF-A | 0
SF-B | 1
SF-C | 2
SF-D | 3
Overlap table
SF-A | (SF-D,4)
SF-B | (SF-C,6)
SF-C | (SF-B,6)
SF-D | (SF-A,3)
TMigrate
Core 0: Stats table
SF-A | 2 | 10 | Heat SF-A
Core 1: Stats table
SF-B | 4 | 10 | Heat SF-B
Core 2: Stats table
SF-C | 4 | 10 | Heat SF-C
Core B: Stats table
SF-D | 2 | 10 | Heat SF-D

Figure 6: *SchedTask*: High level design

if the cosine similarity of the execution fractions for the last two epochs is less than 0.98.

Next, *TAlloc* creates the *overlap table* from the *Page-heatmap* values of different *superFuncTypes*. For each *superFuncType*, *overlap table* stores a list of tuples <*superFuncType, Page overlap*> in the decreasing order of the *Page overlap* values. We do not calculate the *Page overlap* values between OS-specific and application *superFuncTypes*.

5.3 TMigrate

TMigrate is the OS function that handles the execution of different *SuperFunctions* on each core. Algorithm 1 shows the high level approach of *TMigrate* and the functions that it executes.

TMigrate maintains three in-memory data structures for each core: (1) *executing*SuperFunction, a pointer to the structure of the *SuperFunction* that is being executed, (2) *runnable queue*, a queue of all SuperFunctions that are ready to run, (3) *waiting queue*, a queue of SuperFunctions that are waiting for some event; eg: a *read* system call handler that is waiting for the disk interrupt controller to bring a page from the disk to the main memory.

Collecting execution statistics: The *TMigrate* function calls the *startStatsCollection* function before and the *stopStatsCollection* after each *SuperFunction*. These functions calculate the execution statistics for each individual *SuperFunction* and add them to the corresponding *superFuncType*'s entry in the *stats table*.

Start a* SuperFunction*: Before executing a *SuperFunction*, a thread first creates its structure and then calls *TMigrate*. *TMigrate* refers to the *allocation table* to decide which cores should execute the new *SuperFunction*. If there is an option to choose one among multiple cores, then the core that has the least waiting time is selected. The waiting time of a core is equal to the sum of the average execution time of all *SuperFunctions* that are present in its *runnable queue*. *TMigrate* migrates the *SuperFunction* to the selected core by appending it to the core's *runnable queue*. The *runnable queue* is updated using a lock-free implementation. If the *allocation table* does not contain any entry for the *SuperFunction's superFuncType*, it is executed on the local core.

Algorithm 1 TMigrate

```
procedure EXECUTENEXTRUNNABLETASK
    if runnableQueue.isEmpty() then
        $stealWorkOfOtherCores()$
    $currentSF \leftarrow runnableQueue.removeHead()$
procedure STEALWORKOFOTHERCORES
    $sf \leftarrow stealWork(strategy = SAME\_WORK\_ONLY)$
    if sf!=NULL then
        $addToRunnableQueue(sf)$ return
    else
        $multipleSF \leftarrow stealWork(strategy = SIMILAR\_WORK\_ALSO)$
        if multipleSF!=NULL then
            $addAllToRunnableQueue(multipleSF)$ return
        else
            $idle()$
procedure STARTSUPERFUNCTION(SuperFunction s)
    $cores \leftarrow getAllocationTableEntry(allocationTable, s.superFuncType)$
    if cores.isEmpty() then
        $addToRunnableQueue(currentCore, s)$
    else
        $core \leftarrow selectCoreThatHasLeastWaitingTime(cores)$
        $addToRunnableQueue(core, s)$
        if isInIdleState(core) then
            $sendInterProcessorInterrupt(core)$
procedure STOPSUPERFUNCTION(SuperFunction s)
    $deallocateRecord(s)$
    $executeNextRunnableTask()$
procedure STARTSTATSCOLLECTION
    $startTimeForSF \leftarrow getCurrentTime()$
    $clearFuncHeatMapRegister()$
    $initializeStateForRunning(currentSF)$
procedure STOPSTATSCOLLECTION
    $entry \leftarrow getPerCoreStatsTableEntry(currentSF.superFuncType)$
    $entry.execTime = entry.execTime + (getCurrentTime() - startTimeForSF)$
    $entry.funcHeatMap = bitwiseOR(entry.funcHeatMap, funcHeatMapRegister)$
procedure TMIGRATE(requestType, requestPayload)
    $stopStatsCollection()$
    if requestType==START_SUPER_FUNCTION then
        $startSuperFunction(requestPayload)$
    else if requestType==STOP_SUPER_FUNCTION then
        $stopSuperFunction(requestPayload)$
    else if requestType==PAUSE_SUPER_FUNCTION then
        $pauseSuperFunction(requestPayload)$
    else if requestType==WAKEUP_SUPER_FUNCTION then
        $wakeupSuperFunction(requestPayload)$
    $startStatsCollection()$
```

Stop a* SuperFunction*: After completing a *SuperFunction*, *TMigrate* resumes the execution of its parent *SuperFunction*. It then executes the *SuperFunction* at the head of its *runnable queue*. If the *runnable queue* of the local core is empty, then *TMigrate* tries to steal *SuperFunctions* from the *runnable queue* of other cores. It tries two levels of work stealing in the following order:

(1) Steal same work only: This is the simplest choice to make. Steal only those *SuperFunctions* whose *superFuncType* is mapped to the local core. This strategy does not increase the possibility of i-cache pollution and yet reduces core idleness. Given multiple cores to steal from, an idle core always steals from the core with the maximum waiting time. If no such thread is found, then *TMigrate* tries the next level of work stealing.

(2) Steal similar work also: *TMigrate* now tries to steal *SuperFunctions* from the *runnable queues* of other cores. The stealing algorithm gives a higher priority to those *SuperFunctions* whose *superFuncTypes* have a high overlap with the ones that are allocated to the local core. *TMigrate* first combines the *overlap table* entries of all *superFuncTypes* that are mapped to its local core. It then iterates over this list in the decreasing order of the *Page overlap* value. The iteration stops when a *SuperFunction* with a particular *superFuncType* is found in the *runnable queue* of another core. If there are multiple such *SuperFunctions* in the remote core's *runnable queue*, then *TMigrate* steals half of them. Initially such *SuperFunctions* will suffer from a low i-cache hit rate. To amortize this effort, the stealing thread typically steals a few more similar SuperFunctions from other cores. This strategy is used as the default scheme for evaluation.

Pausing a* SuperFunction*: When the executing *SuperFunction* goes to the waiting state, *TMigrate* adds it to the *waiting queue* and calls *executeNextRunnableTask*.

Waking up a* SuperFunction*: When the executing *SuperFunction* wants to wake up another *SuperFunction*, we merely move the other *SuperFunction* from the *waiting queue* to the *runnable queue*.

5.4 Modifications

Software: The scheduler routines *TAlloc* and *TMigrate* are implemented in the Linux kernel. *TMigrate* is called by special hooks that are added to the start of all *superFuncTypes*.

Hardware: *SchedTask* requires hardware modifications to maintain the *Page-heatmap* of each *superFuncType*. These modifications are: (1) adding a 512-bit register, (2) implementing the hash function to map a *PFN* to a bit in the heatmap register, and (3) special assembly instructions to load and store heatmap values. These changes are minimal (require $< 0.01\%$ core area) and they do not interfere with the critical path of any instruction.

6 RESULTS

6.1 Comparison: Performance Improvement

We compare the performance benefits gained through five core specialization techniques: (1) *SelectiveOffload* [32], (2) *FlexSC* [36], (3) *DisAggregateOS* [30], (4) *SLICC* [7], and (5) *SchedTask*. Table 2 shows the details of the baseline system and Table 3 shows the configuration for each core specialization technique. Please read the Appendix [5] for understanding the sensitivity of the results to:

(a) Multi-programmed workloads: multiple OS intensive applications are running simultaneously.

Technique	Configuration
SelectiveOffload [32]	64 core system. Offload system call handlers whose run length is greater than 100 instructions.
FlexSC [36]	Zero cycle delay for user-level scheduler. Specialize cores for all system calls.
DisAggregateOS [30]	Zero cycle delay for micro-scheduling.
SLICC [7]	Zero cycle delay to search for remote tags. Size of the hardware components are taken from the original paper.
SchedTask	Our technique → see Section 5

Table 3: Configuration of all core specialization techniques

(b) Size of the instruction cache (16 KB, 32 KB, 64 KB).
(c) Cache configuration: 2-level, 3-level memory hierarchy.
(d) Number of cores (8, 16, 32, 64) in the system.
(e) Instruction prefetcher [6].
(f) Trace cache [29].

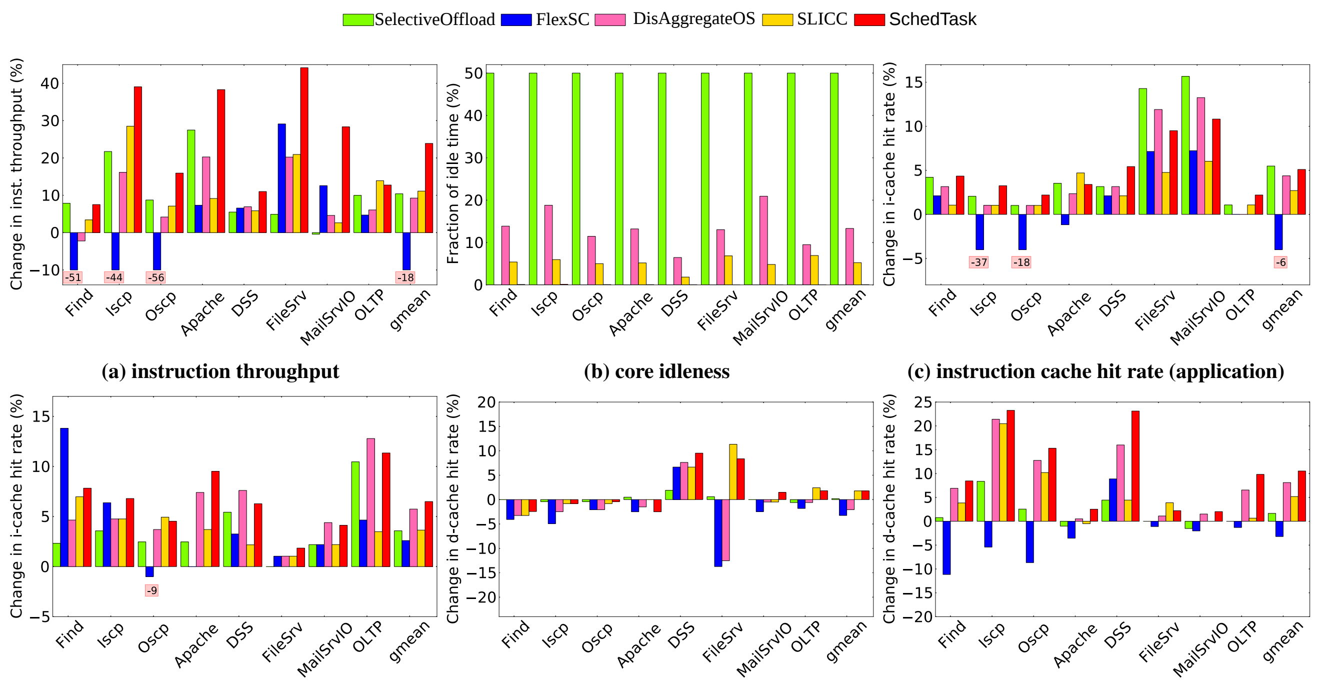
The additional results show that *SchedTask* is the best performing technique across all evaluated configurations.

We evaluate all techniques on a cycle-accurate architectural simulator, Tejas (verified vis-a-vis native hardware). The evaluation has been performed on a simulator because: (a) three of the evaluated techniques (including ours): *SelectiveOffload*, *SLICC* (state of the art) and *SchedTask* propose non-trivial architectural changes and hence, they cannot be evaluated with a purely software based framework, and (b) the evaluated system (32 core out-of-order system) has not been released by any vendor. Hence, we use the same approach as the one used by highly cited recent work [7, 8, 18, 19, 27, 32]. Like our proposal, they also use architectural modifications to improve the performance of OS-intensive applications. We inserted hooks in the Linux kernel to invoke all the *SchedTask* routines. The extra hardware support required for *SchedTask* (special assembly instructions, Bloom filter, and *Page-heatmap register*) was emulated using a patched version of the full system emulator, Qemu. Qemu provides execution traces to the architectural simulator, Tejas.

A system running OS-intensive workloads runs a lot of threads; if some threads are not ready, then a few cores may remain idle. Hence, the standard practice employed in such systems is to spawn more application threads than the number of cores in the system. Hence, for all results shown here, we do this (also done by [7, 36]). We treat the ensemble of all the individual benchmarks discussed in Section 4.2 as the baseline workload and in our experiments we double it. For single-threaded applications, this means spawning twice the number of applications, and for multi-threaded applications, it means spawning twice the number of threads. In Section 6.3, we discuss the impact of varying a benchmark's workload on the performance of these techniques.

Figure 7 shows the impact of each core specialization technique on the application's performance as compared to a baseline system that employs the standard Linux scheduler. We calculate an application's performance as the number of application-specific events that it performs in one second of system execution. For *Find*, an application-specific event is searching an i-node entry; *iscp* and *oscp*, it is receiving/transmitting a data packet; *Apache*, it is serving a web page, *DSS* and *OLTP*, it is processing a query; *FileSrv*, it is completing a file-operation; *MailSrvIO*, it is completing a mail-operation. We instrument the source code of each benchmark to count the number of such events.

The mean (geometric) improvements in the application's performance for these techniques are: *SelectiveOffload* (10.62%), *FlexSC*



(a) instruction throughput **(b) core idleness** **(c) instruction cache hit rate (application)**

(d) instruction cache hit rate (OS) **(e) data cache hit rate (application)** **(f) data cache hit rate (OS)**

Figure 8: Impact of core specialization techniques on microarchitectural parameters

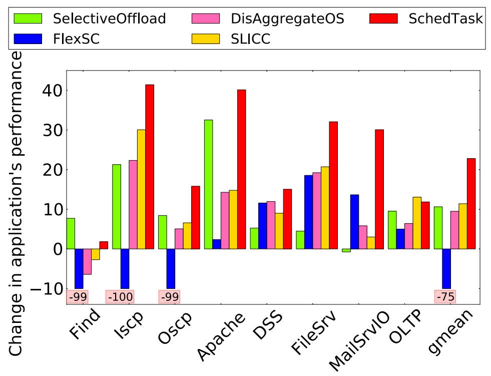


Figure 7: Impact of core specialization techniques on application's performance

(-75% for all benchmarks and +10.08% for multi-threaded benchmarks), *DisAggregateOS* (9.49%), *SLICC* (11.39%), and *SchedTask* (22.79%). *SchedTask* outperforms the state of the art technique *SLICC* by 11.4%. We study six microarchitectural parameters to understand the application performance results that are reported in Figure 7:

(a) instruction throughput (#insts/second) (Figure 8a),
(b) fraction of time a core remains idle (Figure 8b),
(c) i-cache hit rate when the application code is executing (Figure 8c), and when the OS code is executing (Figure 8d), and
(d) d-cache hit rate when the application code is executing (Figure 8e), and when the OS code is executing (Figure 8f).

On comparing Figure 7 and Figure 8a, we observe that the impact of each technique on the application's performance is roughly the same as its impact on the instruction throughput. We observe two changes. *FlexSC*'s impact on the performance of single-threaded applications is much worse than its impact on the instruction throughput. This is because *FlexSC* executes the OS scheduler each time a single-threaded application executes a system call. Hence, the system executes a lot of *additional* kernel instructions that contribute towards the calculation of instruction throughput but do not contribute to actual application work. Next, we observe that the application performance of the proposed technique, *SchedTask*, is lower than its instruction throughput by around 0.5-1%. This is because *SchedTask* executes *additional* kernel instructions for the *TMigrate* routine; they do not count as application's work. The performance gap between *SchedTask* and the state of the art *SLICC* is still high: 11.4 and 12.7 percentage points in terms application's performance and instruction throughput respectively.

The macro benchmarks considered in this work have sufficient number of threads to overlap compute and IO work. Also, as the evaluation of such benchmarks in previous work [11, 20, 21, 23, 25, 36] suggests, even with a substantial increase in the compute speed, such macro benchmarks do not saturate the bandwidth of the IO devices. Hence, increasing the compute speed leads to an increase in the applications' performance. Let us now analyze the performance of each core specialization technique in detail.

SelectiveOffload: Since the *SelectiveOffload* technique lacks a work stealing algorithm, the idle time fraction of *SelectiveOffload* scheme is high: 50% (see Figure 8b). Owing to aggressive work stealing algorithms, the idle time fractions of *FlexSC* and *SchedTask* are almost 0%. *SelectiveOffload* executes only one application thread on each application core. Hence, among all the evaluated techniques, the i-cache hit rate of the application code is the highest for the

Workload	Technique	Find		Iscp		Oscp		Apache		DSS		FileSrv		MailSrvIO		OLTP		geom. mean*	
		Idle	Perf	Idle	Perf	Idle	Perf	Idle	Perf	Idle	Perf	Idle	Perf	Idle	Perf	Idle	Perf	Idle	Perf
1 X	SelectiveOffload	50	7	50	21	50	8	50	27	50	5	50	3	50	0	50	10	50	10
	FlexSC	0	-51	0	-44	0	-55	0	13	0	5	0	5	0	18	0	4	0	-18
	DisAggregateOS	39	-8	50	-4	41	-11	43	5	25	5	40	-6	53	-8	37	-1	41	-3
	SLICC	41	-9	41	1	41	-8	37	-14	43	10	42	-12	39	-9	41	0	41	-5
	SchedTask	18	4	13	17	20	14	11	17	5	1	0	42	7	22	7	2	10	14
2 X	SelectiveOffload	50	7	50	21	50	8	50	27	50	5	50	3	50	0	50	10	50	10
	FlexSC	0	-51	0	-44	0	-55	0	7	0	6	0	26	0	13	0	3	0	-18
	DisAggregateOS	13	-1	19	13	11	5	13	20	6	6	13	18	20	4	9	4	13	8
	SLICC	5	4	5	25	5	8	5	9	1	5	6	22	4	2	6	13	5	11
	SchedTask	0	8	0	34	0	17	0	39	0	10	0	42	0	28	0	10	0	23
4 X	SelectiveOffload	50	7	50	21	50	8	50	27	50	5	50	3	50	0	50	10	50	10
	FlexSC	0	-51	0	-44	0	-55	0	20	0	7	0	6	0	14	0	4	0	-18
	DisAggregateOS	5	3	8	10	3	9	5	28	1	8	4	24	7	14	2	3	4	12
	SLICC	1	3	1	17	0	5	1	22	0	5	0	15	0	4	1	9	0	10
	SchedTask	0	12	0	33	0	17	0	52	0	12	0	42	0	31	0	11	0	25
8 X	SelectiveOffload	50	7	50	21	50	8	50	27	50	5	50	3	50	0	50	10	50	10
	FlexSC	0	-51	0	-44	0	-55	0	30	0	8	0	47	0	15	0	6	0	-13
	DisAggregateOS	1	11	4	-4	0	18	1	35	0	8	1	51	2	20	1	5	1	17
	SLICC	0	9	0	4	0	12	0	29	0	5	0	4	0	6	0	12	0	10
	SchedTask	0	19	0	16	0	27	0	58	0	11	0	42	0	34	0	14	0	27

Idle $\rightarrow$ fraction of idle time (%). Perf $\rightarrow$ change in instruction throughput (%) relative to the baseline with the same workload

Table 4: Impact of the workload on the instruction throughput and idle time fractions

SelectiveOffload technique (see Figure 8c). *SelectiveOffload* has a coarse grained mapping for OS cores; it executes all system call handlers on the same core. Since different system call handlers evict each other's lines in the i-cache and the d-cache, *SelectiveOffload* results in low i-cache and d-cache hit rates when the OS code is executing (Figure 8d and Figure 8f). Overall *SchedTask* outperforms *SelectiveOffload* because of lower core idleness, and higher i-cache and d-cache hit rates for the OS code.

FlexSC: As discussed in Section 2, *FlexSC* executes the Linux scheduler each time a single-threaded application executes a system call. Hence, it's performance for single-threaded applications is low (mean: -98%); however, it's performance for multi-threaded benchmarks is high (mean: 10.08%). For multi-threaded applications, a better performance of the *SchedTask* technique (mean: 25.37%) can be attributed to two reasons: (1) a fine-grained core mapping, and (2) a smarter work stealing algorithm. While *FlexSC* specializes cores for system call handlers, it does not eliminate the i-cache pollution due to interrupt and bottom half handlers. Additionally, *FlexSC* migrates tasks from one core to another when there is an imbalance in the run-queue sizes of different cores. While this strategy ensures that the core idleness is minimal, it regularly migrates the OS threads (that execute system call handlers) between different cores. This decreases the data locality of the OS thread, and thereby leads to a lower d-cache hit rate.

DisAggregateOS: Due to a fine-grained core mapping, the i-cache hit rate for the application and the OS code are high for *DisAggregateOS*. Additionally, because it maps system call handlers that access the same d-cache lines on the same core, the d-cache hit rate for OS code is also high for *DisAggregateOS*. However, owing to the low idle time fraction, *SchedTask* outperforms *DisAggregateOS*.

SLICC performs well on the four parameters: i-cache hit rate for application as well as OS code, and d-cache hit rate for application as well as OS code. However, the mean idle time fraction for the SLICC technique is around 5%, mainly because SLICC does not allow an idle core to steal threads from other cores.

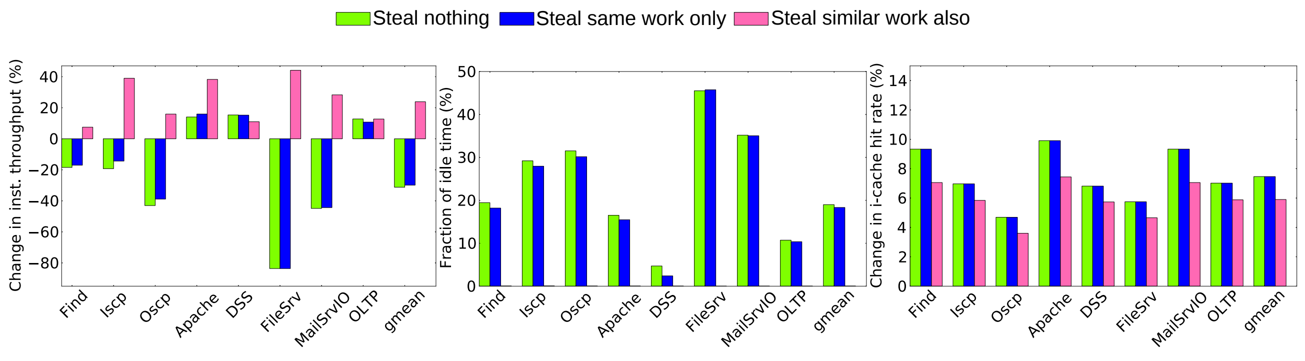
SchedTask performs the best among all the compared techniques. The technique performs well on almost all the studied parameters. Due to a fine-grained core mapping, the i-cache hit rates for the application as well as OS code are high. Additionally, since *SchedTask* uses a smart work stealing algorithm, its idle time fraction is low (almost 0%) and its d-cache hit rates for application and OS code are high.

Data locality: In a baseline Linux system, a system call handler gets executed on the same core on which it got invoked. So if it got invoked on multiple cores, the OS data structures that it accesses are fetched into the respective data caches. If fetching that data incurs any stalls, the overhead will be incurred on all of the cores. In addition, if a shared cache line got modified by one of the cores, there will be an additional overhead due to cache coherence. Since *SchedTask* executes different instances of a system call handler on the same core, the data used by the handler is loaded once into the d-cache and reused in later executions of the handler. Hence, as shown in Figure 8f *SchedTask* significantly improves the d-cache hit rate of the OS code. A similar effect is also observed for the application code for multi-threaded benchmarks.

Other statistics: Let us now discuss some more results, and present aggregate statistics instead of benchmark wise results due to a lack of space.

(1) ***SchedTask related overheads:*** The *SchedTask* technique replaces the Linux scheduler with two components: *TAlloc* and *TMigrate*. We observe that the system spends a negligible amount of time ($< 0.01\%$) executing the *TAlloc* function. This is because *TAlloc* is executed only once during each time-epoch of $3ms$. On the other hand, the *TMigrate* function is called at the start and the end of each SuperFunction. Hence, it contributes much more (around 3.2%) to the system's execution. This is roughly the same amount of time that the baseline Linux system spends executing its scheduler; impressive speed ups in the program execution (around 24%) compensate for the execution of *SchedTask* related routines. Through simulation studies, we observe that the data state maintained by *SchedTask* causes a reduction in the d-cache hit rate of the non-*SchedTask* codes by 0.78%.

(2) ***TLB hit rates:*** Due to a reduction in the instruction and data footprints on each core, the hit rates of the iTLB and dTLB also increase by 0.98% and 0.65% respectively.



(a) instruction throughput **(b) core idleness** **(c) i-cache hit rate (overall)**

Figure 9: Impact of work stealing strategies on different system parameters

(3) ***Interrupt latency:*** Since *SchedTask* steers many interrupts to the same core, sometimes the interrupts have to wait before being serviced. This increases the mean interrupt latency by around 0.53%. This is miniscule.

(4) ***Fairness of scheduling:*** We measure the fairness of a schedule by comparing the instruction throughput of all threads using Jain's fairness index [24]; its value ranges from $\frac{1}{\#threads}$ (for a completely unfair schedule) to 1.0 (for a completely fair schedule). The mean fairness index for *SchedTask* is 0.99 indicating that *SchedTask* allocates almost equal execution times to all threads. This is because we use the FCFS strategy in the *TMigrate* routine.

6.2 Thread Migrations

Figure 10 shows the number of inter-core thread migrations per billion retired instructions. The baseline system employs the standard Linux scheduler. Linux's scheduler tries to allocate the same amount of work to all cores and it migrates a thread from one core to another only if there is a significant imbalance of work across cores. Since almost all threads of the considered benchmarks are uniformly stressed, we observe minimal thread migrations in the baseline system. In contrast, the core specialization techniques migrate threads too often. It must be noted that migrating a thread from one core to another does not decrease its performance if there is a concomitant increase in instruction and data locality. Hence, in spite of an increased number of thread migrations, owing to fine-grained scheduling decisions and a smart work stealing algorithm, *SchedTask* outperforms other techniques.

6.3 Impact of the Workload on Performance

Table 4 shows the impact of a benchmark's workload on the idle time fraction and the instruction throughput of different core specialization techniques. 1*X* refers to the ensemble of individual workloads as described in Section 4.2 and 2*X* refers to two times this workload.

1X: The idle time fraction of all techniques is high for a 1*X* workload. While the *SelectiveOffload* technique gives the best performance, it also employs twice the number of cores as compared to other techniques. As mentioned in Section 6.1, *FlexSC* performs poorly for single threaded benchmarks. For multi-threaded benchmarks, *FlexSC* outperforms *DisAggregateOS* and *SLICC* by 10-15 percentage points, primarily on account of its low core idleness. The results clearly show that *DisAggregateOS* and *SLICC* are not

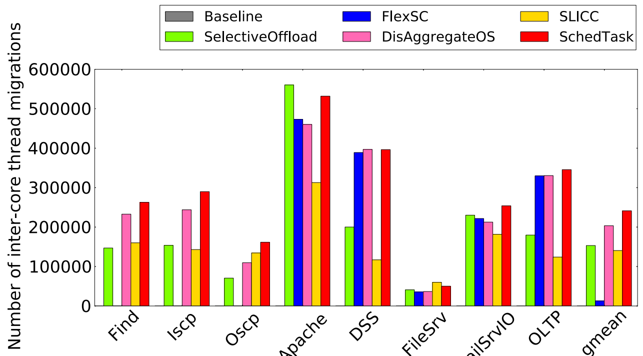


Figure 10: Number of inter-core thread migrations

suitable for servers that have less work. As discussed in Section 6.1, *SchedTask* has low core idleness and a high i-cache hit rate. Hence, in spite of a higher idle time fraction than that of *FlexSC*, *SchedTask* outperforms *FlexSC* even for multi-threaded benchmarks.

2X: As we increase the workload from 1*X* to 2*X*, the idle time fractions of *DisAggregateOS*, *SLICC*, and *SchedTask* drop significantly. Due to fine-grained core mapping, both *SLICC* and *SchedTask* outperform the *SelectiveOffload* technique (*SchedTask* being the best).

For **4X** and **8X** workloads, the idle time fraction for all techniques except *SelectiveOffload* is almost 0%. For such workloads, the technique with the most fine-grained core mapping and the best work stealing strategy will perform the best. Hence, despite having almost the same idle time fraction, *SchedTask* outperforms *DisAggregateOS* and *SLICC*.

Conclusion: For *4X* and *8X* workloads *SchedTask* is still the best. Beyond an 8*X* workload, we observe that the d-cache pollution among application as well as OS threads becomes high. This leads to lower performance and is counter productive.

6.4 Impact of Work Stealing

Figures 9a, 9b, and 9c show the impact of different work stealing strategies on the instruction throughput, idle time fraction and the overall i-cache hit rate respectively. The first strategy is to not allow an idle core to steal any work from other cores. Due to reduced

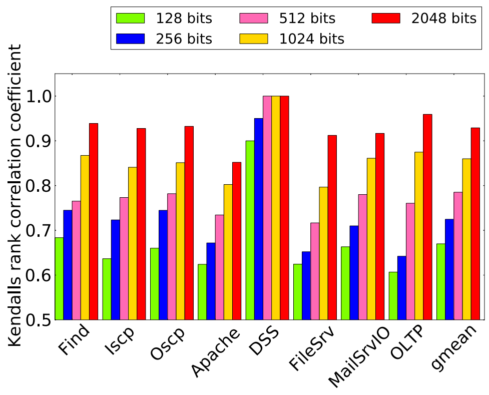


Figure 11: Impact of the size of the *Page-heatmap register* on the quality of its ranking

i-cache pollution between different *SuperFunctions*, the i-cache hit rate for this strategy is high. However, there are periods of time when all threads and applications execute the same *SuperFunction* and hence, they wait in the *runnable queue* of just one or two cores; thereby leading to a high idle time fraction of 19%. The other two bars represent the performance of the two levels of work stealing that we discussed in Section 5.3.

As *Steal same work only* strategy steals *SuperFunctions* with the same instruction footprint as allotted to the local core, it does not increase i-cache pollution and still reduces the idle time fraction of cores by around 0.7% (compared to no work stealing). *SchedTask* tries the *Steal similar work also* strategy only if the *Steal same work only* strategy does not have any *SuperFunction* to execute. This strategy reduces the idle time fraction of *FileSrv* by a massive 45%. We note that *FileSrv* executes a lot of bottom-half handlers (see Figure 4) whose average length is around 24,000 instructions. In the *Steal same work only* strategy, most threads in the system wait to execute their bottom half handler. Since this strategy ensures that an idle core always tries to steal similar *SuperFunctions* first, it reduces the i-cache hit rate by a small amount (around 1%). However, this is adequately compensated by reducing the core idleness to almost 0%. Hence, *SchedTask* uses this as the default work stealing strategy. An alternate strategy is to focus only on core idleness: always steal work from the core with the maximum waiting time. However, this strategy causes higher i-cache pollution and hence has modest performance benefits (mean: 10.77%).

6.5 Impact of the Page-heatmap Register

As discussed in Section 3.2, we use a Bloom filter to approximate the set of i-cache lines that two *superFuncType*'s have in common. Given a *superFuncType*, we compute its Hamming weight against each other *superFuncType* and then compute a ranking (ordered list of *superFuncTypes* in decreasing order of Hamming Weight). We measure the effectiveness of this approximation (using a Bloom filter) by comparing the quality of its ranking against one generated using the actual set of i-cache line addresses. We compare two ranked lists using the *Kendall's rank correlation coefficient* τ_B [35]; its value ranges from -1 (opposite ranking) to +1 (same ranking).

Figure 11 shows the value of τ_B for different sizes of the *Page-heatmap register*. An exponential increase in the size of this register leads to a linear increase in its τ_B. However, the mean performance benefits of *SchedTask* with different sizes of the *Page-heatmap register* do not follow the same trend: *128 bits* (15.87%), *256 bits* (19.37%), *512 bits* (22.79%), *1024 bits* (22.63%), and *2048 bits* (22.71%). The performance benefits for a *Page-heatmap register* of 1024 and 2048 bits are lower than that for 512 bits because of two reasons: (1) increased d-cache pollution for *TAlloc* and *TMigrate* routines, and (2) fewer chances of stealing *SuperFunctions* with higher overlap values at run time. Hence, we choose a *Page-heatmap register* of 512 bits in all our experiments. The mean performance benefit while using the ideal ranking and not facing any d-cache pollution is 24.99%.

7 CONCLUSION

In this work, we proposed *SchedTask*, a fine-grained scheduling scheme for OS-intensive applications. We began by decomposing the combined execution of the OS and the applications into sequences of instructions called SuperFunctions. We proposed a hardware technique (Bloom filter of 512 bits per core) to identify the overlap between the instruction sequences of different types of *SuperFunctions*. We then proposed a hierarchical scheduler that schedules similar *SuperFunctions* on each core. Our scheduler also contains a novel work stealing algorithm that reduces i-cache misses and also reduces the core idleness. Through extensive evaluation over a suite of 8 OS-intensive applications, we demonstrated a performance improvement of up to 29 percentage points (mean: 11.4 p.p.) over the nearest competing state of the art proposal, (SLICC [7]), in this area.

ACKNOWLEDGMENT

We thank Omais Pandith and Himani Raina for providing us their Tejas model of "Trace Based Instruction Caching"; it helped us evaluate the impact of Trace Caches on different core specialization techniques. We also thank Hadi Brais, Rajshekar Kalayappan, Rajesh Kedia, and Neetu Jindal for providing valuable feedback on the initial version of the manuscript; it allowed us to immensely improve our paper.

REFERENCES

[1] 2016. Filebench. (2016). https://github.com/filebench/filebench/wiki
[2] 2016. Linux Security Fix against Rowhammer Vulnerability. (2016). https://lwn.net/Articles/642069/
[3] 2016. Project Zero: Exploiting the DRAM rowhammer bug to gain kernel privileges. (2016). https://googleprojectzero.blogspot.in/2015/03/exploiting-dram-rowhammer-bug-to-gain.html
[4] 2016. TPC-H. (2016). http://www.tpc.org/tpch/
[5] 2017. Sensitivity Analysis of Core Specialization Techniques. (2017). https://arxiv.org/abs/1708.03900
[6] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. 2003. Call Graph Prefetching for Database Applications. *ACM Transactions on Computer Systems* (2003). https://doi.org/10.1145/945506.945509
[7] Islam Atta, Pinar Tozun, Anastasia Ailamaki, and Andreas Moshovos. 2012. SLICC: Self-Assembly of Instruction Cache Collectives for OLTP Workloads. In *ACM/IEEE Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2012.26
[8] Islam Atta, Pinar Tözün, Xin Tong, Anastasia Ailamaki, and Andreas Moshovos. 2013. STREX: boosting instruction cache reuse in OLTP workloads through stratified transaction execution. In *ACM International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/2508148.2485946
[9] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In

ACM Symposium on Operating Systems Principles (SOSP). https://doi.org/10.1145/1629575.1629579

[10] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*.

[11] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. (2010). http://dl.acm.org/citation.cfm?id=1924943.1924973

[12] Rohan Bhalla, Prathmesh Kallurkar, Nitin Gupta, and Smruti R Sarangi. 2014. TriKon: A Hypervisor Aware Manycore Processor. In *IEEE International Conference on High Performance Computing (HiPC)*. http://ieeexplore.ieee.org/document/7116710/

[13] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* (1970). https://doi.org/10.1145/362686.362692

[14] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, et al. 2008. Corey: An Operating System for Many Cores. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. http://dl.acm.org/citation.cfm?id=1855741.1855745

[15] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, Nickolai Zeldovich, et al. 2010. An Analysis of Linux Scalability to Many Cores. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[16] Koushik Chakraborty, Philip M Wells, and Gurindar S Sohi. 2006. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/1168919.1168893

[17] S. Chandran, P. Kallurkar, P. Gupta, and S.R. Sarangi. 2014. Architectural Support for Handling Jitter in Shared Memory Based Parallel Applications. *IEEE Transactions on Parallel and Distributed Systems* (2014). https://doi.org/10.1109/TPDS.2013.127

[18] Nachiappan Chidambaram Nachiappan, Praveen Yedlapalli, Niranjan Soundararajan, Mahmut Taylan Kandemir, Anand Sivasubramaniam, and Chita R Das. 2014. GemDroid: a framework to evaluate mobile platforms. *ACM SIGMETRICS Performance Evaluation Review* (2014). https://doi.org/10.1145/2637364.2591973

[19] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. 2011. Proactive Instruction Fetch. In *ACM/IEEE Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/2155620.2155638

[20] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. 2012. ELI: bare-metal performance for I/O virtualization. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012). https://doi.org/10.1145/2248487.2151020

[21] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. http://dl.acm.org/citation.cfm?id=2387880.2387894

[22] Stavros Harizopoulos and Anastassia Ailamaki. 2004. STEPS towards cache-resident transaction processing. In *International Conference on Very Large Databases (VLDB)*. http://dl.acm.org/citation.cfm?id=1316689.1316747

[23] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. 2014. Rekindling network protocol innovation with user-level stacks. *ACM SIGCOMM Computer Communication Review* (2014). https://doi.org/10.1145/2602204.2602212

[24] Raj Jain, Arjan Durresi, and Gojko Babic. 1999. *Throughput fairness index: An explanation*. Technical Report. Tech. rep., Department of CIS, The Ohio State University.

[25] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. http://dl.acm.org/citation.cfm?id=2616448.2616493

[26] Prathmesh Kallurkar and Smruti R Sarangi. 2016. pTask: A Smart Prefetching Scheme for OS Intensive Applications. In *ACM/IEEE Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2016.7783706

[27] Aasheesh Kolli, Ali Saidi, and Thomas F Wenisch. 2013. RDIP: return-address-stack directed instruction prefetching. In *ACM/IEEE Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/2540708.2540731

[28] Alexey Kopytov. 2004. SysBench: a system performance benchmark. (2004).

[29] Robert F Krick, Glenn J Hinton, Michael D Upton, David J Sager, and Chan W Lee. 2000. Trace based instruction caching. (2000). US Patent 6,018,786.

[30] Min Lee. 2013. *Memory region: a system abstraction for managing the complex memory structures of multicore platforms*. Ph.D. Dissertation. Georgia Institute of Technology.

[31] Pierre Michaud. 2004. Exploiting the cache capacity of a single-chip multi-core processor with execution migration. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2004.10026

[32] David Nellans, Rajeev Balasubramonian, and Erik Brunvand. 2009. Interference Aware Cache Designs for Operating System Execution. *University of Utah, Tech. Rep. UUCS-09-002* (2009).

[33] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*. Austin, TX. https://www.usenix.org/node/197211

[34] S. R. Sarangi, Kalayappan Rajshekar, Kallurkar Prathmesh, Goel Seep, and Peter Eldhose. 2015. Tejas: A Java based Versatile Micro-architectural Simulator. In *IEEE International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*. https://doi.org/10.1109/PATMOS.2015.7347586

[35] Pranab Kumar Sen. 1968. Estimates of the Regression Coefficient Based on Kendall's Tau. *J. Amer. Statist. Assoc.* (1968).

[36] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. http://dl.acm.org/citation.cfm?id=1924943.1924946

[37] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. 2012. Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE). In *ACM International Symposium on Computer Architecture (ISCA)*. http://dl.acm.org/citation.cfm?id=2337159.2337184

[38] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *ACM SIGSAC Conference on Computer and Communications Security*. https://doi.org/10.1145/2976749.2978406

[39] David Wentzlaff and Anant Agarwal. 2009. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *ACM SIGOPS Operating System Review (OSR)* (2009). https://doi.org/10.1145/1531793.1531805

[40] Matthew Wilcox. 2003. I'll Do It Later: Softirqs, Tasklets, Bottom Halves, Task Queues, Work Queues and Timers. In *linux. conf. au*.