Governing with Insights: Towards Profile-driven Cache Management of Black-Box Applications

Golsana Ghaemi
Boston University, USA

Dharmesh Tarapore
Boston University, USA

Renato Mancuso
Boston University, USA

Abstract

There exists a divide between the ever-increasing demand for high-performance embedded systems and the availability of practical methodologies to understand the interplay of complex data-intensive applications with hardware memory resources. On the one hand, traditional static analysis approaches are seldomly applicable to latest-generation multi-core platforms due to a lack of accurate micro-architectural models. On the other hand, measurement-based methods only provide coarse-grained information about the end-to-end execution of a given real-time application.

In this paper, we describe a novel methodology, namely Black-Box Profiling (BBProf), to gather fine-grained insights on the usage of cache resources in applications of realistic complexity. The goal of our technique is to extract the relative importance of individual memory pages towards the overall temporal behavior of a target application. Importantly, BBProf does not require the semantics of the target application to be known — i.e., applications are treated as black-boxes — and it does not rely on any platform-specific hardware support. We provide an open-source full-system implementation and showcase how BBProf can be used to perform profile-driven cache management.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture

Keywords and phrases Profiling, Cache Interference, Multi-core, C2IS

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2021.4

Funding *Renato Mancuso*: The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant number CCF-2008799. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

1 Introduction

The evolution of multi-core architectures and the ever-widening gap between the performance of processor and memory has rendered the adoption of system-level management strategies for shared memory resources a must. Indeed, inter-core interference is a fundamental challenge for the practical adoption of multi-core systems in safety-critical real-time applications, as extensively surveyed in [26]. In a nutshell, the problem of inter-core interference arises due to priority- and criticality-agnostic arbitration for the allocation of and access to shared memory components of application workload deployed in parallel on multiple cores. Important achievements have been accomplished by the research community in the design of practical memory management techniques to mitigate inter-core interference.

Unfortunately, however, the most widely used techniques rely on the enforcement of strict resource partitioning — e.g., shared cache space coloring [23], sustainable memory bandwidth partitioning [39, 37]. Often times, the rigidity of strict resource partitioning results in what is known as the *one-out-of-m* multi-core problem [20]. That is, the performance loss resulting from enacting strict partitioning outweighs its benefits. We argue that at the

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021).
Editor: Björn B. Brandenburg; Article No. 4; pp. 4:1–4:25
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

core of the problem is a fundamental lack of methodologies to analyze exactly *how* realistic, data-intensive applications interact with and benefit from the complex hierarchy of memory resources in modern high-performance embedded systems.

The goal of this paper is to provide one such methodology that goes under the name of *Black-Box Profiling*, or BBProf for short. Specifically, we propose a profiling strategy that can be used to accurately understand how an application's temporal behavior is affected by the presence/absence in the cache of individual memory pages. This sets our work apart from other profiling strategies that compute only end-to-end metrics such as the total cache hit/miss rate, number of bus accesses, resulting runtime when adopting a given resource partitioning scheme, and so on. The BBProf methodology is designed to operate without requiring a micro-architectural model, which is often unavailable (or just too complex) for high-performance systems. The proposed BBProf adopts a measurement-based approach that does not rely on any platform-specific hardware support, and can be ported to virtually any platform.

With this paper, we make the following contributions. First, we propose a novel profiling methodology that requires no special hardware support to produce insights about the relative importance of each memory page towards the overall timing of a target application. Second, we describe how said methodology can be applied to profile realistic, pre-compiled black-box applications without requiring any source-level or compile-time modifications. Third, we propose a proof-of-concept, open-source, full-system implementation and show its capability of profiling real-world vision applications. Fourth, we demonstrate that profile-driven shared cache management is enabled by our BBProf methodology and highlight its benefit in two scenarios: (1) to enact flexible interference mitigation with absolute guarantees that are comparable to strict partitioning; and (2) as an efficient solution to the previously undocumented problem of Contention-Induced Instruction Stall (C2IS).

2 Related Work

Research interest for workload-aware cache management has been spurred a large body of works targeting real-time systems and general-purpose systems alike. A number of works have proposed techniques to estimate the working-set size (WSS) of applications for the purpose of performing informed cache management. One such work is [8], where the WSS of a periodic application is estimated by computing the average per-activation number of cache misses. This information, albeit coarse, is proven useful to avoid concurrently scheduling applications with incompatible WSS. In a spirit quite similar to our BBProf, the work in [40] proposes a technique to detect *hot* memory pages and to dynamically perform re-coloring to improve average performance. Hot pages detection is performed by periodically scanning the *accessed-bit* in all the page-table entries that belong to the target application. This methodology, however, only provides an indirect estimation of the importance of each page that depends on the frequency of sampling. It also relies on the presence of the accessed-bit, which is an Intel-specific hardware feature. The work in [32] uses a similar approach that relies on PowerPC-specific sampled-address data registers (SDAR).

Several works [19, 17, 6] propose scheduling models where the balance between loss of performance due to smaller cache partitions and performance improvements thanks to reduced cache interference is studied. Generally, these model assume that certain intrinsic properties — e.g. their characteristic miss rates — of the applications under analysis are known. In this case, the BBProf methodology proposed hereby could be used to determine key behavioral parameters required to instantiate such and similar analytical frameworks. More recently, a seminal piece of work has proposed an approach to jointly profile an application's sensitivity to cache size and resulting increase/decrease in the requirement for main memory bandwidth [37]. In many ways, the information collected through the sensitivity study represent an experimentally driven profile. Yet, the workload characterization is quite coarse grained and cannot be directly used, for instance, to determine which specific pages of an application need to be shielded from interference.

BBProf shares many similarities, at least in terms of the end goal, with a number of well-established performance analysis toolkits. The survey in [5] provides a good overview of popular toolkits such as Oprofile [9], Dprof [30], Zoom [1], DynamoRIO [7], Valgrind [28], and Pin [24]. The latter three employ dynamic binary instrumentation (DBI), i.e. the ability to translate and instrument on the fly a target binary. DBI-based tools require extensive platform-specific porting. Translation layers for multiple platforms are already provided in Valgrind and DynamoRIO. DBI heavily impacts the timing of an application, so profiling of memory pages has to be performed by instrumenting all the memory references and then conducting a frequency analysis. To the best of our knowledge, the only work that uses one of these tools — the Lackey sub-tool in Valgrind — in this manner is [25]. In [25], a list of hot memory pages to be locked in cache is constructed via meomory tracing, but due to extreme performance degradation incurred, the evaluation is limited to small benchmarks. Lastly, DBI frameworks meant for general-purpose systems seldomly work out of the box on embedded systems due to the complex tree of library dependencies that they rely on, as also reported in [22]. Oprofile, Dprof, and Zoom rely on hardware performance counters to collect information. Oprofile records a variety of statistics such as the mix of hit/miss for L1/L2 caches. It relies on runtime sampling and provides a configurable trade-off between accuracy and overhead. Zoom and Dprof operate on similar principles but the development of Zoom has been discontinued in 2015, while Dprof relies on AMD-specific debug registers. Similarly, the profiling approach proposed in the recently published CacheFlow toolkit [34] relies on the hardware-specific ability, available in a subset of `Aarch64` CPUs, to snapshot the full content of CPU caches.

Since BBProf follows a measurement-based approach, it shares some similarities with the vast literature on measurement-based WCET estimation tools. For instance, the work in [31] aims at producing more accurate WCET estimates by designing synthetic benchmarks that stress different hardware resources in the target system. The purpose of BBProf is not to construct WCET estimates, but rather to extract the importance of each page for the timing of an application. This information can then be used to perform more fine-grained cache management. WCET analysis should be performed after a given management strategy has been applied, and it thus represents an orthogonal goal.

In light of the discussion above, what sets the proposed BBProf methodology apart is its unique capability of extracting fine-grained statistics on the contribution of each memory page to the overall runtime of an application under analysis. It does so without leveraging any hardware-specific support, by requiring no source- or compiler-level manipulation, and by operating directly on the black-box binary of the target application. Moreover, we demonstrate that the profile acquired through our BBProf can be used to enact advanced cache management techniques beyond strict task-level or core-level cache partitioning.

3 Background

In this section, we summarize the inner workings of the system components utilized by our tool for unfamiliar readers. We first present a brief overview of multi-level set-associative caches. Next, we review the notion of cache coloring, before concluding with a conspectus on memory representation and management in modern computing architectures.

Multi-Level Set-Associative Caches: Modern computing architectures implement several levels of caching. The L1 cache resides closest to the CPU and is private to a specific core. A cache miss in L1 triggers a lookup in the level below (L2, in this instance). Some architectures restrict the L2 cache to specific cores, making them private similar to the L1. A miss in the L2 cache may trigger a lookup in the level below (L3 and subsequently, L4) if it exists or failing that, a memory lookup. We constrain our discussion here to a normative ARM-based cache, with private L1 caches and a globally shared, last-level L2 cache.

At all levels, caches adhere to a set-associative modality where a set-associative cache with associativity W consists of W identically-structured *ways*. Blocks of consecutive bytes are stored in *lines* referred to as *cache blocks*. The constant L_S denotes the number of bytes in a cache line, with most line sizes being 32 or 64 bytes. Memory addresses in the cache are divided into three groups of bits: the *offset, index*, and *tag* bits that affect the specifics of a cache lookup. Shared cache levels are physically indexed and physically tagged (PIPT), meaning all addresses used for cache lookups must be physical addresses.

Memory Abstractions in Operating Systems: Most modern operating systems employ a combination of hardware and software features to effectively encapsulate physical addresses into virtual addresses. Virtual addressing allows each process an exclusive view of the system's memory, alleviating problems such as memory fragmentation or the limited availability of physical memory. The OS maps virtual and physical addresses using *page tables*. When a process references a virtual address, the Memory Management Unit (MMU) performs a *page table walk* to locate the entry (PTE) — if any — that points to the corresponding physical memory page. If the walk is successful, the accessed virtual address is resolved into a physical address and the result of the translation is stored in the Translation Lookaside Buffer (TLB). If the address is not found, a page fault is triggered by the MMU and handled by the OS. If the access is legitimate, a new physical memory page is allocated and mapped to the process (*demand paging*); if it falls outside any valid range of virtual addresses, a segmentation fault (SIGSEGV) signal is delivered to the offending application.

Linux defines and manages the layout of legitimate contiguous regions of virtual memory by representing them as *virtual memory areas* or VMAs. VMAs consist of a range of start and end addresses, allowing for fine-grained control of virtual memory regions on a per-VMA basis. They have been a part of the Linux kernel since version 2.6 [10].

Cache Coloring: A major source of interference in multicore systems is LLC contention. One of the solutions to this problem is cache coloring, a purely software-based partitioning technique. With cache coloring, memory pages are assigned "colors" based on the cache sets they map to, which is determined by the value of the index bits. It is possible to allocate virtually-contiguous memory pages to physically discontiguous pages that have the same color. By doing this on a per-application or per-core basis, one can achieve strict cache partitioning, which is a well-known mitigation strategy for cache interference [14]. In multicore embedded SoCs that support two-stage address translations, the OS entirely manages the translation of the first layer address (user virtual address) into the intermediate physical address (IPA). The second stage of translation, however, is controlled by the hypervisor [29, 11] which maps IPAs to physical addresses. Hypervisor-level coloring is advantageous to transparently color entire guest OS's, as demonstrated in [27, 21, 15].

4 Design

In this section, we describe the main principles that comprise the design of the proposed BBProf. We describe the operational approach and functional components that allow it to carry out a fine-grained experiment-driven memory analysis of generic applications. While we advocate for the benefits of the proposed BBProf as a methodology for memory analysis, we have also carried out a proof-of-concept open-source implementation [13]. As we show in Section 7, the information extracted by our BBProf toolkit opens new avenues to perform fine-tuned management of shared memory resources.

In a nutshell, the main goal of the proposed BBProf toolkit can be formulated as follows. To consider a target application's memory footprint decomposed into its smallest manageable entities — individual memory pages. And with that, to produce a ranking that captures and quantifies how crucial is each page for the temporal behavior of the application. In other words, BBProf allows extracting the relative importance of memory pages towards the overall temporal behavior of a target application. Importantly, our BBProf should be able to handle applications of realistic complexity, while requiring minimum knowledge and understanding of the application itself — i.e., by largely treating the application as a black-box.

4.1 Core Principles

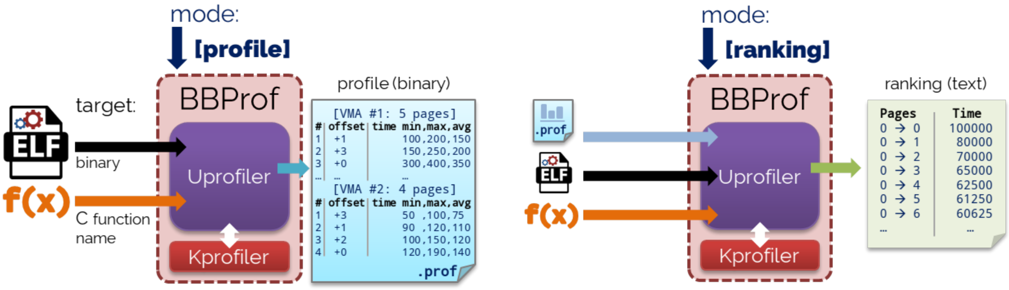
The core principles that have driven the design of the BBProf methodology can be summarized as follows.

Model-free Operation: Modern high-performance embedded systems are soaring in complexity. Additionally, manufacturers are often wary of providing exhaustive platform implementation details, as many of them constitute corporate intellectual property. Even if a formal micro-architectural model can be constructed, the high degree of complexity — in both software and hardware layers — can result in a state-space explosion even with simple workloads. It follows that, unfortunately, traditional static analysis methods might not be easily applicable to the considered class of embedded systems. In light of this, we aim to design a methodology that can be used in an arbitrarily complex system without the need for a micro-architectural model.

Platform Independence: A key design-time constraint we impose is for our BBProf methodology to be feasible regardless of the specific target platform. In other words, our BBProf should not rely on hardware support that exists only in a fraction of existing and future platforms. Instead, it should leverage widely available hardware features that are exposed by embedded and general-purpose platforms alike, and that are unlikely to be phased out in future generations.

Usable for Realistic, Unknown Workload: There exists a fundamental lack of practically viable toolkits that are industry-ready and capable of carrying out the memory analysis of complex applications in complex embedded platforms. The proposed BBProf aims that bridging such a gap with a solution that can be immediately adopted to better characterize the behavior of realistic applications. This implies that not only a minimal understanding of the target application should be required to perform profiling; but also that BBProf should be capable of handling widely used system-level features such as dynamically linked libraries and dynamic virtual memory allocation.

Linear-time Profiling: To be practically useful, we impose our BBProf methodology to be able to operate in linear time with respect to the memory footprint of the application under analysis. Because our strategy is centered around a runtime measurement-based approach, we deem as viable an analysis strategy with a linear time complexity that is



(a) Profiling mode workflow.

(b) Ranking mode workflow.

Figure 1 High-level workflow of BBProf in two of the main modes of operation.

impacted by (1) the runtime of the core logic of the application under analysis; and (2) the size of the memory footprint of the target application.

4.2 High-level BBProf Workflow

The proposed BBProf methodology pivots around the idea that it is possible to manipulate the memory allocation policy on a per-memory page basis. Thus, for a target application, it is possible to understand the importance of individual pages towards application timing by changing the allocation policy one page at a time. Albeit this idea is generic, the specific set of memory allocation policies depends on the type of analysis to be conducted. For the remainder of this paper we direct our focus to shared CPU cache analysis, which is a primary target of this work. Therefore, cacheability is the memory policy of choice to isolate the impact of a single memory page on the timing of an application.

Figure 1 provides a high-level overview of the logical workflow of BBProf in its two main modes of operation. In the *profile* mode described in greater detail in Section 4.3 and depicted in Figure 1a, the required inputs to BBProf are (1) the path to the binary of the ELF executable to be profiled; and (2) the name of the C function whose timing needs to be profiled. This function corresponds to the *observation segment* defined below. The full list of optional operational parameters are described in [13]. The output produced in this mode is a binary file[1] encoding the relative importance recorded for each page of each considered VMA. BBProf allows performing multiple profiling runs and will aggregate the result of all the runs into the same file keeping track of max, min, and average statistics on a per-page basis. BBProf includes a number of other analysis modes described in Section 4.4. These modes require a profile file previously obtained on the target application. For instance, Figure 1b depicts the high-level workflow of the *ranking* mode which produces a human-readable output describing the runtime of the target function as an increasing number of most important pages are made cacheable.

We base our analysis on the presence of a single aforementioned observation segment, which represents a segment of logic whose temporal behavior is of interest. Although the observation segment can be extended to cover the entire application's logic, in practice this is often not the case. Realistic applications are typically characterized into three main phases: (1) an initialization phase where parameters and inputs are parsed and pre-processed; (2) the

[1] The binary profile can be translated into human-readable format using the `-t` parameter as described in [13].

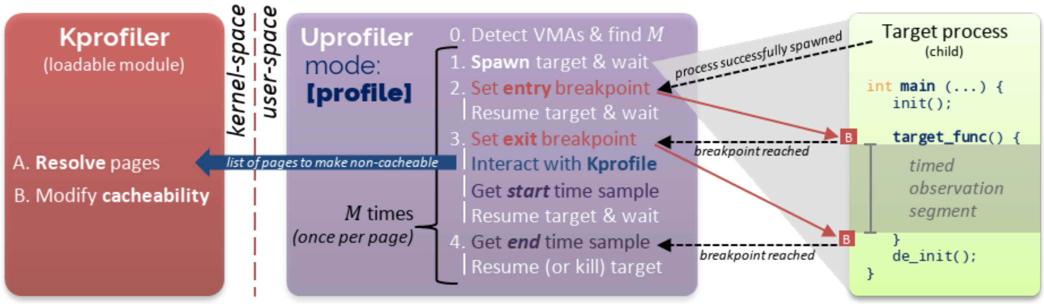


Figure 2 Logical interplay between modules of BBProf in *profile* mode.

main computational payload, which might be executed multiple times in a periodic fashion; and (3) a teardown phase where any acquired resource is released. The observation segment corresponds to the main computational payload of the target application. For the sake of simplicity, we assume that such a phase is encapsulated into a single function called the `target_func`, and hence that the target application has a structure similar to what depicted in the right-hand side of Figure 2. Any initialization and de-initialization logic is excluded from the analysis.

4.3 Profiling Strategy

When operating in profiling mode, the adopted strategy is visualized in Figure 2 and described in the following. (1) Perform a first run of the target application to identify its virtual memory layout; (2) re-execute the target application as many times as the number of memory pages M that comprise its memory footprint; (3) at each re-execution and before the invocation of the `target_func`, switch memory allocation policy for all the pages except the one under analysis; and (4) collect the impact of the selected policy over the execution time of the `target_func`. It is crucial that the profiling of an application is conducted in isolation, i.e., with the lowest possible amount of noise in the target system.

For instance, consider an application whose memory footprint is comprised of 4 pages and assume that its runtime when all the pages are marked as non-cacheable is some time T_{nc}. BBProf first detects the footprint of the application. Next, it performs 4 iterations. In the first iteration, only the first page is marked as cacheable, while all the others are marked as non-cacheable. Then, it measures the runtime of the `target_func` which will be of the form $(T_{nc} - x_1)$, with x_1 being the performance gain that arises from having the first page in cache. We then repeat the same steps for the remaining three pages to extract the terms x_2, x_3, and x_4 in the same way.

To accomplish the strategy outlined above, our methodology relies on the definition of two components, as also depicted in Figure 1: a *user-space driver* and a *kernel-space driver*, which we refer to as UProfiler and KProfiler, respectively. Intuitively, the UProfiler is responsible for launching and collecting data about the temporal behavior of the target application, while the KProfiler is used to enforce the selected memory allocation policy. The main key design principles for the two components are reviewed in the following.

4.3.1 User-Space Driver (UProfiler)

The design of the UProfiler component shares a number of similarities with a typical debugger. Indeed, it operates by taking in two pieces of information — which are the only ones strictly required to launch profiling. These are (1) the location of the executable binary (and any parameters it requires) of the target application; and (2) the name of the target function that corresponds to the observation segment.

First, UProfiler parses the provided binary executable to translate the name of the function into the address that corresponds to the first instruction of the target function — i.e., the beginning of the observation segment. With this information at hand, UProfiler can launch the target application and set a breakpoint, called the *entry breakpoint* right at the beginning of its computational payload (Figure 2, step 1). As soon as the entry breakpoint is reached, UProfiler pauses the target application and performs a sequence of preparatory actions, called the *entry sequence*. The actions performed in the entry sequence depend on the type of analysis being carried out.

As part of the entry sequence, UProfiler always detects the end of the observation segment. This is done by inspecting the return address of the target function. With this information, an *exit breakpoint* is installed by UProfiler (Figure 2, step 2). Before resuming the execution of the target application, UProfiler removes the entry breakpoint and snapshots the current `start` timestamp (Figure 2, step 3). In a similar way, as soon as the exit breakpoint is reached, UProfiler immediately snapshots the current `end` timestamp (Figure 2, step 4); removes the exit breakpoint, and performs a variable sequence of actions — the *exit sequence*.

During the very first run of the target application (iteration 0), UProfiler detects its layout and the number of memory pages M that comprise its footprint. This information is collected during the entry sequence and double-checked during the exit sequence. Additional implementation-specific details about this step are provided in Section 5.

In the generic profiling iteration i, the entry sequence is used by UProfiler to prepare a descriptor that determines the memory policy to be applied to each of the pages subject to profiling. Given the current focus on cache analysis, the descriptor prepared at profiling iteration i instructs the KProfiler to turn all the considered pages non-cacheable except for the i-th page. In the exit sequence, the difference between `start` and `end` timestamp is recorded and associated to page i.

Here, the use of timestamps represents the preferred metric for two main reasons. First, it allows UProfiler to be a valid methodology regardless of the target platform, since time sampling primitives are commonplace in (modern) hardware platforms. Second, it allows UProfiler to directly correlate the impact of the selected memory policy on the timing of the observation segment. Nonetheless, UProfiler can be easily extended to capture additional platform-specific performance metrics such as number of cache references, hits, misses, number of retired instructions, instructions-per-cycles, and so on.

4.3.2 Kernel-side Driver (KProfiler)

The KProfiler encapsulates all the logic that requires elevated kernel-level privileges to manipulate the properties of the memory pages mapped to the target application.

Following the proposed design, the KProfiler defines a communication interface exposed to the UProfiler (Figure 2, step 3). As needed — usually during the entry sequence — the interface is used to pass a descriptor with the list of changes to be applied to the target memory pages. Because absolute memory addresses change from run to run, UProfiler and KProfiler use relative addressing to uniquely identify memory pages across runs. Pages are grouped by the memory policy modification to be carried out over them.

It is responsibility of the KProfiler module to leverage appropriate kernel-level APIs to apply the requested memory policy modifications for the target pages. So far we have only discussed the most basic operation mode of the proposed BBProf. In this case, the descriptor passed by the UProfiler always follows the same structure. Only one page is selected to be kept cacheable, while all the others are requested to be made uncacheable.

4.4 Additional Operational Modes

So far we have described the design of UProfiler and KProfiler with respect to the main operational mode, which is page-level cache profiling. Our current design includes two additional modes that are briefly described in the following.

Page Ranking Analysis: Once per-page statistics have been extracted, it is possible to globally rank all the memory pages that comprise an application's footprint. Intuitively, those pages that led to the best time improvements will be ranked as *more important* towards the temporal behavior of our target. The page ranking analysis allows to understand the *cumulative* benefit of selecting the top-ranked k pages to be cacheable, where $0 \leq k \leq M$. Notably, the case $k = M$ corresponds to the default case where all the memory pages are considered cacheable. Expectedly, as we increase k, the observed runtime of the observed segment will generally decrease. Importantly, however, if a threshold of $k^* < M$ is found where the resulting runtime already approaches the case $k = M$, then k^* corresponds to the working-set size (WSS) of the target application.

Page Migration Analysis: A final useful operation provided in our design is the possibility of changing the physical location of a group of pages based on the information collected during profiling and ranking. For instance, consider a platform that includes a block of scratchpad memory. First profiling and ranking is performed to identify the pages that comprise the working-set of the target application. Next, our BBProf toolkit can be used to test what-if scenarios where all or a part of this group of pages is migrated to scratchpad memory. We will demonstrate two concrete use-cases where page migration can be used to efficiently mitigate inter-core cache interference.

5 Implementation

We hereby review the main details concerning a proof-of-concept Linux implementation of the proposed BBProf toolkit.

5.1 UProfiler Implementation

As we mentioned in Section 4, the UProfiler component is designed to act akin to a debugger. For this purpose, it leverages the `ptrace` family of system calls to manipulate the flow of a child process. Indeed, launch a new run of the target application, UProfiler performs the following sequence: (1) a `fork` system call to spawn a new child process, (2) a `ptrace(PTRACE_TRACEME)` in the spawned child allowing the parent to trace the child's execution, (3) an `exec` system call to execute the target application under tracing.

The `ptrace` system call represents a standard Linux interface. Albeit it is Linux-specific, it is possible to achieve a similar behavior even in a bare-metal system or RTOS by relying on basic debugging features. Indeed, the only features used by UProfiler are (1) the ability to set/remove breakpoints, and (2) the ability to read the content of CPU registers. These capabilities are available even in simple microcontrollers.

Breakpoint Handling: To set a breakpoint in an architecture-independent way via the `ptrace` interface, one can replace (`PTRACE_POKETEXT`) the instruction at the desired breakpoint address with any illegal opcode. This way, when the execution of the tracee reaches the modified instruction, the process is paused by a `SIGILL` POSIX signal and a

SIGCHLD signal is delivered to the parent process — i.e., to our UProfiler. Before setting the breakpoint, UProfiler records the value of the instruction being replaced (PTRACE_PEEKTEXT) so that it can be restored once the breakpoint is reached. As soon as the breakpoint is hit, UProfiler records the value of the tracee's program-counter (PC) register. To allow the tracee to resume from the breakpoint, UProfiler (1) restores the original instruction at the breakpoint address and (2) rewinds the PC of the tracee to the recorded address. Accessing the tracee's CPU registers can be done via a combination of PTRACE_GETREGS/PTRACE_SETREGS operations[2].

As discussed in Section 4, UProfiler only sets two breakpoints. The entry breakpoint is set upon launching the target application and at the first instruction of the target function. The exit breakpoint is installed at the address to which the target function is set to return. To find the address of the entry breakpoint, UProfiler accepts as a command-line parameter the name of the target function whose body corresponds to the observation segment. It then uses the LibELF[3] library to translate the provided function name into the corresponding instruction address by performing a lookup in the target ELF's symbols table (SHT_SYMTAB). The address of the exit breakpoint is only known once the tracee hits the entry breakpoints. In ARM32 and ARM64, it is enough to read the content of the link register (LR) to retrieve the return address of the target function.

Layout Detection and Enforcement: In a generic POSIX-compliant application, there is a number of system calls that can dynamically modify the memory layout of an application. Most notably, sbrk is internally used by the libc to implement functions that perform dynamic memory (de)allocation, such as malloc and free. Calling the sbrk can affect the size of the heap virtual memory area (VMA). Similarly, the mmap and unmap system calls can cause the addition, deletion, or modification of VMAs in the tracee's layout. Importantly, the libc uses mmap instead of performing a heap extension when applications allocate large buffers. For the final output of our BBProf to be valid, it is crucial that no memory layout changes occur during the execution of the observation segment. This is not a concern with applications written for embedded/safety-critical systems where memory is always statically allocated. Nonetheless, UProfiler includes logic to enforce a deterministic memory layout even on applications that use dynamic memory allocation primitives.

To achieve that, when the tracee is spawned for the first time, UProfiler runs the tracee a first time and records the peak amount (VmPeak) of data that was used during the target function. Once the maximum amount of memory required by the observation segment is known, all the subsequent runs of the target application are performed by setting two environmental variables that modify the behavior of the libc memory allocation routines. These are (1) the MALLOC_TOP_PAD_ and (2) the MALLOC_MMAP_MAX_ variables. The former allows setting an initial size for the heap and is set to the peak memory size detected by UProfiler in the first run. The latter is set to 0 to disable the use of mmap to handle dynamic memory allocations.

All the subsequent runs of the target application can be used to perform profiling. In the first of such runs, UProfiler further detects the actual memory layout that results from setting the aforementioned environmental variables. It does so by querying the /proc/PID/maps interface as soon as the entry breakpoint is reached. Additional launch parameters are

---

[2] Note: this is true for many platforms, including x86, x86_64 and ARM32. Equivalent operations can be carried out in ARM64 through PTRACE_GETREGSET and PTRACE_SETREGSET.

[3] LibELF is part of the elfutils open-source project which is a toolkit to read, create and modify Executable and Linkable Format (ELF) binaries.

accepted by UProfiler to include/exclude certain types of VMAs in the profiling. For instance, in order to make profiling faster, one might want to exclude VMAs that belong to shared libraries and that are not used during the observation segment.

Single-page Profiling: Once UProfiler has computed the number of pages M in the target VMAs, the single-page profiling phase can be initiated. Of course, the M pages can be distributed across multiple VMAs (e.g. `text`, `heap`, `stack`). Moreover, their absolute address will change from run to run due to address space layout randomization (ASLR). To operate even with ASLR in place, UProfiler uses a run-independent relative encoding to express the coordinate of memory pages. Specifically, we use two indices to identify each page: (1) the index v of the VMA that contains the page; and (2) the offset o of the page from the beginning of the VMA.

To profile a generic page $i \in \{1, \ldots, M\}$ with coordinates $\langle v, o \rangle$, the UProfiler prepares a descriptor to instruct the KProfiler module to modify the cacheability of the pages in the target VMAs. In profiling mode, this descriptor contains the list of all the VMAs under analysis. For each of them, a list of pages whose cacheability attributes need to be modified is included, with an opcode field that determines how the cacheability attributes should be altered. In this case, the cacheability of page i is unchanged, but that of all the other pages is the target VMAs is set to become non-cacheable. The descriptor prepared as mentioned above is then passed to KProfiler to apply the necessary changes once the entry breakpoint is reached. The target application is resumed only once all the pending changes are effective. Note that any timestamp acquisition is performed after the cacheability changes have been applied, so that the overhead required to switch the cacheability attributes is excluded from the time measurements.

Time Measurements: Albeit extensible, the current use of the BBProf toolkit is to analyze the relative importance of individual memory pages toward the overall temporal behavior of the observation segment. The most direct and platform-independent way to extract this information is by acquiring timing samples of the target function as we vary which page is allowed to be allocated in cache. In order to be as precise as possible, UProfiler directly reads CPU cycle counters instead of relying on system primitives.

Time measurements are acquired right before resuming the application from the entry breakpoint and right after it reaches the exit breakpoint. Moreover, since timestamps can be affected by random system noise, UProfiler allows specifying an arbitrary number of samples to be collected for the same profiled page. System noise originates from workload on other cores, interrupt handlers, non-deterministic hardware behavior, and inaccuracy of time sampling instructions. Various mitigations strategies can be adopted to reduce the magnitude of system noise, such as turning off other cores and disabling peripherals. The only mitigation strategy used by BBProf is running UProfiler and the target process with the `SCHED_FIFO` Linux policy and with a high real-time priority. As we evaluate in Section 7.2, the observed degree of noise was negligible and did not impact the validity of our profiles. The final profile stores, for each page, the maximum, minimum, and average runtime of the observed segment across all the acquired samples. Note that with this infrastructure in place, it is straightforward to extend UProfiler to collect additional metrics such as hardware counters for micro-architectural events — e.g. cache references, misses, hits, bus accesses, to name a few. This can be done in a platform-agnostic fashion by leveraging the `perf` infrastructure [12].

Page Ranking and Migration: The implementation of the other two modes of operation is similar to what has been discussed above, hence much of the details are omitted. To perform page ranking and migration, it is assumed that a profile has been previously acquired for the target application. The pages in the profile are then arranged in a sorted set in descending order of their impact on the timing of the target application. Examples of the output produced by a ranking experiment are provided in Figure 8.

In the ranking phase, UProfiler performs M runs where in run k, the top k pages in the sorted set are requested to be kept cacheable by the KProfiler, while all the remaining pages in the set are turned non-cacheable. The timing of the M runs is collected and stored for later analysis.

In a similar way, a page migration experiment requires a pre-acquired profile. The M pages in the target VMAs are sorted according to the same criterion described above. In this case, however, a single run is performed where the UProfiler instructs the KProfiler module to migrate the top k pages in the sorted set to a new location in physical memory. The value of k represents a parameter supplied by the user. The destination of the migration is determined by the KProfiler, as we discuss below. The support to conduct page migration directly from the profiler allows quick testing of what-if scenarios for the allocation of important pages. As part of our future work, we plan to directly modify the way applications are launched to take advantage of profiling information without the need to go through the profiler.

5.2 KProfiler Implementation

The KProfiler component is implemented as a Linux kernel module. Our current implementation targets Linux 5.4. At startup, a communication channel with the UProfiler is created in the form of a file in the `proc` pseudo-file system. Whenever the UProfiler needs to trigger a kernel-side operation, the `write` system call is used to pass the content of the aforementioned operation descriptor. The descriptor also contains the PID of the tracee that will be targeted for the current operation. A combination of `find_get_pid` and `get_pid_task` kernel APIs is used to retrieve the descriptor of the tracee's process given the provided PID. Moreover, the descriptor contains redundant information about the structure of the memory layout of the tracee as detected by UProfiler. This is used to perform a sanity-check in the KProfiler and ensure that the desired operations are performed on the right VMAs and pages.

Cacheability Modification: For the profiling and ranking phases in which only the cacheability of the target page(s) is changed, no changes to the source code of the Linux kernel are required.

For each VMA in the passed descriptor, the KProfiler retrieves the corresponding `vm_area_struct` descriptor by scanning the kernel-maintained linked list of tracee's VMAs. It then ensures that any page that will be affected by the current operation is present in physical memory. This is done by *faulting-in* the target pages that can be achieved via the kernel API rev`get_user_pages_remote` and with flags `FOLL_POPULATE`, `FOLL_TOUCH` and `FOLL_MLOCK`. Next, the kernel API `apply_to_page_range` is used to invoke a custom function for each page on which a change in cacheability attributes needs to be carried out. Such a function already invokes our custom routine with a pointer to the Page Table Entry (PTE) that needs to be manipulated to change the cacheability attributes of the page.

Given a page that is set to be made non-cacheable, the following steps are performed. First, a new PTE is prepared to mirror the same exact value of the existing PTE, but where the page attributes have been switched to encode for normal, non-cacheable memory. Next, we clean and invalidate data and instruction caches to make sure that any dirty line is written back to main memory. Then, we install the newly created PTE to replace the previous entry. Finally, we invalidate any TLB entry (if any) for the current page on all the online CPUs.

Page Migration: Being able to support page migration requires some changes to the kernel sources[4]. A total of around 200 lines have been modified to implement the required changes. Specifically, we have generalized the existing support for the migration of physical memory pages across NUMA nodes used to implement the `move_pages` system call. We have introduced a new exported kernel API with the following prototype:

```
int move_pages_to_pvtpool(struct mm_struct *mm, unsigned long nr_pages,
                          unsigned long * vaddrs, new_page_t get_new_page,
                          unsigned long private);
```

Here `mm` is the virtual address space descriptor of the process targeted for page migration, `nr_pages` is the number of pages to be migrated, `vaddrs` is an array of `nr_pages` virtual addresses of pages to be migrated, `get_new_page` is a function pointer used by the internal routines to allocate destination pages, and `private` is a parameter to be passed to the allocation function.

At load time, the KProfiler module internally maps an area of memory reserved at boot for page migration. The reservation is performed via a modified Device Tree Blob (DTB). Here we use the `reserved-memory` attribute [5] to exclude a given range of physical addresses from the default Linux allocator — the Buddy System. We do not mark this region with the `no-map` attribute to allow the kernel to initialize the necessary page descriptors to correctly map kernel virtual addresses and physical addresses in the reserved region.

If a valid reservation is found by the KProfiler at load time, the module uses a combination of `memremap` and `gen_pool_create` kernel APIs to instantiate a new general-purpose memory allocator over the reserved memory region [6]. The former produces a valid kernel virtual address that can be used to access the reserved memory region, while the latter enables the allocation of new pages from the region.

With our custom allocator in place, whenever UProfiler requests the migration of a set of pages, a set of initial steps similar to those required to change the cacheability attributes is performed. But instead of manipulating the cacheability attribute of the exiting pages, a list of pages to be migrated is compiled and the newly introduced `move_pages_to_pvtpool` API is invoked. When doing so, a wrapper to a `gen_pool_alloc` call is passed as the `get_new_page` function pointer to allow internal book-keeping.

We describe in Section 7.4 how profile-driven page migration can be used to enact advanced techniques to manage inter-core interference in the shared cache. Nonetheless, the implications of profile-driven page migration are deeper than what presented in Section 7.4. Indeed, this support allows defining a distinct memory pool for each heterogeneous memory component available in the system, e.g. scratchpad memory, in-FPGA block RAM, non-volatile memory, reduced-latency DRAM blocks (RL-DRAM) [16], to name a few. By leveraging profiling information, one can then decide which pages need to be mapped to the various memory resources.

6 System Instantiation

In this section, we review the full-system setup that was carried out to evaluate the potential of the proposed BBProf approach and proof-of-concept implementation.We have deployed

[4] The modified kernel sources are available at `https://github.com/rntmancuso/linux-xlnx-prof`.
[5] See `https://www.kernel.org/doc/Documentation/devicetree/bindings/reserved-memory/reserved-memory.txt`.
[6] See `https://www.kernel.org/doc/html/v5.4/core-api/genalloc.html`.

the implemented UProfiler and KProfiler modules on an `ARM64` platform that we also use for all our experiments. Specifically, we use a Xilinx-ZCU102 development platform featuring a Zynq UltraScale+ XCZU9EG MPSoC [36] with a quad-core ARM Cortex-A53 [4] 64-bit CPU operating at 1.5 GHz and implementing the ARMv8-A [2] architecture profile. The L1 cache consists of a split cache with a 32 KB 2-way instruction (I) cache plus a 32 KB 4-way data cache. The L2, which is also the last-level cache (LLC) is unified and 1 MB in size; it has associativity 16, and it is shared among all the A53 cores. The cache line size is 64 bytes for both L1 and L2.

Profiling and ranking analysis can be carried out directly under Linux. Conversely, to evaluate the ability to enact advanced memory management via profile-driven page migration, we additionally deploy a thin partitioning hypervisor, namely Jailhouse [3]. Jailhouse is used to perform cache coloring [38, 19, 27, 21] in a way that remains transparent to the Linux environment where we conduct our experiments. Our goal is to conduct a series of experiments centered around the problem of shared cache management. To achieve this, we have reproduced the setup described in [21] on the ZCU102 system, where dynamic re-coloring of the Linux environment is available. We use coloring in two ways. First, in a traditional way to statically restrict the applications running in the Linux environment to only a subset of the available colors — we vary this amount from two to 15, with 16 being the maximum value and corresponding to no partitioning. In this case, Linux is restricted to use only one CPU. Moreover, when strict coloring is used, interfering workload (INTERF) consists of bare-metal memory-intensive synthetic applications deployed on all the other cores as stand-alone virtual machines (VM).

We then use Jailhouse and page coloring to illustrate a new technique enabled by the profiler to mitigate the problem of shared cache interference. The setup, illustrated in Figure 4, essentially defines two contiguous ranges of intermediate physical addresses (IPA). The first corresponds to all the memory that Linux uses for legacy memory allocations through the Buddy System and is mapped by Jailhouse to $12/16 = 3/4$ of the available colors. The second IPA range is mapped to pages with the remaining $4/16 = 1/4$ of the available colors. The latter is then used by the KProfiler to instantiate a privately managed allocation pool. It follows that pages can be allocated in the pool only through explicit profiler-driven page migration. We refer to this setup with the PVT+SH short-hand notation. Note also that this setup provides page-level granularity over memory allocated in the private cache pool. This sets this work apart from the large literature on colored page allocators proposed in the past that assign colors at the process or core granularity [18, 20, 19, 23].

In terms of workload, apart from the aforementioned INTERF workload, an equivalent synthetic memory-intensive application, namely `bandwidth` from the IsolBench suite[7], is used to generate cache contention when no other VMs are active in the system and Linux is used in SMP mode on all the cores. For the purposes of building confidence in the ability of the profiler to characterize the importance of memory pages, we use the STAIRCASE synthetic benchmark described more in detail in Section 7.2. For our observed realistic workload, we used the San Diego Vision Benchmark (SD-VBS) suite [35]. While we conducted all our experiments on all the benchmarks, due to space constraints we only include a subset of the results that capture the more interesting cases. We also limit our discussion to the input sizes SQCIF, QCIF, CIF, and VGA. We exclude the FULLHD sizes as the runtime of the benchmarks on the target platform is excessively high. As we mentioned in Section 5, the observed system noise was quite negligible which resulted in the timing of the profiled

[7] See `https://github.com/CSL-KU/IsolBench/blob/master/bench/bandwidth.c`.

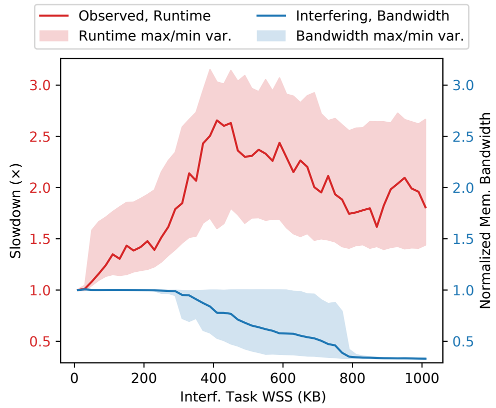


Figure 3 Interference as a function of WSS.

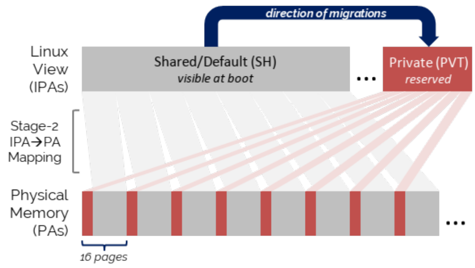


Figure 4 Overview of PVT+SH setup.

applications to be remarkably deterministic. Thus, five independent runs were sufficient to acquire each profile. For production systems with worse signal-to-noise ratios, we expect that a much larger number of runs might be needed to construct meaningful profiles.

7 Evaluation

In this section, we describe the evaluation that we have carried out on the system setup described in the previous section. We focus our attention on four main aspects. First, in Section 7.1 we evaluate the amount of shared cache contention that can be suffered by applications in this platform and understand the ability of strict cache coloring to mitigate such interference. Next, we show in Section 7.2 that our proof-of-concept BBProf implementation is capable of extracting useful profiling information for the considered synthetic and real-world applications. Third, we discuss how profile-driven migration can be used efficiently to solve the problem of contention-induced instruction stall in Section 7.3. Finally, we evaluate in Section 7.4 how profile-driven page migration can be used to controllably mitigate shared cache contention in real-world applications.

7.1 Interference and Mitigation via Strict Partitioning

In the experiments presented in this section, we focus on cache contention. Generating cache contention for an application under analysis is done by deploying a set of interfering synthetic memory-intensive applications on all the other cores. In order to set the WSS of the interfering workload with the goal of maximizing contention, we have conducted the experiment depicted in Figure 3. In this experiment, the application under analysis is MSER from the SD-VBS suite with input size SQCIF. Three interfering applications deployed on the remaining cores continuously perform cache-allocate store operations over a buffer of increasing size (x-axis). We plot on the left y-axis (red) the runtime normalized to the case in which MSER runs in isolation (*solo* case) in the system. We display the memory bandwidth observed by the interfering workload on the right y-axis (blue). A clear trend emerges that highlights how the cache interference is maximized (both in average and maximum terms) when each interfering application accesses a buffer of around 420 KB, i.e. access in a total of about 1.23 MB.

In light of the results highlighted above, we have set our interfering tasks to have a WSS of 420 KB. With this in mind, we want to understand how well strict coloring is able to mitigate cache interference. We have conducted a study where all the strict coloring configurations described in Section 6 are explored for all of our SD-VBS benchmarks and

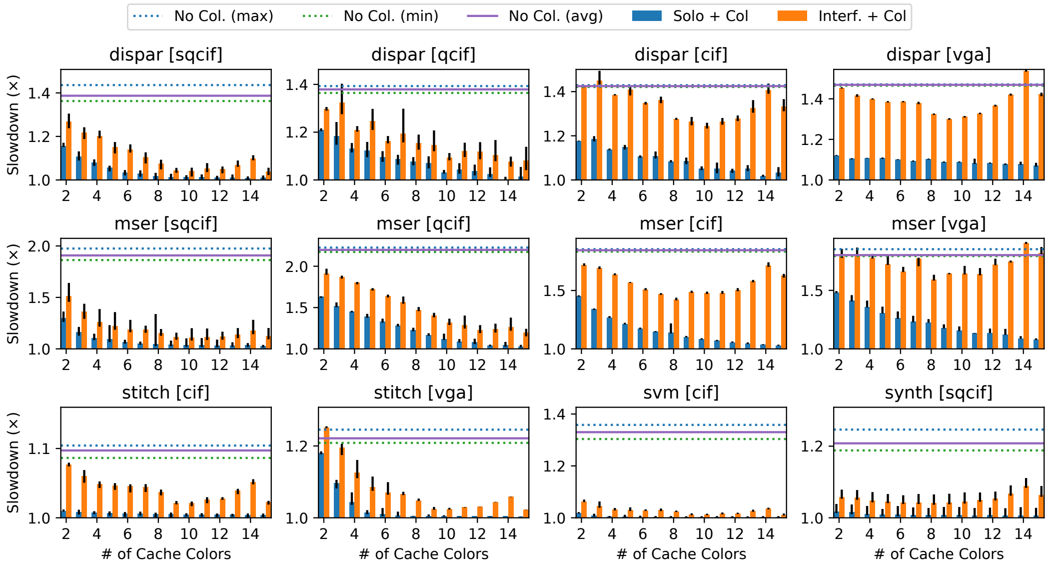


Figure 5 Performance of SD-VBS benchmarks under strict partitioning with (orange) and without (blue) cache contention.

considered input sizes. The most interesting nine cases are presented in Figure 5. In all the sub-plots, the vertical bars represent the slowdown of the application under analysis when no cache partitioning is performed. The blue bars (resp., orange) report the runtime of the application under analysis in the solo case (resp., under interference). It emerges that partitioning leads to significant improvements in certain circumstances, especially for workload with L2-sensitive footprint such as DISPARITY and MSER with input sizes QCIF and SQCIF, and for STITCH with input size CIF. However, the ability to mitigate cache contention with coloring alone is limited in some cases. This is due to contention over memory bandwidth which exacerbates as larger partitions are given to large-footprint applications — see DISPARITY and MSER with input sizes CIF and VGA. Indeed, the stress over the main memory subsystem placed by the interfering workload increases as it is confined to a smaller cache partition. Traditionally, bandwidth throttling techniques are used to solve this problem, such as MemGuard [39, 33].

But an important takeaway from this study is that strict partitioning is just too rigid to (1) be able to efficiently mitigate cache contention for a wide variety of tasks deployed on the same core. And (2) that over-throttling of the interfering workload might be required to compensate for the lack of flexibility in coloring-based cache partitioning. Conversely, as shown in the following, the proposed BBProf toolkit can be used to strike a balance between strict partitioning and unregulated interference.

7.2 Profiling of Staircase and SD-VBS benchmarks

The first step toward profile-driven cache management is to use the proposed BBProf toolkit to acquire the page-level profile about the applications to be managed. As a first step to build confidence on the correctness of BBProf, we have designed the STAIRCASE benchmark[8]

[8] The code of the STAIRCASE benchmark is available in the project repository [13].

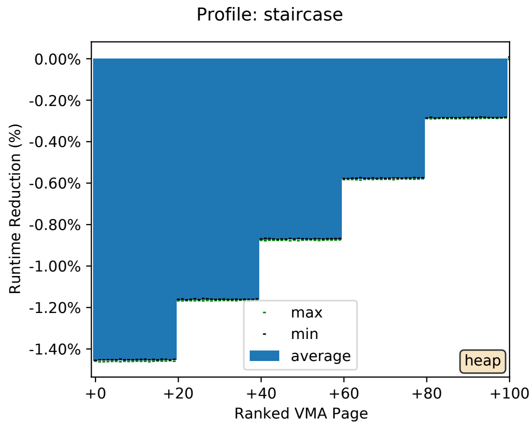


Figure 6 Profile of STAIRCASE benchmark.

to exhibit a well-recognizable behavior in terms of memory accesses that can serve as the *ground truth* on the extracted profile. Specifically, the benchmark allocates a buffer of 100 heap memory pages. It then performs a total of 1000 iterations reading over the buffer. In the first 200 iterations, the buffer is read entirely; in the next 200 iterations, the first 20 pages are skipped; after 200 additional iterations, the first 40 pages are skipped and so on. The result is that the second group of 20 pages is accessed 2× more than those at the beginning of the buffer. The third 20-pages group 3× more, and so on. Thus if we were to plot the importance of each page from beginning to end, the resulting plot would resemble a *staircase*, hence the name. Figure 6 provides a visualization of the extracted profile focused on the heap VMA. In the figure, the x-axis represents the index of the page under profiling. The blue bars from the top of the plot visualize by how much (in percentage) the runtime of the benchmark is reduced when each page is kept cacheable while all the others are not. A taller bar signifies a page with relatively higher importance for the temporal behavior of the application under analysis. For all the bars, the normalization baseline is always taken as the application's runtime when none of the pages in the target VMAs is made cacheable. The pages are sorted based on their importance rather than their offset in the VMA. Because of the by-importance sorting, the most-accessed pages appear to the left-hand side of the plot, with the recognizable staircase characterization having been reconstructed by BBProf. One can also note that the gap between min and max in each profile sample is quite small, thus leading to the conclusion that the overall measurement noise is negligible.

Next, we have acquired a profile for all the benchmarks in the SD-VBS suite, one for each of the considered input sizes. Due to space constraints we only visualize the three most representative profiles, namely those for DISPARITY, MSER with input size QCIF, and for SVM with input size CIF. These are displayed in Figure 7, where we limit the plots only to the `heap` and `stack` VMAs. The style of the sub-plots in Figure 7 is identical to that of Figure 6, with the only difference that the bars of stack pages are color-coded in red and that we have omitted max/min error bars to avoid over-plotting. From the figure it emerges that in all the cases there exists a small group (1-3 pages) of heap pages that has a large impact on the runtime of the application. From left to right, these alone cause a reduction of around 1.8%, 69%, and 7.9% when kept cacheable. Moreover, the temporal behavior of MSER and STITCH is more heavily impacted by `stack` pages; the DISPARITY benchmark has a core set of around 65 `heap` pages that comprise its working-set. Taken individually, the presence in cache of each of these pages alone contributes to a runtime reduction between 1.25% and 1.5%.

To further understand the relationship between important pages and overall application

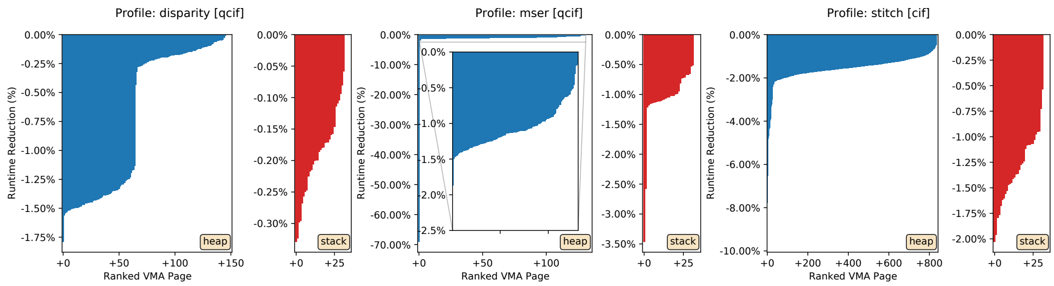


Figure 7 Profile of DISPARITY (left), MSER (center), and STITCH (right) — heap, stack pages only.

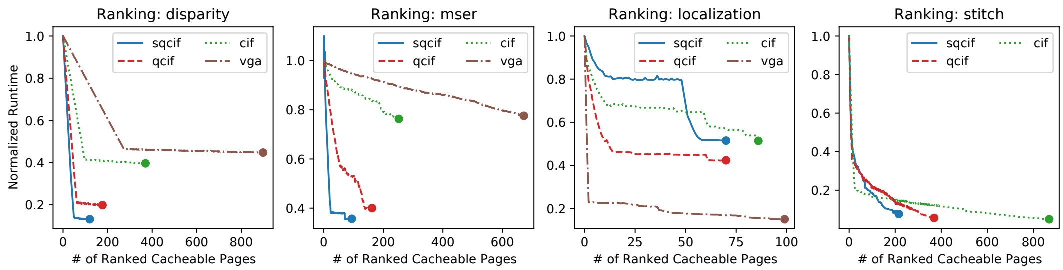


Figure 8 From left to right, ranking analysis of DISPARITY, MSER, LOCALIZATION, and STITCH.

runtime, we conduct a ranking analysis (see Section 4.4) given the profiles obtained at the previous step. In Figure 8 we depict the result of the ranking analysis conducted on DISPARITY, MSER, LOCALIZATION, and STITCH. In each subplot, the x-axis reports the number of pages, sorted in order of importance, that are made cacheable. The y-axis reports the resulting normalized runtime of the application under analysis. The normalization baseline is the runtime when only the most important page is made cacheable. A stark contrast emerged in the behavior of the considered applications. Specifically, DISPARITY features a block of pages with comparable importance that produces a constant slope in the runtime reduction as more pages are made cacheable. It is also possible to appreciate how the WSS size increases as the input size goes from SQCIF to VGA. Conversely, the WSS of MSER is concentrated in a very small set of pages for the SQCIF and QCIF case, and increases rapidly for input sizes CIF and VGA. Next, LOCALIZATION is characterized by *quantized* temporal improvements unlocked only when a certain threshold of pages is allocated in cache. Finally, STITCH appears to be relatively insensitive to caching as long as a core set of about 10 pages is allocated.

Once the profile has been acquired, it is important to understand if the set of memory pages deemed *important* remains the same as when the content of the input images changes while their size remains the same. In the general case, this might not be true while for some applications the profile might transcend the specific data input. We hereby conduct a sample evaluation to understand in which category the considered benchmarks fall. Note that this is not meant to represent an exhaustive evaluation. For this experiment, we consider the profiles acquired on the default ("def") input images provided with the SD-VBS suite. In terms of benchmarks, we limit ourselves to DISPARITY, MSER, TRACKING, and STITCH. Compared to Figure 8, we have replaced LOCALIZATION with TRACKING because the latter uses images as input while the former takes as input a text file with an unknown format. The selected input size is VGA for DISPARITY, MSER, and TRACKING and CIF for STITCH

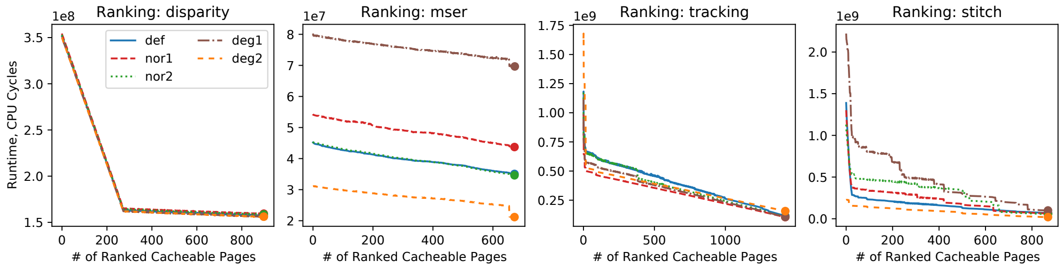


Figure 9 From left to right, ranking analysis of DISPARITY, MSER, TRACKING, and STITCH with profiles acquired under "def" and varying input images.

because the latter runs for too long over the VGA input size. For each benchmark, we have produced four additional input images. The first two called "nor1" and "nor2" are meaningful (*normal*) scenes, while the last two, namely "deg1" and "deg2" are scenes that correspond to corner (*degenerative*) cases. Specifically, "deg1" corresponds to random noise while "deg2" to a solid-color frame. Due to space contraints, we refer the reader to the project repository [13] for the full list of images used in this experiment.

Figure 9 provides the same type of analysis used to construct Figure 8. The key difference here is that for each of the considered benchmarks we construct the displayed ranking curves using the profile originally acquired with the "def" input images. To more clearly appreciate the difference in absolute runtimes as we vary the images supplied in input, the runtimes are not normalized and are instead expressed in CPU cycles. Among the four considered benchmarks, the runtime of MSER is the most heavily affected by the content of the input data. Nonetheless, the general trend in terms of runtime reduction as an increasing number of ranked pages is made cacheable is consistent across experiments. In the DISPARITY case, all the curves remain quite consistent. This suggests that the benchmark remains quite insensitive to the input image and that the profile acquired with the default input captures well the relative importance of individual memory pages regardless of the supplied input images. The TRACKING case is quite similar to the DISPARITY case, with the trend of the curve remaining consistent across experiments. Conversely, STITCH shows visible variations in the relative importance of memory pages, especially when comparing between the "deg1" and "deg2" cases. In this case, the profile obtained with the "def" input images does not generalize well. We can conclude that what captured by BBProf remains mostly accurate for three out of the four benchmarks considered in this experiment. The fourth case (STITCH) displays important dependencies between input images and memory usage, in which case the profile constructed by BBProf does not generalize.

7.3 Mitigation of Contention-induced Instruction Stall

We hereby want to bring to the attention of the community a previously understudied problem, namely the problem of *contention-induced instruction stall*, or C2IS, for short. We also demonstrate that profile-driven page migration represents an effective strategy to mitigate the problem.

In a nutshell, C2IS can occur in platforms with small L1 caches and shared, unified L2/LLC caches. The problem manifests itself when a process operates in a periodic fashion over a large block of instructions (e.g. a long function) that spans more pages than the size of the L1 instruction cache. For instance, in the target ZCU102 platform, the size of the L1 cache can hold up to eight pages. When such a threshold is crossed, instruction pages

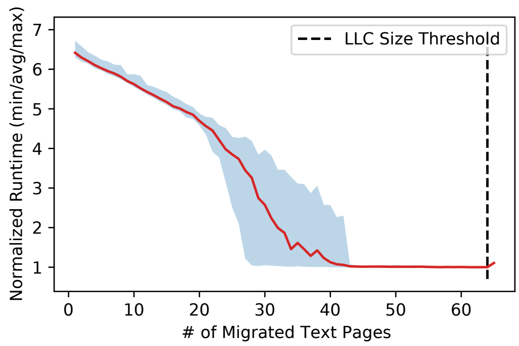


Figure 10 Inteference mitigation via migration of instruction pages.

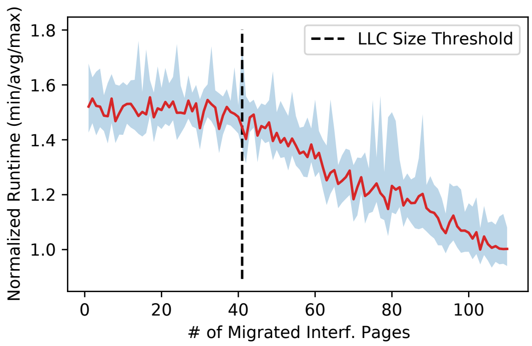


Figure 11 Interference mitigation via migration of data pages.

spill over L1 and are allocated in L2. But when the L2 is shared, these instruction pages are subject to be evicted by data fetched by any interfering workload. Unlike with missed over data items, an L1 and L2 miss during an instruction fetch cannot be hidden by the micro-architecture, which causes an immediate pipeline stall. The resulting impact on the runtime of the application under analysis can be dramatic.

We observed this effect in the wild and created a synthetic benchmark, namely C2ISBM, to isolate and study the C2IS problem. Our C2ISBM is a process that invokes a long function that spans through 65 `text` pages — i.e., it performs around 64,000 nops. Using as a baseline its solo performance, the runtime increases by a factor of 6.5× when INTERF workload is activated on all the other cores. We extract a profile of the C2ISBM benchmark, where the instruction pages are identified as important. We then configure our system in the PVT+SH mode (see Section 6), and progressively select the instruction pages to be migrated to the PVT pool. Recall that in the PVT+SH configuration, the PVT pool is exclusively allocated to 1/4 of the L2 cache. Gradually migrating the profiled instruction pages to the private pool allows us to gradually de-conflict these pages and to create an equivalent L2 instruction cache with a size that is proportional to the number of migrated pages. The resulting impact on the runtime of the C2ISBM process is plotted in Figure 10. A sharp improvement in runtime can be observed until around 43 pages are migrated. After that, the benchmark becomes unaffected by the interfering workload as around 51 (43 + 8 in the I-cache) of the 65 instruction pages are deterministically present in the cache. It can be noted that a slight runtime increase is visible when more than 64 pages are migrated because the private pool can hold up to 1/4 of the L2 cache size, i.e. 64 pages.

In the presented use-case, being able to identify those pages that are crucial for the application's performance and selectively migrate them to a reserved portion of the cache, space is an efficient solution to the C2IS problem. By contrast, strict coloring would force all the pages of the application to share the same color, which would require the allocation of a much larger cache partition to achieve the same degree of interference mitigation.

7.4 Controllable Mitigation of Cache Interference

In the last set of experiments, we use our BBProf toolkit and PVT+SH setup to demonstrate that (1) profile-driven interference mitigation is effective for real-world applications, and (2) that, albeit more flexible, its effectiveness is comparable to strict partitioning. For this experiment, we leverage the fact that we can profile the interfering workload and progressively

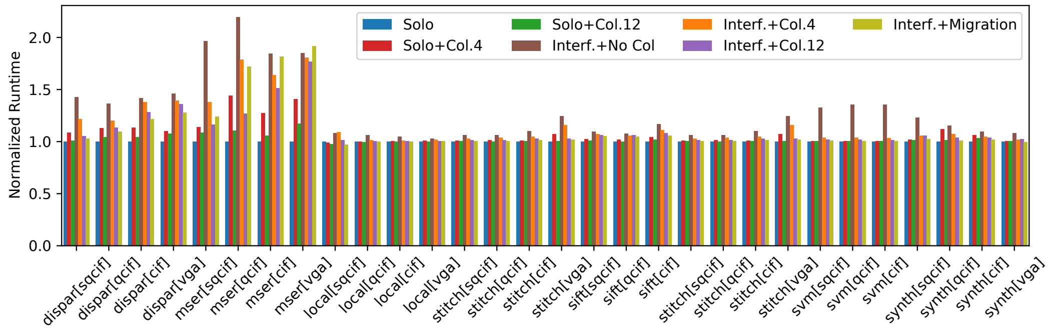


Figure 12 Mitigation of cache interference with profile-driven migration of interfering data pages.

migrate to the private pool the pages that are responsible for the generated cache contention, while we keep the pages of the application under analysis in their original location. Doing this allows cache-sensitive applications to benefit from 12/16 of the LLC space. First, we study the temporal behavior of the MSER benchmark with input size SQCIF in Figure 11. On the x-axis we track the number of pages migrated to the private pool for each of the three INTERF benchmarks — hence the total size of migrated pages is three times this value. The timing behavior of MSER starts to improve after 123 pages from the INTERF benchmarks are migrated away. That is because each INTERF process accesses a total of 315 pages (420 KB each, see Section 7.1), meaning that only 192 pages are left to migrate, which is exactly 12/16 of the total LLC size.

Lastly, Figure 12 summarizes the behavior of the most interesting benchmarks when a full migration of interfering pages is performed — see last bar of each cluster ("Interf.+Migration"). The resulting runtime is compared against a number of notable cases: (1) the "Solo" case where no INTERF is deployed and no cache partitioning is performed. This is also the normalization baseline for all the other cases; (2) and (3) the solo runtime where only four ("Solo+Col.4") or 12 ("Solo+Col.12") cache colors are assigned to the application under analysis; (4) the "Interf.+No Col" case where INTERF is deployed on all the other cores and no partitioning is enforced; (5) and (6) the cases "Interf.+Col.4" and "Interf.+Col.12" that correspond to (2) and (3) but with INTERF active on all the other cores. Profile-driven migration has comparable performance to the case where 12 page colors are dedicated to the application under analysis. In a few cases (see MSER with input sizes SQCIF and QCIF) migration does worse. The reason is likely interference over shared Linux meta-data (e.g. page tables, kernel code and data structures). This kind of contention does not occur with strict partitioning because the INTERF workload operates in a different, fully colored VM.

8 Known Limitations

The proposed method and current implementation present a number of limitations. First (i), BBProf is not designed to handle multithreaded applications, or applications comprised by multiple processes with complex data sharing, synchronization and dependencies. Second (ii), for applications that that exhibit strong dependencies between inputs and memory usage, the profile produced by BBProf on a given input might not generalize well to the entire input space. Third (iii), the only piece of information used by BBProf to construct profiles is timing. While this is a deliberate choice that allows BBProf to better generalize on many COTS platforms, we envision that being able to integrate additional metrics (e.g.

L1/L2 cache/miss count, consumed main memory bandwidth, energy consumption) might be useful to characterize page importance along additional dimensions beyond timing. In our current implementation, we only provide sample code to integrate calls to Perf [12] APIs during the entry/exit protocols, but more comprehensive handling of the additional metrics that can be collected is required. Fourth (iv), our current implementation relies on a number of Linux-specific features, such as PTRACE and the `proc` filesystem. Thus, while porting to other non-Linux OS's or even bare-metal environments is possible, some heavy re-engineering is required. We expect that PTRACE might need to be replaced with direct interaction with platform-specific debug registers, while memory layout information currently collected via `proc` interfaces might need to be exported at compile-time. Next (v), BBProf does not rely on any hardware features that are not widely available. Nonetheless, a few architecture-dependent features are leveraged, requiring some porting effort when moving to different architectures. These are (1) cacheability manipulation, (2) sampling of CPU clock cycles, and (3) cache maintenance operations. Lastly (vi), the time required to carry out profiling is strictly dependent on the WSS of the target application and on the runtime of the observation segment. Thus, BBProf might become impractically slow at profiling large-footprint and/or long-running applications. Operating on groups of adjacent pages instead of individual pages might mitigate this problem, but the trade-off between loss in granularity and speed-up needs to be investigated.

9 Concluding Remarks

In this work, we introduced BBProf, a methodology and toolkit to extract the importance of individual memory pages towards the runtime of a target application. The proposed BBProf does not rely by design on any hardware-specific feature, and thus it can be implemented on any platform where (1) it is possible to change cacheability attributes at a single-page granularity; and (2) it is possible to acquire time samples. Additionally, BBProf can operate on the unmodified, pre-compiled binaries of complex applications, and includes strategies to cope with the use of dynamic memory allocation primitives. We have performed and described an open-source full system implementation and setup on a state-of-the-art high-performance embedded platform. With this setup, we have shown three main aspects. First, that BBProf is capable of extracting the profile of real-world complex vision applications. Second, that the extracted page-level profiles can be used to enact fine-grained shared cache management. Third, that a previously undocumented variant of inter-core interference, namely contention-induced instruction stall can arise in multi-core embedded platforms; in which case profile-driven selective page migration represents an efficient mitigation strategy.

As part of our future work, we intend to relax some of the limitations described above. For instance, we aim at expanding the capabilities of BBProf to capture additional per-page properties. Moreover, we plan to develop strategies to use profiling information for OS-driven mapping of pages to heterogeneous memory resources — e.g., scratchpad memory, FPGA BRAM. Finally, we plan to further improve the level of detail of the collected information by identifying how each page impacts the runtime of multiple code sub-segments.

References

1. Zoom by Rotate Right. URL: `http://www.rotateright.com/`.
2. Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile (version G.a), author=ARM Holdings, 2011.
3. Siemens AG. Jailhouse, 2014. URL: `https://github.com/siemens/jailhouse`.

4 ARM Holdings. Cortex-A53 MPCore technical reference manual (r0p4), 2018. URL: https://developer.arm.com/documentation/ddi0500/j/.

5 I. Ashraf, M. Taouil, and K. Bertels. Memory profiling for intra-application data-communication quantification: A survey. In *2015 10th International Design Test Symposium (IDT)*, pages 32–37, 2015. doi:10.1109/IDT.2015.7396732.

6 F. Bouquillon, C. Ballabriga, G. Lipari, and S. Niar. A wcet-aware cache coloring technique for reducing interference in real-time systems. *CoRR*, abs/1903.09310, 2019. URL: http://arxiv.org/abs/1903.09310, arXiv:1903.09310.

7 D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, page 265–275, USA, 2003. IEEE Computer Society.

8 J. M. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 194–204, 2009. doi:10.1109/ECRTS.2009.13.

9 W. Cohen. Multiple Architecture Characterization of the Build Process with OProfile, 2003. URL: http://oprofile.sourceforge.net.

10 J. Corbet, J. Edge, and R. Sobol. Kernel Development. Linux Weekly News – https://lwn.net/Articles/74295/, 2004. [Online; accessed 7-May-2019].

11 C. Dall and J. Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 333–348, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2541940.2541946.

12 The Linux Foundation. perf: Linux profiling with performance counters. URL: https://perf.wiki.kernel.org/index.php/Main_Page.

13 R. Mancuso G. Ghaemi, D. Tarapore. BU Black-box Profiler. https://github.com/rntmancuso/black-box-profiler, 2021.

14 G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 48(2), November 2015. doi:10.1145/2830555.

15 G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In Sophie Quinton, editor, *31th Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:25, Stuttgart, Germany, July 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECRTS.2019.27.

16 M. Hassan. On the off-chip memory latency of real-time systems: Is ddr dram really the best option? In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 495–505, 2018. doi:10.1109/RTSS.2018.00062.

17 H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 80–89, 2013. doi:10.1109/ECRTS.2013.19.

18 H. Kim and R. Rajkumar. Real-time cache management for multi-core virtualization. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2016. doi:10.1145/2968478.2968480.

19 H. Kim and R. (Raj) Rajkumar. Predictable shared cache management for multi-core real-time virtualization. *ACM Trans. Embed. Comput. Syst.*, 17(1), December 2017. doi:10.1145/3092946.

20 N. Kim, B. C. Ward, M. Chisholm, C. Fu, J. H. Anderson, and F. D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016. doi:10.1109/RTAS.2016.7461323.

21 T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14, 2019. doi:10.1109/RTAS.2019.00009.

22 Y. Kwon, X. Zhang, and D. Xu. Pietrace: Platform independent executable trace. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 48–58, 2013. doi:10.1109/ASE.2013.6693065.

23 J. Liedtke, H. Haertig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, RTAS '97, page 213, USA, 1997. IEEE Computer Society.

24 C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005. doi:10.1145/1064978.1065034.

25 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, 2013. doi:10.1109/RTAS.2013.6531078.

26 S. Mittal. A survey of techniques for cache partitioning in multicore processors. *ACM Comput. Surv.*, 50(2), May 2017. doi:10.1145/3062394.

27 P. Modica, A. Biondi, G. Buttazzo, and A. Patel. Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1651–1657, 2018. doi:10.1109/ICIT.2018.8352429.

28 N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007. doi:10.1145/1273442.1250746.

29 A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi. Embedded hypervisor xvisor: A comparative analysis. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 682–691, 2015. doi:10.1109/PDP.2015.108.

30 A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 335–348, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1755913.1755947.

31 P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F.J. Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4), January 2012. doi:10.1145/2086696.2086713.

32 L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 258–269, 2008. doi:10.1109/MICRO.2008.4771796.

33 P. Sohal, R. Tabish, U. Drepper, and R. Mancuso. E-warp: A system-wide framework for memory bandwidth profiling and management. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 345–357, Los Alamitos, CA, USA, dec 2020. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/RTSS49844.2020.00039, doi:10.1109/RTSS49844.2020.00039.

34 D. Tarapore, S. Roozkhosh, S. Brzozowski, and R. Mancuso. Observing the invisible: Live cache inspection for high-performance embedded systems. *IEEE Transactions on Computers*, pages 1–1, 2021. doi:10.1109/TC.2021.3060650.

35 S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, Oct 2009. doi:10.1109/IISWC.2009.5306794.

36 Xilinx, Inc. Zynq ultrascale+ mpsoc data sheet: Overview (v1.8), 2019. URL: https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

37 M. Xu, R. Gifford, and L.T. Xuan Phan. Holistic multi-resource allocation for multicore real-time virtualization. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3316781.3317840`.

38 Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: A dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392, 2014. `doi:10.1145/2628071.2628104`.

39 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013. `doi:10.1109/RTAS.2013.6531079`.

40 X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 89–102, New York, NY, USA, 2009. Association for Computing Machinery. `doi:10.1145/1519065.1519076`.