MxU: Towards Predictable, Flexible, and Efficient Memory Access Control for the Secure IoT

Runyu Pan
The George Washington University
panrunyu@gwu.edu

Gabriel Parmer
The George Washington University
gparmer@gwu.edu

ABSTRACT

The advanced functionality requirements of modern embedded and Internet of Things (IoT) devices – from autonomous vehicles, to city and power-grid management – are driving an ever-increasing software complexity. At the same time, the pervasive internet connections of these systems necessitate the fundamental design of security into these devices. The isolation of complex features from those that are critical through protection domains is an effective means to constrain the scope of faults and security breaches. Common hardware-provided memory facilities to enforce protection domains through memory access control – including Memory Management Units (MMUs) usually found in microprocessors, and Memory Protection Units (MPUs) usually found in microcontrollers – must meet the goals of enabling *flexible, efficient and dynamic management of memory*, and must enable *tight bounds on the worst-case execution* of critical code. Unfortunately, current system memory management facilities are ill-prepared to handle this challenge: MMUs that use extensive caches to achieve strong average-case performance suffer from debilitating worst-case and even average-case behavior under hefty interference, while MPUs struggle to provide flexible memory management.

This paper details MxU, a memory protection and allocation abstraction that integrates temporal specifications into the memory management subsystem, to enable *portable* code to achieve both predictable, tightly-bounded execution and dynamic management across both MMU- and MPU-based systems. We implement MxU in the Composite microkernel, and evaluate its flexibility and predictability over two different architectures: a MPU-based Cortex-M7 microcontroller and a MMU-based Cortex-A9 microprocessor using a suite of modern applications including neural network-based inference, SQLite, and a javascript runtime.

For MMU-based systems, MxU reduces application TLB stall by up to 68.0%. For MPU-based systems, MxU enables flexible dynamic memory management often with application overheads of 1%, increasing to 6.1% under significant interference.

ACM Reference Format:
Runyu Pan and Gabriel Parmer. 2022. MxU: Towards Predictable, Flexible, and Efficient Memory Access Control for the Secure IoT. In *Proceedings of EMSOFT 2019 (EMSOFT '19).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

EMSOFT '19, October 13 2019, New York

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The common denominator of Industry 4.0, the Internet of Things (IoT), and "smart" infrastructure is the ubiquitous connection of embedded systems to the network. This promises new levels of coordination, orchestration, situational awareness, and environment analyzability. At the same time, we're moving increasing functionality to embedded devices, which pushes these systems beyond the simplicity of traditional embedded systems. This is often driven by Size, Weight, (Cost), and Power (SWaP) constraints that encourage consolidation of many functions onto a single system. Applications requiring increased feature sets include Machine Learning (ML), embedded virtualization, and blockchain, which are often scaled-down versions of their general-purpose variants.

Unfortunately, a network connection exposes devices to malicious actors. The ever-increasing complexity of embedded-systems driven by the addition of low-assurance code, necessitate a deep and pervasive focus on both security and reliability. Though we cannot practically ensure that all code is free of bugs, we *can* ensure that different system functionalities are mutually isolated in time and space so as to limit the impact of their failure or compromise. Hardware memory isolation mechanisms are wide-spread, including the microprocessor Memory Management Units (MMUs) and microcontroller Memory Protection Units (MPUs). They focus on constraining the scope of each application's accessible memory, thus providing *memory access control*.

Unfortunately, MMUs and MPUs are ill-prepared to provide both the flexibility necessary for dynamic, feature-rich applications, and the predictability necessary for embedded systems. MMU-based systems use the page-based memory mapping facilities to control application address spaces. They use a memory-defined page-table representation paired with a pipeline-integrated Translation Lookaside Buffer (the TLB) cache to define the subset of memory accessible by the application. MMUs exhibit significant variance in memory access latency as access control decisions access memory-based page tables. A load or store to a application data in L1 cache (often < 10 cycles) can exhibit multiple 100s of cycles if the corresponding page-table is not in cache. On the other hand, MPU-based systems provide protection of only a small, limited number of memory ranges. This can easily prevent their use with feature-rich applications that use heap-based, dynamic memory allocations, and shared memory.

To provide the security and reliability constraints required by modern embedded systems, while ensuring both tight Worst-Case

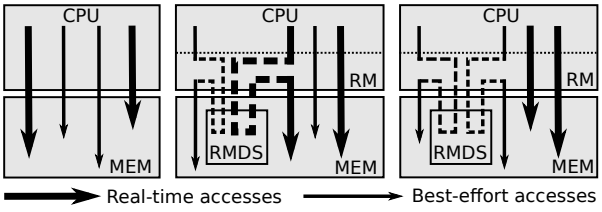

Figure 1: CPU accesses to memory, and the memory access control resource monitor (RM). The left system has no protection, center uses the RM which must access its own data-structures (RMDS) in memory (dashed lines) which increase the latency of some accesses. The right system is MxU which integrates temporal specifications into memory allocation to enable tight execution bounds for critical applications.

Execution Time (WCET) bounds and good average-case performance, this paper introduces MxU. MxU integrates temporal specifications into the memory allocation facilities, and ensures that they map down to hardware mechanisms to provide the corresponding behavior (see Figure 1). Additionally, MxU enables the necessary dynamic memory management for complex applications even on constrained microcontrollers. MxU provides a unified API that controls the otherwise loose WCET bounds from MMU systems, and makes MPU systems more dynamic and flexible. We believe this will enable the strong security that modern embedded systems require, while providing both more practical predictability and flexibility.

Contributions. We begin with a background on MMUs and MPUs (§2), then focus on this paper's contributions:

- we introduce the design of MxU (§3) and its integration of temporal constraints into the OS's handling of MMUs (§3.4) and MPUs (§3.5);
- a prototype implementation of MxU in the Composite microkernel in as a system-wide memory manager (§4);
- an evaluation of the MxU prototype to understand the fundamental predictability and efficiency properties of the system's operations (§5); and
- a thorough evaluation of MxU's use with applications focusing on data-management, processing, and machine-learning (§5.3).

2 MEMORY PROTECTION MECHANISM BACKGROUND

To further understand the challenges in achieving predictability and flexibility with MMUs and MPUs while providing strong memory protection, we review the hardware organization of each in detail.

2.1 Memory Management Units

MMUs provide flexible memory access control facilities by enabling page-granularity memory protection and allocation. Different applications are given different subsets of system memory, thus providing OS-controlled memory isolation. Application's accessible memory is expanded and contracted on a page-granularity only limited by the size of the address space, thus enabling *flexibility* in allocation. MMUs enable applications to execute in overlapping address spaces using the virtualization of addresses. This is achieved by controlling the mapping of the virtual addresses that are exposed to applications, to the backing physical memory. This enables ease of *dynamic memory allocation* as non-contiguous physical memory can be used for single, contiguous virtual allocations.

The implementation of MMUs has a significant impact on the performance and WCETs of task execution. To provide efficient resolution of memory access permissions, MMUs must be integrated into the processor pipeline. To achieve efficient pipeline integration and *flexibility* in mapping a large number of pages, a typical MMU includes a TLB cache of virtual to physical translations, and the hardware logic for walking a page table data structure (though software-driven page table walkers exist, they are less common). When a memory access is issued (*e.g.* a load or store) to a virtual address, the MMU finds a translation for the address in its TLB. If such a translation is not found, the in-memory page table is walked to find a translation.

Unfortunately, MMUs impact the average-case performance of applications and their ability to execute within *tight WCET bounds*. The latency of the access control decision has significant jitter depending on if it hits in the TLB, or it walks the page-table. The latter involves multiple memory accesses for each level in the page-table, thus significantly increasing memory access jitter. On processors that are heavily dependent on the efficiency of their caches, this overhead is in the thousands of cycles [18]. The memory access pattern of an application alone is not sufficient to understand cache behavior, as the MMU's state must also be considered. This is challenging as TLB eviction policies are often opaque and unpredictable (often explicitly based on random replacement or pseudo-least recently used – P-LRU). To avoid these challenges, many embedded systems choose to forego isolation, and use a single identity mapped address space (where all virtual and physical memory are identical) consisting of very large superpages.

To make things worse, the MMU's TLB is shared among multiple applications. When switching between applications, TLB entries are transient and are not explicitly saved and restored as part of the context switch. As such, the TLB must be flushed to prevent the previous application's mappings from being accessed from the next. This causes TLB misses when switching back to an application, further harming the performance and forcing pessimistic WCET estimates to maintain predictability. Alternatively, many modern processes maintain tagged-TLBs, in which each application, and each TLB entry is associated with an Address Space ID (ASID). Only entries with the current application's ASID provide valid translations, thus providing access control within the TLB. This enables the avoidance of flushes on context switches, but the worst-case impact still exists if another application with a large working set evicts an application's entries. As such, *inter-application interference* in shared TLB caches forces pessimistic assumptions about full TLB eviction on context switch, thus significantly inflating the WCET. If this interference is strong and persistent, the average-case performance can be adversely affected as well. This interference in a fixed-priority system is on the order of $M \times L \times S \times \sum_{p_h \in higher(i)} \lceil p_i / p_h \rceil$ for task τ_i where $higher(i)$ is all higher-priority tasks, M is the worst-case memory access time, L is the number of levels in the page table, and S is the minimum of the TLB residency and number of TLB entries.

2.2 Memory Protection Units

MPUs provide a limited form of memory access control, and are typically found in smaller microcontrollers. They use a register-based design in which specific entries are directly retrieved and programmed. Each entry contains an address range and access permissions (*e.g.* read, write, execute), and enables the corresponding accesses only to those addresses. If the access is out of the designated range, or violates the access permissions, an exception will be delivered to the kernel.

The MPU registers are directly saved and loaded by the operating system, thus the contents are explicit, and accesses to address ranges are always deterministic and predictable (validated by, or in violation of, the MPU entries). Memory access control checks never make extra memory accesses. This leads to *tighter WCET bounds*, thus a high-degree of predictability of memory accesses as access control decisions are made using on-chip registers. Due to the *explicit programmability and visibility* of the MPU registers, they can be saved and restored with other thread registers. This completely *mitigates inter-application interference* at the cost of increased context switch times.

Unfortunately, several properties of MPUs limit their *flexibility*, especially for systems with dynamic memory requirements. The number of MPU entries is bounded, and often quite small (between 4 and 32), and the size of the address range for each entry is constrained (often to be a power-of-2, and aligned on that boundary). Additionally, common MPUs do not map virtual addresses to physical addresses, which implies that contiguous memory is required for allocations. This leads to many MPU-based systems completely forbidding *dynamic memory allocation* from global memory. Some MPU implementations do allow region-based virtual address to physical address translation, which eases global allocation, but they are still constrained by the limited and small number of MPU regions.

Infrastructure	Dyn. Alloc	Flexibility	Tight WCET	No Interf.
MMU only	✓	✓	✗	✗
w/ASID	✓	✓	✗	✗
MPU only	✗	✗	✓	✓
w/virt	✓	✗	✓	✓
MxU	✓*	✓	✓	✓

Table 1: Memory access properties for different memory access-control hardware mechanisms. * is clarified below.

2.3 MMU and MPU Summary

Current MMU-based embedded systems either avoid the unpredictability of memory accesses by offloading real-time computations to microcontrollers, or accept the pessimistic bounds necessary to accommodate uncontrolled TLB misses. In contrast, MPU-based systems often offload more complex computations with complex memory usage to MMU-based systems. This draws a false-dichotomy between these systems, even though many MPU-based systems are just as computationally capable as their MMU counterparts (*e.g.* Cortex-M7 and Cortex-A9).

Table 1 summarizes some of the key dimensions differentiating MMU and MPU variants, and clarifies the goals of MxU. MxU aims to provide a uniform abstraction for memory access control that provides tight WCET bounds, flexibility, dynamic, system-wide memory allocation, and controls inter-application interference. MxU extends the memory allocation interfaces to enable dynamic, system-wide memory management. However if hardware doesn't support address virtualization, it is limited by the availability of contiguous ranges of memory.

3 MXU DESIGN FOR FLEXIBILITY AND PREDICTABILITY

3.1 MxU Abstraction

MxU is a high-level abstraction that integrates temporal specifications into flexible memory management facilities, and maps them down to the memory access control hardware. This is motivated by the strong security and reliability requirements of modern embedded systems. MxU increases both the tightness of worst-case bounds for and the flexibility of protection domains, thus enabling the liberal use of hardware-provided isolation to limit the scope of faults and compromises. Average-case performance is also improved for cases where heavy interference persists.

MxU integrates new mechanisms into the core hardware-software co-design of memory protection facilities, and guides them by adding an abstraction layer on top that integrates temporal specifications. The hardware mechanisms for memory access control integrate in-pipeline, efficient and predictable structures (*e.g.* CAMs), with in-memory data-structures to track broader state. MMUs use page tables as a medium for coordination between hardware and OS, and MPUs use explicit region registers loaded from OS-defined data-structures. Each memory access that must go out to the access control, in-memory data-structures incurs the significant overhead and latency spikes from additional memory accesses (that may miss the cache in the worst-case). This causes significant memory accesses jitter compared to solely on-chip accesses.

On-chip caches are used to increase the efficiency of memory accesses, and embedded chips often enable limited control over cache contents. MxU's focus is to control the latency of memory access control decisions by controlling cache contents at all levels: are necessary entries in pipeline-integrated caches (TLB, or MPU region), in L1/L2/L3, or in memory? This increased control enables the latency of memory access control decisions to be bounded and significantly tighter than the existing worst-case of having to access off-chip memory. MxU uses a uniform data-structure representation for memory access control, and through the temporal specification used for memory allocation, ensures that the required access control decisions are bounded by cache accesses at a specified level.

3.2 Guarantees and Aims

The MxU abstractions makes a number of *guarantees* upon which secure embedded systems can be built. Unfortunately, the limitations of hardware prevent some desirable properties, thus we include a number of *aims* that are satisfied on a subset of MxU's hardware.

MxU guarantees:

G1: Integration of temporal properties into memory management. All memory allocation operations are parameterized by a temporal specification which corresponds to the level of the storage hierarchy at which memory access control decisions

are guaranteed to be made. This enables applications to limit the access control's increase in memory access latencies.

G2: Admission control for memory access timeliness. An allocation is successful if and only if the hardware is capable of allocating such memory with the appropriate access latency guarantees. This should enable static guarantees in cases where all allocations are known a-priori.

G3: Flexibility of allocations. The number and size of allocations is not constrained by specific hardware implementations, and is only constrained by available memory.

G4: Portable abstraction. The MxU abstraction and programming interfaces is uniform, and shared across MMU and MPU-based systems, thus enabling portable embedded system implementations.

G5: Memory protection and controlled sharing. To create secure embedded systems, protection domains must enable access to disjoint sets of memory with the controlled exception of the shared memory API. Shared memory also requires agreement on the temporal specifications for that memory.

MxU aims:

A1: Fully deterministic accesses. If the hardware provides means to control lookups in in-pipeline structures, then specific regions are accessed with determinism.

A2: Controlled interference between protection domains. If the hardware allows efficient direct programming of and visibility into memory access control state, interference in between lookups for different protection domains is eliminated.

A3: Address virtualization. If the hardware can translate virtual addresses to physical addresses, MxU harnesses this feature, thus enabling non-contiguous physical memory to satisfy single allocations.

3.3 API Design

Operation	Description
`rt_malloc(sz, ts)`	Allocate memory with a given temporal specification (`ts`).
`rt_malloc_at(addr, sz, ts)`	Similar, but allocate at a specific address.
`rt_shmget(id, sz, align, ts)`	Allocate shared memory associated with an `id`.

Table 2: The main functions in the MxU API.

MxU aims to extend conventional APIs, enabling it to slot in to existing software ecosystems easily. Table 2 shows the most notable functions in the API. They all focus on dynamically allocating memory and associating it with some temporal specification. Note that this API is more flexible than those exposed for most MPU-based systems that are architecture-specific and explicitly program the protection registers. Though the API is similar to typical `POSIX` interfaces, `rt_malloc_at` can be used by the lower-level OS components which allocate and map application's code and data to control the timing properties of even applications that require only static allocation. Legacy APIs are supported by fixing the temporal specification for all of an application's allocations.

MxU temporal specification. This specification includes a number of flags: (1) `MXU_DETERMINISTIC` to require that access control checks for this memory must be deterministic, and use only pipeline-integrated (sub-cycle latency) hardware, (2) `MXU_L`x requires that access control checks never touch memory going beyond the level-x cache, and (3) `MXU_INDEPENDENT` to signify that inter-application interference must be prevented, thus enabling the application to be analyzed independently. The temporal specification focuses explicitly on cache behaviors, rather than on latency or time values. The application's timing behavior is a composition of its execution and memory access properties, and the additional delays due to the memory access control checks. This API gives guarantees for the latter, so that a timing analysis of the application (beyond the scope of this paper) can bound interference from memory access control.

3.4 Design for MMU-based Systems

MMU-based systems use virtual memory (**A3**) to support dynamic, often physically non-contiguous, allocation (**G3**). The MxU design for MMU-based systems exploits hardware features such as TLB lockdown, on-chip memory, and cache layouts for coloring, to lower the worst-case MMU access control bounds. Unlike other past work that use such features to control application data [1, 8, 11, 12, 14, 18, 24, 25], MxU focuses on controlling placement of access control data-structures as these are checked for *every* memory operation, thus can have a disproportionately large impact on worst-case execution time. MxU leverages hardware features, where available, to explicitly place page-table nodes at the cache-levels of the temporal specification (**G1**), or returns failure if those features aren't available, or are expended due to other previous requests (**G2**).

Deterministic memory accesses (with zero overhead from access control operations) are supported only in those chips where TLB lockdown or coloring [18] are possible (**A1**). Perhaps most difficult to support with MMUs is the controlled interference between protection domains (**A2**). As the TLB's contents are not OS-controlled, even ASIDs do not prevent a protection domain's contents from being flushed (by capacity evictions). Regardless, MxU supports ASIDs to improve average, if not worst-case, performance.

3.5 Design for MPU-based Systems

Given the explicit programmability of MPUs, MxU has complete control over the MPU's contents (**G1** and **A1**), and tracks if it runs out of regions (**G2**). Unlike existing systems, the regions are saved for *each application* along with their registers, thus providing complete inter-application independence (**A2**). To increase the flexibility of allocations, the MxU design for MPU-based systems takes advantage of the software MPU exception handler to virtually extend the number of regions (**G3**). When access is made outside of an allowed MPU region, the CPU triggers an OS exception that can load in new MPU regions on-demand. Thus, much like page-tables, MxU uses a set of in-memory data-structures from which to retrieve valid MPU regions. For these, the same optimizations around placement are relevant, as is hardware support for controlling cache contents. Only a relatively limited number of microcontrollers have MPUs that support address mapping virtualization (**A3**).

4 MXU IMPLEMENTATION

4.1 MxU-enabled Composite Kernel

We implement MxU in the Composite [23] microkernel that has strong security based on capability-based access control. All system resources including threads, communication end-points, and protection domains, are only referenced through *unforgeable tokens* called capabilities, which are *delegated* in a controlled manner between user-level *components* (Composite's protection domains). Authorized protection domains boot up with full access to system resources, and they delegate capabilities to those resources to others in accordance with their policies. These "management" components minimally include the system scheduler (there is no in-kernel scheduler [7]), and the memory and I/O managers.

MxU is implemented as the system's memory manager component. Application components use synchronous invocations (the highly-optimized, component-equivalent of IPC) to request memory services. Semantically, these invocations take the form of function calls to the API in §3.3. The MxU memory manager then uses the kernel system call operations to modify the client's protection domain (*e.g.* add or remove memory access).

In the style of seL4 [4], Composite requires user-level components to manage kernel memory using a safe form of memory retyping. Even the construction of memory access control data-structures (*e.g.* page tables) is carefully performed by components with proper access to resources, and the ability to retype memory to be the appropriate memory type. This allows the memory manager to define the policy to control the physical addresses of kernel data-structures such as page tables (within the bounds of what resources it has capabilities to). A crucial guarantee that the kernel ensures (independent of any component's logic) is that memory that has been typed as kernel (*i.e.* that backs kernel data-structures) cannot also be typed as user memory (thus accessible to components) – this has the effect of ensuring the consistency of kernel data-structures. Pan et al. [17] introduced the use of the space-efficient Path-Compressed Radix Trie (PCTrie) data-structure to program MPU regions. MxU uses PCTries as the uniform data-structure to describe both MMU and MPU systems (page tables are a specific configuration of PCTries).

MxU expands the kernel API to provide operations on PCTrie nodes that mimic the temporal specifications in §3.3. Specific PCTrie nodes optionally have cache locality specifications when created, which the kernel uses to place them appropriately using whatever hardware mechanisms are available. The MxU memory manager tracks how much hardware (*e.g.* cache ways, TLB entries, MPU regions) has been allocated for existing resources, and performs admission control on these limited resources for new requests. Though the kernel provides the facilities for manipulating processor resources, the memory manager is given a *hardware capability* allowing the caches of various levels to be manipulated. All these features are orchestrated by the memory manager to provide MxU services across MMU-based and MPU-based architectures, which satisfies **G1**, **G4** and **G5**.

Though the Composite system has a number of benefits that eased the development of MxU including the explicit layout control of kernel data-structures, fast IPC and user-level definition of MxU policies, and existing support for portable access control data-structures between MPUs and MMUs, comparable modifications could be made in other existing systems such as Linux.

4.2 MMU-based MxU

On MMU-based systems, the core idea is to lower worst-case memory access latencies (**G1**). To make sure that memory access decisions are made in the worst-case at a certain cache level, page table nodes must be stored in the corresponding cache-level. This requires the utilization of relatively common hardware features and characteristics to control cache contents [15] including coloring [18], way locking, and pinning. In our current implementation, we are using the XC7Z020-1CLG400I, which features an ARM Cortex-A9 dual-core processor, but only one core is used.

Item	Description
Speed	767MHz, 2.5 DMIPS/MHz
MMU	4k page support, 8-bit ASID support
L1 TLB	32 I/D, fully associative
L2 TLB	128 2-way, plus 4 lockable entries
L1 cache	32k I/D, 4-way.
L2 cache	512k, 8-way, lockable by way.
On-chip SRAM	256k at L2 speed.
Off-chip SDRAM	1GB DDR3-1066.

Table 3: The detailed hardware configuration of XC7Z020-1CLG400I platform (processor subsystem).

We can see that the processor includes the following different levels of cache: level-1 TLB (**L1T**), level-2 TLB (**L2T**), L1 data cache (**L1C**) and L2 cache (**L2C**). The processor also features on-chip SRAM **(OCM)**. The MMU supports 4k pages and features a hardware page table walker. This introduces the following different cache levels where the memory access control decisions may be made.

L1T: The L1 TLB is fully associative, which makes TLB coloring infeasible and the entries are not lockable. Thus, MxU cannot control this cache.

L2T: The L2 TLB will be accessed if there is an L1 TLB miss. The L2 TLB is not fully associative (only 2-way set associative), which allows for effective TLB coloring [18]. Additionally, Cortex-A9 provides 4 extra individually lockable entries. In this paper, we make use of this feature for guaranteed TLB access (**G1**) in response to `MXU_DETERMINISTIC` specifications.

L1C: On Cortex-A9, L2 TLB misses trigger the hardware page table walker which starts walking the page tables from L1 data cache. The L1 data cache is also not fully associative (4-way set associative), which makes cache coloring possible. However, the L1 data cache is too small to retain permanent data, thus we forgo such an option. In this case, applications that specify `MXU_L1` will be TLB-pinned, or the allocation will return an admission control failure (**G2**).

L2C: The L2 cache is not fully associative as well (8-way set associative), which makes cache coloring possible but less effective. Additionally, this processor provides way locking which prevents evictions of the way's contents. We prefer way locking versus page coloring as it doesn't put strong constraints on physical memory allocation. There is a trade-off: locking a way still

reduces the cache size available to the regular code and data, which can harm application performance.

OCM: On-chip SRAM is accessed at L2 cache speed is assigned a physical memory range. Composite enables the explicit typing of OCM memory as kernel page tables, thus OCM can be used to access them at L2 cost. The MxU implementation on this hardware favors the use of the OCM for `MXU_L2` specifications as it doesn't effectively shrink the cache.

This processor also supports ASIDs which are used for processes that specify the `MXU_INDEPENDENT` flag. However, caches are shared between processes, thus this only partially satisfies **A2**.

4.3 MPU-based MxU

On MPU-based systems, the emphasis is to enhance flexibility by providing a virtually infinite number of protected memory regions. As representative hardware based on an MPU, we use the STM32F767IGT6, which features an ARM Cortex-M7 processor. Its MPU does not provide virtual address translations (thus it cannot provide **A3**). The detailed hardware configuration is listed as follows:

Item	Description
Speed	216MHz, 2.14 DMIPS/MHz
MPU	8 regions, pow-of-2 size/alignment requirement
L1 cache	16k I, 2-way, 16k D, 4-way.
On-chip SRAM	512k.
On-chip Flash	1M, with ART accelerator.
Off-chip SDRAM	32M 108MHz.

Table 4: The detailed hardware configuration of the STM32F767IGT6 platform.

In MPU-based MxU, the PCTrie is used to store accessible memory ranges. A per-component representation of these ranges is stored in the top-level PCTrie node that is used to directly program the MPU registers (efficiently, using multi-register load and store instructions). When the component is switched to, this representation is loaded into the MPU. The PCTrie contains a large number of accessible memory ranges, while the MPU has only 8 regions, with two used to protect access to an on-chip timer we use as a timestamp counter (`TIM2`) and executable memory. MxU partitions the remaining regions of the MPU into two sets: Static and Dynamic. Static regions are used in response to `MXU_DETERMINISTIC` requests, and are *always* present in the MPU while the component is executing. Dynamic regions are managed as a cache while multiplex the remaining PCTrie ranges. We extend Composite such that if a PCTrie range is not present in the MPU (a load/store causes a miss), a MPU exception triggers a software handler that finds the desired range, performs an eviction from the Dynamic regions, and adds the range to the MPU. We implement three eviction policies: round-robin, random, and Bit-based Pseudo Least Recently Used (Bit-PLRU). While Static regions ensure deterministic requests (**G1**), Dynamic ensure memory management flexibility (**G3**). MxU enables the number of Static MPU regions to be determined by application allocation requests, and uses the rest for Dynamic. If a Static allocation would leave no Dynamic regions, the allocation returns failure (**G2**). MxU tracks and saves MPU state per-component, thus they all satisfy `MPU_INDEPENDENT` requests (thus **A2**).

5 EVALUATION

Hardware configurations. We use the MMU-based system described in §4.2, and the MPU-based system described in §4.3. These two architectures have significant market share and have many resembling features: both are 32-bit, dual-issue, the number of pipeline stages are close to each other, and both have a dynamic branch predictor. Though many other architectures exist, we do believe that the results obtained from these systems apply to other systems as well due to the same underlying principles.

We'll refer to the XC7Z020-2CLG400I system as S_{mmu}, and the STM32F767IGT6 system as S_{mpu}. For all evaluations, the gcc version 5.4.1 targeting the ARM architecture is used with the -O2 optimization flag. S_{mmu} is run at 767MHz, and S_{mpu} is run at 216MHz. The cache of both systems are always enabled, and for S_{mpu}, the flash accelerator is enabled along with execute-in-place support. In synthetic benchmarks, the prefetchers of S_{mmu} are disabled to maximize the accuracy of memory access latency measurements, while in application evaluation they are enabled.

All graphs in this section depict the average (the dot or the darker bottom bar) and standard deviation (the error bar displayed on the average bar) measurements. In all line graphs, only the upper standard deviation bar is shown for clarity. All bar graphs additionally depict the maximum measured value (the lighter top bar).

5.1 Miss overheads

The basic page or region filling operation on MMUs or MPUs are the deciding operations that force pessimistic bounds on memory access latencies on both of the platforms. To understand the overhead of TLB misses and dynamic MPU region replacement, we first investigate the hardware overheads for these operations. These measurements will provide us with the loose – but often necessary – upper memory access latency on our hardware.

We measure the overhead of a single load instruction whose target data content is in L1 cache, while varying if we flush the TLB and if we use Dynamic regions, and report the 99th percentile values to rule out noise (*e.g.* due to timers). On S_{mmu}, a hit in the TLB results in 6 cycles, while a TLB miss takes 434 cycles. On S_{mpu}, accessing a region in the MPU takes 2 cycles, while missing on a dynamic region and executing MxU's exception handler (assuming a simple filling of the region without any replacement policy cost) takes 360 cycles.

Discussion. Both systems show orders of magnitude slowdowns for misses. Conservative WCET analysis must consider the upper bounds of missing, thus deriving likely quite loose memory accesses bounds. This motivates MxU to better control the overhead of cache misses, thus avoid such increases in the execution bound of tasks.

5.2 Synthetic Benchmarks

To investigate the overhead of memory accesses for different MxU temporal specifications, and in the presence of interfering tasks, we evaluate a number of synthetic workloads. Four synthetic benchmarks are used: sequential access (`seq`), random access (`ran`), stride access with a stride of 4kB & accessing 16 integers at a time (`str`), and a simple 128*128 integer matrix self multiplication (`mat`). A total of 262144 accesses are made for the first 3 benchmarks. The

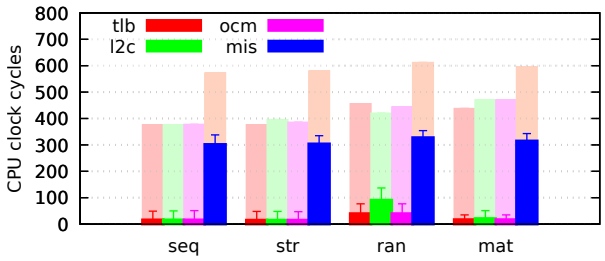

Figure 2: MMU temporal specifications' performance on different benchmarks. The horizontal axis is the different benchmarks, while the vertical axis is the memory access latency.

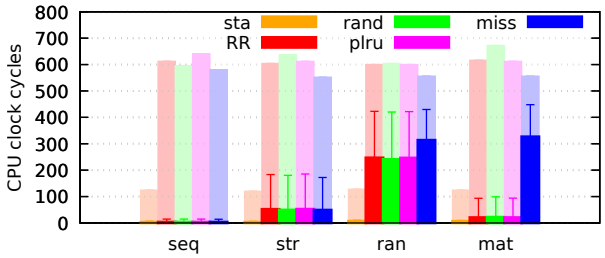

Figure 3: MPU region replacement policies' performance on different benchmarks. The horizontal axis is the different benchmarks, while the vertical axis is the memory access latency.

matrix multiplication benchmark only computes first 17 lines of results, which performs 557056 accesses. On S_{mmu} they do not wrap around, while on S_{mpu}, they wrap around in 64kB SRAM.

Memory access latencies. To evaluate the effect of different MxU mechanisms on the benchmarks, we run the four benchmarks with different temporal specifications on both platforms, and with different eviction policies on S_{mpu}. We compare against the costs of uniformly missing in access control caches. Here we wish to evaluate the effectiveness of MxU at controlling the pessimism of memory access control check latencies.

On S_{mmu}, we evaluate three temporal specifications: `MXU_DETERMINISTIC` which locks the accessed pages into the TLB (`tlb`), `MXU_L2` using way locking which locks seven ways of the L2 cache for page table storage (`l2c`) and OCM for page table placement (`ocm`). We also plot the case where a TLB and a L2 miss always occur for each memory access (`mis`). Figure 2 shows the results.

On S_{mpu}, we evaluate three Dynamic MPU region replacement policies: random eviction (`rand`), round-robin eviction (`RR`), and Bit-PLRU eviction (`plru`). These policies are also compared with Static mappings (`sta`) for `MXU_DETERMINISTIC`, and misses (`miss`). All policies are evaluated with three Dynamic MPU regions. The page size used is 4kB, and a region holds two pages. Figure 3 plots the results.

Discussion. In S_{mmu}, we see the using `MXU_L2` has a significant ability to lower the worst-case access latency when compared with full misses. Though the average-case access latency also shrinks, this is largely due to the data caches not being flushed in cases other than full misses. As the `ran` case shows, using way locking

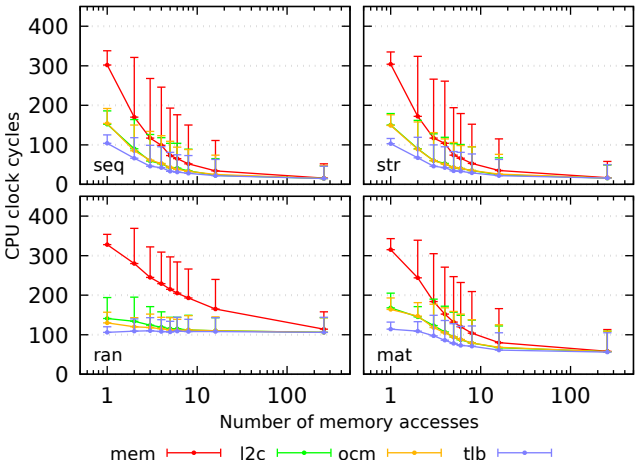

Figure 4: S_{mmu}'s memory access latency under different interference inter-arrival time. The four subfigures show the four benchmarks respectively. The horizontal axis is the interference inter-arrival time (in number of consecutive memory accesses), while the vertical axis is the memory access latency.

demonstrates increases in both average and worst-case latencies as we are using most of the cache for page tables, thus causing increased miss rates for benchmark data. We study the trade-off in way usage in the application evaluation later.

In S_{mpu}, the three replacement policies all perform similarly, which is in accordance with the fact that all three are widely used today. For this reason, we only run the `plru` policy for all the following evaluations on S_{mpu}. `MXU_DETERMINISTIC` specifications indeed control average and worst-case latency, while misses are quite pronounced, especially in matrix multiplication. For many of the workloads, the average performance of the Dynamic memory ranges is close to that of Static, thus showing the practical effectiveness of extending the MPU to a virtually unlimited number of regions.

Interference between different workloads. A significant component in the worst-case behavior of applications is how much co-running tasks and interrupt handlers can interfere with the cached MxU state. In cases where interference is strong, average-case performance may be hampered as well. To investigate the interference between different workloads, we run an adversarial workload alongside the four synthetic benchmarks. This interference represents the execution of interrupts or other applications.

Figure 4 shows interference measurements in S_{mmu}. We run the benchmarks in one thread, and an adversarial workload which flushes all the data cache and TLB in another component. We test the page table caching policies (for `MXU_DETERMINISTIC` and `MXU_L2`), and compare against memory-backed page tables in the `mem`. In the `l2c` case, seven L2 ways are locked down for page table storage.

S_{mmu} Discussion. All cases follow a decreasing curve with increasing interference periods. The TLB-pinned cases consistently show the lowest interference, followed by the OCM and L2 options. Importantly, these techniques – driven by the temporal specifications

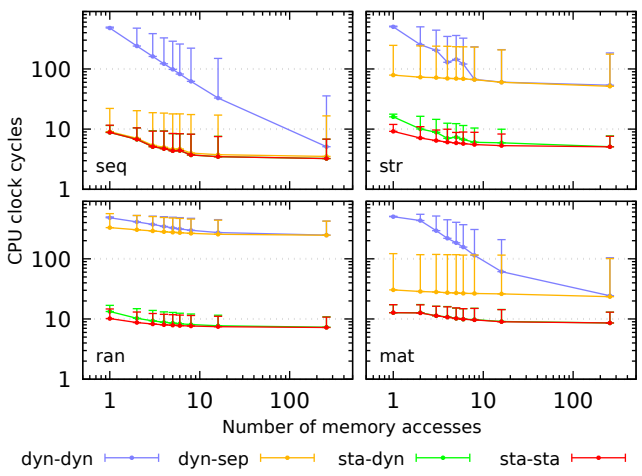

Figure 5: S_{mpu}'s memory access latency under different interference inter-arrival time. The four subfigures show the four benchmarks respectively. The horizontal axis is the interference inter-arrival time (in number of consecutive memory accesses), while the vertical axis is the memory access latency.

of MxU – all show significantly lower interference values than traditional, in-memory page table placements. This result is somewhat surprising: though TLBs are shared among applications, MxU's ability to place nodes at different levels in the cache hierarchy significantly decreases the impact of interference.

On S_{mpu}, the workload runs in a thread, and adversarial workload flushes all MPU regions in a different thread. The two threads will use different memory temporal specification and component combinations, and we report the memory access latencies under different interference periodicities. We run the system with `MXU_DETERMINISTIC` (for Static mappings) and `MXU_L2` which uses Dynamic mappings. In Figure 5 we compare the four test cases listed in Table 5.

Case	Adversarial	Benchmark	Component
`dyn-dyn`	Dynamic	Dynamic	Same
`dyn-sep`	Dynamic	Dynamic	Different
`sta-dyn`	Dynamic	Static	Same
`sta-sta`	Static	Static	Same

Table 5: The four test cases for S_{mpu}.

S_{mpu} Discussion. For S_{mpu}, the dynamic regions are generally significantly more susceptible to interference than the static regions. Further, dynamic regions can show even more interference when shared within the same component (the difference between `dyn-dyn` and `sta-sta`) as they share the same "cache" of dynamic entries. All of the Static workloads exhibit predictable execution without significant extra MPU misses. The Static lines have a decreasing trend along the x-axis mainly as the interference flushes normal application cache-lines which need to be reestablished, and *not* because of MPU misses. It is noteworthy that in the `str`'s `dyn-dyn` case, an inter-arrival time of 4 outperforms 5, due to the fact that 4 is a divisor of 16 which is the stride length. We don't show the case

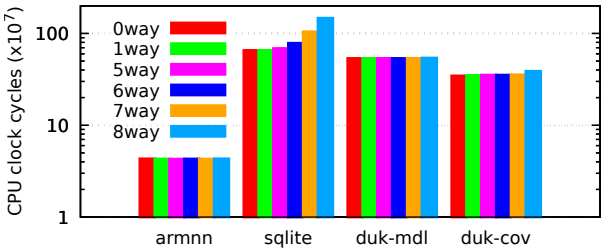

Figure 6: S_{mmu}'s L2 locked down ways and application running time. `0way-8way` means 0-8 L2 ways locked down respectively. The horizontal axis is the different applications, while the vertical axis is the applications' running time.

of interference across components with Static mappings as there is none.

5.3 Application Evaluation

In this section we present real-world application evaluations using the ARM's ARMNN, a neural network package, SQLite3, a database engine, and Duktape, a javascript engine commonly used in IoT systems. We use ARM CMSIS-NN V1.0.0 (`armnn`) to do image recognition using a CIFAR-10 configuration which has three convolution layers, then ReLU activation and max pooling layers, finally followed by a fully-connected layer. The input to the network is a 32x32 pixel color image which classifies to number 0-9. Sqlite V3.25.2 (`sqlite`) is run with a in-memory database that has one table, which has three columns containing integer primary key ID, string and integer respectively. The SQL workload inserts 16k lines into the table, deletes the lines with ID=3n, then updates the lines with ID=3n+1, then queries the lines with ID=3n+2, and at last deletes all lines in the table. Duktape V2.2.0 is run with two scripts: one is CPU-bound and picked from the Duktape testbench computing the Mandel pattern (`duk-mdl`), the other is from the jStat library that calculates covariance of two arrays containing 75000 numbers each (`duk-cov`). The total application running time is measured in all evaluations unless otherwise noted.

Application performance vs. locked down L2 ways. For S_{mmu}, L2 ways are locked down for page table preservation in `MXU_L2` with the `l2c` policy, and this reduces the L2 cache available to applications and may cause performance degradation. To investigate the impact, we lock down different numbers of L2 ways and run the four applications, and the results can be found in Figure 6.

Discussion. It is noteworthy that all the applications are long-running, thus their average-case and worst-case execution time are very close, making it hard to distinguish them from the each other; the standard deviation bar is also negligible. For this reason, we do not discuss worst-case and average-case performance separately in this section.

For S_{mmu}, locking fewer than five L2 ways for page tables has minimal impact on application performance, less than 4.9%. For applications that have a small working set such as `armnn` and `duk-mdl`, locking down even more ways does not take a heavy toll thanks to the L1 cache. However, the overhead impacts `sqlite` performance if the ways locked down exceed five, leading to 124.6%

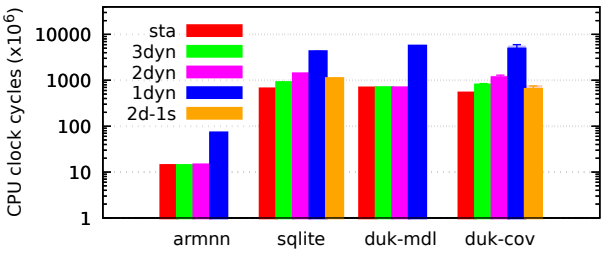

Figure 7: S_{mpu}'s Dynamic region number and application running time. `sta` means fully Static regions, `1dyn-3dyn` means 1-3 Dynamic regions respectively, and `2d1s` means running one half of the application workload in Static regions and the other half in Dynamic regions. `2d1s` is only shown for `sqlite` and `duk-cov`. The horizontal axis is the different applications, while the vertical axis is the applications' running time.

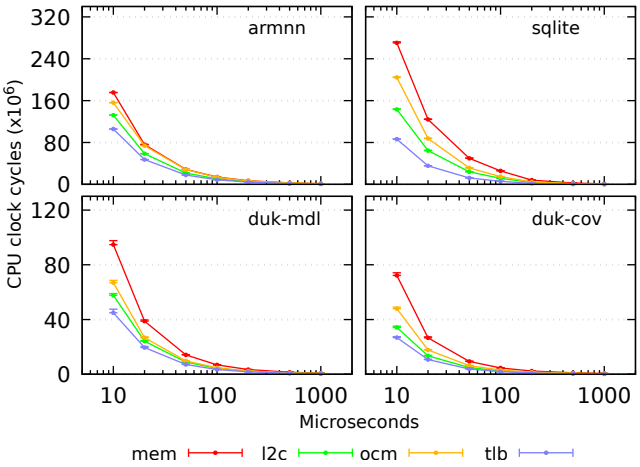

Figure 8: S_{mmu}'s application data side TLB stall time under different interference inter-arrival time. The four subfigures show the four applications respectively. The horizontal axis is the interference inter-arrival time, while the vertical axis is the data main TLB stall time.

slowdown if all are locked. This is due to the fact that `sqlite` has a large working set. On contrast, `duk-cov` accesses three arrays linearly and locking ways does not significantly impact performance thanks to L1 prefetcher.

Locking page-tables into L2 ways is an effective means of decreasing memory access latencies, but comes at the cost of effectively decreasing cache size for application data. Here we see that it varies per application how much this impacts performance. MxU relies on OCM and TLB pinning primarily, thus minimizing the need to use L2 ways. The admission control policies can reject allocation requests to result in too many pinned ways.

Application performance vs. dynamic region number. For S_{mpu}, the relationship between the available Dynamic region number and real-world application performance is investigated in this section, with results shown in Figure 7. `armnn` is run using internal SRAM; `sqlite`, `duk-mdl` and `duk-cov` are run with a 4MB heap in external SDRAM which uses 64kB pages, and each region holds two pages.

Discussion. For S_{mpu}, the number of Dynamic regions available does have an impact on performance, depending on the application type. For CPU-bound applications that have a small working set such as `armnn` and `duk-mdl`, the performance gap between three dynamic regions, and only static (`3dyn` versus `sta`) is negligible, at less than 1%. For memory-bound applications that have a larger working set such as `sqlite` and `duk-cov`, `3dyn` is less performant than `sta`, with 36.9% and 49.7% overhead respectively, and performance degrades sharply as the number of available regions shrink. In the case of `duk-cov`, livelocks occur if only one region is available to it due to double-word memory access instructions such as `ldrd` and `strd` that do accesses across region boundaries. The `1dyn` data shown for `duk-cov` is obtained by supplying additional compiler options to suppress the emission of these instructions, decreasing performance further. Thus, when `ldrd` and `strd` are used in an application, at least two Dynamic regions are required. As the `2d1s` case shows, even partially using Static regions can lower application execution time.

These results are somewhat surprising. Though dynamic regions do impose some overhead, using a small number of dynamic regions, especially if paired with some static allocations, can remove much of the dynamic overhead of MPU miss processing. Though we do not investigate it here, changing the memory range represented by each region is another means to control overhead.

Application performance vs. interference. To investigate the effectiveness of MxU under interference for real-world applications, we run an adversarial workload alongside the four applications. To make measurements accurate for `armnn`, we accumulate its results for 50 consecutive runs.

For S_{mmu}, we run the application and the adversarial workload which flushes L1 cache, L2 cache, and TLB in two different components. On *average*, many applications may not have significant performance degradation due to mutual TLB evictions. However, in the worst case, they effectively flush the TLB entries of a hard real-time application, and can do so at the rate of the minimum interrupt inter-arrival. Thus, to assess this impact, we flush all the caches and TLB to measure the worst case. To avoid impacting application performance only four L2 cache ways are locked down in the `l2c` case. We switch to the application first for the interference inter-arrival time, then switch to the adversarial workload to complete a cache flush, and then repeat the process until the application finishes running. To understand the MMU overhead, we use a Cortex-A9 integrated performance counter that directly measures the *data-side TLB stall overhead*, which we leverage to directly show MMU overheads. By using this performance counter, the overhead of TLB misses are isolated from other system overheads including instruction cache and non-page-table data cache misses. Figure 8 shows the application interference measurements in S_{mmu}.

S_{mmu} Discussion. For S_{mmu}, `tlb` has lower TLB stall cycles than `l2c` and `ocm` in all cases, which both control overhead more than `mem`, especially for cases where the interference is frequent, *e.g.* 10-1000 μs. Note that the maximum TLB interference is related to the minimum interrupt inter-arrival which can be quite small for some

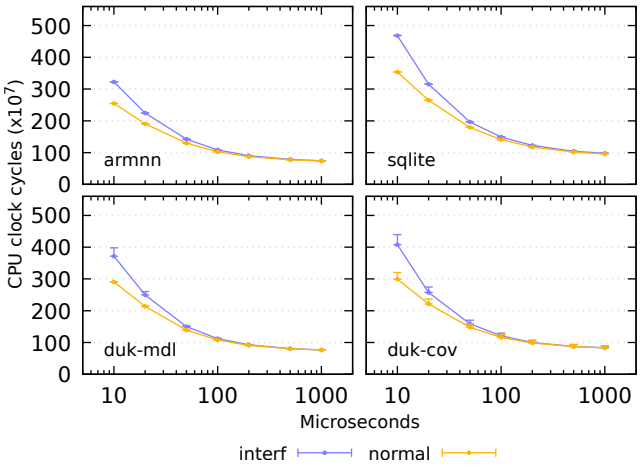

Figure 9: S_{mpu}'s application running time under different interference inter-arrival time. The four subfigures show the four applications respectively. The horizontal axis is the interference inter-arrival time, while the vertical axis is the application running time.

devices. Such high interrupt frequencies may originate from, for example, high-speed off-chip analog-to-digital converters (ADCs). For the `sqlite` application, a TLB stall improvement of 68.0% is reached with interference at a 10μs periodicity. For all cases, the TLB stall cycles have a decreasing trend along the x-axis, showing the expected decreased interference with longer inter-arrival time. The performance difference between `l2c` and `ocm` is due to their memory organization, which is not further discussed here.

For S_{mpu}, we run the application and the adversarial workload in the *same* component with three Dynamic regions available for them. Note that MxU *completely prevents inter-component interference* on MPU-based systems, so here we focus on studying contention for dynamic regions within a component (a much more predictable activity). We compare the case where the adversarial workload evicts all MPU regions (`interf`) with the case where it does not evict regions (`normal`). We use the same reciprocal running pattern described in S_{mmu}, and we run the adversarial workload for a fixed amount of time in every interference to cancel out its running time, then the difference between the two cases' application running time shows the impact of interference on execution, as shown in Figure 9.

S_{mpu} Discussion. For S_{mpu}, the applications running on Dynamic memory are susceptible to interference, especially when the application have a large working set (`sqlite` and `duk-cov`). However, in the common case where the inter-arrival time is more than 100 μs, the Dynamic overhead due to interference is only up to 6.1% for all cases. This means that Dynamic regions is practical for many best-effort applications, increasing their flexibility without much performance loss. For all cases, the application running time has a decreasing trend along the x-axis, showing decreased interference with longer inter-arrival time. We don't show the case for Static memory, nor inter-component interference as there is no MPU-induced interference.

6 RELATED WORK

MPU as cache. Though we know of no open-source, scientifically evaluated systems that treat the MPU as a per-process cache with dynamic regions, some systems have attempted this support. PXROS-HR [9], for the TriCore architecture, allegedly keeps caches of MPU contents for different processes. It allegedly allows user-level handling of MPU faults, and changing the mappings on the fly to emulate a larger number of regions. However, no in-kernel support of the dynamic swapping features are supplied, and no generic interface is provided across different architectures; it also does not provide any pointers to algorithms or data structures for region replacements. As this system is not open source, we cannot evaluate against it, nor attest to its capability.

Another implementation which employs per-process MPU cache and allows swapping of MPU entries is Emcraft's Cortex-M ucLinux port [5], which runs orders of magnitude slower than MxU in Composite as it must execute entirely from external DRAM. It has hard-coded kernel policy to only lock the stack segment for each process. Dynamic regions are tracked in a linear table, leading to $O(n)$ cost misses.

Memory access emulation. Memory access emulation was also proposed to circumvent the MPU region number or alignment constraints. In [2], automatic application compartments are deployed on an ARM Cortex-M4 platform with code generated by a modified LLVM backend. When a compartment makes an access to the stack segment that is not directly accessible by its MPU settings, the access is permission checked then performed within a fault handler routine. Though it allows accessing memory regions that are not covered by the current MPU region settings, it triggers a fault on every such memory access. This can cause significant overheads on program execution, while our MxU technique can largely avoid such overheads.

TLB lockdown and TLB coloring. TLB coloring was proposed by [18] to provide more predictable management of the TLB. In the work, they designed and implemented an intelligent allocator that takes physical page address to TLB entry mappings into account. Their allocator can guarantee that no two pages in different processes are mapped to the same TLB entry.

Some work also explored the benefits of TLB lockdown [10]. They conducted the work on an ARM-based processor and conclude that the static lockdown provides little TLB miss overhead.

In contrast to these previous works, MxU provides a unifying abstraction for predictable memory access control, that uses whatever hardware mechanisms are available, including TLB coloring and pinning. We compare not just the ability to decrease memory access execution times via TLB pinning, but also the utility of using data caches for page table nodes.

Cache partitioning. Cache partitioning is a standard approach to isolating the working sets of different processes [15, 19, 21]. If the cache lines of different processes are disjoint, the different working sets will not interfere with each other, thus reducing inter-application thrashing. This research focuses on application data, and not on the access control mechanism data-structures which are accessed on *each application load/store*. In [16], compiler infrastructure is leveraged to arrange application code and data to

distinct cache partitions so that the real-time portions' WCET is improved. In [8], intra-application code partitioning is also considered to map different functions to different cache lines, which reduces the instruction cache misses across function call boundaries. Such approaches are complementary to MxU.

Multi-core. Cache partitioning and TLB coloring can also be applied when there are multiple processors. In [11] and [14], last-level cache (LLC) partitioning is considered in multi-core systems to increase the schedulability of applications. In [25] this is also considered but only with regards to hot pages, which reduces online analysis overheads and is less constrained than full page coloring. In [1] and [24], the allocation techniques are extended to mixed-criticality (MC) systems which are also multi-core. In [12], cache allocation is dynamically performed, and virtualization extensions are leveraged to achieve this. Their techniques focus on application data and is orthogonal to our MxU approaches. Applying both may help avoid data cache pollution across processors or different criticality levels when page table walks are triggered. MxU does not currently address multi-cores, however this provides interesting future work for both MMU and MPU.

Scratchpad management. Some existing works allow the applications to allocate scratchpad memory for performance boost. In [3], frequently used code and data pages are dynamically mapped to the scratchpad, resulting in a 32% energy saving and a 47% acceleration. In [20], the latencies of moving code and data to scratchpad is hidden by a synergy of dedicated hardware and a modified LLVM, which improves WCET. In [22], integer linear programming and heuristics is applied to make static scratchpad allocations that can minimize the task's WCET. Other research also integrates scratchpad management into the compiler infrastructure [13], using graph coloring to allocate application arrays to the scratchpad. Unlike MxU, these works still put emphasis on application data placements and not on access control mechanism data structures.

Temporal specifications. Some recent existing work also explore the possibility of assigning temporal specifications to memory allocations. In [6], a deterministic memory abstraction similar to MxU is described. However, their abstraction are dedicated to MMU-based systems, and requires both OS and hardware extensions. Also, multiple levels of predictability are lacking in their temporal specification. On the contrary, our MxU abstraction remains uniform across MMU and MPU, and accommodates more than two levels of predictability.

7 CONCLUSIONS AND FUTURE WORK

This paper introduces MxU, a memory access control abstraction that generalizes MMUs and MPUs, integrates temporal specifications into the memory allocation API, and ensures flexible memory management, good average-case performance, and the ability to tightly bound the impact of memory access control on the WCET on tasks. For MMU-based systems, MxU tighten bounds on the TLB stall cycles by enabling the control of the cache placement of page tables, reducing application TLB stall by up to 68.0% under 100kHz interference. For MPU-based systems, MxU demonstrates the use of Dynamic regions to provide a virtually unlimited number of regions, enabling flexible dynamic memory management with an application overhead of down to 1% and up to 6.1% with 10kHz interference. Given the modern security requirements that are pressuring embedded and IoT systems, we believe that MxU enables more feature rich microcontrollers, and more realistically predictable microprocessors.

This paper addresses neither multi-core nor program data related cache or scratchpad management. Combining these facets with MxU certainly leads to even more variants of temporal specifications, which we reserve as a direction for interesting future work.

REFERENCES

[1] Micaiah Chisholm, Bryan C Ward, Namhoon Kim, and James H Anderson. 2015. Cache Sharing and Isolation Tradeoffs in Multicore Mixed-criticality Systems. In *RTSS*.

[2] Abraham A. Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. ACES: Automatic Compartments for Embedded Systems. In *USENIX SEC*.

[3] Bernhard Egger, Jaejin Lee, and Heonshik Shin. 2008. Scratchpad Memory Management in a Multitasking Environment. In *EMSOFT*.

[4] Kevin Elphinstone and Gernot Heiser. 2013. From L3 to seL4 what have we learnt in 20 years of L4 microkernels?. In *SOSP*.

[5] Emcraft. 2019. ucLinux: https://github.com/EmcraftSystems/linux-emcraft, retrieved 4/12/19.

[6] Farzad Farshchi, Prathap Kumar Valsan, Renato Mancuso, and Heechul Yun. 2018. Deterministic Memory Abstraction and Supporting Multicore System Architecture. In *ECRTS*.

[7] Phani Kishore Gadepalli, Robert Gifford, Lucas Baier, Michael Kelly, and Gabriel Parmer. 2017. Temporal Capabilities: Access Control for Time. In *RTSS*.

[8] Amir H. Hashemi, David R. Kaeli, and Brad Calder. 1997. Efficient Procedure Mapping Using Cache Line Coloring. In *PLDI*.

[9] HighTec EDV-Systeme. 2019. PXROS-HR: https://hightec-rt.com/en/products/real-time-os.html, retrieved 4/12/19.

[10] Takuya Ishikawa, Toshikazu Kato, Shinya Honda, and Hiroaki Takada. 2013. Investigation and improvement on the impact of TLB misses in real-time systems. In *OSPERT*.

[11] Hyoseung Kim, Arvind Kandhalu, and Ragunathan Rajkumar. 2013. A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems. In *ECRTS*.

[12] Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. 2019. Deterministic Memory Hierarchy and Virtualization for Modern Multi-core Embedded Systems. In *RTAS*.

[13] Lian Li, Lin Gao, and Jingling Xue. 2005. Memory coloring: A compiler approach for scratchpad memory management. In *PACT*.

[14] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. 2013. Real-time Cache Management Framework for Multi-core Architectures. In *RTAS*.

[15] Sparsh Mittal. 2016. A Survey of Techniques for Cache Locking. *ACM Trans. Des. Autom. Electron. Syst.* (2016).

[16] Frank Mueller. 1995. Compiler Support for Software-based Cache Partitioning. In *LCTES*.

[17] Runyu Pan, Gregor Peach, Yuxin Ren, and Gabriel Parmer. 2018. Predictable Virtualization on Memory Protection Unit-based Microcontrollers. In *RTAS*.

[18] Shrinivas Anand Panchamukhi and Frank Mueller. 2015. Providing Task Isolation via TLB Coloring. In *RTAS*.

[19] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO*.

[20] Muhammad Refaat Soliman and Rodolfo Pellizzoni. 2017. WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching. In *ECRTS*.

[21] G. E. Suh, L. Rudolph, and S. Devadas. 2004. Dynamic Partitioning of Shared Cache Memory. *J. Supercomput.* (2004).

[22] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. 2005. WCET centric data allocation to scratchpad memory. In *RTSS*.

[23] Qi Wang, Yuxin Ren, Matt Scaperoth, and Gabriel Parmer. 2015. Speck: A Kernel for Scalable Predictability. In *RTAS*.

[24] Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. 2013. Making Shared Caches More Predictable on Multicore Platforms. In *ECRTS*.

[25] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards Practical Page Coloring-based Multicore Cache Management. In *EuroSys*.