Intermediate Address Space: virtual memory optimization of heterogeneous architectures for cache-resident workloads

QUNYOU LIU, Embedded Systems Laboratory (ESL), École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

DARONG HUANG, Embedded Systems Laboratory (ESL), École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

LUIS COSTERO, Dpto. of Computer Architecture and Automatics, Universidad Complutense de Madrid (UCM), Spain

MARINA ZAPATER, Institute of Reconfigurable & Embedded Digital Systems (REDS), School of Engineering and Management Vaud, HES-SO University of Applied Sciences and Arts Western Switzerland, Switzerland

DAVID ATIENZA, Embedded Systems Laboratory (ESL), École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

The increasing demand for computing power and the emergence of heterogeneous computing architectures have driven the exploration of innovative techniques to address current limitations in both the compute and memory subsystems. One such solution is the use of *Accelerated Processing Units* (APUs), processors that incorporate both a *central processing unit* (CPU) and an *integrated graphics processing unit* (iGPU).

However, the performance of both APU and CPU systems can be significantly hampered by address translation overhead, leading to a decline in overall performance, especially for cache-resident workloads. To address this issue, we propose the introduction of a new *intermediate address space* (IAS) in both APU and CPU systems. IAS serves as a bridge between *virtual address* (VA) spaces and *physical address* (PA) spaces, optimizing the address translation process. In the case of APU systems, our research indicates that the iGPU suffers from significant *translation look-aside buffer* (TLB) misses in certain workload situations. Using an IAS, we can divide the initial address translation into front- and back-end phases, effectively shifting the bottleneck in address translation from the cache side to the memory controller side, a technique that proves to be effective for cache-resident workloads. Our simulations demonstrate that implementing IAS in the CPU system can boost performance by up to 40% compared to conventional CPU systems. Furthermore, we evaluate the effectiveness of APU systems, comparing the performance of IAS-based systems with traditional systems, showing up to a 185% improvement in APU system performance with our proposed IAS implementation.

Furthermore, our analysis indicates that over 90% of TLB misses can be filtered by the cache, and employing a larger cache within the system could potentially result in even greater improvements. The proposed IAS offers

a promising and practical solution to enhance the performance of both APU and CPU systems, contributing to state-of-the-art research in the field of computer architecture.

CCS Concepts: • **Software and its engineering** → **Virtual memory**; • **Computer systems organization** → *Multicore architectures*.

Additional Key Words and Phrases: Computer architecture, CPU, GPU, virtual memory, TLB, Cache

1 INTRODUCTION

In the post-Moore era, modern computer systems face increasingly diverse workloads and ever-changing computing demands. The explosion of data generated by IoT devices, social networks, and other sources makes applications more computationally intensive and memory-hungry by the day [21]. Server workloads such as social network analytics, web search engines, and biomedical data processing, and therefore their most crucial kernel, graph processing, have gained widespread prominence in contemporary applications [8]. CPU-only systems are increasingly facing computing performance and energy efficiency issues. Therefore, to cope with increasing computing requirements, specialized computing units with different acceleration functions, such as *Graphics Processor Units* (GPU) and *Data Processing Units* (DPU), have been developed and integrated into the systems.

These heterogeneous computing units collaboratively work to meet the challenges of varied and intensive workloads. However, heterogeneous architectures face the problem of data transmission overhead, with some works reporting more than 50% GPU performance loss [8] due to this issue. In 2010, AMD proposed the Deep-Integrated Accelerated Processing Unit [14] to alleviate the data transfer overhead of traditional heterogeneous systems. An APU is a system that comprises a CPU and an iGPU, both integrated on the same die and sharing certain processing units, including the memory controller and last-level cache. The iGPU, as a co-processor of the CPU, executes specialized computing tasks upon receiving instructions from the CPU. In 2022, AMD revealed the first APU data center products in the world, showing great potential in the future [2].

Although the APU system alleviates the data transmission problem, the rapidly increasing memory capacity of such systems, already reaching terabytes [11], results in a growth in the memory address translation overhead between virtual and physical memory. Virtual memory is a crucial feature of modern computer systems. It provides processes with the illusion of a private memory space, which is maintained and converted into the PA space by the operating system and *memory management unit* (MMU). This is particularly important for modern operating systems and applications but also brings-in some problems. Among them, VA to PA translation can consume more than 30% of overall system performance for server workloads [25] and exhibits an increasing trend.

To alleviate the memory address translation overhead, researchers in [11] introduced a proof-of-concept IAS into a CPU-only system. However, this IAS technique is just a theoretical proposal, never evaluated and implemented in a full-system cycle-accurate simulator able to run full workloads on top of an operating system, and lacks of support for heterogeneous systems. Therefore, to solve the above challenges, this paper proposes the following contributions:

- We introduce a new IAS approach for both CPU and APU systems. The proposed IAS has two stages of MMU, namely front-end and back-end MMU. The front-end MMU is responsible for converting VAs into *intermediate addresses* (IAs), while the back-end MMU translates the
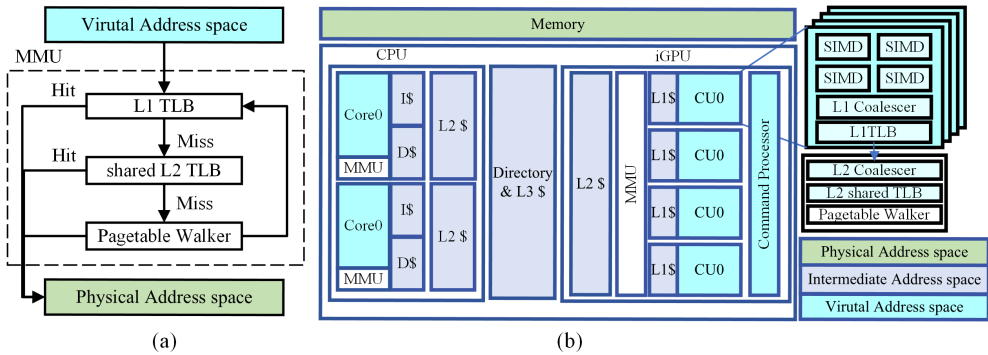
Fig. 1. (a) Traditional MMU design with two levels of TLB and a page table walker and (b) its use in the APU architecture. $ is a common sign used to denote cache.

IAs to PAs. By adopting IAS into the computer system, we shift the bottleneck of address translation from the core side to the memory controller side, allowing us to increase the performance of cache-resident workloads.

- We extensively evaluate our proposed IAS architecture by modifying a state-of-the-art cycle-accurate full-system simulator gem5 [10] to carefully investigate its performance in both CPU and APU systems. To our knowledge, this is the first research work in the field to implement and evaluate IAS with a complete system cycle-accurate architectural simulator capable of running full applications. We target to release this complete IAS simulation infrastructure as open source within the gem5 framework to enable further research in this area.
- By comparing a conventional system to a system with IAS, we show that the use of an IAS can improve the overall performance of CPU systems by up to 40% and achieve up to 185% improvements in APU systems.
- Furthermore, we explore the architectural design options for the IAS system. By varying the size of the cache, we investigate the impact of the cache on the IAS system, showing how a properly sized cache can filter more than 90% of the TLB misses. By varying the size of the TLB for conversion from IA space to PA space, we show the importance of a suitable TLB size and cache coverage, which can reduce more than 95% TLB misses and improve overall performance by more than 25%.

The remainder of this paper is organized as follows. Section 2 provides the background on VA and virtual-PA translation, introduces the APU system, and the motivation for our approach. Section 3 explains the IAS and its implementation in both CPU and APU systems. The following two sections describe our evaluation methodology and the results obtained. Finally, we conclude with a summary of our findings and potential future developments in Section 6.

2 BACKGROUND AND RELATED WORK

2.1 Virtual memory in modern systems

Contrary to the original single-tasking Operating Systems (OS) that only supported the execution of a single program at the same time, modern multi-tasking operating systems running on current CPUs allow the execution of multiple independent processes simultaneously. Although this improves drastically the user experience, it has a huge impact on how the OS manages the different processes, as each of them has to be mapped to a different memory location, and therefore, uses

different memory addresses. To reduce the burden on programmers, the virtual memory abstraction allows every process in the system to use the same memory addresses, being the OS responsible for translating the addresses used by each process (VA) to a different and unique address in the memory (PA).

The part of the CPU that is responsible for this translation is called the *Memory Management Unit* (MMU). A general structure of an MMU is illustrated in Fig. 1(a). The interior of the MMU consists mostly of two types of components: *Page table Walker* (PTW) and *Translation Lookaside Buffers* (TLB). The PTW operates by following a chain of page table entries, each of which describes a portion of the VA space and its corresponding PA. The page table entries are stored in a hierarchical structure known as the multi-level page table, which is managed by the OS. In multi-level page tables, the VA space is divided into several levels, each level containing a separate page table. When a VA needs to be translated to a PA, the translation process starts at the highest level page table and *walks* its way down through the hierarchy until it reaches the final lowest level page table. The PTW is also responsible for ensuring that memory accesses are properly authorized and directed to the correct physical memory location. It does this by checking the page table entries to ensure that the program has the necessary privileges to access the memory location, and by checking for any memory access violations or errors.

However, translating the PA every time is not efficient. Therefore, the second component, the TLB, is introduced and responsible for storing recently utilized VA to PA mappings. The TLB serves as a cache to speed up the VA-to-PA translation process so that if the same VA is accessed in a short period of time, the MMU can quickly retrieve the corresponding PA from the TLB without having to perform the entire translation process. However, due to area and latency constraints, the size of TLBs is usually small, compared to the size of caches, and often a VA can not be found in the TLB (called *TLB miss*). In these cases, the MMU will then perform the translation using the page tables stored in the main memory, which is a slow process due to the long latency of the modern main memory structure. Results from [24] show that a TLB miss can incur an average of 135 cycles overhead on page walks on a modern x86-64 architecture, especially bottlenecking memory-intensive workloads. The trend of having larger and larger caches to enhance data locality results in lower TLB cache coverage and therefore higher TLB misses, with the subsequent overhead in memory translation. Furthermore, the use of virtual memory today is not limited to CPUs but rather frequently used in accelerators and co-processors, such as APUs, where the same bottlenecks can be found, yet more exacerbated.

2.2 APU architecture

Fig. 1(b) illustrates the APU's general architecture [8], comprising a multi-core CPU and an integrated GPU (iGPU). Each core on the CPU has its own private data and instruction L1 caches and a unified private L2 cache. The iGPU component includes a command processor, and several Compute Units (CUs). The command processor is an embedded microprocessor within the iGPU that is capable of performing the majority of tasks traditionally handled by the operating system. It is responsible for receiving instructions from the CPU, keeping track of GPU states, and sending interrupts to the CPU. The CUs are in charge of computing-related duties through the multiple SIMDs inside. All CUs share the same L2 cache.

The advantage of APU architecture with respect to a system with a discrete GPU is its shared memory address space, reducing programming complexity and performance overhead by sharing the memory blocks and enabling the CPU and iGPU to access the same data simultaneously. Compared with a system comprising a discrete GPU (directly attached to the system bus), the APU allows for fine-grained data sharing between the CPU and iGPU, offering significant benefits in various computing scenarios, as well as decreasing the power consumption [27]. Furthermore, due

Table 1. APU System Configuration

Component	Parameter	Value
CPU	—	Out-of-order @2GHz
	L1 - Data cache	32 KB, 2 cycles latency
	L1 - Instr. cache	32 KB, 1 cycle latency
	L2 cache	2 MB, 24 cycles latency
iGPU	—	4 CUs, 64 lanes per CU @1GHz
	L1 cache - Scalar	32 KB, 1 cycle latency
	L1 cache - SQC	32 KB, 1 cycle latency
	L1 cache - TCP	16 KB, 4 cycles latency
	L2 cache - TCC	256 KB, 8 cycles latency
L3 cache - shared	—	16 MB, 20 cycles latency
TLBs	private	32 entries, 1 cycle latency
	L2 TLB	512 entries, 10 cycle latency

to these features, the APU shows more potential with respect to other heterogeneous systems for supporting IAS and exhancing the performance of server workloads.

2.3 Performance advantages of the APU and its large address translation overhead

The escalating demand for server workloads, particularly for social network analytics, web search engines, and biomedical applications, underscores the growing significance of their core component, graph processing [8]. Graph processing typically uses sparse data formats such as Compressed Sparse Row/Column (CSR/CSC) to manage a large amount of data efficiently. Then, *Sparse Matrix-Vector Multiplication* (SpMV) is used to manipulate and process data. It is well-known that SpMV, and of course, graph processing, are computing and memory-intensive tasks in the field [8]. Therefore, heterogeneous systems, (like APU systems), have been proposed to bring performance improvements for such workloads by introducing dedicated acceleration units, i.e., iGPU. Compared with CPU, APUs can exhibit greater benefits.

To closely examine the advantages of APU over CPU and at the same time evaluate the room for improvement, we performed an analysis of the performance obtained from the state-of-the-art graph benchmarks Pannotia [7] running with the gem5 architectural simulator [5]. The gem5 simulator is configured to mimic a real system. Table 1, shows the parameters used to run the experiments, most of which come from the default simulator settings, and some which are scaled down to better match the behavior of a real system.

Fig. 2 gives the speedup of the APU system against the CPU-only system for each application in the Pannotia benchmark. The results depicted in the figure unequivocally show that the APU system surpasses the CPU system by a considerable margin, with an overall performance improvement of approximately 132 times. Based on the performance evaluation of APU and CPU systems using the Pannotia benchmark, it is evident that the GPU-based APU system substantially outperforms the traditional CPU system, even when handling irregular GPU workloads. This observation underscores the significance of embracing GPU-accelerated architectures, especially in application domains characterized by irregular workloads, in order to leverage the exceptional performance and efficiency advantages they provide.
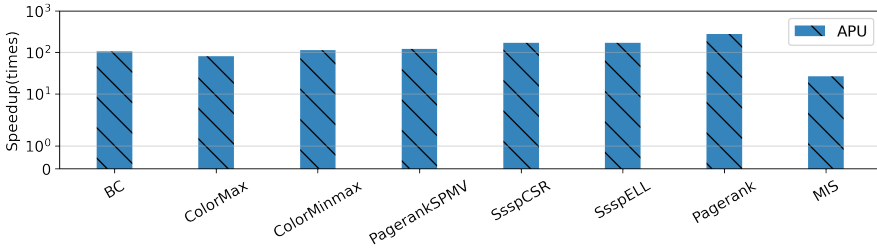
Fig. 2. Execution time improvement using APU against CPU.

Despite the tremendous advantages brought by the APU system, the irregular memory access patterns of sparse data lead SpMV and graph processing applications to have a large memory address translation overhead and performance degradation [11, 16, 29].

The fundamental reason behind this phenomenon is the fact that while the cache sizes have increased over the decades to mitigate the impact of data cache misses, TLBs have not followed the same growth, and the extent to which physical memory addresses can be accommodated within the TLB has not kept the pace, thus, leading to low TLB coverage on the cache and a large number of TLB misses and page walks. Due to the performance cost of a PTW, this eventually introduces large memory address translation times and performance overhead. Furthermore, a larger cache makes the problem more severe, as the PTW needs to consume additional cycles to access the cache hierarchy to map virtual memory addresses used by applications to physical memory addresses used by the hardware. The work in [11] shows that the address translation overhead is increasing in the existing systems with increasing cache sizes.

This phenomenon can be observed in Fig. 3, where we show the number of misses per thousand instructions (MPKI) for both TLB and last level cache (left y-axis) for each application of the Pannotia benchmark. The TLB misses induce MMU overhead, which we define as the percetage of total CPU cycles spent on address translation, and that is plotted as a red dashed line in the right y-axis.

On the one hand, benchmarks like *BC*, *Colormax*, and *Colormin* have relatively low MPKI values for both TLB and Cache, suggesting efficient utilization of memory resources. The low MPKI values may be indicative of regular or semi-regular memory access patterns and good spatial and temporal data locality. These properties help to minimize memory misses and thereby increase the overall performance of the benchmarks. On the other hand, benchmarks like *FW*, *PagerankSPMV*, *Pagerank*, or *SsspELL*, have higher MPKI values for either TLB, cache, or both, which suggests more complex or irregular memory access patterns. This could be due to the nature of the problems they are solving, which may involve complex data structures or irregular algorithms. High MPKI values indicate poor spatial and temporal data locality, leading to increased cache and TLB misses, which shows great pressure for cache and MMU.

Furthermore, we compared the experimental results with a system featuring an Ideal MMU (i.e., an MMU with 0 cycle address translation overhead), to investigate the TLB misses induced overhead. The overhead is depicted as the red dash line in Fig. 3, with the right y-axis. *PagerankSPMV* and *SsspELL*, as the two benchmarks with the highest TLB missrate, show the most dramatic overhead of approximately 50% and 37% respectively, consistent with their previously high TLB MPKI values. These overheads underscore the importance of effective memory management for irregular workloads and reveal that the APU is significantly affected by address translation, leading to a degradation in overall performance.
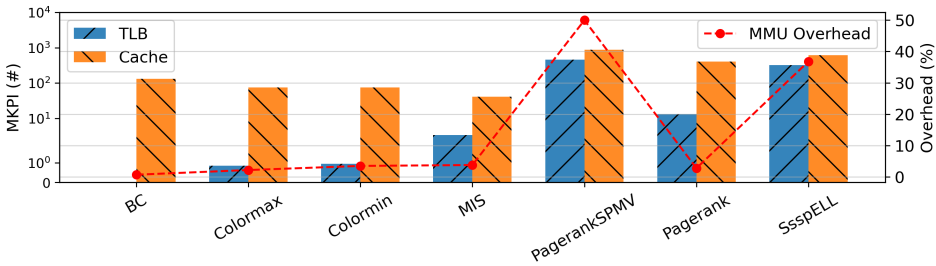
Fig. 3. Access Pattern and Address Translation Overhead for APU for the Pannotia benchmarks.

Similar behavior is present in CPU systems. For example, [11, 31] show that traditional CPU-only systems suffer from around 17% address translation overhead, reaching more than 30% as the size of the cache increases.

Recognizing the promising capability of the APU system in the computation of irregular workloads and similar access pattern problems they are facing, we are motivated to implement the IAS into the APU system. The anticipated benefits of this integration will be elucidated in the subsequent sections. In summary, the results show that address translation is a serious problem in modern systems (both CPU and APU). In the next section, the state-of-the-art to mitigate the translation overhead will be discussed.

2.4 State-of-the-art

Researchers from around the world have proposed a variety of solutions to address the translation overhead problem. First, to extend the address range the TLB can reach, Kwon *et al.* propose a framework called Ingens [19] for huge page automatic support in operating systems. Ingens promotes or demotes huge pages according to the number of physically resident pages and their access frequency. The experimental results demonstrate that Ingens has the ability to mitigate tail latency and memory bloat, significantly improving performance for essential applications such as Web services and Redis [19]. However, the idea of huge pages can also cause several other problems, which, for example, can lead to internal fragmentation and memory waste.

To improve the efficiency of address translation of the GPU system, researchers propose a second method to address the address translation overhead called Mosaic [3]. They propose using address-translation-aware caches and memory management algorithms that significantly reduce address-translation overhead. However, the proposal cannot handle intermediate sizes larger than 4kB. [20] shows that Mosaic does not work well with low-contiguity pages and struggles with the workloads of large memory footprints due to the 2MB page-size limitation.

Several researchers have proposed the idea of virtual caches. In particular, Wood *et al.* [28] propose the use of a global VA space for addressing caches. However, their approach involves translations from virtual to global virtual address spaces using fixed-size program segments. More flexible paging systems have since replaced these segments. Their simulation methodology is based on a trace-driven simulator and cannot estimate the overall full-system performance. A similar idea is also introduced in GPU-addressing. Yoon *et al.* [29] proposes to turn the physical cache system into a virtual cache system for GPU systems. Virtual caches are designed to take the TLB off the critical path, thereby moving address translation to the memory side. In such systems, processes are addressed in the cache using their private VA as a namespace. Using a virtual index cache, the GPU can immediately retrieve data from the cache, and, therefore, the cache can filter most of the TLB misses. This strategy can significantly reduce address translation overhead [29].

However, the method requires the system to handle the problem of control logic required to resolve synonyms and homonyms across VAs (i.e., multiple VAs mapping to the same PA, or a single VA mapping to multiple PAs). Consequently, virtual hierarchies are difficult to incorporate into modern systems [11].

The use of IAs is another promising proposed solution. Zhang *et al.* [30] propose an idea of an IA space that translates the VAs at 256MB granularity. However, it works only well for GB-scale memory. When faced with TB-scale memory, it seems powerless. Although they simulate performance using a complete system simulator called Mambo [6], it is not open source, only available on limited platforms, and only supports the PowerPC architecture. Hajinazar *et al.* [13] propose an IA space that consists of fixed-size virtual blocks for application use. However, to implement such an address space inside the system, additional tools, and software modifications are needed. The simulation is based on a modified DRAM simulator, called Ramulator [17], which simulates the performance by using the collected trace of representative regions of the benchmark, which cannot reflect the overall performance improvement.

Designed for the modern data center system, Sid *et al.* propose an IA approach called the Midgard address space [11]. This idea uses variable VMA sizes in a flexible manner, converting the two address spaces into three address spaces. The main contribution of this work is that, by using three different address spaces in the system, the address translation overhead can be greatly reduced.

The mapping mechanism from VAs to IAs is depicted in Fig. 4(a). Midgard IA space employs the operating system concept of *Virtual Memory Areas* (VMAs) to produce a single IA space in which all processes' VMAs can be mapped uniquely. A VMA is a contiguous range of memory used by an operating system to manage and keep track of the VA space allocated to processes. When many processes are present in a system, the shared library will be assigned to the same IA, while the rest of the private data will be assigned to other IAs. Each private VA will be mapped to a unique IA through the IA mapping process. Unlike the conventional system, which translates VAs to PAs in a unit of fixed size, the IAS system translates VAs into IAs in the unit of VMAs. However, in real-world computer systems, programs use much fewer VMAs than pages to represent their VA space. Thus, fewer hardware resources are required during the transition from VA to IA space in the granularity of VMAs than with conventional virtual-physical translation. Similarly, the number of mappings from VMAs to *Intermediate Memory Addresses* (IMAs) is fewer, and it brings fewer TLB misses for front-end translation, which is lightweight. At the back-end translation from IA space to PA space, due to the translation granularity of pages, more hardware resources are needed, like TLB entries and multilevel page tables. Besides, facing TLB miss at the back-end MMU, the PTW needs to walk through the page table level by level, first from the cache and then from memory if the cache misses. Furthermore, the latency of accessing memory is much higher than that of accessing cache, which is a heavyweight translation. However, thanks to a larger cache, it can filter a high number of TLB misses. We show the impact of cache in the next section. By implementing this kind of mapping mechanism using QEMU [4], [11] evaluates the full potential of Midgard address space, which shows more than a 30% decrease in address translation overhead compared to the conventional system. However, their work is based on tracing information from emulation and using average memory access time as a metric, which can not show the cycle-accurate result and overall system performance improvement. To evaluate the overall performance improvement, we introduce IAS into the cycle-accurate full-system simulation gem5. We will discuss the detailed implementation of IAS with a little more depth in the next section.

3 INTERMEDIATE ADDRESS SPACE FOR BOTH CPU AND APU

In this section, we start by introducing the implementation of an IA space for CPU systems. Then, a description of how to extend this idea to APU systems is explained.
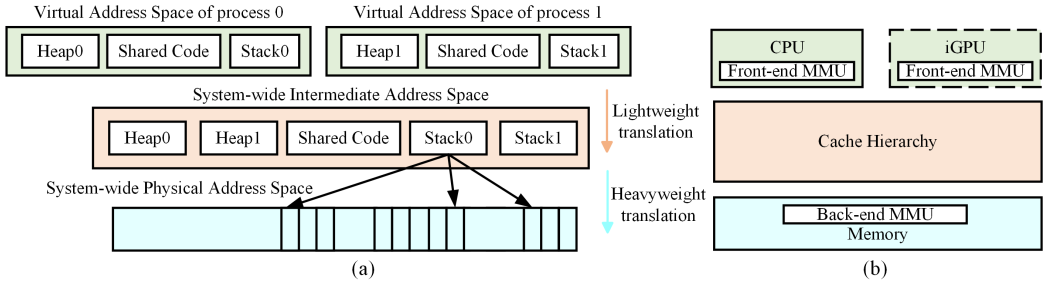
3.1 Intermediate address space for CPU

Fig. 4. (a) Address Mapping and (b) System Architecture for Intermediate-address Space Systems.

Fig. 4(b) shows how the high-level abstract architecture of IAS is implemented in this work. We first introduce the CPU-only system, which comprises the three blocks on the left of Fig 4(b) (without the iGPU block in the dashed box). The IAS comprises three address spaces: VA space, IA space, and PA space. The processor's core, cache, and memory utilize VAs, IAs, and PAs, respectively.

To handle translation, two stages of MMUs are introduced into the system, each responsible for a different phase of the translation process. Hereafter, we shall refer to the MMU that converts VAs into IAs as the front-end MMU and the MMU that converts IAs into PAs as the back-end MMU. During the process of address translation, initially, the CPU translates the VA into an IA, using it to access the cache and fetch the desired data or instructions. In the event of a cache miss, the back-end MMU converts the IA into a PA and retrieves the data from the memory. By incorporating an IA into the CPU system, IAs filter heavy-weight translations to PAs, limiting them only to memory references that miss in the last-level cache. This part and how we have implemented it in our proposed IAS is carefully explained in section 3.3.

3.2 Unified Intermediate Address Space for APU

The use of an IAS in the APU system follows the same ideas as in the CPU system. As mentioned in the previous section, IAS can enhance performance when used together with a large cache system. The motivation for introducing an IA space into APUs is supported by three key ideas:

- In an APU system, the CPU and iGPU share the same address space. This shared address space serves as a prerequisite for introducing the IA into the APU. By utilizing a shared address space, the system eliminates the need for address conversion between the CPU and iGPU.
- Second, GPU tasks typically comprise a single process or a small number of processes with hundreds or even thousands of threads. This implies that when the GPU is in operation, it only needs to handle a limited number of VMAs. Employing IA translation reduces the quantity of address translation correspondences, thus alleviating the load on the front-end translation.
- In an APU system, the CPU and iGPU share the same memory controller. This means that only one back-end MMU is required on the memory controller side to convert the IA to a PA. Consequently, the TLB can be shared by both the CPU and iGPU, resulting in increased efficiency. Moreover, the large cache can filter more MMU requests, leading to further performance improvements.
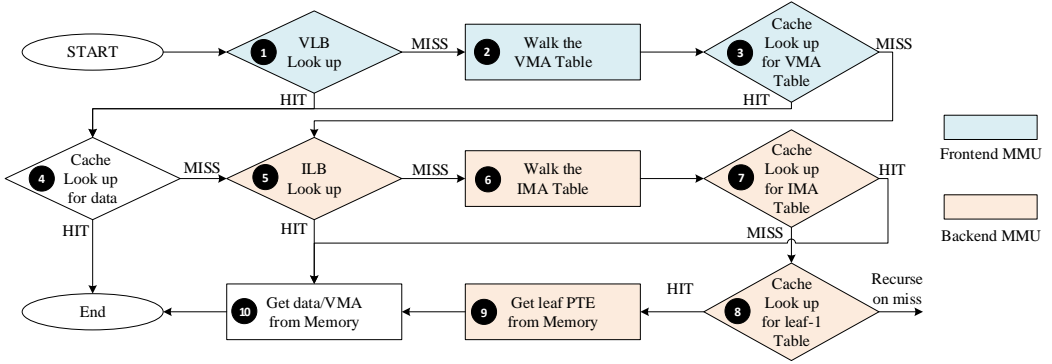
Fig. 5. Intermediate address space finite state machine.

These three factors offer a theoretical benefit in introducing IAS into the APU system. As shown in Fig.1 (b), we therefore introduce the IAS into the cache hierarchy, i.e., adding the dashed box on the top left of the figure, which comprises the original iGPU model augmented with a front-end MMU for the iGPU.

3.3 Intermediate Address Space MMU design

An experimental evaluation of this proposal is performed in the cycle-accurate gem5 simulator. However, to properly simulate the IAS for the system, a set of changes to gem5 are needed:

- We modify the conventional MMU to adapt the behavior of the proposed front-end MMU responsible for the VA to IA translation.
- We modify the cache hierarchy and structure in the simulator to make it IAS-compatible.
- We modify the memory controller to incorporate the back-end MMU inside, enabling the IA to PA address translation.
- We design new TLB entries for front-end and back-end MMU, respectively, to be compatible with the IAS.
- Finally, to evaluate the overhead of the MMU, we tune the conventional MMU to emulate the behavior of the ideal MMU.

Figure 5 provides a comprehensive description of how the implemented IA space works within the system. The process begins with a *virtual look-aside buffer* (VLB), labeled ① in the figure. The VA is mapped to an IA in the unit of VMAs partially stored in VLB. This phase is light-weighted. If the VA is found in the VLB, the system attempts to fetch data from the cache with the IA, labeled as ④ in the figure. If there is a cache hit, the cache returns the data from the desired address, and the process ends.

If the VLB cannot initially locate the required VA translation, the system must call on the PTW from the front-end MMU to have a VMA table walk for it (②). If the desired address is successfully found in the cache, the translation process is complete at ③, and the system proceeds to fetch data from the cache ④. In contrast, if the address is not found in the cache at ③, resulting in a cache miss, the system needs to fetch data from the next memory level, the main memory. The system enters the back-end translation. The data stored in memory are physically addressed. The system turns to the back-end MMU to translate the IA to PAs. The back-end MMU receives requests only when a cache miss happens. It works using the same clock domain as the memory controller, it is triggered every time a memory request happens, and shares the same requests as the memory

controller, with a different execution order. The back-end MMU consists of two parts, TLB and PTW. The back-end MMU, firstly, checks the *intermediate look-aside buffer* (ILB), labeled ⑤ in the diagram. If hit, the system will fetch the data from memory by its PA (⑩). On the other hand, if missed, the system needs to walk the multi-level IMA table (⑥). The back-end MMU, first of all, inquires about the cache for the desired *page table entry* (PTE). If hit, it fetches the data from the memory as in the previous step. If missed, it needs to fetch PTE level by level, similarly to conventional systems. These two stages of address translation work together to implement the virtual to physical address translation. Please note that these two stages operate autonomously, with their management being independent. Modifications or invalidations within the front-end or back-end translation mechanisms do not affect the other stage, in alignment with the address translation mechanism illustrated in Figure 4(a). The Translation Lookaside Buffer (TLB) functions solely as a cache for address mapping, with the mapping relationship exclusively managed by the operating system kernel. The coordination between address spaces is facilitated through a collaborative interplay between software and hardware components.

4 EVALUATION METHODOLOGY

4.1 Simulator Platform

We utilize the state-of-the-art cycle-accurate simulator gem5 [10] to evaluate our implementation. Gem5 is a popular full-system simulator that supports a highly configurable simulation, multiple ISAs, and diverse CPU and GPU models [5]. For CPU part performance evaluation, we choose ARM ISA, as the ARM core has better support on the address translation and it is more practical than the X86 ISA system, which only has a simplified translation mechanism. In the APU system evaluation, we select the X86 ISA for the CPU system due to the limitation that the APU system is currently only supported by AMD in Gem5. In addition, AMD Research has developed an APU model with gem5 by incorporating a GPU timing model capable of running the *Graphics Core Next* (GCN) generation 3 machine ISA [12] [1]. Our experiments are based on the AMD's APU model with heavily customized functions and modules. To support IAS in gem5, different modifications were needed: changes to the hardware source code, new custom blocks, and modification of the simulator microarchitecture. We modify the microarchitecture of the front-end MMU to incorporate the IAS and modify the Ruby subsystem inside Gem5 to adapt the IA and solve the cache coherency problem inside IAS. Also, we introduce the back-end MMU to process the IA to PA translation. From the perspective of the software inside the OS, the IAS is transparent. Therefore, no modifications to the applications running on the system, nor to any user-space code are required. By adopting IAS into a modified system on gem5, we are able to run detailed simulations of the full system with accurate timing and gather microarchitecture event information, such as front-end / back-end misses and the number of page table walks. We simulate three different systems for comparison purposes: (i) a conventional system used as baseline (called *Conventional* in the following), (ii) a system with an ideal MMU, which has zero address translation overhead, i.e., zero latency in translation at both front-end and back-end (called *Ideal*), representing the maximum theoretical attainable improvement, and (iii) our proposed implementation of IAS (called *IAS*).

The OS is a key element in implementing full-system IAS support. From the OS perspective, address translation cannot be transparent, since the OS is the software component in charge of managing the MMU. In the IAS system, the OS should at least be aware of the existence of this two-level address translation. However, modifying a full-fledged Linux-based OS to support IAS is a huge development effort going beyond the scope of this paper. In contrast, in this work, our aim is to use the research undertaken in [11] by taking a deeper look at the impact of IAS on full-fledged single-process applications. Modifying the OS, we believe is the next natural step once architectural

exploration proves IAS to be relevant and allows us to understand the impact of latency at the front-end and back-end MMU.

To solve this chicken-and-egg problem and simulate IAS in gem5, we choose to walk around this issue by introducing the mandatory modifications and additional information required to manage IAS directly into the packet structure in gem5. In this way, we can keep using a conventional OS without IAS support, and modify the basic data structure of gem5 packets to include information of both VAs and PAs.

4.2 System Architecture

In this study, we introduce IAS into the CPU systems based on the ARM architecture, both on an in-order core and an *out-of-order* (OoO) core. The OoO core allows for the dynamic reordering of instructions during execution, enhancing the overall system performance by exploiting instruction-level parallelism and reducing the impact of pipeline hazards. The in-order core executes the program in order with lower performance but better energy efficiency.

In addition, we modify the APU model to adapt to IAS. We configure it with the X86 CPU timing model and iGPU timing model. The iGPU model utilizes the GCN3 microarchitecture and consists of four computing units. Also, we design a new back-end MMU for the memory controller. We configure the scalar cache, *sequencer cache* (SQC), and *texture cache per pipe* (TCP) cache on the integrated GPU side to 32KB, 32KB, and 16KB, respectively. Moreover, we set the *texture cache per channel* (TCC) cache size to 256KB. The last level cache is set to 16MB. As the last level cache is CPU-private in our designed protocol, we do not modify its size in the following experiment. For the first phase of translation, the private TLB is built with 32 entries and a 4KB page size, whereas the second-level TLB is configured with 512 entries and the same page size. The hit latency and miss latency of the back-end MMU are 3 and 500 cycles, respectively. Most of the numbers are configured using the default value from gem5 since the AMD APU group sets the value, while others are scaled down according to realistic systems. All these values imitate the values present on modern processors. Table 2 details the configuration specifications. In the CPU, the miss latency for MMU is not a fixed number, as it may depend on the address and location of the target address as PTW fetches it step by step (this feature is simulated by gem5). According to our analysis, the average miss penalty of the CPU side is around 120 cycles.

4.3 System Workload

4.3.1 CPU Workload. Graph processing workloads are widely used in evaluating MMU modifications [11]. Indeed, most of the applications used nowadays in data centers, such as social network analytics, web search engines, or biomedical applications are based on graph processing algorithms. However, the large footprint of such workloads and benchmarks is not feasible to run with a cycle-accurate architectural simulator (mainly due to its slow simulation speed). To avoid this problem, while being accurate in the target architectural conclusions, we select *sparse matrix-vector multiplication* (SpMV) workloads, the crucial kernel behind graph processing workloads [8] to run the experimental evaluation. These workloads present three advantages, namely: (i) they have a small footprint, making the simulation process feasible on the gem5 simulator, (ii) they present a highly irregular access pattern, similar to graph processing workloads, and (iii) they are a fundamental linear algebra operation playing a crucial role in a multitude of different scientific, engineering, and machine-learning applications. To assess the performance of our IAS-equipped CPU system, we have chosen several sparse matrix benchmarks, i.e., *NLR*, *kron_g500*, from the SuiteSparse Matrix Collection [18] to stress the MMU. Thanks to the *Address Space Identifier* (ASID), vastly supported in modern CPUs to maintain multiple different address spaces in TLB simultaneously and avoid frequent TLB flush, our solution does not incur a larger overhead in context switches.

Table 2. CPU system and APU system configuration used on the experiments

Component	Parameter	Value	Latency
CPU	—	ARM OoO & in-order @1GHz	
	L1 data cache	32 KB	2 cycles
	L1 instruction cache	32 KB	1 cycles
	L2 cache	Configurable size	Configurable
Front-end MMU	L1	48 entries	1 cycle
	L2	1024 entries	10 cycles
Back-end MMU	—	4096 entries	3 cycles
APU	—	4 CUs, 64 lines per CU @1GHz	
	L1$ - Scalar	32 KB	1 cycle
	L1$ - SQC	32 KB	1 cycle
	L1$ - TCP	16 KB	4 cycle
	L2$ - TCC	256 KB	8 cycles
L3$ - shared		16 MB	20 cycles
Front-end MMU	L1	32 entries	hit 1 cycle, miss 5 cycles
	L2	512 entries	hit 10 cycles, miss 750 cycles
Back-end MMU	—	256 entries	hit 2 cycles, miss 500 cycles

4.3.2 APU Workload. In order to rigorously evaluate the performance of our design, we employ a wide range of test benches and workloads in our simulation, which encompasses various application domains and computing challenges, ranging from regular GPU workloads to irregular workloads. We explore the performance impact of using IAS in different domains:

- Pannotia [7]: A set of irregular GPU workloads representing non-uniform and complex computational patterns. This benchmark allows us to test our design's ability to manage irregular memory access patterns and control flow, which are often encountered in real-world applications.
- DNNMark [9]: A benchmark suite specifically tailored for *machine learning* (ML) and *deep learning* (DL) applications. It includes a collection of representative *deep neural network* (DNN) primitives that capture the computational characteristics of typical ML/DL workloads, enabling us to assess our design's efficiency in handling such tasks.
- HeteroSync [26]: A benchmark suite, comprised of *HeteroSyncLFBarr* (HSL) and *HeteroSync-sleepMutex* (HSM), focusing on fine-grained synchronization in tightly coupled GPU architectures. This testbench evaluates our design's performance in handling efficient synchronization and communication between GPU threads, which is crucial for maintaining high performance in parallel computing environments.
- Lulesh [15]: A hydrodynamic modeling application used in scientific computing. Lulesh (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) is a proxy application that simulates the behavior of materials under extreme conditions. This benchmark helps us gauge our design's ability to handle complex numerical simulations and computational fluid dynamics problems.
- HACC (Hardware/Hybrid Accelerated Cosmology Code) [23]: An application designed to simulate the evolution of the universe. HACC is a large-scale, high-performance cosmological
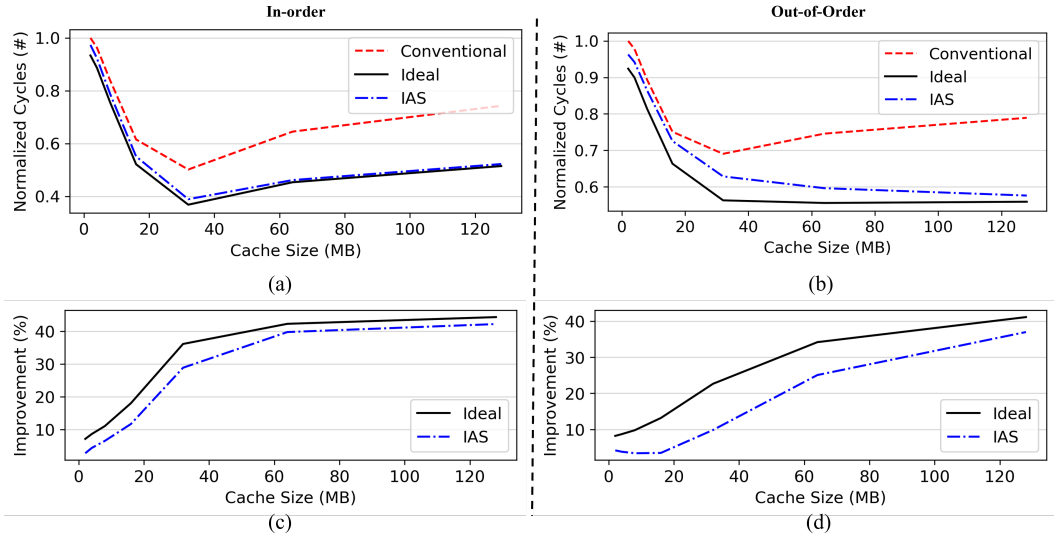
Fig. 6. When changing the cache size, execution time varies for different MMU configurations in (a) In-order core and (b) OoO core. Relative performance improvements of Ideal and IAS against Conventional MMU in (c) In-order core and (d) OoO core.

N-body code that enables us to test our design's performance in the context of astrophysical simulations and computationally intensive scientific applications.

As described, the Pannotia benchmark is representative of the typical irregular benchmark with irregular access patterns, while the others are regular benchmarks for ML and scientific computation showing some different performance results. We implement these GPU workloads on our customized Gem5 simulation, as they represent a wide set of applications. By leveraging this diverse set of benchmarks and workloads, we can thoroughly examine our proposed design's performance across various application domains and computing challenges, ensuring its efficiency, robustness, and suitability for various use cases. To evaluate the performance of the iGPU, we run the full benchmark traces without any skipped instructions, and we acquire the microarchitecture statistics of the iGPU side with that of the CPU side ignored. Then, to make a fair and consistent comparison with CPU workloads, we choose one subset from Pannotia, *Pagerank_SPMV* (PR_SpMV) as our target benchmark, due to the similar access pattern, to further analyze the performance of the APU system.

5 EXPERIMENTAL RESULTS

5.1 Evaluation results for IAS-equipped CPU-only system

5.1.1 Overall execution time. First, we compare the overall execution time of the three different systems: *Conventional*, *Ideal*, and *IAS*. Fig. 6(a) and (b) show the results of running the SPMV benchmark on these different systems with different cache sizes, both for in-order and OoO cores respectively.

When the cache size is small, the execution time for all three systems is considerably high, which can clearly be attributed to this fact, resulting in a high cache miss rate and causing the cache controller to fetch data from the main memory frequently. In addition, for systems with IAS, the high number of requests to the back-end MMU contributes to the increase in the execution time.

Due to the limited cache size, there is a significant demand for IA-to-PA conversions, resulting in considerable address translation overhead.

As the cache size increases, for the in-order core, the execution time of all three systems initially decreases, followed by an increase as the size of the cache continues increasing (see Fig. 6(a)). There are two main factors that can impact the overall performance of the system: cache overhead and translation overhead. The cache overhead is the time spent fetching data or instructions from the cache hierarchy, excluding page-table entries. The translation overhead is the time spent accessing TLB and fetching page table entries. For conventional systems, it comes only from the front-end translation. However, for systems with an IAS, it comes mainly from the back-end translation and slightly from the front-end translation. Indeed, when the cache size is small, the workload size cannot be totally placed in the cache. As the cache size enlarges, so does the cache hit rate. During this phase, the size of the cache still has a larger impact on the performance than the cache access latency. The cache overhead decreases greatly much more than the increase of the translation overhead for the IAS system. After that, when the cache size reaches a certain point, the workload can reside entirely in the cache hierarchy. The cache overhead increases during this phase due to the longer access latency. Therefore, the execution time of the IAS system and the ideal system increases. The conventional system also has to face the increasing overhead from front-end translation. Therefore, the conventional system increases much larger than the ideal system and IAS system.

For an OoO core, the performance of the IAS system improves and eventually stabilizes when the cache size increases (Fig. 6(b)). At the same time, the overall execution time of the *Conventional* system and the *Ideal* MMU system initially decreases but then experiences a slight increase. The decrease in the IAS system comes from the decrease in back-end translation. Even though the workload can fit into the cache hierarchy, the corresponding page table may not be able to reside in the cache entirely, which means that, during the second phase of cache increasing, the cache can still filter a part of page table requests. Therefore, facing increasing cache size, the translation overhead decreases and the decrease is more significant than the increase in cache overhead.

Regarding performance comparisons, we calculate the performance improvement for the *Ideal* and *IAS* system based on the *Conventional* system and plot them in Fig. 6 (c) and (d). From the figures, the performance improvement of both systems increases with the cache size. The performance of the *Ideal* MMU system always outperforms that of the IAS MMU system. The main reason for the translation overhead is the frequent back-end MMU requests. It is worth noting that there is a gap between the ideal and IAS system and that the gap of the OoO core is much larger than that of the in-order core. The gap is the translation overhead from IAS, which mainly comes from the back-end MMU (explained later in Section 5.1.5). For the OoO system, the memory access requests and the translation requests are overlapped due to the OoO ability. Therefore, the back-end translation overhead is smaller than that of the in-order core.

Compared to a system with an OoO core, the in-order-core system is more sensitive to cache latency; the fluctuation of execution time is greatly impacted by cache size because of the execution of in-order instructions. When an out-of-core core encounters a long-latency event, like a cache miss or TLB miss, the cores can execute other instructions instead of waiting for the stalled one, which can hide a large part of the latency. Instead, when the in-order core meets a long-latency event, due to the in-order mechanism, it has to wait until the previous stalled instruction resumes working. Therefore, the in-order and OoO cores show different performance sensitivities to cache size and latency.

5.1.2 Cache size impact on MMU accesses and misses. As mentioned in the previous section, only when a cache miss occurs, the system will call the back-end MMU to translate the desired addresses.

Table 3. Filter rates (%) of different cache sizes for both In-order and OoO cores

Cache size (MB)	2	4	8	16	32	64	128
in-order	92.1	92.7	93.9	95.9	97.5	97.7	97.8
OoO	92.5	93.1	94.1	95.6	96.8	97.5	97.9

Therefore, the cache can filter most of the back-end translation during the process. To study the ability of cache size, we tested the system with different cache sizes. We derive a new parameter called the filter rate by taking the ratio of the difference between the back-end access count in the system with an IA space and TLB requests in the conventional system to the TLB requests in the conventional system, shown as Eq. 1.

$$\text{filter rate } = \frac{N_{tlb\_conv} - N_{tlb\_IASback}}{N_{tlb\_conv}} \tag{1}$$

N_{tlb_conv} is the number of TLB requests in conventional system,
$N_{tlb\_IASback}$ is the number of back-end TLB requests in IAS system

The results are shown in Table 3. It shows that the cache can actually filter most MMU requests, which is consistently more than 90%. With the increasing cache size, an increasing number of MMU requests are filtered, which is one of the advantages of IAS.

5.1.3 TLB size impact on performance improvement. To understand the TLB size's influence on system performance, we analyze the overall performance of these three different systems with different back-end TLB entries.

Running benchmarks on systems with varying back-end MMU TLB entries from 1024 to 16384, we obtain the number of TLB misses and generate Figs. 7(a) and (b). The figures illustrate a decrease in the number of TLB misses as the cache size increases. When there are only 1024 TLB entries, the number of TLB misses is substantial. However, as the cache size increases, the cache becomes more effective at filtering out TLB misses. In other words, translation overhead can be mitigated by increasing the cache size.

By comparing with the *Conventional* system, Fig. 7 (c) and (d) show the performance improvement brought about by increasing cache size for different numbers of TLB entries. When the TLB size is small, the system faces a huge performance degradation, which is even 40%. The main reason is that with the limited size of TLB entries, the system faces a huge miss rate, especially when the cache size is small. In this situation, the number of requests for address translation is very high due to the high miss rate caused by the limited cache size. Then, the back-end MMU needs to use the PTW to fetch data from main memory frequently, which causes a huge translation overhead. The time spent on PTE fetch consumes much more time and addresses translation time than that spent in the *conventional* system, which causes performance degradation in the end. However, when increasing cache size, the time spent fetching PTE becomes smaller and smaller. Performance can be improved when the time is shorter than the front-end translation time spent in the *Conventional* system. By comparing systems with different TLB sizes, when the TLB size reaches 4096, the performance is always better than that of the *Conventional* system. When the cache size is larger, the overall performance improvement is greater than 40%. When comparing Figs. 7(c) and (d), the performance of the in-order system, similar to an OoO system, is sensitive to the TLB size in the back-end MMU. It is worth noting that when the cache size reaches a certain level, the differences between varying entry sizes become minimal. The in-order core system is more sensitive to cache size than the OoO system due to the in-order execution mechanism.
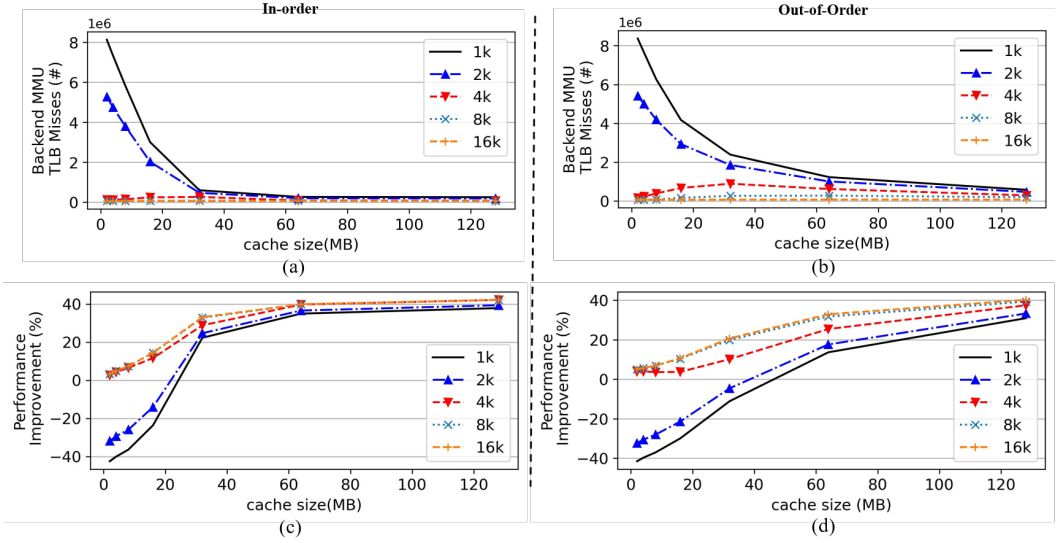
Fig. 7. When changing the cache size, Back-end MMU Misses for different TLB size configurations in (a) In-order core and (b) OoO core. Relative performance improvements of different TLB sizes in (c) In-order core and (d) OoO core.
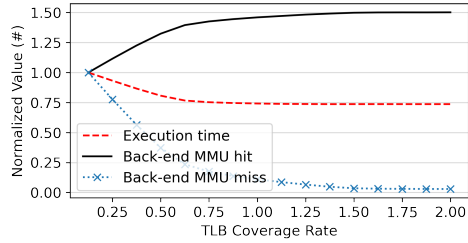

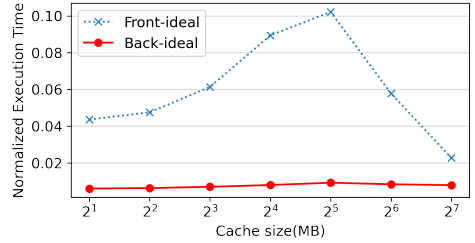Fig. 8. Impact of TLB coverage rate on execution time, back-end MMU hit and misses.

Fig. 9. Translation Overhead from front-end and back-end.

To understand the reason why the TLB size makes some impact on the overall performance, we analyze the relationship between TLB and the address range it can cover, that is, the TLB coverage.

5.1.4   Study of coverage rate. To comprehensively assess the influence of the back-end TLB size, we performed simulations of our program within a cache-fixed OoO system. We fix the cache size at 32MB, while progressively varying the TLB size in the back-end MMU from 1024 to 16384 in finer granularity. We introduce a new metric, termed the *coverage rate*, computed as the ratio of the maximum data address coverage of the TLB and the maximum data volume accommodated within the last cache level.

As illustrated in Fig. 8, we observe an inverse relationship between the coverage rate and the overall execution time. As the coverage rate increases, the execution time exhibits a downward trend and subsequently plateaus. At the same time, the count of TLB misses decreases until it stabilizes, while the number of TLB hits increases until it reaches a plateau. As depicted in the figure, with increasing coverage rate from 0.25 to 1.0, more than 87% MMU misses can be filtered,
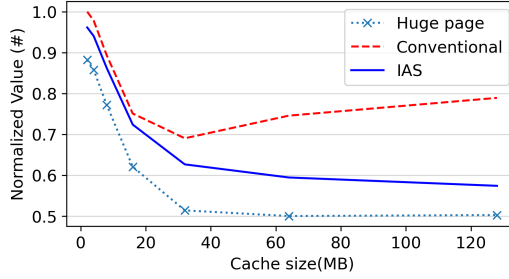
Fig. 10. Execution time comparison between huge pages, Conventional, and IAS systems.

and overall execution time can be reduced by around 25%. Increasing the coverage rate to 2.0 can reduce more than 95% of the back-end MMU misses while the execution time is nearly constant. After quantitatively analyzing the relationship between size and hit rate, we show the importance of adequately sizing the TLB.

5.1.5 Translation Overhead Analysis. As mentioned in previous sections, the translation phase of the system with IAS is composed of two parts, the front-end translation and the back-end translation. To analyze the source of translation overhead, we set up a set of experiments to split the front-end and back-end translation overhead. First of all, we modify the system to have the ideal front-end translation with back-end translation latency. Then we set the back-end translation to have zero translation overhead, while the front-end keeps the same. We change the size of the last level cache, run the NLR benchmark on these two systems, and compare the results. We normalize the execution time to the ideal (zero address translation overhead) system. As shown in Fig.9, *Front-ideal* represents the former situation, where the front-end translation is ideal, and the *Back-ideal* represents the situation where the back-end translation is free. As shown in the figure, the overhead that comes from back-end translation is much larger than that from front-end translation. The overhead from *Front-ideal* generally increases as the cache size goes up from 2MB to 32MB. However, after that, it decreases as the cache size increases further. The most significant drop is observed between sizes 64MB and 128MB. The *Back-ideal* shows a steady, gradual increase as the cache size increases from 2MB to 32MB. However, there is a slight reduction in time at cache sizes 64MB and 128MB compared to 32MB, indicating some efficiency or optimization at larger cache sizes, but not as profound as in *Front-ideal*. The main difference comes from increasing cache size, which brings longer hit/miss latency and filter rate. Overall, within the IAS system, the overhead of address translation comes mainly from the translation in the back-end, which is greatly impacted by cache size.

5.1.6 Comparison of results against huge pages. Large pages are a feature that certain operating systems use to manage virtual memory using larger granularity. In most systems, the default page size is 4KB. Huge pages use larger page sizes (e.g., 2MB or 1GB) to extend TLB coverage and reduce address translation overhead. We compare the execution time of a system using IAS when using different cache sizes in the three different systems, with a conventional system and a system using huge pages. Figure 10 shows that the performance of the system with IAS lies between the conventional system and the system with many pages.

   As expected, the performance of the IAS system sits in between that of the conventional system (outperforming it) and the system with huge pages (not achieving it). However, huge pages have well-known problems that prevent their usage in current systems, and that IAS does not exhibit.
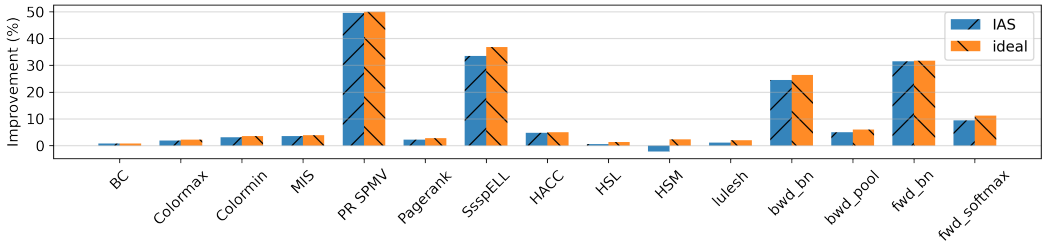
Fig. 11. Execution time improvement for applications in both *Pannotia* and *DNNmark* benchmarks using ideal MMU and IAS MMU.

For example, large pages cause memory fragmentation and memory bloating [22]. The reason is that because huge pages are much larger than standard pages, they require a contiguous block of physical memory. Over time, as processes allocate and deallocate memory, it becomes difficult to find these large contiguous blocks, causing fragmentation.

However, for the IAS system, the back-end MMU still uses the 4kB page size, which is much smaller than the huge page and will not cause fragmentation problems. The front-end implements the address translation in the unit of VMA, which is a continuously addressed memory range with no fragmentation problem.

5.2 Evaluation results for IAS-equipped APU system

5.2.1 Benefits of intermediate address space on APU. We then simulate the aforementioned benchmarks on an APU system with an IAS and compare its results with those of the Conventional APU system.

Fig. 11 shows that the implementation of an IA space can lead to substantial performance improvements. In these figures, *Ideal* represents the ideal MMU, while *IAS* denotes the MMU of the IA space. Nearly every benchmark benefits from the IA space, with the improvement of the ideal MMU and the IA space being comparable. The PR_SpMV program gains the most significant enhancement, with a greater increase of 50%. In contrast, the HSL program only achieves a performance boost of 0.5%. The results indicate that the IA space can effectively reduce translation overhead in certain scenarios, particularly when the TLB MPKI is large and the cache MPKI is small.

Moreover, it can be observed that HSM exhibits negative optimization. Upon analyzing its access pattern, we determined that HSM has a very small footprint. And its cache MPKI is 100 times larger than TLB MPKI. As a result, when the IA space is employed, the back-end MMU faces a greater address translation overhead than the benefits brought by cache increasing. In contrast, using PR_SpMV involves a larger TLB MPKI, providing substantial performance improvement. We can conclude that the IAS system significantly improves the performance of a program when it has a lower cache MPKI.

5.2.2 MMU access times benefits. Next, we compare the access frequencies of the front-end and back-end MMUs using *Pannotia* benchmarks as an example. We set the cache size to 4MB and perform a comparative analysis between traditional systems and systems with an IA space, calculating the access frequencies of the front-end MMU and back-end MMU in these two different scenarios. Our findings reveal that the implementation of caching can significantly filter out a large number of MMU access requests.

Table 4. Filter rates of different applications in Pannotia benchmark

Application	BC	Colormax	Colormin	FW	MIS	PR_SpMV	Pagerank	SsspCSR	SsspELL
Filter rate (%)	99.99	99.99	99.98	96.2	99.96	93.57	98.94	99.95	99.91

We examine the filter rate for different benchmarks. As depicted in Table 4, the use of an IA space enables the cache to filter a considerable number of address translation requests. In various test sets, up to 99% of address translation requests are filtered, with the minimum filtering rate being 94%. This demonstrates that when an IA space is incorporated, the cache offers exceptionally high filtering efficiency.

5.2.3   iGPU Cache Size impact on APU. Moreover, as previously mentioned, the cache plays a crucial role in this system. Therefore, we analyze the performance of the system and the back-end MMU by increasing the capacity of the second-level cache on the GPU side. We incrementally increased the size of the second-level cache on the iGPU side from 2kB to 4MB. We selected one of the benchmarks, PR_SpMV, for analysis, to keep consistency with the experiments in IAS CPU systems. We examine the overall execution time in Fig. 12(a), and the number of cache misses and back-end MMU misses in Fig. 12(b).

As cache size increases, the overall execution time of three distinct scenarios decreases from Fig. 12(a). The gap between a *Conventional* system and an *Ideal* MMU system widens as the cache capacity expands. When the cache size is small, the *Conventional* system outperforms the system with IAS. Due to the small size of the cache, most of the data or instruction requests are missing in the cache. Then the back-end MMU has to finish a large number of the IA to PA translations, which can great pressure upon TLB and *page table walker* (PTW), and cause significant translation overhead. In the worst-case scenario, where the cache size is only 1KB, the performance of the IA System is 13% worse than that of the conventional one.

However, as we increase the size of the cache beyond 512KB, the performance of the IA System significantly exceeds the conventional one. Due to the size of the upper-level cache, in the IAS system, most of the data requests are filtered by the cache. The back-end MMU faces less pressure. In the best-case scenario, the performance of the IAS system is 185% better than that of the conventional one.
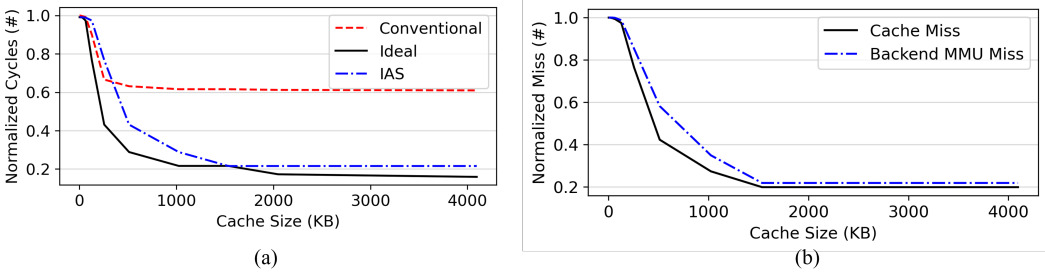

Fig. 12. (a) APU Overall execution time for different MMU configurations and (b) cache, MMU misses of IAS system with increasing Cache size.

As seen in Fig. 12(b), when the size of the second-level cache increases, the number of cache misses decreases significantly, reducing the translation pressure on the back-end MMU. In addition, the number of IAS MMU TLB misses decreases as the cache capacity increases. On the contrary, if

the size of the cache is small, the back-end MMU will be under large pressure. The request to the back-end will increase and the large miss number will bring more translation overhead. Therefore, when the cache size is small, the IAS will exacerbate the performance of the system.

6   CONCLUSIONS

In this work, we have introduced and thoroughly evaluated an innovative IAS architecture tailored to CPU and APU systems. In particular, by utilizing the IAS we have effectively divided the address translation process into two phases, front-end and back-end. This dual-phase approach effectively shifts the address translation bottleneck from the core to the memory controller side, enhancing the performance of cache-resident workloads. A detailed cycle-accurate evaluation has been performed at the microarchitectural level by modifying gem5 to add customized functions and components, thus simulating the IAS system in a full-system simulation mode. We thoroughly evaluate the IAS system from different aspects. Our analysis and experimental results demonstrate that, for CPU systems, the proposed architecture can increase performance to 40%, and filter most page table walks. To the best of our knowledge, it is the first work to show how to incorporate IAS into APU systems and analyze the performance impact from different architecture levels, such as cache, TLB size, and overhead resources. Overall, as part of the APU system, the proposed IAS architecture can achieve substantial performance improvements, with a maximum increase of 185%. Moreover, our comparison of MMU requests reveals that the IA space can significantly reduce MMU accesses by up to 99%.

Our study also investigates the impact of varying cache sizes on the performance of the IAS architecture. Our findings indicate that larger cache sizes contribute to enhanced performance, and the degree of improvement increases as the cache size increases. This observation highlights the importance of optimizing cache size to maximize IAS benefits. Furthermore, our results show that the size of the back-end TLB can also impact the overall performance of the CPU and APU system. In addition, we analyze the source of address translation overhead, and our result shows that the translation overhead comes mainly from the back-end translation.

As a result, the proposed IAS architecture offers a promising solution to alleviate the performance degradation in both APU and CPU systems caused by address translation overhead. By implementing this approach without requiring modifications to the original applications, we facilitate the adoption of IAS in existing APU and CPU systems. We believe that our work lays a solid foundation for future research and development efforts to optimize address translation mechanisms in heterogeneous computing systems and further advance computer systems' capabilities. Furthermore, we target to release as open source the modified gem5 with the new IAS architecture in https://github.com/esl-epfl/midgard-ias to enable the computer architecture community to further explore this new design space to improve the performance of the memory hierarchy, particularly in new APU systems.

REFERENCES

[1] Advanced Micro Devices, Inc. 2016. *AMD GCN3 Instruction Set Architecture*. https://www.amd.com/system/files/ TechDocs/gcn3-instruction-set-architecture.pdf Version 1.1.

[2] Paul Alcorn. 2023.   AMD instinct MI300 data center APU pictured up close: 13 chiplets, 146 billion transistors.   https://www.tomshardware.com/news/amd-instinct-mi300-data-center-apu-pictured-up-close-15-chiplets-146-

billion-transistors

[3] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 136–150. https://doi.org/10.1145/3123939.3123975

[4] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[6] Patrick Bohrer, James Peterson, Mootaz Elnozahy, Ram Rajamony, Ahmed Gheith, Ron Rockhold, Charles Lefurgy, Hazim Shafi, Tarun Nakra, Rick Simpson, Evan Speight, Kartik Sudeep, Eric Van Hensbergen, and Lixin Zhang. 2004. Mambo: A Full System Simulator for the PowerPC Architecture. *SIGMETRICS Perform. Eval. Rev.* 31, 4 (mar 2004), 8–12. https://doi.org/10.1145/1054907.1054910

[7] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 185–195. https://doi.org/10.1109/IISWC.2013.6704684

[8] Mayank Daga, Ashwin M Aji, and Wu-chun Feng. 2011. On the efficacy of a fused CPU+ GPU processor (or APU) for parallel computing. In *2011 Symposium on Application Accelerators in High-Performance Computing*. IEEE, 141–149.

[9] Shi Dong and David Kaeli. 2017. DNNMark: A Deep Neural Network Benchmark Suite for GPUs. In *Proceedings of the General Purpose GPUs* (Austin, TX, USA) *(GPGPU-10)*. Association for Computing Machinery, New York, NY, USA, 63–72. https://doi.org/10.1145/3038228.3038239

[10] gem5. 2023. GCN3. https://www.gem5.org/documentation/general_docs/gpu_models/GCN3 Accessed: 2023-05-27.

[11] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. 2021. Rebooting Virtual Memory with Midgard. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 512–525. https://doi.org/10.1109/ISCA52012.2021.00047

[12] Anthony Gutierrez, Bradford M. Beckmann, Alexandru Dutu, Joseph Gross, Michael LeBeane, John Kalamatianos, Onur Kayiran, Matthew Poremba, Brandon Potter, Sooraj Puthoor, Matthew D. Sinclair, Mark Wyse, Jieming Yin, Xianwei Zhang, Akshay Jain, and Timothy Rogers. 2018. Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 608–619. https://doi.org/10.1109/HPCA.2018.00058

[13] Nastaran Hajinazar, Pratyush Patel, Minesh Patel, Konstantinos Kanellopoulos, Saugata Ghose, Rachata Ausavarungnirun, Geraldo F. Oliveira, Jonathan Appavoo, Vivek Seshadri, and Onur Mutlu. 2020. The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 1050–1063. https://doi.org/10.1109/ISCA45697.2020.00089

[14] 2010 9:28 pm UTC Jon Stokes; Feb 8. 2010. AMD reveals fusion CPU+GPU, to challenge Intel in Laptops. https://arstechnica.com/information-technology/2010/02/amd-reveals-fusion-cpugpu-to-challenge-intel-in-laptops/

[15] I Karlin. 2012. LULESH Programming Model and Performance Ports Overview. (12 2012). https://doi.org/10.2172/1059462

[16] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, and Wolfgang Lehner. 2017. Big Data Causing Big (TLB) Problems: Taming Random Memory Accesses on the GPU. In *Proceedings of the 13th International Workshop on Data Management on New Hardware* (Chicago, Illinois) *(DAMON '17)*. Association for Computing Machinery, New York, NY, USA, Article 6, 10 pages. https://doi.org/10.1145/3076113.3076115

[17] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[18] Scott P Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. 2019. The suitesparse matrix collection website interface. *Journal of Open Source Software* 4, 35 (2019), 1244.

[19] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 705–721.

[20] Jiwon Lee, Ju Min Lee, Yunho Oh, William J. Song, and Won Woo Ro. 2023. SnakeByte: A TLB Design with Adaptive and Recursive Page Merging in GPUs. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1195–1207. https://doi.org/10.1109/HPCA56546.2023.10071063

[21] Farhad Mehdipour, Hamid Noori, and Bahman Javadi. 2016. Chapter Two - Energy-Efficient Big Data Analytics in Datacenters. In *Energy Efficiency in Data Centers and Clouds*, Ali R. Hurson and Hamid Sarbazi-Azad (Eds.). Advances in Computers, Vol. 100. Elsevier, 59–101. https://doi.org/10.1016/bs.adcom.2015.10.002

[22] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. 2019. MEGA: Overcoming Traditional Problems with OS Huge Page Management. In *Proceedings of the 12th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '19)*. Association for Computing Machinery, New York, NY, USA, 121–131. https://doi.org/10.1145/3319647.3325839

[23] U.S. Department of Energy. 2018. Coral-2 Benchmarks. https://asc.llnl.gov/coral-2-benchmarks

[24] Adarsh Patil. 2020. TLB and Pagewalk Performance in Multicore Architectures with Large Die-Stacked DRAM Cache. *ArXiv* abs/2002.01073 (2020).

[25] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. *SIGARCH Comput. Archit. News* 42, 1 (feb 2014), 743–758. https://doi.org/10.1145/2654822.2541942

[26] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2017. HeteroSync: A benchmark suite for fine-grained synchronization on tightly coupled GPUs. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*. 239–249. https://doi.org/10.1109/IISWC.2017.8167781

[27] Tuan Ta, David Troendle, Xiaoqi Hu, and Byunghyun Jang. 2017. Understanding the Impact of Fine-Grained Data Sharing and Thread Communication on Heterogeneous Workload Development. In *2017 16th International Symposium on Parallel and Distributed Computing (ISPDC)*. 132–139. https://doi.org/10.1109/ISPDC.2017.16

[28] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton. 1986. An In-Cache Address Translation Mechanism. In *Proceedings of the 13th Annual International Symposium on Computer Architecture* (Tokyo, Japan) *(ISCA '86)*. IEEE Computer Society Press, Washington, DC, USA, 358–365.

[29] Hongil Yoon, Jason Lowe-Power, and Gurindar S. Sohi. 2018. Filtering Translation Bandwidth with Virtual Caching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 113–127. https://doi.org/10.1145/3173162.3173195

[30] Lixin Zhang, Evan Speight, Ram Rajamony, and Jiang Lin. 2010. Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation. In *Proceedings of the 24th ACM International Conference on Supercomputing* (Tsukuba, Ibaraki, Japan) *(ICS '10)*. Association for Computing Machinery, New York, NY, USA, 159–168. https://doi.org/10.1145/1810085.1810109

[31] Yufeng Zhou, Xiaowan Dong, Alan L. Cox, and Sandhya Dwarkadas. 2019. On the Impact of Instruction Address Translation Overhead. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 106–116. https://doi.org/10.1109/ISPASS.2019.00018