Integration of RISC-V Page Table Walk in gem5 SE Mode

Mirco Mannino
mannino@diism.unisi.it
University of Siena
Siena, Italy

Yinting Huang
justin.huang20@imperial.ac.uk
Imperial College London
London, United Kingdom

Biagio Peccerillo
peccerillo@diism.unisi.it
University of Siena
Siena, Italy

Alessio Medaglini
medaglini@diism.unisi.it
University of Siena
Siena, Italy

Sandro Bartolini
bartolini@dii.unisi.it
Univerisity of Siena
Siena, Italy

ABSTRACT

gem5 is a popular architectural simulator, for both academic and industrial researchers. It can be used in two configurations: *Full System* mode and *Syscall Emulation* mode. The former requires running a real kernel to achieve realistic results, at the cost of increased user effort. In contrast, the latter emulates operating system functionalities, which improves usability but is more prone to producing less accurate results. Due to the absence of a genuine kernel in *Syscall Emulation* mode, the simulator model of virtual address translation remains inaccurate. In the current gem5 version (v23.0.1.0), the address translation is performed through the lookup of a flat structure that stores all the virtual-to-physical mappings. However, this approach does not reflect the behaviour of a real multi-level page table, lacking the additional latency associated with page walks. In this paper, we present our implementation of the page walk functionality in *Syscall Emulation* mode for the RISC-V ISA. We show how our page walker affects the performance of simulated benchmarks and also its sensitivity on the TLB size. Furthermore, we make our work publicly available, inviting fellow researchers to utilize and build upon the model to suit their specific requirements.

CCS CONCEPTS

• **Software and its engineering → Virtual memory**; • **Computing methodologies → Simulation tools**.

KEYWORDS

gem5, architectural simulator, page table walk, virtual memory, address translation

ACM Reference Format:
Mirco Mannino, Yinting Huang, Biagio Peccerillo, Alessio Medaglini, and Sandro Bartolini. 2023. Integration of RISC-V Page Table Walk in gem5 SE Mode. In *Proceedings of 16th Workshop on Rapid Simulation and Performance Evaluation for Design Optimization: Methods and Tools (RAPIDO '24).* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

RAPIDO '24, January 17–19, 2024, Munich, Germany

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION AND MOTIVATION

Computer architecture researchers need efficient and accurate simulators to evaluate novel architectures. One of the most widely used architectural simulators is gem5 [3, 7]. It can be used in two distinct configurations: *Full System* mode (FS) and *Syscall Emulation* mode (SE). When used in FS mode, gem5 needs a kernel binary and a disk image that allows simulating a machine with an operating system and applications running on top of it. In contrast, SE mode focuses on simulation of specific applications, or workloads, neglecting the interaction with the OS. This emulation mode is particularly useful for performance profiling, where the focus is on the behavior of microarchitectures and user-level applications and the role of the operating system is considered marginal. It allows researchers to analyze performance metrics without the overhead of simulating a fully-fledged operating system, and thus, it is suitable for initial evaluation and proof-of-concepts within a short time frame. Despite the benefit of low simulation overhead and usability, the absence of an operating system, its intricacies, and its interaction with user applications, may lead to unfaithful representation of system behavior. One notable shortcoming of gem5 SE mode is the absence of a proper page walk and address translation mechanism. In the current gem5 version, the address translation relies on a hashtable lookup, which does not take into account the page walk latency. Additionally, it does not provide statistics related to the Translation Lookaside Buffer (TLB), such as its miss rate. Furthermore, a lacking of page walk during address translation leads to an inaccurate memory access pattern. In practical system, the page table walker interacts with the memory system by storing page table entries (PTEs) in the cache hierarchy. If page walk is not simulated, the cache hierarchy is not polluted by PTEs as in real systems.

Research in virtual memory management has gained popularity in recent years due to its potential to alleviate critical bottlenecks in modern workloads. For instance, the overhead of page table walks can reach up to 50% [6]. This is a result of emerging data-intensive workloads driven by widespread applications of machine learning, big-data analytics, and scientific simulations. In contrast, the dimensions of memory system, such as caches and TLBs, fail to maintain a similar pace of growth. A simulation tool that facilitates research on the virtual memory subsystem is vital for developing new techniques and methodologies to address this performance bottleneck. Using the SE mode allows researchers, especially in the initial stages, to focus on their proposals without having to deal with OS compatibility.

```
src/arch/riscv/pagetable.hh                    |  78 +++++++
src/arch/riscv/pagetable_walker.cc             |   5 +
src/arch/riscv/process.cc                      |  41 +++-
src/arch/riscv/tlb.cc                          |  47 ++---
src/arch/riscv/tlb.hh                          |   1 +
src/base/loader/elf_object.cc                  |   4 +-
src/base/loader/memory_image.cc                |   2 +-
src/base/loader/memory_image.hh                |  13 +-
src/mem/multi_level_page_table.hh              |   7 +-
src/mem/page_table.cc                          |   2 +-
src/mem/page_table.hh                          |   2 +-
src/mem/port_proxy.hh                          |   6 +-
src/mem/se_translating_port_proxy.cc           |  10 +-
src/mem/translating_port_proxy.cc              |   6 +-
src/mem/translating_port_proxy.hh              |   3 +-
src/python/SConscript                          |   3 +
.../components/boards/se_binary_workload.py    |   3 +
src/sim/process.cc                             |   4 +-
src/sim/process.hh                             |   2 +-
```

Figure 1: gem5 files modified to support page walk in Syscall Emulation mode.

In this paper, we present the integration of the page table walk mechanism in gem5 SE mode. We choose RISC-V ISA for our implementation, since it is a popular open source ISA, although the work can be extended to other ISAs. Our objective is to improve the SE mode memory system by enabling page walk in address translation while prioritizing the use of existing components within FS mode. Figure 1 shows the files we modified to achieve our objective.

The main contributions of this work can be summarized as follows:

- We describe the limitation of address translation in current gem5 SE;
- We present our implementation of a page walk mechanism for RISC-V ISA;
- We make the source code publicly available [4], enabling other researchers to utilize and extend our work.

The remainder of the paper is organized as follows. In Section 2, we briefly describe Virtual Memory and page walking, with particular attention to the RISC-V ISA. In Section 3, we analyze the current address translation mechanism in gem5 SE mode. In Section 4, we describe in detail our proposal and analyze its impact on the performance of simulated benchmarks in Section 5. Finally, in Section 6 we conclude.

2 VIRTUAL MEMORY

Physical memory is divided into fixed-size chunks called *pages*. A memory address comprises a *page number* and an *offset* that denotes the position within that page. Each process employs its own virtual address space for memory management, introducing a layer of abstraction from the actual physical memory (i.e., physical address space). The operating system establishes a mapping between physical pages and virtual pages, allowing processes to access a greater and segregated portion of memory than what is physically available [9].

Address translation is executed by consulting a data structure known as the *page table*. The page table stores the mappings required for converting virtual addresses into physical addresses. In modern systems, it is implemented in a hierarchy structure, with multiple levels of indirection to map the virtual address space to the physical address space. Therefore, traversing the page table hierarchy (i.e., *page walk*) necessitates multiple memory accesses (e.g., three accesses for a three-level page table). Figure 2 illustrates the three-level page table used in the RISC-V Sv39 virtual memory scheme.

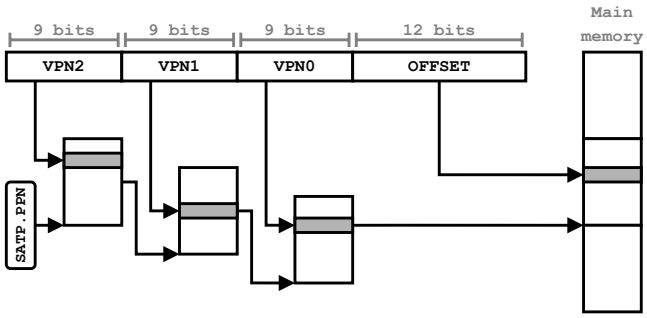

Figure 2: Sv39 virtual memory scheme.

Modern systems adopt caching mechanisms to avoid high memory access latency, and reduce the address translation overhead. Each core is equipped with a Translation Lookaside Buffer (TLB) hierarchy. TLB stores the virtual to physical mapping, allowing the core to resolve the translation in, at most, few tens of cycles. Due to the critical position of the TLB, it contains a limited number of entries (e.g., few thousands of entries), since higher capacity is associated to higher access latency. Another caching structure used for this purpose is the so-called *translation cache* [1]. It is used to store partial information on the virtual address, reducing the number of memory accesses involved during page walk. Moreover, PTEs can be stored in the cache hierarchy or in dedicated caches to reduce data retrieval latency.

2.1 RISC-V Sv39

RISC-V ISA defines three virtual memory schemes: Sv39, Sv48, and Sv57, corresponding to 39, 48, and 57 bits for the virtual address, respectively. A virtual memory scheme that uses all 64 bits for virtual address will be defined in a future version of the standard [10].

gem5 adopts the Sv39 address format for the FS mode. For consistency and compatibility, we use the same address in our implementation. However, our changes can be easily adapted to support other virtual memory schemes.

Sv39 is a 3-level virtual memory scheme with a default page size of 4KB. The 12 least significant bits define the *offset* within the page. The upper 27 bits define the offset within each level of the page table (9 bits per level): *VPN2*, *VPN1*, and *VPN0*. Figure 2 shows the format of the Sv39 address translation scheme.

The RISC-V standard defines the algorithm used to perform page walk by leveraging information stored in PTEs. PTE format for Sv39 includes physical page number (PPN) and permission bits to control access rights during a page walk. A fundamental register used to perform page walk is the so-called *Supervisor Address Translation and Protection* (`SATP`) register. It is used for storing information about page table root address (`SATP.PPN`), process

Table 1: Encoding of PTE.X, PTE.W, and PTE.R fields [10].

X	W	R	Description
0	0	0	Pointer to next level of page table
0	0	1	Read-only page
0	1	0	Reserved
0	1	1	Read-write page
1	0	0	Execute-only page
1	1	0	Reserved
1	1	1	Read-write-execute page

identifier (SATP.ASID), and virtual address translation scheme (SATP.MODE).

Page walk starts by reading the physical page number stored in SATP, concatenated with *VPN2* field of the virtual address. After a PTE is fetched, permission bits are checked to evaluate whether the page access is valid, otherwise a page fault is raised. PTE.X, PTE.W, and PTE.R are used to check execute, write, and read permissions, respectively. They are also used to specify whether the PTE is a leaf or an intermediate PTE. Table 1 shows the encoding of these three fields in the RISC-V standard.

3 ADDRESS TRANSLATION IN GEM5 SE

In gem5 (v23.0.1.0), the Memory Management Unit (MMU) receives translation requests that are propagated to either I-TLB or D-TLB, based on the type of data contained in the page to be translated. The behavior of RISC-V TLB varies depending on the simulation mode.

In FS mode, after some preliminary checks, the TLB is looked up. In case of a TLB miss, the page table walker is invoked. During this process, the TLB statistics such as TLB miss counters are updated accordingly. The page table walker reads the SATP register information, which is set by the kernel during context switch, and initiates the page walk as described in the algorithm in the RISC-V standard [10]. The page table walker is connected to the memory system (e.g., page walk cache), allowing it to send memory requests and retrieve PTEs. The walker terminates the page walk with either the correct translation or a page fault that is handled by the underlying kernel.

The approach used in SE mode is completely different. The lack of a real operating system led gem5 developers to develop components to emulate certain behaviors. For the page table, there is an object of type EmulationPageTable (defined in src/mem/page_table.hh) that contains all the standard operations of a page table, such as map, remap, and unmap. The main structure used to store the virtual-to-physical mapping is a hashtable indexed with virtual addresses. Whenever the TLB receives a translation request in SE mode, the per-process hashtable pTable is looked up. If the virtual address is present, the hashtable instantaneously returns the corresponding physical address, otherwise a page fault is raised from the TLB. In this implementation, there is no page walk during address translation. The latency of accessing page table is not modeled in simulations. Moreover, TLB is not consulted and therefore, there are no updates to the related statistics. As a consequence, if an application exhibits a data access pattern involving a high TLB miss rate, the performance degradation due to page walk is not reflected in the simulation results.

In Section 4, we describe the main changes included to support the page table walk mechanism in gem5 SE mode.

4 PAGE WALK IN GEM5 SE

During the development of page walk support in gem5 SE, our aim was to leverage as much of the existing FS mode code as possible. In this section, we describe the main changes we made. In the final version of our implementation, we refrain from altering the page table walker code used in FS mode. Instead, we focused on adapting the relevant components in the simulator to align with the already established and thoroughly tested code. The main changes involve the definition of a multi-level page table, the process initialization, and the internal logic of the TLB to solve address translations.

The implementation presented in this paper is also used to obtain the experimental results of a previous work [8].

4.1 Multi-level page table

As discussed in Section 3, a proper multi-level page table is missing in gem5 SE mode. In gem5, there is the definition of a class called MultiLevelPageTable[1] which is a child of EmulationPageTable class. We leverage that class to define the multi-level page table for RISC-V ISA, establishing the foundation for page table traversal. We define the class HierarchySv39 which encapsulates the information about PTE fields, along with public methods for modifying PTE read, write, and execute permission bits. This serves as a building block for the template class MultiLevelPageTable. The initialization of the multi-level page table involves preparing the root page table for each level and allocating a physical page from the free pool of physical pages. When the simulator encounters an unmapped page during initialization, it triggers a GenericPageFault and the page fault handler allocates a new physical page. The page is mapped to the physical page by calling the MultiLevelPageTable::map method. After the allocation of a new page in the page table, the corresponding PTE fields are initialized accordingly.

A key aspect in PTE initialization lies in setting the permission bits. As discussed in Section 2.1, the PTE.X, PTE.W, and PTE.R fields are used to determine whether the PTE is a leaf or not. This information is written after that the physical page is allocated and the PTE is mapped in the page table. For a non-leaf PTE, PTE.X, PTE.W, and PTE.R are initialized to zero. For a leaf PTE, the bit values depend on the permission of the page. We exploit the permission information contained in the program header of the Executable and Linkable Format (ELF) object. It contains segment-dependent flags which specifies the execute, write, and read permission for each segment. We encode information about execute permission in the MemoryImage::Segment struct during page mapping. The execute permission is needed by the page table walker to understand if the page contains data or instructions. In the current version of our implementation, the write and read permissions in the ELF object header are not propagated to the page table-related objects since we set those bits to 1 every time a page is initialized.

[1] actually, it is used only by the x86 ISA combined with KVM CPU type, but it is adaptable to the RISC-V ISA.

Table 2: Value of `stack_base` and `mmap_end` for every value of `useArchPT`.

	useArchPT	
	false	true
stack_base	0x7FFF'FFFF'FFFF'FFFF	0xFFFF'FFFF'FFFF'0000
mmap_end	0x4000'0000'0000'0000	0xFFFF'FFEF'FFFF'0000

The multi-level page table is initialized during process initialization (see Section 4.2).

4.2 Process initialization

We modify the process initialization to include the allocation of a multi-level page table. Our changes are specific to the 64-bit RISC-V process, `RiscvProcess64`. Within the process object, there already exists a boolean attribute named `useArchPT`, which is disabled in the current gem5 version. This attribute determines whether to activate the page table walker (`useArchPT=true`) or not (`useArchPT=false`). We use this attribute, in the process object constructor, to allocate a conventional `EmulationPageTable` or a `MultiLevelPageTable`. Figure 3 shows the modifications we made to the constructor of `RiscvProcess` object, from which `RiscvProcess64` inherits. We show both the current gem5 version and our implementation.

As part of the process initialization, the multi-level page table is initialized as described in Section 4.1. This involves the allocation of the top-level page table, with the resulting page table root address being stored in the `SATP` register. Subsequently, the per-process `SATP` register is configured. In this step, the root address of the page table, the virtual scheme mode, and the process ID are written in the corresponding fields of `SATP` register.

Certain reference addresses, like `stack_base`, are defined during process initialization. As mentioned in RISC-V Sv39 scheme [10], the 25 most significant bits that are not used in the virtual address are filled by copying the value of the 38^{th} bit. To accommodate this, the current reference addresses employed in gem5 SE mode needs to be adjusted. In particular, we modify the original value of `stack_base` and `mmap_end`. These addresses are assigned based on the `useArchPT` value. Table 2 shows the values used in the current gem5 version and the new values in the modified gem5 version.

4.3 TLB logic

The internals of the TLB object are modified in order to lookup the information in the TLB and, in case of a miss, to start a page walk.

In timing mode, the MMU forwards the translation request to the TLB using the `TLB::translateTiming` method, which subsequently calls `TLB::translate`. Following this, in FS mode, in case of TLB hits, the physical address is returned, otherwise the page table walker is triggered. However, in SE mode, the address translation is resolved using the page table implemented as a hashtable (i.e., `Process::pTable`). We modify the `TLB::translate` method so it follows the same paging mechanism as FS mode when the boolean attribute `useArchPT` is set.

RISC-V process constructor in gem5 v23.0.1.0

```
RiscvProcess::RiscvProcess(const ProcessParams &params, loader::ObjectFile *objFile) :
                Process(
                    params,
                    new EmulationPageTable(params.name, params.pid, PageBytes),
                    objFile
                )
{
    fatal_if(params.useArchPT, "Arch page tables not implemented.");
}
```

(a) Original

RISC-V process constructor with integration of page table walk in SE mode

```
template class MultiLevelPageTable<HierarchySv39<38, 30>,
                                   HierarchySv39<29, 21>,
                                   HierarchySv39<20, 12> >;

typedef MultiLevelPageTable<HierarchySv39<38, 30>,
                            HierarchySv39<29, 21>,
                            HierarchySv39<20, 12> > ArchPageTable;

RiscvProcess::RiscvProcess(const ProcessParams &params, loader::ObjectFile *objFile) :
                Process(
                    params,
                    params.useArchPT ?
                        static_cast<EmulationPageTable*>(
                        new ArchPageTable(params.name, params.pid,
                        params.system, PageBytes)
                        ) :
                        new EmulationPageTable(params.name, params.pid, PageBytes),
                    objFile
                )
{ }
```

(b) Modified

Figure 3: Our modifications to the `RiscvProcess` constructor in gem5.

Figure 4 depicts the flow diagram for processing address translation requests, reflecting the incorporation of our modifications into the `TLB::translateTiming` method. The flow diagram shows, in green, the new if-statement that enables the use of the page walker, as in FS mode. It is possible to insert the latter thanks to the previous creation of a multi-level page table.

During initialization operations (e.g., loading binaries), the MMU invokes `TLB::translateFunctional` method. Similarly to timing mode, the current gem5 version does not involve the use of the page table walker. We modify the code in order to start the functional page walk when `useArchPT` is set, as in FS mode. It is worth mentioning that the functional page walk does not add latency when it accesses memory, it is just used to solve address translations during the initialization phase. The flow diagram for `TLB::translateFunctional` is similar to the one showed in Figure 4.

During page walk, a page fault may occur because the page may be not allocated before. The TLB object provides a method to trigger a page fault (i.e., `TLB::createPageFault`). In the gem5 current version, the method returns an `AddressFault` pointer. This type of fault is designed for FS mode, and expects the underlying kernel to handle the page fault. In SE mode, page faults are handled by generating a `GenericPageTableFault`, which, when invoked, can either allocate and map a new page or halt the simulation, depending on the address that raised the fault. Our implementation leverages `GenericPageTableFault` to handle page faults as described in Subsection 4.1.

Our implementation also allows the use of TLB statistics in SE mode, and not only in FS mode. We improve the TLB statistics, including the miss rate in addition to the conventional counters on

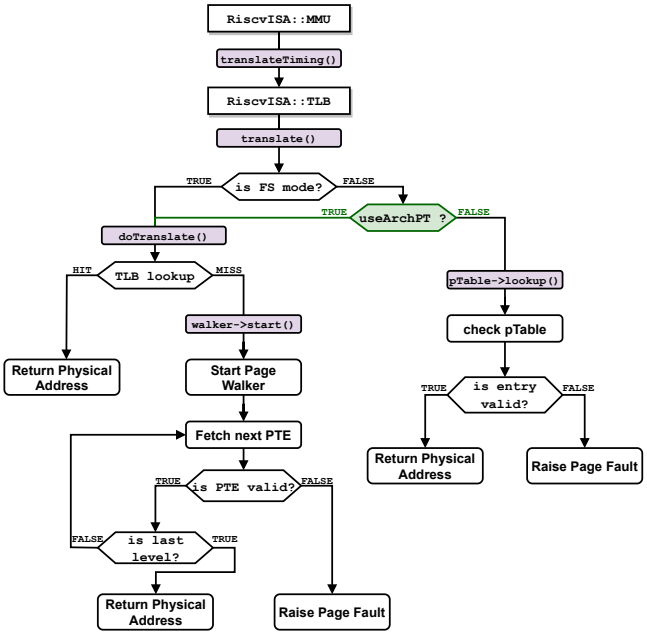

Figure 4: Flow diagram for processing address translation requests, after the changes to `TLB::translateTiming` method. The green if-statement allows for executing page walk code, as in FS mode

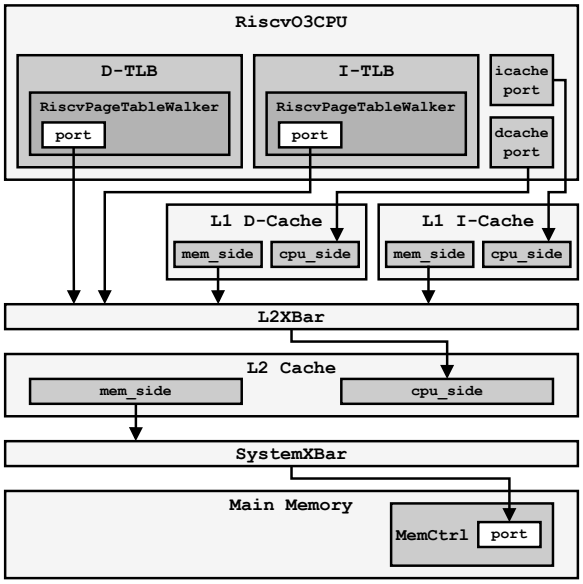

Figure 5: Overview of the simulated board used to obtain experimental results.

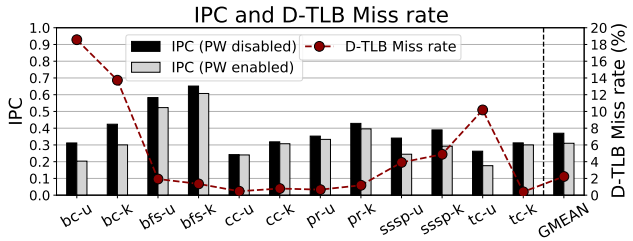

Figure 6: IPC and TLB miss rate obtained with both page walk disabled (gem5 standard version) and enabled (our changes).

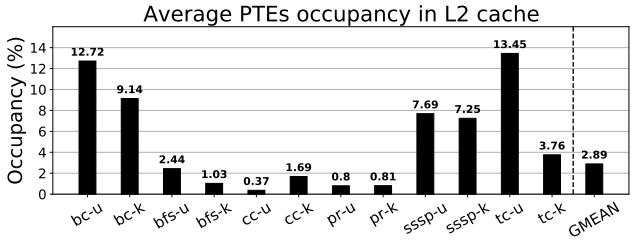

Figure 7: Average occupancy of PTEs in L2 cache.

number of misses, hits, and accesses. Moreover, we identify and report a bug [5] that affects the count of TLB misses and hits. In particular, the current gem5 version updates TLB statistics twice after a TLB miss (i.e., once for the first lookup and once after concluding the page walk).

5 EVALUATION

5.1 Methodology

We compare the results obtained using the page walk in gem5 SE against those obtained using the standard gem5 version. The simulated board consists of a RISC-V O3 core and a two-level cache hierarchy (32KB L1 I-cache, 32KB L1 D-cache, and 256KB L2 cache). The core has a D-TLB and I-TLB, both with 1024 entries. Moreover, we change the TLB size and analyze the performance. The page table walker port is connected to the L2 bus, allowing storing PTEs in the L2 cache. It is worth mentioning that this is just a design choice, it is possible to either connect the page table walker port to the L1 cache or to a dedicated page walk cache. Figure 5 shows the board design.

We use the GAP Benchmarks Suite [2] for evaluations. It includes six graph processing benchmarks. We use two sets of input graphs: uniformly distributed graphs (indicated by "-u"), and Kronecker graphs (indicated by "-k"). All the input graphs include 1 million edges.

5.2 Comparison against standard gem5

The first analysis concerns the performance difference between our implementation and the current gem5 version. Figure 6 shows the IPC and the TLB miss rate obtained using both standard gem5 and our implementation. Default D-TLB and I-TLB sizes of 1024 are used for comparison. It is worth noting that, although slightly in some cases, the IPC of our version is always lower than standard gem5. This implies that there is a overhead induced by the memory accesses of the page walk. Another aspects that emerges from these results is the correlation between performance degradation and the

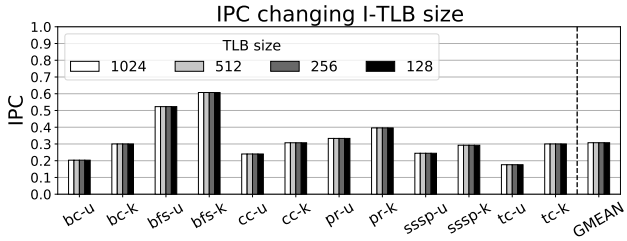

Figure 8: IPC obtained changing the I-TLB size (1024, 512, 256, and 128 entries).

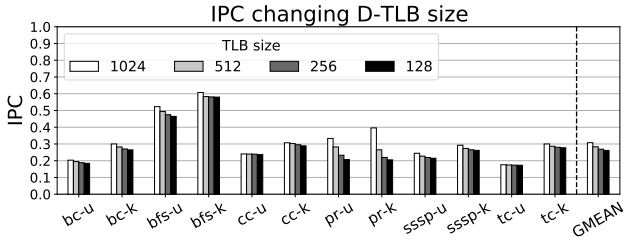

Figure 9: IPC obtained changing the D-TLB size (1024, 512, 256, and 128 entries).

TLB miss rate. When the TLB miss rate is higher, the difference between the IPCs is higher. For instance, for the *bc-k* benchmark, that shows a TLB miss rate of 14%, IPC obtained is about 30% lower than the one obtained using gem5 standard version. When the TLB miss rate is low, e.g., *cc-u* benchmark, the IPCs are similar, indicating that the page walk doesn't impose significant overhead, as a substantial portion of address translations are efficiently handled by the TLB.

The use of a page table walker that is able to send memory requests through the cache hierarchy allows having more realistic simulation results, that are not possible to obtain in the gem5 current version using SE mode. Figure 7 shows the average percentage of L2 cache blocks occupied by PTEs. The geometric mean of all the benchmarks is 2.89%. The highest occupancy is obtained using *tc-u* benchmark (i.e., 13.45%). These findings demonstrate that, in certain scenarios, cache performance may be influenced by PTE-related pollution, enhancing the importance of simulating such behaviors with an effective page table walker.

5.3 Effect of I-TLB and D-TLB size

In order to evaluate the integration of page table walker in gem5 SE, we simulated a system with different I-TLB and D-TLB with different sizes (i.e., 128, 256, 512, and 1024 entries).

Figure 8 shows the IPC of all the benchmarks obtained varying the size of the I-TLB and keeping D-TLB size unchanged (1024). The figure shows that decreasing the I-TLB size, even up to 128 entries, the performance are not affected. These results are in line with the I-TLB miss rate, that is almost zero in all analyzed configurations. Indeed, selected benchmarks show a high page locality for instruction pages that can be covered even by relatively small I-TLB.

The variation in performance is different when the number of D-TLB entries changes. Figure 9 shows the IPC of all the selected benchmark obtained changing the D-TLB size and keeping I-TLB size unchanged (1024). Some benchmarks do not show performance degradation when the TLB size is reduced, as can be seen in the case of *cc-u* benchmark. The reason is that, for this kind of benchmarks, the TLB miss rate is low even with the smallest TLB size (i.e., 128 entries). Table 3 shows the TLB miss rates obtained. On the other hand, benchmarks that exhibit an increasing D-TLB miss rate, when the D-TLB size decreases, show a consequent IPC decrease. This is an expected results, and it highlights that when the TLB miss rate is *low*, the need for page table walks is reduced.

Table 3: D-TLB miss rate obtained using different D-TLB size (1024, 512, 256, and 128 entries)

Benchmark	TLB size			
	1024	**512**	**256**	**128**
bc-u	18.57	23.41	26.07	27.49
bc-k	13.73	21.64	25.16	26.48
bfs-u	1.93	3.87	5.23	5.96
bfs-k	1.35	2.58	2.64	2.71
cc-u	0.47	0.50	0.57	1.08
cc-k	0.79	1.44	2,18	2.85
pr-u	0.66	22.65	33.85	39.44
pr-k	1.18	27.92	37.51	40.98
sssp-u	3.91	10.54	13.01	14.33
sssp-k	4.88	9.33	11.18	11.97
tc-u	10.18	10.76	11.10	11.28
tc-k	0.40	1.37	1.81	2.02

6 CONCLUSIONS

In this paper, we present our changes to the popular gem5 simulator aimed at integrating the page walk functionality in RISC-V *Syscall Emulation* mode. We first analyze the address translation mechanisms and limitations in current gem5 SE mode. Subsequently, we introduce the methodology behind our work, focusing on three crucial aspects: the definition of multi-level page tables, process initialization, and TLB logic. Through comprehensive benchmarking, we demonstrate the influence of our modifications by comparing instruction per cycle and TLB miss rate before and after enabling the page walk functionality. Moreover, we conduct an analysis to show the effects of changing TLB size. To facilitate further research exploration, we have made the source code of our modification publicly available [4]. This enables fellow researchers to delve into new virtual memory-related strategies, with the flexibility to design the page table walker and TLB configurations in both SE and FS modes. In the pursuit of more realistic simulation results, our future work aims to address another limitation of gem5 memory system: the absence of a two-level TLB hierarchy. This enhancement, while requiring significant gem5's source code changes, will further improve the accuracy and fidelity of our simulations. Additionally, we aim to incorporate support for other virtual memory schemes and integrate a translation cache object, enhancing the versatility of our modified gem5 simulator.

ACKNOWLEDGMENTS

The authors would like to thank all members of Huawei Technologies R&D Cambridge (UK) research centre for their support and helpful guidance during development.

REFERENCES

[1] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France) *(ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 48–59. https://doi.org/10.1145/1815961.1815970

[2] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[3] Nathan Binkert et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[4] Mirco Mannino et al. 2023. Integration of RISC-V Page Table Walk in gem5 SE Mode - Github Repository. https://github.com/mircomannino/gem5/tree/rapido24

[5] gem5 GitHub repository. 2023. issue n. 484. https://github.com/gem5/gem5/issues/484

[6] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman Ünsal. 2015. Redundant memory mappings for fast access to large memories. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 66–78.

[7] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. 2020. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[8] Mirco Mannino, Biagio Peccerillo, Andrea Mondelli, and Sandro Bartolini. 2023. Energy and Performance Improvements for Convolutional Accelerators Using Lightweight Address Translation Support. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*. 84–90.

[9] James L Peterson and Abraham Silberschatz. 1985. *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc.

[10] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A Patterson, and Krste Asanovic. 2015. The RISC-V instruction set manual volume II: Privileged architecture version 1.7. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-49* (2015).