Enabling Virtual Memory Research on RISC-V with a Configurable TLB Hierarchy for the Rocket Chip Generator

Nikolaos Charalampos
Papadopoulos*
National Technical University of
Athens
ncpapad@cslab.ece.ntua.gr

Vasileios Karakostas
National Technical University of
Athens
vkarakos@cslab.ece.ntua.gr

Konstantinos Nikas
National Technical University of
Athens
knikas@cslab.ece.ntua.gr

Nectarios Koziris
National Technical University of
Athens
nkoziris@cslab.ece.ntua.gr

Dionisios N. Pnevmatikatos
National Technical University of
Athens
pnevmati@cslab.ece.ntua.gr

ABSTRACT

The Rocket Chip Generator uses a collection of parameterized processor components to produce RISC-V-based SoCs. It is a powerful tool that can produce a wide variety of processor designs ranging from tiny embedded processors to complex multi-core systems. In this paper we extend the features of the Memory Management Unit of the Rocket Chip Generator and specifically the TLB hierarchy. TLBs are essential in terms of performance because they mitigate the overhead of frequent Page Table Walks, but may harm the critical path of the processor due to their size and/or associativity. In the original Rocket Chip implementation the L1 Instruction/Data TLB is fully-associative and the shared L2 TLB is direct-mapped. We lift these restrictions and design and implement configurable, set-associative L1 and L2 TLB templates that can create any organization from direct-mapped to fully-associative to achieve the desired ratio of performance and resource utilization, especially for larger TLBs. We evaluate different TLB configurations and present performance, area, and frequency results of our design using benchmarks from the SPEC2006 suite on the Xilinx ZCU102 FPGA.

KEYWORDS

RISC-V, Rocket Chip Generator, TLB, Memory Management Unit

---

*Contact author

1 INTRODUCTION

Rocket Chip Generator (RCG) is a tool that uses the open RISC-V ISA to produce configurable SoCs. RCG supports fully-fledged Unix-like operating systems, and features important RISC-V extensions and accelerators. RCG is designed to target a wide range of application domains, ranging from embedded up to complex and multicore systems. To support this wide range of application domains, most of the processor components have been implemented as configurable templates in the Chisel high-level hardware construction language (HCL). However, some of the Rocket Chip Generator components are still missing support for configurability. In this paper we focus on the Memory Management Unit (MMU) and specifically on the Translation Lookaside Buffer (TLB) hierarchy that lacks such configurability support. TLBs are essential in terms of performance because they mitigate the overhead of frequent page table walks, but may harm the critical path of the processor due to their size and/or associativity. Furthermore, a configurable TLB hierarchy might be useful for performance scaling for faster processors such as the out-of-order BOOM [8].

In the original Rocket Chip implementation only the number of TLB entries is configurable; the L1 Instruction and Data TLBs can only be fully-associative and the shared L2 TLB direct-mapped. However, that approach is not optimal for applications with large memory footprints that require larger TLB reach with many entries because (i) increasing the number of the fully associative L1 TLB may increase the processor critical path and can impact the operating frequency of the entire design, and (ii) a direct-mapped L2 TLB can experience many conflict misses, leaving significant room for application performance improvement with the use of increased associativity. Clearly, this lack of configurability in the TLB may limit the efficient applicability of Rocket Chip SoCs for applications with large memory footprints that stress the TLB hierarchy.

In this paper we lift these restrictions and design and implement configurable, set-associative L1 and L2 TLB templates that can create any organization from direct-mapped to fully-associative to achieve the desired ratio of performance and resource utilization, especially for larger TLBs. Furthermore, we modify existing replacement policies to be compatible with our design, offering flexibility for performance and resource usage trade-offs.

We modify the L1 and L2 TLB mechanisms and specifically how TLB lookups, refills, flushes, and replacements are handled. Chisel allows the programmer to produce circuit generators that are easily configurable. With our approach, just by adjusting the number of the sets and the ways of the L1/L2 TLB, all the TLB circuitry is properly configured. Corner cases such as direct-mapped and fully-associative organizations are included, and the design is tailored to remove unnecessary components for these cases. For example, if a direct-mapped organization is selected there is no need for replacement policy, so our Chisel code removes it altogether.

We use different L1/L2 TLB configurations to evaluate our design with benchmarks from the SPEC2006int suite [10]. We show that the largest evaluated TLB configuration improves performance by up to 15.4%, with minimal impact in area and frequency.

In summary the main contributions of this paper are:

- We implement a fully configurable Instruction/Data L1 TLB and shared L2 TLB that can output any design from direct-mapped to fully-associative, lifting the initial restrictions of configurability only by the number of entries. This leads to better scaling of performance and resources, especially for large TLBs. We will make our design publicly available to enable further research on the active topic of virtual memory support for RISC-V.

- We present a case study in which we evaluate the performance and resource usage of the Rocket Chip [4] processor with different TLB configurations, by running benchmarks from the SPEC2006int [10] suite on the Xilinx ZCU102 FPGA.

2 BACKGROUND

Here we provide information on virtual memory, the Chisel hardware description language, and the Rocket Chip Generator.

2.1 Virtual Memory

Virtual memory is an essential concept for processor design because it provides the illusion of a very large and private address space to each process running in the system. Virtual memory offers security through process isolation and also benefits programmer productivity since the operating system manages the memory mappings and the hardware accelerates the translations.

RISC-V supports different Virtual Memory systems depending on the size of the address space (e.g. RV32 Sv32, RV64 Sv39/Sv48 [1]), in this paper we focus on RV64 Sv39 (39-bit address space) which supports 4KB base pages but also 2MB, 1GB super pages; the page table, that stores the memory mappings of each process, is implemented as a multi-level radix tree (3-level page table in RV64 Sv39). A processor register called SATP (Supervisor Address Translation and Protection register) holds the root of the page table. The physical address is obtained after performing a sequential lookup in each page table level. The page table walker (PTW) that performs the virtual-to-physical address translations is typically implemented in hardware for improved performance.

To accelerate address translation without accessing the page table on every memory reference, a Translation Lookaside Buffer (TLB) is used which keeps the recently used translations. The TLB lies on the critical path of the processor and as a result its size and associativity are essential for the overall performance. To overcome
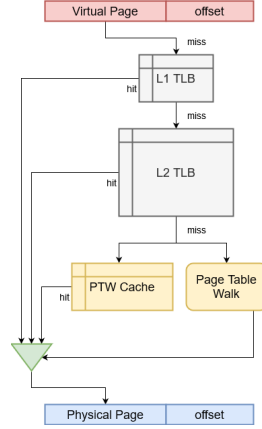

Figure 1: Overview of the MMU in Rocket Chip Generator.

this problem without sacrificing the hit rate, multi-level TLB organizations are used; the first level TLB (L1) is usually small (32-128 entries) but very fast, while the second level TLB (L2) is usually larger (128-1024 entries) but slower. Finally, a Page Table Walk cache is usually implemented to hold non-leaf intermediate translations of the page table to avoid searching levels of the page table (TLBs hold the leaf translations). Figure 1 shows these structures.

2.2 Chisel

Chisel [5] is a high-level Hardware Construction Language (HCL) embedded in the Scala language. Chisel enables the design of powerful circuit generators by utilizing Scala's high-level programming concepts like object-orientation, functional programming, parameterized typed and type inference. Chisel can generate synthesizable Verilog for both FPGA simulation and ASIC implementation. It can also output cycle-accurate C++ simulators which are very useful for hardware simulation and debugging.

2.3 Rocket Chip Generator

The Rocket Chip Generator (RCG) [4] generates RISC-V ISA [1, 2] based systems using Chisel. It can also be considered as a library of processor parts that can easily be reused with any design written in Chisel. By default, the Rocket Chip Generator instantiates Rocket, an in-order core implementation, but also supports various core implementations including the BOOM out-of-order processor [8]. Rocket is a simple, 5-stage, in-order processor that implements the RISC-V ISA, including an MMU that supports page-based virtual memory, TLBs, instruction and data caches, and a frontend that features dynamic branch prediction with configurable sizes.

3 TLB HIERARCHY DESIGN

In this section we provide an overview of the original implementation of the Instruction/Data L1 and shared L2 TLB in the Rocket Chip Generator. Then, we present the design and implementation of our proposed configurable L1 and L2 TLB. Our design can output any organization ranging from direct-mapped up to fully-associative TLBs.
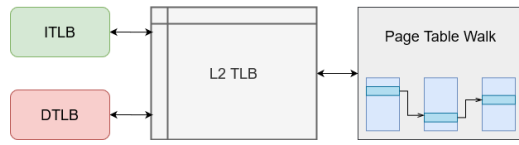
Figure 2: Rocket Chip MMU organization.

3.1 Original TLB overview

Each processor has its own TLB hierarchy, as shown in Figure 2. The L1 Instruction and Data TLB hold address translations for the process code and the process data respectively. The L1 Instruction/Data TLBs are built based on the same Chisel template in the RCG and only have minor differences regarding access privileges to pages. The L2 TLB is shared among the L1 Instruction/Data TLBs and can contain both Instruction and Data page translations.

3.1.1 L1 TLB. The L1 Instruction/Data TLB stores the page translations in registers using a vector of `Reg` elements which create an array of positive-edge-triggered registers that output a copy of the input signal delayed by one clock cycle, depending on its activation signal. The original L1 TLB is fully-associative with configurable number of entries and uses a Pseudo-LRU Replacement Policy. The L1 TLB responds with a hit/miss indication on the next cycle and stores virtual-to-physical page translations of 4KB pages but also 2MB/1GB super pages.

3.1.2 L2 TLB. The Chisel template for the Page Table Walker (PTW) incorporates the shared L2 TLB. The PTW is connected with the L1 Instruction and Data TLBs though a Round-Robin Arbiter that selects the target virtual address to be translated. The shared L2 TLB is direct-mapped with configurable number of entries. Because of the direct-mapped organization there is no need for a replacement policy. The L2 TLB stores the page translations using Chisel's `SyncReadMem/SeqMem` construct, which can be synthesized to FPGA Block RAM or ASIC SRAM. `SyncReadMem` basically creates a synchronous-read, synchronous-write memory, in this case with one read and one write port. Because of the `SyncReadMem` construct data are fetched on the next cycle; `SyncReadMem` outputs to a register with a purpose of performing a synchronous read operation. In order for the L2 TLB to sync with the rest of the PTW mechanism there are intermediate stages until the L2 TLB informs for a hit or miss.

3.1.3 Page Table Walk Cache. The PTW Cache is a small fully-associative cache that stores the non-leaf virtual-to-physical page translations. In this paper we focus on the TLBs and leave the PTW Cache for future work.

3.1.4 Limitations. In the original Rocket Chip implementation, only the number of TLB entries is configurable. However, that approach is not optimal for applications with large memory footprints that require larger TLB reach. Increasing the number of the fully associative L1 TLB significantly increases the critical path of the processor and can impact the operating frequency of the entire design. This happens because fully associative TLBs are typically implemented as CAMs. However, CAMs are a resource- and power-hungry structures, in both ASICs and FPGAs [25]. Considering this, the original fully-associative L1 TLB is constrained and does not

scale with application requirements. Increasing the size of L1 TLBs at lower associativity may increase the TLB reach and reduce the number of TLB misses without affecting the the overall resource usage/frequency.

Furthermore, because the L1 TLBs need to be fast, they are implemented using discrete registers that are generally precious resources both for ASIC and FPGA implementations. To mitigate the miss overhead of a relatively small L1 TLB, a larger but slower L2 TLB is introduced that stores translations in FPGA Block RAM or ASIC SRAM. However, a direct-mapped L2 TLB can experience many conflict misses. In addition, L2 TLB misses are even more costly than L1 TLB misses, because they are resolved through page walks that incur increased latency. Associativity may reduce the number of conflict misses and improve the application performance.

To summarize, this lack of configurability in the TLB may limit the applicability of Rocket Chip Generator for workloads with large memory footprints that stress the TLB hierarchy.

3.2 Configurable L1 TLB Architecture

To develop a configurable L1 TLB we must consider a set of factors and trade-offs. More specifically, the configurable Instruction/Data TLB should use registers using Chisel's `Reg` element for fast lookup time. In addition, the configurable Data/Instruction TLB should be built by the same Chisel template with minor differences regarding the access privileges as mentioned earlier. Our implementation adheres to the aforementioned requirements and is compatible with the original implementation. Next we describe how lookups, refills, replacements, and flushes are handled in our configurable L1 TLB.

3.2.1 Lookup. Whenever an address translation is requested, we obtain a *tag* and an *index* by splitting the *VPN*. Using the *index* we locate the target set and perform there a fully-associative search that matches the *tag*. We modify the valid bit array and construct it as a `Vec` of registers, so every set has its respected valid bit array and can address it using the `index`.

3.2.2 Refill. When a TLB refill is requested, we locate the target set that the virtual/physical address must be inserted using the *index*. In case the set is not full, we select the first free slot. Otherwise, if the set is full we perform a Pseudo-LRU replacement.

3.2.3 Replacement Policies. We modify the existing pseudo-LRU replacement policy and implement a set-associative alternative that uses the `Reg` construct. Support for a random replacement policy is already provided. A random replacement policy is an attractive alternative option thanks to its simplicity and can be also applied to TLBs; however, it may increase the TLB miss rate and hence degrade performance.

3.2.4 Flushing the L1 TLB. When the OS modifies the page table, the stale TLB entries must be flushed. This happens when the OS executes the `sfence.vma` instruction to invalidate an entry. Using the index we retrieve the set that includes the entry to be flushed and perform a fully-associative lookup within that set using the *tag*. The flushing of the TLB is done by zeroing the valid bit of the specified entry.

Conf. No	DTLB	ITLB	L2 TLB	DTLB Reach	ITLB Reach	L2 TLB Reach
I	fully-assoc., 32 entries	fully-assoc., 32 entries	-	128KB	128KB	-
II	fully-assoc., 32 entries	fully-assoc., 32 entries	4-way, 128 entries	128KB	128KB	512KB
III	fully-assoc., 32 entries	fully-assoc., 32 entries	4-way, 512 entries	128KB	128KB	2MB
IV	8-way, 64 entries	8-way, 128 entries	8-way, 1024 entries	256KB	512KB	4MB
V	8-way, 128 entries	8-way, 64 entries	8-way, 1024 entries	512KB	256KB	4MB

Table 1: Rocket Chip L1 Instruction/Data TLB and shared L2 TLB configurations (Associativity /Size).

3.2.5 Limitations. We initially developed the configurable L1 TLB in an older Rocket Chip edition that supported both base and super page sizes in the same TLB. A constraint of a set-associative TLB structure that we must address concerns the page size: when the page size is unknown it is difficult to determine the least significant bits of the VPN in order to select a set [22, 23]. Therefore, we select to implement a configurable L1 TLB only for 4KB fixed page size. We also ported the configurable L1 TLB in a recent edition of the Rocket Chip in which this restriction is lifted, the TLB mechanism is separate for base/super pages, so our implementation of the configurable L1 TLB does not affect the superpage mechanism. More details about the Rocket Chip versions/commits that we modified are presented in Section 4.

3.3 Configurable L2 TLB Architecture

The L2 TLB was originally direct-mapped, an organization very simple in terms of replacement policies and TLB flushing. An address translation maps only to a unique TLB entry and as a result there is no need for a replacement policy. The *valid bit array* is kept in register banks and not in the `SyncReadMem` that the TLB entries are stored. Obtaining a value from a register bank is completed in the same cycle in contrast with the `SyncReadMem` that has a cycle delay. As a result the valid bit array of the L2 TLB can be read and updated on the same cycle. This has the benefit of manipulating the valid bit without accessing the TLB array. The valid bit array is constructed as a `Vec` of registers the same way as in the L1 TLB.

3.3.1 Lookup. The L2 TLB lookup mechanism is similar to that of the L1 TLBs. The only difference is that the lookup in the L2 TLB introduces additional cycle delay due to the `SyncReadMem` construct. As a result we use registers to hold intermediate state.

3.3.2 Refill. In case of a refill, the L2 TLB handles it similarly with the L1 TLB. The only difference is the use of *masks* to update a specific way in a set. Masks are a feature of the `SyncReadMem` construct to ease updating specific indexes inside a set.

3.3.3 Replacement Policies. To choose a replacement policy we must make a trade-off between area and performance. The pseudo-LRU replacement policy must keep track of the way access history and as a result impacts the total area when the TLB is large. On the other hand, using a random replacement policy has a nearly zero impact on the total area but may degrade performance. We implement both replacement policies for the L2 TLB. In Section 5 we choose to evaluate our set-associative design with the random replacement policy in favor of area constraints.

3.3.4 Flushing the L2 TLB. Flushing a TLB entry on a set-associative organization means that the entry must be located inside the selected set. In order to fetch the tags of the selected set there must be a cycle delay because of the `SyncReadMem` construct. To overcome this overhead and keep the flushing mechanism simple, we select to flush the whole set. Another approach would be to block the L2 TLB for one cycle to retrieve the set, and then flush the specific entry. We are considering implementing that in the future.

4 METHODOLOGY

In this section we describe our evaluation methodology, including the hardware/software tools, metrics, and configurations. We initially developed the configurable TLB Hierarchy on an older Rocket Chip commit (7cd3352, April 3, 2018) that supported the Xilinx ZCU102 platform. Our contributions consists of about 80 and 70 lines of Chisel code added for the L1 and L2 TLB. Unfortunately that repository does not track the recent changes in the Rocket Chip Generator. As a result, we opted to use the old Rocket Chip version for our evaluation with the Xilinx ZCU102 platform. In addition, to ensure the relevance and compatibility of our approach with more recent versions of Rocket Chip, we ported our design to a more recent version (27120ee, Jan 22, 2020) that also features new mechanisms such as a sectored L1 TLB to further improve the TLB reach for 4KB pages, and a separate fully-associative L1 TLB for super pages. Our changes amount to about 50 and 70 lines of Chisel code for the configurable L1 and L2 TLB respectively in the recent version. We validated our ported design using Verilator simulations, and we plan to evaluate it on other supported FPGA platforms.

4.1 Software and Hardware tools

We follow a two-step process during the development of our TLB hierarchy. At first, we evaluate the L1/L2 TLB using Verilator [24] to validate the correctness of our design and to remove any bugs; afterwards, we use the Vivado tools to compile our design for the Xilinx ZCU102 FPGA. In more detail, the development phase includes the following:

4.1.1 Verilator. Verilator is an open-source tool that produces high-performance cycle-accurate C++/SystemC hardware models. Using `assert-printf` statements debugging becomes easier as Verilator produces logs of high verbosity. We use the official riscv-tests [20] as a sanity check, and then orchestrate specific assembly tests that run upon the riscv-pk [19] (lightweight proxy kernel) which provides virtual memory support. Unfortunately, the downside of using Verilator is the slow emulation speeds in contrast with FPGAs.

4.1.2 Software tools. We use Sifive's Freedom-U-SDK [21] which sets up a minimal Linux environment. The Rocket Chip SoC boots

the lightweight Buildroot [7] distribution on top of Linux kernel 4.15.0 with 4KB pages. We add new Buildroot packages that include simple TLB tests to verify that our design is working as expected, tools to retrieve performance counter results, and finally the SPEC2006 benchmarks [10] (compiled using Speckle [9]). We modify the Berkeley-Boot-Loader (BBL) [19]–which initializes machine registers and then boots the linux kernel–to set up several performance counters such as ITLB/DTLB and L2 TLB misses using the `mhpmeventXX` registers.

4.1.3   FPGA Flow. We use Vivado 2018.1 Design Suite for synthesis and placement. Vivado provides also results regarding resource usage. To evaluate the impact of the TLB hierarchy on application performance, we run a subset of the SPEC2006int [10] benchmarks with different L1 Instruction/Data TLB and shared L2 TLB configurations. In all configurations we use a 4-way 32KB instruction cache and a 4-way 16KB data cache.

4.2   Metrics and Benchmarks

To evaluate our configurable TLB hierarchy we use the following metrics: (i) FPGA resource usage, i.e., flip-flops, look-up-tables (LUTs), and block RAM, (ii) TLB performance, i.e., TLB Misses-Per-Kilo-Instructions (MPKI), and (iii) System performance, i.e., Instructions-Per-Cycle (IPC), a performance metric that isolates the impact of TLB implementation on the critical path, ignoring the processor frequency. To evaluate the TLB and system performance we use benchmarks from the SPEC2006int suite [10]. We use them with the test input set due to the limited physical memory (512MB) that our Xilinx ZCU102 platform exposes to the programming logic.

4.2.1   Configuration scenarios. We evaluate our configurable TLB hierarchy using different configurations for the L1 Instruction/Data TLB and shared L2 TLB. Table 1 summarizes the evaluated configurations. We choose these configurations to cover a range of systems from small and embedded up to modern high-performance general-purpose systems. The TLB reach (i.e., number of entries × page size) covered by the L1 ranges from 128KB to 512KB, and for the L2 is up to 4MB. In the most lightweight configuration we choose not to include an L2 TLB to quantify the performance and area differences of the different configurations. Finally, in the most performant TLB configurations (Configurations IV, V) we swap the size of the Data and Instruction TLB to identify possible changes in performance without changing the L2 TLB. Note that in our evaluation we do not include a PTW Cache. Finally, the configuration scenarios are chosen to resemble well-known architectures:

   I. Vanilla Rocket Chip without L2 TLB
  II. Vanilla Rocket Chip including small L2 TLB
 III. ARM Cortex A57 [3]
  IV. Intel Skylake [11]
   V. Intel Skylake with swapped Instruction/Data TLB sizes.

5   RESULTS

In this section we evaluate our configurable TLB hierarchy. The purpose of our evaluation is twofold: (i) to show that the generated designs have minimal impact on area and frequency, and (ii) to show how TLB configurability affects performance.

5.1   Area and Frequency Results

Figure 3 shows the area results for the various configurations. We present the total area of the Rocket Chip SoC as reported by the Vivado 2018.1 Implementation stage. Note that the Instruction/Data L1 TLB structures use FFs and the shared L2 TLB uses BRAMs.
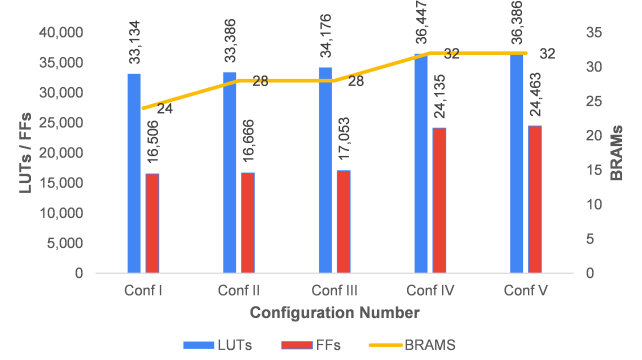

Figure 3: Area results for different TLB configurations.

Configuration	I	II	III	IV	V
Frequency (Mhz)	189	187	186	188	186

Table 2: Maximum operating frequency per configuration.

In the most lightweight scenarios (Conf I, II, II) Vivado 2018.1 reports that the full Rocket Chip SoC occupies 12% of the total LUTs, 3% of the total FFs, and 3% of the total BRAMs of the Xilinx ZCU102. Tuning up to the most performant configurations (Configuration IV, V) in terms of TLB hit rate, the Rocket Chip SoC occupancy increases to 13% for total LUTs, and 4% for total FFs/BRAMs. The FF usage is increased in Conf IV, V in order to accommodate the new TLB entries.

Table 2 shows the maximum frequency achieved with all configurations. The results show that the impact on the maximum operating frequency ranges from 0.53%-1.59%. In particular, Configuration IV has a 2× larger DTLB, 4× larger ITLB, and a 1024 entry L2 TLB, but exhibits only a 0.53% drop in frequency compared to Configuration I.

5.2   Performance Results

We now present the results of the SPEC2006int benchmarks that we obtained on the Xilinx ZCU102 FPGA board.

Figure 4 shows the results of MPKI in the L1 Instruction/Data TLBs for the various configurations. We observe that gobmk, hmmer, sjeng and libquantum exhibit similar behavior in L1 TLB MPKI even with larger TLB configurations. The most demanding in terms of TLB miss rate is mcf, and even with the largest Configuration V the miss rate is still high. For Configuration IV - V the miss rate is nearly the same, with Configuration V performing better in all tests. Most misses come generally from the Data TLB.

Figure 5 shows the results of the MPKI in the L2 TLB for the various configurations. The Configuration I is not included, as it lacks an L2 TLB. We observe that the L2 TLB MPKI for most benchmarks is nearly zero, particularly for the larger Configurations IV and V, thanks to the larger reach of the L2 TLB. There is also a
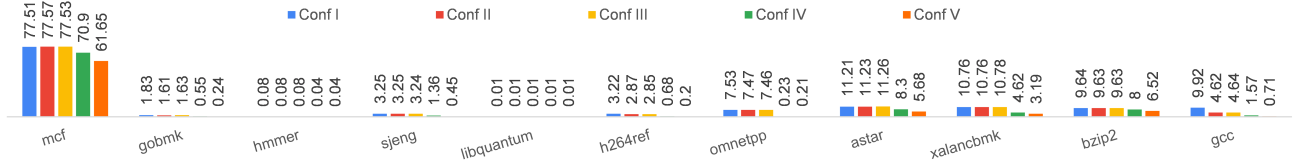
Figure 4: Aggregated MPKI of the L1 Data/Instruction TLBs for the various TLB configurations.
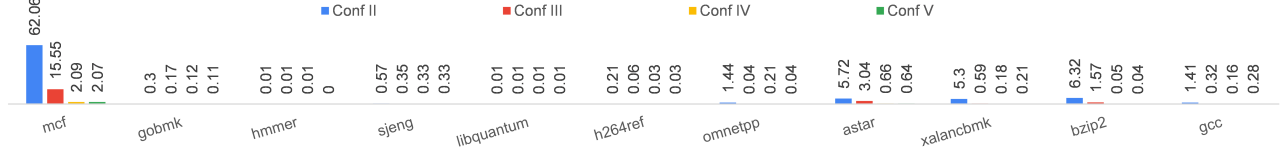

Figure 5: MPKI of the L2 TLB for the various TLB configurations.

major improvement in mcf which stresses the most the L2 TLB. On average, the miss rate for the L2 TLB is nearly zero with the larger Configurations IV and V.

Focusing on the impact of associativity, Table 3 shows the number of L2 TLB misses for mcf as we increase the L2 TLB associativity but keep the number of L2 TLB entries constant. The L1 Instruction/Data TLB parameters are based on those of Configuration V. We observe that there is an 82.8%/83.3% reduction in TLB misses when associativity changes from direct-mapped to 4-way/8-way. This behavior highlights the possible impact on the miss rate that a direct-mapped TLB can have due to conflicting entries, and the benefits of using a set-associative TLB. Note, however, that such behavior depends on the working set of the application and its access pattern, and that our results are for the Spec2006int benchmarks with the rather small test input set, as explained in Section 4.

L2 TLB Associativity	Direct-mapped	4-way	8-way
#TLB Misses for mcf	40.2M	6.9M	6.7M

Table 3: Number of L2 TLB misses for mcf as L2 TLB associativity increases, with Conf. V and fixed 1024-entry L2 TLB.

Finally, Table 4 summarizes the absolute IPC value with Configuration I, and the IPC speedup for the configurations II-V with respect to Configuration I. As we can see the IPC performance increases by up to 15.4 % depending on the demand of TLB resources and access patterns of every benchmark.

6 RELATED WORK

Prior work has focused on developing new MMU features for the Rocket Chip Generator in order to improve performance (e.g., Direct Segments for RISC-V [14]) while future work could investigate alternative techniques (e.g., Coalesced [18] and Clustered TLBs [17], Redundant Memory Mappings [12], and Hybrid TLB Coalescing [16]) to enhance the MMU performance. Another line of prior work has focused on bridging the FPGA-to-ASIC performance in order to gain more insights about the actual performance of a processor (e.g., [6, 13, 15]) to be fabricated and to also lower resource usage. Furthermore, Content-Addressable-Memories (CAMs) are known to be resource-hungry structures [26]. Magyar et al. proposed Golden Gate [15] to create Decoupled FPGA-accelerated

Benchmark	I	II	III	IV	V
mcf	0.13	-	7.7 %	15.4 %	15.4 %
gobmk	0.44	-	-	2.3 %	2.3 %
hmmer	0.58	-	-	-	-
sjeng	0.55	1.8 %	1.8 %	1.8 %	3.6 %
libquantum	0.44	-	-	-	-
h264ref	0.77	1.4 %	1.4 %	2.6 %	2.6 %
omnetpp	0.35	2.9 %	5.7 %	5.7 %	5.7 %
astar	0.36	-	-	2.8 %	2.8 %
xalancbmk	0.36	2.8 %	8.3 %	8.3 %	8.3 %
bzip2	0.51	2.0 %	4.0 %	5.9 %	5.9 %
gcc	0.44	2.2 %	2.2 %	4.5 %	4.5 %

Table 4: Absolute IPC values for Conf. I and percentage of IPC increase for Conf. II to V with respect to Conf. I.

Simulators by replacing FPGA-hostile CAMs with multi-cycle models, thus reducing resource utilization. As fully associative TLBs are typically implemented as CAMs, future work on resource optimization for large fully associative TLB organizations could leverage such FPGA-simulated research frameworks.

7 CONCLUSIONS

In this paper we explored the Memory Management Unit of the Rocket Chip Generator and lifted its implementation limitations in the TLB hierarchy. We implemented a fully configurable L1 and L2 TLB, that can output any design from direct-mapped to fully-associative. Our design enables design space exploration and allows the Rocket Chip Generator to instantiate cores with TLBs that match the needs of TLB intensive applications. We will make our design publicly available to enable further research on the active topic of virtual memory support for the RISC-V architecture.

ACKNOWLEDGMENTS

REFERENCES

[1] A. Waterman and K. Asanovic. 2017. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.10.

[2] Andrew Waterman and K. Asanovic. 2017. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2.

[3] ARM. [n.d.]. ARM Cortex-A57 Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0488c/DDI0488C_cortex_a57_mpcore_r1p0_trm.pdf

[4] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[6] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanovic. 2019. FASED: FPGA-Accelerated Simulation and Evaluation of DRAM. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 330–339. https://doi.org/10.1145/3289602.3293894

[7] Buildroot. 2017. Buildroot manual. https://buildroot.org/downloads/manual/manual.html

[8] Christopher Celio, David A. Patterson, and Krste Asanović. 2015. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Technical Report UCB/EECS-2015-167. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[9] Christopher Celio. [n.d.]. Speckle: A wrapper for the SPEC CPU2006 benchmark suite. https://github.com/ccelio/Speckle

[10] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[11] Intel. [n.d.]. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

[12] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal. 2015. Redundant Memory Mappings for fast access to large memories. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 66–78.

[13] Donggyu Kim, Christopher Celio, David Biancolin, Jonathan Bachrach, and Krste Asanovic. 2017. Evaluation of RISC-V RTL with FPGA-Accelerated Simulation.

[14] Nikhita Kunati and Michael M Swift. 2018. Implementation of Direct Segments on a RISC-V Processor. (2018).

[15] A. Magyar, D. Biancolin, J. Koenig, S. Seshia, J. Bachrach, and K. Asanović. 2019. Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8.

[16] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 444–456. https://doi.org/10.1145/3079856.3080217

[17] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 558–567.

[18] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, USA, 258–269. https://doi.org/10.1109/MICRO.2012.32

[19] RISC-V Foundation. [n.d.]. riscv-pk. https://github.com/riscv/riscv-pk/tree/master/bbl

[20] RISC-V Foundation. [n.d.]. riscv-tests. https://github.com/riscv/riscv-tests

[21] Sifive. [n.d.]. Freedom-U-SDK. https://github.com/sifive/freedom-u-sdk

[22] Madhusudhan Talluri and Mark D. Hill. 1994. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*. Association for Computing Machinery, New York, NY, USA, 171–182. https://doi.org/10.1145/195473.195531

[23] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. 1992. Tradeoffs in Supporting Two Page Sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*. Association for Computing Machinery, New York, NY, USA, 415–424. https://doi.org/10.1145/139669.140406

[24] Wilson Snyder. [n.d.]. Verilator Manual. https://www.veripool.org/wiki/verilator/Manual-verilator

[25] H. Wong, V. Betz, and J. Rose. 2014. Quantifying the Gap Between FPGA and Custom CMOS to Aid Microarchitectural Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 10 (2014), 2067–2080.

[26] H. Wong, V. Betz, and J. Rose. 2014. Quantifying the Gap Between FPGA and Custom CMOS to Aid Microarchitectural Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 10 (2014), 2067–2080.