Implementation of Direct Segments on a RISC-V Processor

Nikhita Kunati
Department of Computer Sciences
University of Wisconsin-Madison
Madison, WI, USA
kunati@cs.wisc.edu

Michael M. Swift
Department of Computer Sciences
University of Wisconsin-Madison
Madison, WI, USA
swift@cs.wisc.edu

ABSTRACT

Page-based virtual memory has been ubiquitous for decades and provides a secure way to manage scarce physical memory. But, with increasing memory sizes, paging may no longer be a good idea. Past analysis shows that big-memory workloads can spend 5%-50% of execution cycles on TLB misses. Previously proposed Direct Segments remove TLB miss overhead by mapping a part of process's virtual address space directly to contiguous physical memory. However, Direct Segments were evaluated using a simple model based on counting TLB misses, and hence the full architectural impacts are not known.

We attempt an implementation of Direct Segments on a RISC-V Rocket core, with software support in the Linux kernel. We add three supervisor level registers—base, limit and offset—and modify TLB miss handling to create TLB entries automatically for addresses within the Direct Segment without a page-table walk. We modify the RISC-V Linux kernel to reserve a region of physical memory and allocate a contiguous region of virtual address space to support Direct Segments and to change segment registers when changing address spaces.

We found that implementing Direct Segments for RISC-V forced us to further refine the segment interface, and consider where best in the translation path to implement segments Our prototype of Direct Segment consists of about 50 lines of Chisel code added to the Rocket core and about 400 lines of code added to the RISC-V Linux kernel.

ACM Reference format:
Nikhita Kunati and Michael M. Swift. 20187. Implementation of Direct Segments on a RISC-V Processor. In *Proceedings of Second Workshop on Computer Architecture Research with RISC-V , Los Angeles, CA USA, June 2018 (CARRV 2018),* 5 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Page based virtual memory, first implemented in the Atlas system [7], has largely been unmodified since the late 1960s when Translation-lookaside buffers (TLBs) were proposed. However, the usage of virtual memory by applications has changed significantly because of the growing physical memory capacities. Servers today are equipped with terabytes of RAM. For example, Windows Server 2016 supports up to 24 terabytes of physical memory, which is 6 times more than its predecessor. This growth in physical memory is expected to continue with improvements in memory technologies, especially non-volatile memory [11].

With an increase in physical memory sizes a number of applications use vast in-memory data like key-value stores in Memcached [5], in-memory databases [9, 12]. A previous analysis [2] shows that these applications spend 5%-50% execution cycles on TLB misses, which can take up to hundreds of cycles and increase energy consumption [3].

Direct Segments [2] help alleviate the TLB miss overhead incurred in these big-memory workloads. The main idea behind Direct Segments is to directly map a contiguous region of virtual memory to a contiguous region of physical memory. Three hardware registers *Base, Offset,* and *Limit* are used to perform the mapping. A virtual address lying in between base and limit bypasses the TLB and is instead translated to a physical address by adding an offset. Virtual addresses that do not fall between base and limit are mapped using conventional page-based virtual memory. The authors [2] emulated Direct Segments entirely in software by using the memory-hotplug feature to reserve a continuous region of physical memory and by triggering a fake page fault on TLB misses to check if the miss address falls in the Direct Segment region. This implementation is incorrect since it affects the contents of TLB.

This leads to our motivation of implementing Direct Segments in hardware and accurately measure the true benefits of using virtual memory that is directly mapped to a contiguous region of physical memory. RISC-V based processors have become a popular platform for architecture research because of their simple, fast and open-source design. We implement our design on a RISC-V based Rocket Chip [1] which leverages the Chisel hardware construction language to provide a library of sophisticated, highly parameterized generators which make up the synthesizable System-on-Chip (SoC). We add the three registers base, offset and limit to the Rocket Core and modify the TLB unit to perform a check on a TLB miss to see if the virtual address lies in between base and limit and add the offset to it to get the Physical address. TLB entries are created for the virtual addresses which are translated using Direct Segments.

We also modify the RISC-V Linux kernel to support Direct Segments. We reserve a contiguous region of physical memory using the contiguous memory allocator [8]. When a process uses a Direct Segment we allocate a contiguous region of virtual memory and populate the base, limit and offset registers. To develop assembly tests and test the kernel changes we implemented the design first on Spike (RISCV ISA simulator) [13] and RISCV-Qemu [4].

The rest of the paper is organized as follows. Section 2 describes the background specifically the main idea behind Direct Segments and how it was implemented and evaluated in prior work. Section 3 explains our design and implementation. Section 4 presents the

CARRV 2018, Los Angeles, CA USA
© 2018 Copyright held by the owner/author(s). This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of Second Workshop on Computer Architecture Research with RISC-V , June 2018*, http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn.

methodology adopted to develop and evaluate our design. Section 5 presents the lessons learned from our implementation and evaluation of Direct Segments. Section 6 concludes this paper.

2 BACKGROUND

Virtual memory has been an active research area for a long time with efficient TLB mechanisms being proposed and support for large pages added for various architectures like MIPS, Alpha, UltraSPARC, PowerPC, and x86. Prior work has produced many proposals to reduce TLB misses (e.g., clustered TLBs [10]). We are focusing on Direct Segments proposed in prior work [2] that enables fast and efficient address translation of big memory workloads by treating a some portions of virtual address space as segments and page mapping the rest.

Hardware support for Direct Segments. In order to enable fast translation for part of a process's address space that does not benefit from page-based memory, previous work introduces Direct Segments, which map a contiguous region of virtual address space directly to a contiguous region of physical address space. The virtual address region can be of any size and and requires only minimal hardware regardless of its size. Virtual addresses that do not fall in this contiguous region are page mapped using the conventional technique. The hardware needs three registers to support Direct Segments

- *Base* holds the start address of the contiguous virtual address region mapped through Direct Segment
- *Limit* holds the end address of the contiguous virtual address region mapped through Direct Segment.
- $Offset$ holds the start address of contiguous physical address region backing the Direct Segment minus the value Base.

The address translation involves the following steps. If the virtual address V falls in the contiguous virtual address range ($Base \leq V < Limit$) the hardware will return physical address as $V + Offset$. A given virtual address can be translated using Direct Segment or page mapping but not both. The OS is responsible for populating the Base, Limit and Offset registers.

Software support for Direct Segments. The OS provides the abstraction of a *primary region* to applications(primary processes), which allows them to specify a region of memory that does not benefit from paging and can use a Direct Segment. In order to support primary regions two things need to be done. First, the OS needs to reserve a contiguous region of physical memory which is usually done at boot time. Second, the OS needs to provision a contiguous virtual address range. It does this during the creation of a process by reserving a portion for memory allocations in the primary region. This partition should be large enough to encompass the largest possible primary region.

Figure 1 shows the virtual address space of a process with a primary region and how the physical address space looks in the presence of both Direct Segments and page-mapped memory. The lightly shaded box represents the primary region and the dark narrow rectangles represent the conventional pages

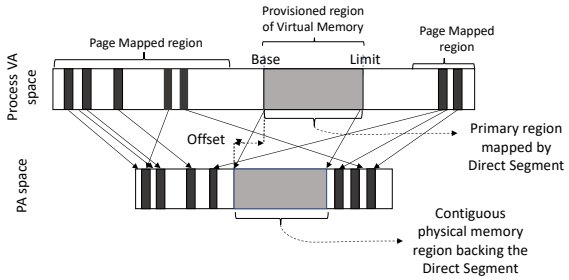

Figure 1: Virtual address and physical address space layouts with primary region.

The OS is also responsible for setting up the *Base*, *Limit* and $Offset$ registers. When a primary process is found the OS reserves a physical memory region and allocates a contiguous range of virtual addresses. On process context switches, the OS saves and restores these registers.

Previous Evaluation of Direct Segments. Direct Segments were emulated in software and by tricking an x86-64 processor into trigger a page fault on every TLB miss using a technique like BadgerTrap [6]. Page tables initially mark all pages invalid. On any reference to memory the processor will trap, at which time the emulation code marks the page valid, touches a byte on the page to add the translation to the TLB, and then marks the page table entry invalid again. Thus, while the page is in the TLB it can be re-referenced without a fault, but once it leaves the TLB, the page-walk code will trigger a fault. The emulation code determines on each page fault whether the faulting address lies within a Direct Segment, and then predicts overall performance from the fraction of TLB misses that would be avoided. This approach is much faster than full-system simulation and allows running workloads with tens of gigabytes of data, but potentially less accurate.

Problems with the previous implementation. The software emulation used to evaluate the original Direct Segments proposal had several inaccuracies that we improve in our work.

- While the original design called for checking for Direct Segment in parallel to L1 TLB lookup, the emulation code only checks a Direct Segment on an L2 TLB miss. Thus, addresses in Direct Segments will pollute both L1 and L2 TLBs and many accesses to a Direct Segment will pay the L2 TLB latency.
- The emulation code cannot determine how many cycles would be saved from access to a Direct Segment, as the CPU may have overlapped other work with a TLB miss that would now be on the critical path.
- The work did not include effects on pipeline timing from adding comparisons to the *Base* and *Limit* registers.

For these reasons, we developed an implementation of Direct Segments on RISC-V.

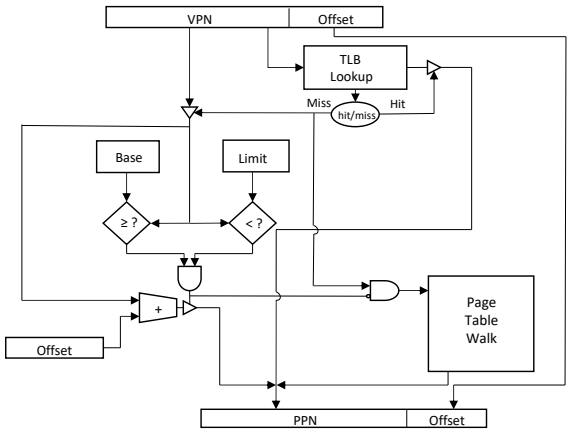

Figure 2: Logical view of address translation with Direct Segment Hardware.

3 DESIGN

We implement Direct Segments as described in the previous section on a RISC-V based Rocket Chip and add software support by modifying the RISC-V Linux Kernel.

3.1 Hardware Support in Rocket Core

We use the Berkeley Rocket chip SoC generator and modify it to create a 64-bit RISC-V core that supports Direct Segments. The Rocket Chip generator supports both an in-order Rocket core as well as an out-of-order BOOM core, and both instantiate a common TLB and page table walker unit. Rocket core has 5 pipeline stages the I-TLB access happens in the instruction fetch stage and D-TLB access happens in the data memory access stage.

We prototype Direct Segments only for data memory accesses. Using the Supervisor level Page Table Register (SPTBR/SATP) as our model, we add support for three supervisor level registers: Supervisor Direct Segment Base (SDSB), Supervisor Direct Segment Limit (SDSL), and Supervisor Direct Segment Offset (SDSO) to store the base, limit and offset required for Direct Segment lookup. The least significant bit of SDSL is the *enable* bit, which although not a big addition was not proposed in the original design and provides the functionality to enable/disable Direct Segments on a per-process basis. Bit[1] of SDSL represents the protection bit. If it is set to 1 we set read/write permissions for the direct segment region and otherwise set read only. Figure 2 shows our hardware design. We perform the Direct Segment lookup on a TLB miss, unlike the original design where it was proposed to be done in parallel with the TLB lookup. We chose this implementation because of the ease of integrating the Direct Segment lookup into the existing TLB unit in Rocket. We plan to implement the Direct Segment lookup in parallel to the TLB lookup, and within the page table walker unit and compare the performance and timing impact of the three designs. Our prototype of Direct Segment consists of about 50 lines of Chisel code added to the Rocket core

We modify the TLB unit in Rocket to check on a TLB miss, if Direct Segments are enabled (*enable* bit in SDSL is set), and then check if a given virtual address lies in between the base and limit. If so, it computes the physical address by adding offset to virtual address. We introduce a new signal *ds_miss* which is set if the Direct Segment lookup is unsuccessful. Figure 3 shows the state diagram of the TLB unit in Rocket. The TLB is in state *s_ready* when there is a translation request(TLB_req) or when the Page Table Walker(PTW) has sent a request to refill the TLB, it moves to *s_request* state when the address request misses in both the TLB and the Direct Segment look up (*ds_miss* is set). The TLB is in *s_wait* state when PTW is ready and is processing a request and TLB is in *s_wait_invalidate* state when a sfence instruction is encountered.

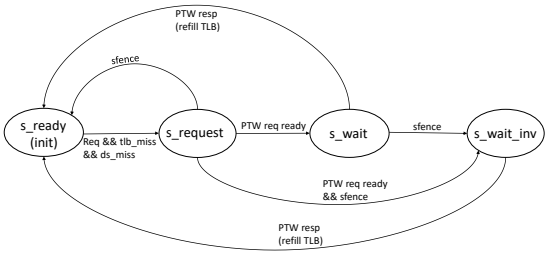

Figure 3: TLB unit's state diagram.

3.2 Software Support: RISC-V Linux kernel

We modify the RISC-V Linux kernel to provide two basic functionalities to support Direct Segments. First, the OS provides a Primary Region as an abstraction for the application to specify which portion can benefit from Direct Segments. Second, the OS reserves a portion of Physical memory and maps to the primary region by configuring the Direct Segment registers.

Primary regions are a contiguous range of virtual addresses with a uniform read-write access permission. We allocate a primary region when a primary process is detected. The size of the primary region is specified by the application with a new system call *create_primary_region()*. Our prototype employs an "opt-out" policy where all memory allocations with read-write permissions are put in the primary region unless explicitly specified by the application. This way, all heap allocations and *mmap()* calls for anonymous memory are placed in the address region reserved for primary region unless explicitly requested otherwise by the application with a flag to *mmap()*.

Managing the Direct Segment Hardware. The OS sets up the Direct Segment hardware by first allocating a contiguous region of physical memory to back the primary region and then it sets up SDSB, SDSL and SDSO for the hardware to perform Direct Segment lookup. To make contiguous physical memory available we use the Contiguous Memory Allocator [8] which allows for large contiguous allocations by requiring just a few changes to the kernel code. We use *_contigous_alloc()* to reserve memory at boot time and use *dma_alloc_from_contiguous()* to allocate CMA memory for a primary process. When a process requests it, we allocate a primary region and set up the Direct Segment registers SDSB, SDSL and SDSO. We also modify the context switch code to save and

restore these registers. Our prototype of Direct Segment consists of about 400 lines of code added to the RISC-V Linux kernel.

4 METHODOLOGY

In this section we describe our methodology for implementing and testing Direct Segments. We first implemented the hardware changes on Spike a RISC-V ISA simulator and developed assembly tests to test the hardware changes. We then implemented the hardware changes in the existing Rocket Core, and used the Verilator to run assembly tests and test the Chisel implementation. In order to test the RISC-V Linux kernel changes we also implemented the hardware changes in RISC-V Qemu.

Spike is a RISC-V ISA simulator that is a functional model of a RISC-V processor. We first implement the Direct Segment hardware changes to Spike before Rocket to check the feasibility of our design and devise some simple tests which can be run and debugged easily on Spike. We modify the *walk()* function which is accessed on an I-TLB or D-TLB miss to perform a direct segment lookup only on data memory access.

RISC-V Qemu: Qemu is an open source machine emulator capable of emulating a machine entirely in software using dynamic translation and supports emulation of multiple architecture including RISC-V. We modify RISCV-Qemu, specifically the *get_physical_address()* function, to perform the Direct Segment lookup before the page table walk. It checks if the virtual address lies in between base and limit and if it is computes the physical address as virtual address + offset and return translation success. We implement Direct Segments in Qemu because of the ease of testing RISC-V Linux Kernel changes on top of it. Booting the kernel on RISC-V takes a few seconds as compared to verlilator where it could take hours.

5 LESSONS LEARNED

In this section we describe our experience using the RISC-V ecosystem to implement Direct Segments. Specifically, we describe the RISC-V ecosystems successes as well as the challenges we faced while using the various components of the RISC-V project.

RISC-V Ecosystem Successes. The RISC-V ecosystem provides all the tools necessary to prototype the Direct Segment hardware well. We were able to start off from a well defined instruction-set with the ease of adding new registers and instructions. Generating a RV64 core required modifying a configuration script. We also used Spike the RISC-V ISA simulator which enabled fast testing of our design before we made the necessary hardware changes to Rocket. The RISC-V assembly test suite is comprehensive and serves as a model to develop new tests which simplifies verification. The Verilator which converts the converts Verilog to cycle accurate C++ model was essential to test the changes we made to the Rocket Core. We were also able to boot our modified RISC-V Linux kernel successfully on the Verilator. The RISC-V ecosystem played an important role in enabling a first year graduate student to implement an efficient virtual memory management design in hardware and develop necessary software for it all in a matter of few months.

RISC-V Ecosystem Challenges. Despite the flexibility provided by the RISC-V ecosystem we faced some challenges. The most significant challenge was to successfully build and test all the components of the RISC-V ecosystem this was because of the rapid pace of development both within each RISC-V project and across the full RISC-V ecosystem. For example, when we started making changes to the TLB unit of rocket we found that within a month a newer version of Rocket was released with significant changes to the TLB unit causing us to redo our implementation. We also faced issues in finding the right version of RISC-V kernel that boots across all the platforms Spike, RISC-V Qemu and Verilator. We found that documentation across RISC-V projects is either missing or not sufficient to build a working system. For example, there were multiple sources listing the steps to build the RISC-V linux kernel but some of the steps were outdated and resulted in either us failing to build the kernel or failing to boot it on a specific platform.

Obtaining performance results from Rocket using was challenging due to the lack of performance monitoring tools. Unlike a simulator that can easily be extended with arbitrary counters, we implemented performance counters within the Rocket implementation and write test code to extract the results.

Finally, making changes to existing Rocket code was not trivial because of the lack of comments explaining the flow in a particular unit and across multiple units. Due to the lack of well documented performance counters we had to add logic in Rocket to enable a performance counter that measures TLB misses.

RISC-V Linux Kernel Challenges. Kernel support is essential for any new architecture to enable operating systems research. Though basic support for RISC-V was added in Linux Kernel 4.15 it was sufficient to boot and not much else. For example, it does not support interrupts thus cannot support devices. We found that porting kernel code developed for x86 faced unexpected challenges because of this primitive support for RISC-V in Linux. For example, we could not reserve physical memory using memory hotplug which was used for the x86 software prototype because this support was not present for RISC-V. We hence found an alternative—Contiguous memory Allocator(CMA). Another challenge we faced was that the RISC-V Linux kernel was constantly under development and getting the right version of the kernel which would boot on Qemu and Verilator was difficult. Had there been coordinated releases of all components (Qemu, Verilator, Linux, etc.), or at least documented stable configurations, this would have been simplified.

6 CONCLUSION

In this paper we present an implementation of an efficient virtual memory—Direct Segments—which can be used along with traditional page-based virtual memory to reduce the TLB-miss overhead found in server workloads. To achieve this, we make changes to the TLB-unit in Rocket and add software support by modifying the RISC-V Linux Kernel. The flexibility of the RISC-V ecosystem has played a major role in enabling our design and implementation of Direct Segments in a full system with an application execution environment. Our preliminary results show that the TLB-Miss overhead has reduced significantly and we plan to do further analysis on where to perform the Direct Segment lookup in hardware to get the best performance. Based on our experience, we believe that RISC-V is a promising platform for future research on Virtual Memory.

REFERENCES

[1] K. Asanovifj, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[2] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 237–248, New York, NY, USA, 2013. ACM.

[3] A. Basu, M. D. Hill, and M. M. Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 297–308, Washington, DC, USA, 2012. IEEE Computer Society.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[5] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, Aug. 2004.

[6] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift. BadgerTrap: A Tool to Instrument x86-64 TLB Misses. *SIGARCH Comput. Archit. News*, 42(2):20–23, Sept. 2014.

[7] S. H. Lavington. The manchester mark i and atlas: A historical perspective. *Commun. ACM*, 21(1):4–12, Jan. 1978.

[8] M. Nazarewicz. A deep dive into CMA. https://lwn.net/Articles/486301/, 2012.

[9] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, Jan. 2010.

[10] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 258–269, Washington, DC, USA, 2012. IEEE Computer Society.

[11] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev.*, 52(4):465–479, July 2008.

[12] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[13] A. Waterman and Y. Lee. Spike - RISC-V ISA Simulator. https://github.com/riscv/riscv-isa-sim.