Hardware Acceleration for Memory to Memory Copies

Howard Mao
Randy H. Katz, Ed.
Krste Asanović, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-2
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-2.html

January 13, 2017

Hardware Acceleration for Memory to Memory Copies

by Howard Mao

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Randy Katz
Research Advisor

(Date)

Professor Krste Asanovic
Second Reader

(Date)

Abstract

As performance improvements from transistor process scaling have slowed, microprocessor designers have increasingly turned to special-purpose accelerators to continue improving the performance of their chips. Most of these accelerators deal with compute-heavy tasks like graphics, audio/video decoding, or cryptography. However, we decided to focus on a memory-bound task: memory to memory copies. Memory to memory copies make up a significant portion of data center workloads, so improving its performance could lead to large savings in operational cost. To that end, we designed a memory copy accelerator which can move data at high bandwidth within the L2 cache and main memory. Unike traditional DMA engines, this copy accelerator is virtual memory-aware and can perform data transfers without any need for page pinning or ahead-of-time page translation. This relieves much of the programming burden from the operating system developer and application programmer. We compared the performance of this accelerator to memcpy() functions implemented with scalar RISC instructions and with vector instructions. Our evaluation showed that the copy accelerator was significantly faster than the scalar implementation for larger transfers, even when accounting for the overhead of page faults. In addition, the copy accelerator matched the performance of the vector implementation, while taking up an order of magnitude less area than the vector coprocessor.

Contents

List of Figures

List of Tables

1 Introduction

As performance gains from transistor shrinks have slowed, microprocessor designers have increasingly turned to adding special-purpose accelerators to continue improving the performance of their chips. These accelerators generally focus on computation-heavy tasks, such as graphics audio/video decoding, cryptography, and, more recently, machine learning. In this work, we focus on accelerating a much more mundane task: data movement.

Data movement within memory makes up a significant proportion of many workloads. An investigation of a Google datacenter by Kanev, et al. [4] found that approximately 5% of CPU cycles were spent inside the memcpy() and memmove() library routines. Memory-memory copies are used especially heavily in kernel IO drivers (such as disk and network device drivers), which spend most of their time copying data between device, kernel, and userspace buffers.

There has been quite a bit of work done to reduce the overhead of copying data between IO devices and main memory. Traditional DMA engines can lower the CPU overhead of copying from device buffers to main memory by offloading this copying to the device itself. However, the DMA engine generally does not have access to the page table, so it cannot use virtual addresses. Since pages in userspace are usually not guaranteed to be resident in memory, the operating system device drivers must allocate kernel-space buffers for the DMA engine to write into and then copy the data again into the userspace buffer.

Some messaging technologies, such as Infiniband, have sought to completely obviate additional copies by having the device write directly into the userspace buffer. However, these so-called "zero-copy" DMA protocols shift a lot of the burden of buffer management from the operating system developer to the application programmer. The application must explicitly allocate memory for DMA and pin the virtual addresses in memory. This is potentially quite expensive, since the operating system must translate all of the virtual addresses and page them into physical memory up front. Also, while the pages are pinned, the physical frames cannot be reused by the operating system for other tasks. The application must also manage any concurrency between the device's use of the memory region and its own. Application frameworks can hide the complexity of low-level API calls, but this usually comes at the cost of a significant performance overhead.

To solve these issues, we propose a coprocessor that can offload memory-memory copies. The coprocessor has access to the CPU's page table walker and thus can perform its own virtual memory translations. It only requires intervention from the processor if the page it is translating is not resident in memory and needs to be paged in. Once the page has been moved into memory, the accelerator can resume the copy operation at the point it left off. This removes the need to allocate a separate kernel buffer or pin userspace buffers in memory.

2 Related Work

The most common form of memory-memory copy acceleration is to use vectorized or SIMD load-store operations. This is the tactic taken by the glibc memcpy() function. On architectures which have these sorts of instructions (such as modern x86 machines), the memcpy

function moves data through the wide SIMD registers. To investigate how our approach compares to this one, we also benchmarked memory-memory copies using Hwacha, a vector coprocessor we developed in our lab.

One interesting design by Duarte and Wong [3] removed the need for any data movement at all by simply keeping an indirection table in the cache. Performing a copy simply placed an entry in the table. Then, for any read access to the destination region, the data would be pulled from the source region's index in the data array. There was additional logic to invalidate the indirection table entries if the source or destination regions were modified. While this improved the performance of copy operations, it was limited by the fact that the memory region to be copied had to fit into the cache. You could not copy a region larger than the cache size using this method. Also, since the indirection table had to be checked for every cache access, there was some additional overhead for regular cache accesses that did not use the indirection. This may end up affecting the critical path of the processor.

A design by Su, et al. [5] overcomes these limitations. Their accelerator actually performs the memory movement from main memory to the cache. Their backend design is very similar to ours. There are multiple DMA channels that can run independently, the read and write requests can be pipelined, and the DMA transactions can be paused at certain points and then resumed. One key difference is that their accelerator sits outside of the core and receives commands via the memory interface rather than through custom instructions.

Intel's I/O Accelerator Technology (IOAT) provides support for copy acceleration through the asynchronous DMA copy engine (ADCE). This is a PCI-enumerated device that can perform the DMA copy asynchronously. Unlike our design, it accesses memory directly instead of going through the L2 cache. Therefore, to keep things coherent, it must probe into the L2 cache to force it to writeback any dirty data. This design might be good for large transfers for which you would like to avoid cache pollution. But for smaller transfers it may be inefficient because the application must go through a potentially lengthy cache refill when it actually reads the data. The ADCE uses only physical addresses and is only accessible from userspace. Work by Vaidyanathan, et al. [7] exposed the ADCE to userspace through a kernel module. The kernel-space software took care of splitting the userspace request on page boundaries and performing the page translation.

Other than the SIMD instructions, all of these prior acceleration technologies lack the ability to translate virtual addresses. Therefore, the operating system must perform the address translation ahead of time and pass physical addresses to the accelerator. In addition, the physical frames must be pinned in memory and cannot be paged out or relocated. A design by Tezuka, et al. [6] attempts to alleviate the overhead and high resource usage of page pinning in a zero-copy DMA system by implementing a pin-down cache. The system specifies a maximum number of pinned pages possible. If the application requests to release a page it has pinned, the page is not immediately unpinned. Instead, it is marked as being a candidate for unpinning and the actual unpinning is deferred until the the number of pinned pages exceeds the maximum allowed. When the application requests that a page be pinned, the system checks if the page is already pinned. If it is, nothing further need be done. If the page is not already pinned, the system checks if pinning the page will cause the number of pinned pages to exceed the maximum allowed. If so, then pages marked as released will be unpinned until the number of pinned pages goes below the maximum. This technique does provide more predictable virtual and physical memory usage and reduces the CPU overhead

of page pinning. However, it also limits the amount of DMA transfers that can be inflight at any given time. If the number of requested pages exceeds the maximum number of pinned pages allowed, some of the requestors will have to wait. For our design, we removed the need for page pinning by allowing the accelerator to perform its own address translation through a translation lookaside buffer. The accelerator can also detect if a page is not resident in memory and pass control back to the CPU for page fault handling.

3 Design

3.1 RISC-V ISA and memcpy() ISA extension

For this work, we rely heavily on the RISC-V instruction set [8]. The RISC-V ISA is an open source RISC instruction set with 32-bit and 64-bit variants. It has a simple base integer ISA and standard extensions for integer divide/multiply, atomic memory operations, and floating-point arithmetic. The instruction set also provides support for virtual addressing modes. In most RISC-V implementations that support virtual addressing, the L1 instruction and data caches have TLBs that perform the address translation. These TLBs are refilled from a page table walker that accesses the L1 data cache. If the pages are modified by the operating system, they need to be explicitly invalidated from the TLB using the FENCE.VM instruction.

A key feature of the ISA is that a part of the opcode space is reserved for custom non-standard ISA extensions. We use these custom instructions to communicate between the processor and the copy accelerator.

The custom instructions sent to the accelerator follow the format shown in Figure 1. The main sections are the opcode, funct field, and register numbers for two source registers (rs2, rs1) and one destination register (rd). The opcode and funct are used to distinguish different instructions. For our memcpy() extension, we always use opcode *custom2* and distinguish the instructions with the funct. The xs2, xs1, and xd fields are booleans used to determine whether or not the corresponding registers are being used by the instruction.

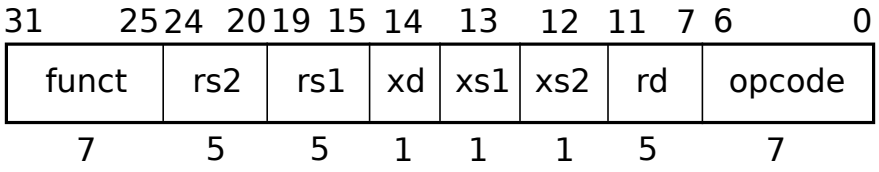

Figure 1: RoCC Instruction Format

The different instructions in the memcpy() ISA extension are shown in Table 1. The "Transfer" instruction initiates a DMA transfer. The first source register provides the destination start address and the second source register provides the source start address. If virtual addressing is turned on, the two addresses will both be virtual addresses. The accelerator can perform address translation using its own TLB. This accelerator TLB is refilled from the same page table walker as the L1 caches and can be invalidated using the FENCE.VM instruction.

The other information needed for a DMA request is kept in control registers. These values can be read using the "Read-CR" instruction, which returns the value of the register at the index given by the first source register. The registers can be modified using the "Write-CR", "Set-CR", and "Clear-CR" instructions, which updates the register at the index given by the first source register with the data supplied by the second source register. "Write" simply overwrites the entire control register, "Set" sets any bits in the CR that are set in the source register, "Clear" clears any bits which are set in the source register.

Instruction	funct	xs2	xs1	xd
Transfer	0	1	1	0
Resume	1	0	0	0
Read-CR	4	0	1	1
Write-CR	5	1	1	0
Set-CR	6	1	1	0
Clear-CR	7	1	1	0

Table 1: memcpy() extension instructions

There are seven control registers used by the accelerator, as listed in Table 2. The first four determine the "shape" of the memory transfer. The copy accelerator supports strided, segmented copies. A set of NSEGMENTS chunks of SEGMENT_SIZE bytes each are copied from the source to the destination. Each segment at the source is separated by SRC_STRIDE bytes and will be separated by DST_STRIDE bytes at the destination. This feature is useful for copying subsections of matrices. The segment size is the size of a row in the submatrix being copied. The number of segments is the number of rows in the submatrix. The stride is the difference between the size of a row in the larger matrix and the size of a row in the submatrix.

The ACCEL_CTRL register contains several bit fields, as shown in Table 3. The first two bits are the allocation bits. They control whether the source or destination data will be cached in the L2 if they are not already. By default, the source data is not cached but the destination data is. This avoids polluting the cache with duplicate data. The application developer may wish to cache the source data if it is going to be copied more than once, or they may wish to not cache the destination data if the transfer size is larger than the L2 size and they would like to avoid blowing out the cache. The pause bit can be set to cause the accelerator to halt its current transfer operation. The paused operation can be restarted later with the "Resume" instruction. To allow precise restarts, the accelerator can only halt at segment or page boundaries. This functionality is mostly used by the operating system in case it wants to deschedule a program that is waiting on an accelerator transfer.

Here's an example of how one would implement copying a subset of one matrix into a smaller matrix using the memcpy() ISA extension.

Index	Name	Description
0	SRC_STRIDE	Stride (in bytes) between source segments
1	DST_STRIDE	Stride (in bytes) between destination segments
2	SEGMENT_SIZE	Size (in bytes) of a single segment
3	NSEGMENTS	Number of segments
4	ACCEL_CTRL	Pause and alloc bits
5	RESP_STATUS	Accelerator page fault status
6	RESP_VPN	Faulting VPN

Table 2: memcpy() Control Registers

Name	pause	dst_alloc	src_alloc
Position	2	1	0

Table 3: ACCEL_CTRL register fields

```
void pagein(unsigned long vpn)
{
    int page_size = sysconf(_SC_PAGESIZE);
    void *vaddr = (void *)(vpn * page_size);
    mlock(vaddr, page_size);
    munlock(vaddr, page_size);
}
void wait_for_completion(void)
{
    int status;
    unsigned long vpn;

    while (true) {
        fence();
        status = read_cr(RESP_STATUS);
        if (status == 0)
            break;
        vpn = read_cr(RESP_VPN);
        pagein(vpn);
        resume();
    }
}
```

```
void copy_matrix(
    int *dst, int *src, int nrows, int dst_ncols, int src_ncols)
{
    write_cr(SRC_STRIDE, sizeof(int) * (src_ncols - dst_ncols));
    write_cr(DST_STRIDE, 0);
    write_cr(SEGMENT_SIZE, sizeof(int) * dst_ncols);
    write_cr(NSEGMENTS, nrows);

    transfer(dst, src);
    wait_for_completion();
}
```

After starting the transfer, the application executes the RISC-V "fence" instruction to synchronize between the accelerator and the CPU. This instruction will halt the CPU pipeline until all in-flight operations in the accelerator have completed. When the fence instruction has completed, the accelerator has either successfully finished the transfer, in which case the response status will be zero, or it has encountered some page translation error, in which case the response status will be non-zero. In the latter case, copies for the pages before the current one will have all completed and the internal state of the accelerator is saved. The faulting VPN is stored in the FAULT_VPN register.

If the application encounters a translation error, the proper response would be to read the faulted VPN from the control register and force the OS to page it in. The easiest way to do this in a Unix OS is to call the mlock() system call to pin the page in memory and then optionally call munlock() to allow it to unpin it. After the page is set up, the application should execute the "Resume" instruction to continue the transfer. This will flush the coprocessor TLB to clear out the old invalid page table entry and begin copying again starting from the page which produced the translation error.

3.2 RocketChip SoC Generator

We developed our memory copy accelerator using the RocketChip SoC generator [1]. This is an open-source SoC generator for processor systems implementing the RISC-V ISA. Using this generator, we are able to generate a processor system with the Rocket CPU, an in-order 64-bit RISC-V core.

The CPU is backed by a cache-coherent memory system. Each core contains separate L1 instruction and data caches. These caches connect over a crossbar to shared L2 cache banks. The physical address space of the processor system is strided across the L2 cache banks, so each cache bank handles a separate set of addresses from the others. The cache banks are backed by one or more DRAM channels. Cache banks are mapped to DRAM channels on an N-to-1 basis. Each L2 cache bank is refilled from a single DRAM channel and each channel handles a unique subset of the cache banks.

Alongside the cache-coherent memory system, there is a non-coherent memory-mapped IO system. This allows the CPU to access various peripherals over an MMIO crossbar. The generator provides a set of standard peripherals including boot ROM, programmable interrupt controller, and system clock. Additional peripherals, such as IO devices, can be

easily added. The MMIO network handles addresses that are distinct from those handled by the coherent memory system. Though the CPU connects to this system through the L1 caches, the data provided through MMIO is not meant to be cached. The data cache will never cache MMIO data. The instruction cache may cache either memory or MMIO data read-only. Unlike the data cache, the instruction cache will not automatically invalidate its stored data if it is modified by another client. To reflect any modifications to instruction cache data (as in the case of self-modifying code), the RISC-V program must execute the FENCE.I instruction, which invalidates all data stored in the instruction cache.

A key feature of RocketChip is the ability to easily integrate an accelerator with the CPU core through the Rocket Custom Co-processor (RoCC) interface. The RoCC accelerator is controlled by custom RISC-V instructions and shares processor resources such as access to the page table walker and L1 data cache. The accelerator can also be given dedicated ports into the L1-to-L2 crossbar for direct access to the L2 cache. This is the approach we take for the copy accelerator. All of the data movement is done through the L2 interface and the L1 interface is not used. We chose this design because the L1 data cache is usually quite small. Since the accelerator may be used to move large amounts of data and the L2 to L1 refill latency is relatively short, we thought it would be best not to risk evicting other important data from the L1.

A block diagram of the RocketChip system with our memory copy accelerator is shown in Figure 2.

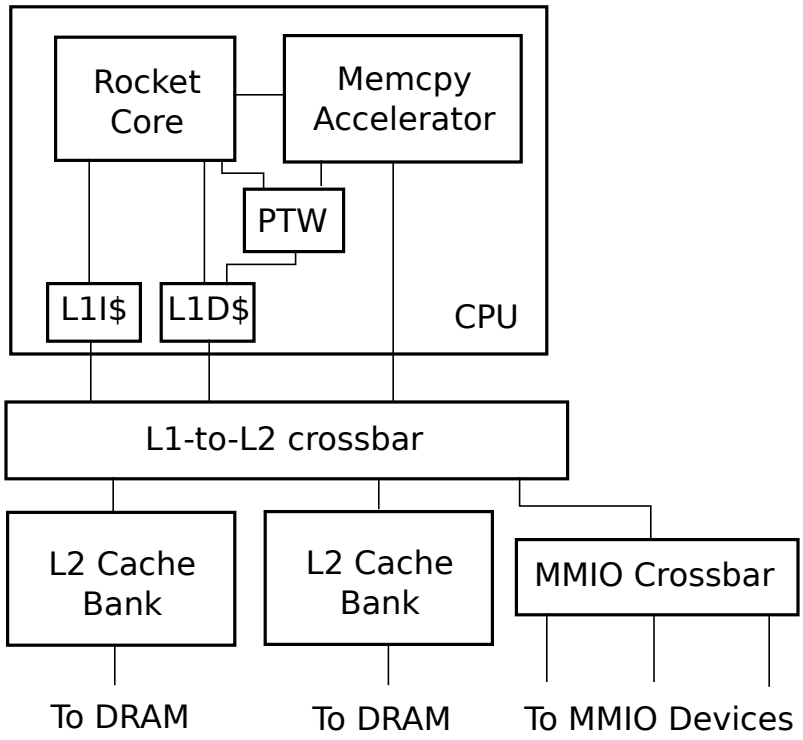

Figure 2: System Diagram

3.3 TileLink Memory Interface

The L1 caches and copy accelerator communicate with the outer memory system through the TileLink coherent memory interface [2]. This interface consists of five different channels between the client (higher level cache or non-caching client) and manager (lower level cache or memory).

- Acquire (client to manager) - Initiates a transaction to obtain increased permissions for the client on a cached block. Also used to read and write cache blocks without caching them.
- Probe (manager to client) - Queries if the client is caching a block. Can also downgrade the client's permissions on the cache block.
- Release (client to manager) - Responds to a probe with the client's current permissions on the block. Acknowledges the downgrading of permissions on the block and writes back dirty data. Also used to voluntarily write back data evicted from the client.
- Grant (manager to client) - Provides data and/or permissions to client in response to an acquire. Also used to acknowledge voluntary releases.
- Finish (client to manager) - Acknowledges the receipt and handling of a permissions upgrade from a grant, indicating to the manager that it is safe to accept further requests for the cache block.

For the copy accelerator, we are concerned with the non-caching protocol, which uses only the acquire and grant channels (although the manager may send probes to other clients in response to the acquire). The acquire channel can carry several types of non-caching messages. The four that we are concerned with are GetBlock, Get, PutBlock, and Put.

A GetBlock message requests data for an entire cache block. The TileLink data width is parameterizable and may not be as large as a cache block. If the data width is smaller, the data response will be sent back broken into several grant messages. These individual grants are called "beats". All of the beats must be transferred in-order and cannot be interleaved with other grant messages.

A Get message also requests data, but the amount requested must be less than or equal to a single beat. This way, only a single grant message must be sent back in response.

A PutBlock message writes the data for an entire cache block. If the data width is smaller than a block, the PutBlock will be broken up into multiple acquire messages (beats). The beats for a single PutBlock must be sent in full (no omitted beats), in order, and must not be interleaved with other acquire messages. The manager responds with a single grant message acknowledging the receipt of the data.

A Put message writes the data for a single beat. Thus, only one acquire message is sent for each Put transaction. A Put message contains a bytemask, which allows it to write a smaller amount of data than a single beat. In previous iterations of the TileLink protocol, any bytemask was acceptable. However, in the current specification, the number of bytes written will only be allowed to be a power of two, and the write address must be aligned to the data size.

A TileLink manager can process multiple in-flight acquires. That is, it can accept an acquire message before sending the grant for a preceding acquire. There is no guaranteed ordering for the grants sent back. To distinguish which grant is for which acquire, each acquire is given a unique client transaction ID. This ID is then echoed back in the grant message. Each beat of a multi-beat transaction should have the same transaction ID. No two inflight transactions from the same client can have the same transaction ID, so the client must not reuse an ID until after it has completed the transaction by sending the finish message.

3.4 Accelerator Microarchitecture

The design of the accelerator itself is show in Figure 3. When a RoCC command comes in from the CPU pipeline, it is sent to either the frontend (if it is a Transfer or Resume instruction) or to the control register file. The frontend is responsible for splitting up the request by segments and performing virtual address translation. This way, a single virtually-addressed non-contiguous transfer may be broken up into multiple physically-addressed contiguous transfers. If a segment crosses a page boundary, it will also be broken up into multiple transfers, since it may not be contiguous in physical memory. For page translation, the frontend sends requests to a TLB similar to that used by the L1 caches. If there is a page translation error, the frontend waits for all outstanding backend transfers to finish, stores a non-zero response status and faulting VPN in the CR file, and signals completion to the CPU. The CPU then reads the faulted VPN from the CR file and pages it into memory. Once the memory has been paged in, the CPU sends the Resume instruction to the frontend. Assuming no other transfer request has been made, the internal state of the frontend will be the same as it was before the error occurred. On resume, the TLB is flushed, the previously erroring translation is retried, and the frontend continues from the point before the error.

The backend is composed of one or more trackers, each of which can independently handle a single request from the frontend. Each tracker is composed of a reader unit and a writer unit, which can independently issue TileLink requests. The data fetched by the read unit is transferred to the write unit through a shared buffer. Each tracker is given its own port into the L1-to-L2 crossbar. The reader and writer in each tracker must share access to the tracker's crossbar port through an arbiter.

The reader unit reads data starting from the source address, re-aligns the data if necessary, and then stores the data in the shared buffer. To avoid unnecessary round trips, the reader unit uses GetBlock acquires as much as possible. If the source address is not aligned to a cache block boundary, there will be a few beat-sized reads at the beginning until the first cache block boundary is reached. Similarly, there might be a few beat-sized reads at the end if the end of the source region does not fall on a cache block boundary. All of the data in between will be read a cache block at a time. The read acquires are issued in-order, but the grants may come out-of-order. To ensure proper ordering in the buffer, the read unit reserves space in the buffer when it sends the acquire. The reader unit saves the allocated index in the buffer and associates it with the transaction ID of the acquire. When the reader receives the grant for the acquire, it uses the grant transaction ID to look up the buffer index so that it can correctly place the data in the buffer.

The writer unit takes data from the buffer in FIFO order, collects the data together into

as large a packet as possible (up to a single beat) and writes them out to memory. As with the reader unit, the writer uses block-size acquires (PutBlock) as much as possible. There may be a few beat-sized writes at the beginning and end if the destination addresses are not aligned. Writes at a smaller granularity than a beat are handled by setting the Put bytemask. Because of the additional stipulation that writes must be power-of-two sized with aligned addresses, there may be several sub-beat writes for each beat of data read. However, for transfers spanning across multiple blocks, the majority of writes will still be block-sized. Because source and destination addresses can be at any alignment within a beat, the writer unit may need to collect two read beats together in order to form a single write beat. The leftover data from the second read beat will be used as the initial portion of the next write beat.

Splitting the tracker in this way allows the read and write requests to be pipelined, thus increasing utilization of the available memory bandwidth. Utilization can be further improved by increasing the size of the buffer. Since none of the blocks in a single transfer are dependent on each other, the trackers can have as many concurrent requests as there is space in the buffer. The performance can thus be improved at the cost of increasing the area and power consumption of the design.

Since messages passing through a single TileLink channel cannot be reordered and multiple beats of the same block-sized transaction cannot be interleaved with beats from other transactions, the backend trackers must perform flow control to avoid deadlocks. The reader and writer unit perform flow control using reservations in the buffer. The reader unit will not initiate a transaction unless it can reserve sufficient space in the buffer for the returned data. Similarly, the writer unit will not initiate a transaction until all of the data for the transaction has been reserved in the queue. The writer unit does not need to wait for all of the data to actually be placed in the queue. If the space has been reserved in the buffer, the read request must have already been sent out. Therefore, it is guaranteed that the data will eventually be returned, no matter what requests the writer sends out. Beginning a PutBlock acquire does not block the Get or GetBlock grant from being accepted. Because the start of the source region and the start of the destination region can have arbitrary alignment, the buffer must be, at minimum, twice the size of a single block for the system to run without deadlocks.

3.5 Hwacha Vector Processor

Besides comparing our accelerator to a software implementation using scalar load/store instructions, we also compared the copy accelerator to an implementation using vector load/store instructions. Our platform for these benchmarks was Hwacha, another RoCC-based coprocessor designed in our lab, which performs vectorized integer and floating-point arithmetic. The microarchitecture of Hwacha is shown in Figure 4.

Hwacha is broken up into a scalar unit, vector lanes, and vector runahead unit (VRU). Instructions coming into Hwacha from Rocket over the RoCC interface are collected in the command queue. These instructions are then processed in FIFO order by the scalar unit, which handles scalar computation and holds the address and scalar registers. The scalar registers are used for computation, whereas the address registers are used only for memory addressing. If the scalar unit gets a **vf** instruction from the command queue, it begins

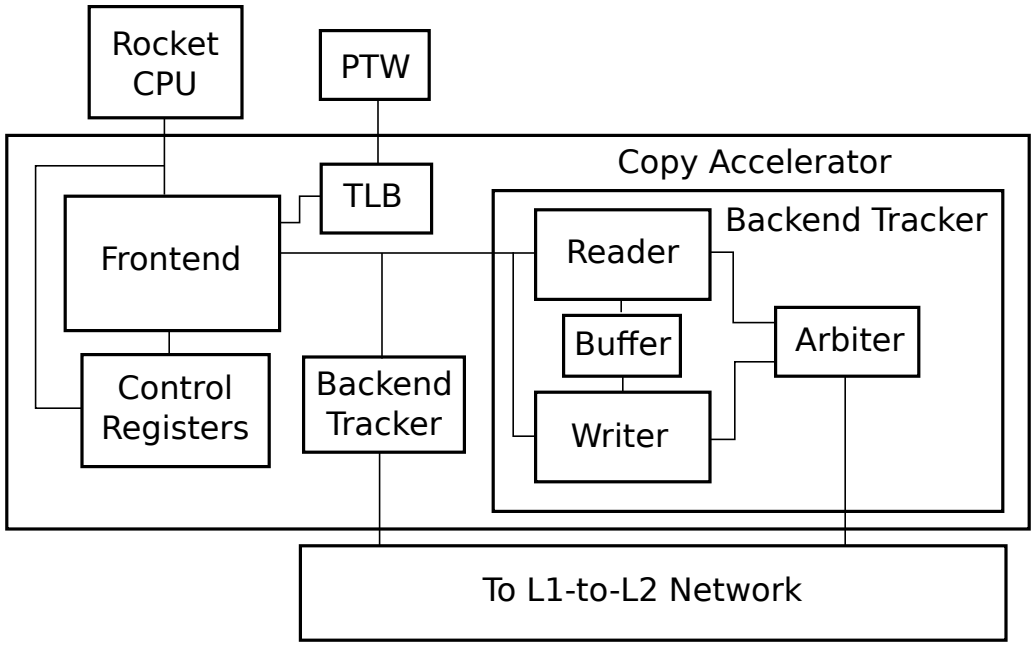

Figure 3: Accelerator Design

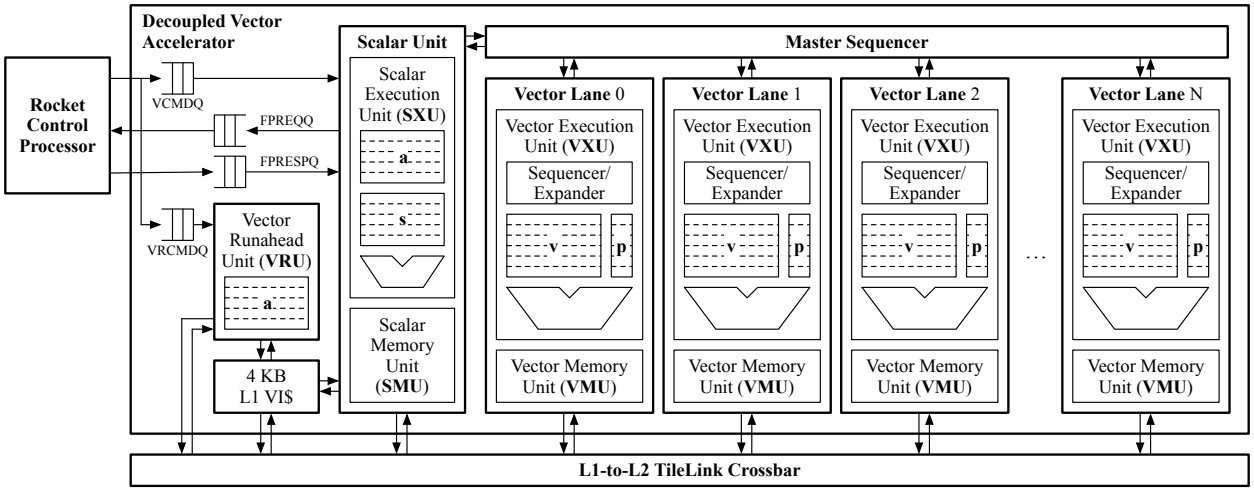

Figure 4: Hwacha Design

fetching instructions from a separate instruction cache. These instructions are then broadcast to the vector lanes, along with the values of the relevant scalar and address registers. The scalar unit will stop fetching from the instruction cache and return to processing the command queue once it executes the **vstop** instruction.

The vector lanes are where the actual vector computation occurs. Each lane contains four 256 x 128-bit SRAM banks, which are connected by a crossbar to several heavily pipelined functional units. The SRAM banks can be logically reconfigured to form up to 256 vector registers. The fewer logical vector registers desired, the larger each one will be. For instance, if only a single logical register is desired, it will have the entire 16 KB of available SRAM to

itself. If two logical registers are requested, each will have 8 KB of SRAM and so on. Each vector lane is also equipped with a vector memory unit (VMU), which has a dedicated port to the L1-to-L2 crossbar.

Hwacha also has a runahead unit which can skip forward in the command queue and instruction memory. It has its own copy of the address registers. Since the address registers can only be changed by command queue instructions, the VRU can be certain that its view of the address registers is the same as the scalar unit's. The VRU skips over most of the instructions and just considers the **vmca** instruction, which sets the address registers; the **vf** instruction and **vstop** instructions for switching between command queue and instruction cache; and the vector load and store instructions. When the VRU sees one of the latter instructions in the instruction cache, it will issue a prefetch acquire to the L2. The L2 will then prefetch the existing data from DRAM so that it can be more quickly supplied when it is later requested by the VMU. Since the VRU "executes" the same instructions that the scalar unit will eventually execute, none of the prefetching will be wasted. It is possible that the VRU may run too far ahead and evict data from the L2 cache before the VMUs have a chance to request them or take up L2 bandwidth that could be used by the VMUs, but the VRU can be intelligently throttled to prevent these events.

Hwacha's utilization is greatly increased by the significant decoupling in the system. The RISC-V processor can submit an instruction to the command queue and continue executing as long as it does not require a response. The scalar unit is decoupled from the vector lanes and can continue executing from the command queue or instruction cache as the vector lanes perform memory operations and computation. When combined with the runahead execution, this decoupling allows the Hwacha coprocessor to easily generate a large volume of memory requests.

4 Benchmarking Methodology

4.1 Design Points

The RocketChip generator allows us to parameterize parts of the design so that we can easily explore many different design points. Hwacha, the copy accelerator, and the outer memory system each have parameters that can be tweaked. For our evaluation, we were interested in two memory system parameters: the number of DRAM channels and the number of acquire transactors per L2 cache bank. The number of DRAM channels determines the total DRAM bandwidth available to the L2. The number of acquire transactors determines the number of concurrent requests that a single L2 cache bank can send to its corresponding DRAM channel. This therefore determines how much DRAM bandwidth the L2 bank can take up, and thus how much bandwidth a single L2 bank can supply to the accelerator.

For Hwacha, the interesting parameter was the number of vector lanes. This determines how many ports into the L1-to-L2 crossbar the accelerator is given, and thus the number of concurrent memory requests that can be sent at a single point in time. This is more interesting for memory systems with multiple DRAM channels. Since the L1-to-L2 interconnect is a full crossbar, two lanes can simultaneously communicate with two separate L2 banks.

For the copy accelerator, the two parameters we investigated were the size of the backend

buffer in each tracker and the number of trackers. The size of the backend buffer determines the number of concurrent requests the accelerator backend can make on a single port, and thus the memory bandwidth that the accelerator can take up. The second parameter is the number of backend trackers, which controls how many L1-to-L2 ports the accelerator is given. As with Hwacha's lanes, having multiple trackers is more useful when there are also multiple DRAM channels.

We investigated the performance with buffers of size 2, 8, or 32 blocks (each block is 64 KB). 2, 4, or 8 L2 acquire transactors; 1, 2, or 4 DRAM channels; and 1, 2, or 4 backend trackers.

4.2 Benchmarking environments

We ran our benchmarks in two different environments. The first was a bare-metal environment with no operating system. In this environment, the virtual pages are set up manually before the test is run, meaning that there are no page faults during the transfer. The source and destination regions are statically allocated in the executable. This was a useful benchmarking environment because it is relatively quick to simulate (no kernel boot overhead). We were able to test a large number of design points fairly quickly. It also helped us see the raw memory performance without any page-fault overhead.

The second environment was a userspace executable running on top of the Linux kernel. In this environment, the source and destination regions are dynamically allocated and the pages are not set up ahead of time, which means that there may be page faults. Benchmarking in this environment helped us determine the effect of page faults on total performance. Due to time constraints (Linux boot in simulation takes a very long time), we only collected data for a single design point in this environment.

4.3 memcpy() implementations

To have a baseline to compare our accelerator against, we wrote a hand-optimized memcpy routine in C and compiled it to the 64-bit RISC-V ISA (RV64).

```
#define UNROLL 8

void fast_memcpy(void *dst, void *src, size_t len)
{
        uintptr_t ptr_mask = (uintptr_t) dst | (uintptr_t) src | len;

        if ((ptr_mask & (sizeof(uintptr_t) - 1)) == 0) {
                uintptr_t *d = dst, *s = src;
                size_t n = len / sizeof(uintptr_t);
                size_t lenu = ((n / UNROLL) * UNROLL) * sizeof(uintptr_t);

                while (d < (uintptr_t *)(dst + lenu)) {
                        //for (int i = 0; i < UNROLL; i++)
                        //      d[i] = s[i];
                        d[0] = s[0];
                        d[1] = s[1];
                        d[2] = s[2];
                        d[3] = s[3];
                        d[4] = s[4];
                        d[5] = s[5];
                        d[6] = s[6];
                        d[7] = s[7];
                        d += UNROLL;
                        s += UNROLL;
                }

                while (d < (uintptr_t *)(dst + len))
                        *(d++) = *(s++);
        } else {
                char *d = dst, *s = src;

                while (d < (char *)(dst + len))
                        *(d++) = *(s++);
        }
}
```

This implementation checks to see if the source and destination are both aligned to 64 bits. If so, it performs an optimized copy of 64 bits at a time, with the loop manually unrolled by 8. We chose this loop unrolling factor because it provided the best performance of the unrolling factors we checked. This unrolling also happens to match the size of a single cache block.

We also wrote an implementation that performs the memory copy in Hwacha by loading into a vector register and then storing out of the vector register to the destination. The vector implementation falls back to a scalar memcpy() if the source, destination, and length are not aligned. Otherwise, the vector memcpy() runs the following stripmining loop implemented in

RISC-V and Hwacha assembly. The arguments to the function are the destination address, the source address, and the number of 64-bit words to be transferred.

```
# a0 -> dst
# a1 -> src
# a2 -> n
vec_copy_d_asm:
  li t0, VCFG(1, 0, 0, 1)
  vsetcfg t0
stripmine:
  vsetvl t0, a2
  vmca va0, a0
  vmca va1, a1
  la t1, vcopy_d
  vf 0(t1)
  // update # of remaining words
  sub a2, a2, t0
  // x8 for # of bytes
  slli t0, t0, 3
  // update addresses
  add a0, a0, t0
  add a1, a1, t0
  // check if we're done
  bnez a2, stripmine
  fence
  ret

.align 3
vcopy_d:
  vld vv0, va1
  vsd vv0, va0
  vstop
```

The instructions in the `vec_copy_d_asm` function are RISC-V and Hwacha scalar instructions. The **vsetcfg** instruction configures the vector registers so that there is only a single vector register using all of the available SRAM resources. The **vsetvl** instruction attempts to set the vector length, which controls how much work will be done by the vector instructions. The application attempts to set the vector length to the number of remaining words to be transferred. The **vsetvl** instruction returns the actual vector length. If the desired vector length is larger than the maximum vector length, the actual vector length will just be the maximum vector length. The two **vmca** instructions copy the current source and destination pointers to the Hwacha address registers. The **vf** instruction switches control to a "vector fetch block", a set of instructions from Hwacha's separate instruction cache. In this case, the vector fetch block is the `vcopy_d` function, which simply performs a vector load from the source address into the vector register and then a vector store to the destination.

It then executes the `vstop` instruction to complete the vector fetch block and return control flow to the command queue. After the scalar core finishes the vector fetch block, it subtracts the actual vector length from the count of remaining words and then increments the source and destination addresses by the number of bytes that have just been transferred. If there are still words remaining to be transferred, it branches back to the top of the stripmining loop. Otherwise, it executes a fence to make sure the vector unit has finished all of its work and then returns to the caller.

For testing the copy accelerator, we used an implementation which is basically the same as the `copy_matrix` implementation shown earlier. The only difference is that there is only a single segment.

```
void accel_memcpy(void *dst, void *src, size_t len)
{
        mlock(dst, len);
        munlock(dst, len);

        write_cr(SEGMENT_SIZE, len);
        write_cr(NSEGMENTS, 1);
        transfer(dst, src);
        wait_for_completion();
}
```

As an additional optimization, we force in the destination pages so that they are less likely to cause exceptions while the accelerator is running. This avoids the overhead of returning control from accelerator to processor and multiple mlock/unlock calls. We do not need to do this for the source because the source region should already be resident in memory (since it was populated with data).

In the bare-metal environment, it is not possible for the DMA transfer to page fault and the mlock()/munlock() system calls are not available, so we replace the wait_for_completion() call with a simple fence instruction.

For the Linux environment, we also ran a benchmark that simply performed the initial mlock/munlock without performing the transfer. This helps us determine how much of the time spent in the accelerated implementation is due to page-faulting and how much due to the actual transfer.

4.4 Cycle Time and Area

To measure the cycle time and chip area taken up by the copy accelerator, we ran our RTL through synthesis using Synopsys 32nm educational process. The design we ran through synthesis consisted of a single CPU tile including the copy accelerator, integer/floating point pipeline, page table walker, L1 data cache, and L1 instruction cache. We used a copy accelerator configuration with a single tracker and 16-block buffer.

5 Evaluation

5.1 Bare-metal microbenchmark results

In our bare-metal microbenchmarks of the various memcpy() implementations, we measured the number of cycles taken to complete a transfer as we increased the number of bytes transferred. We started with a size of 1 KB and kept doubling the size up to 256 KB. A plot of this performance for a single design point is shown in Figure 5. This compares the baseline implementation, Hwacha with a single vector lane, and a copy accelerator with a 2-block buffer and a single backend tracker on a system with two L2 trackers per bank and a single DRAM channel. The copy accelerator is about four times faster than the baseline implementation and comparable to the Hwacha implementation.

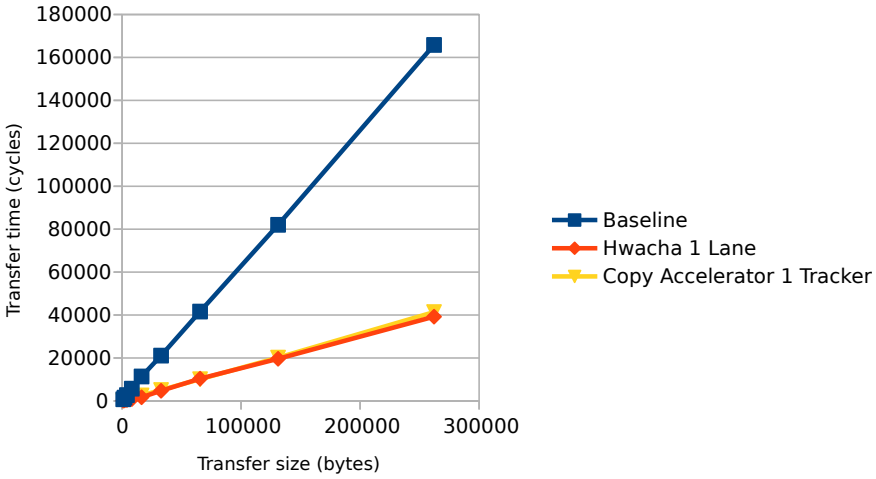

Figure 5: Baremetal memcpy() performance

To make comparison between design points easier, we distilled the benchmark results for each design point to a single number: the average number of bytes copied per cycle. To calculate this, we simply performed a linear regression on the performance curve and took the inverse of its slope.

The bytes per cycle numbers for the baseline implementation are shown in Table 4. The bytes per cycle don't change even as the memory system resources are increased. This suggests that the software implementation cannot fully utilize the memory bandwidth available.

	2 transactors	4 transactors	8 transactors
1 channel	1.6	1.6	1.6
2 channels	1.6	1.6	1.6
4 channels	1.6	1.6	1.6

Table 4: Software baseline results (bytes/cycle)

The Hwacha results are shown in Table 5. As expected, increasing the memory system resources generally does lead to improved performance. However, there are some diminishing

returns. Doubling the number of acquire transactors per bank from four to eight when there are four DRAM channels does not confer any improvement. This may be because Hwacha cannot keep up with the increased bandwidth.

One surprising result is that increasing the number of vector lanes sometimes leads to a decrease in performance. This only occurs in the configurations with fewer memory system resources. A likely explanation is that the L2 cache banks do not have sufficient resources to satisfy the increased volume of requests, but increasing the number of crossbar ports creates extra contention that may degrade performance for each individual lane. It seems that the memory system must have a total of at least eight transactors for additional Hwacha lanes to confer any performance benefit.

	1 Lane	2 Lanes	4 Lanes
1 channel, 2 transactors	6.6	4.1	4.3
1 channel, 4 transactors	12.8	10.3	9.4
1 channel, 8 transactors	14.8	16.4	17.1
2 channels, 2 transactors	11.1	7.6	6.7
2 channels, 4 transactors	15.0	16.5	15.2
2 channels, 8 transactors	15.0	18.2	21.4
4 channels, 2 transactors	13.6	12.5	12.3
4 channels, 4 transactors	15.0	18.2	21.3
4 channels, 8 transactors	15.1	18.2	21.4

Table 5: Hwacha results (bytes/cycle)

Performance results for copy accelerators with 1, 2, and 4 trackers are shown in 6, 7, and 8, respectively. As we saw with Hwacha, increasing the number of backend trackers does not necessarily improve the performance and sometimes degrades it if there aren't sufficient L2 resources. A similar effect is seen for increasing the size of the buffer. Increasing the buffer from two blocks to eight blocks only improves the performance once there are at least a total of eight transactors in the system. Increasing the size from eight to thirty two does not seem to confer much improvement in any configuration. Also similar to Hwacha, we see diminishing returns to increasing the amount of L2 resources. Although the performance gains plateau more quickly if there are smaller buffers or fewer accelerator trackers. Overall, the copy accelerator's performance is quite comparable to Hwacha. It appears that the copy accelerator generally does better when there are more L2 transactors or DRAM channels, while Hwacha does better when the L2 and DRAM resources are more constrained.

5.2 Linux microbenchmark

For the Linux microbenchmark, we picked the configuration with the best performance in the bare-metal microbenchmarks (8 L2 transactors per bank, 4 DRAM channels, 4 backend trackers, 8 block buffer) and compared the software baseline and copy-accelerated implementations. The results of this test are shown in Figure 6. The accelerated version starts out slower than the software version. However, the transfer time grows more slowly for the accelerated implementation, allowing it to catch up to the baseline implementation. Once

	2 blocks	8 blocks	32 blocks
1 channel, 2 transactors	6.4	6.4	6.3
1 channel, 4 transactors	8.9	8.8	9.0
1 channel, 8 transactors	8.7	10.3	10.3
2 channels, 2 transactors	9.5	9.5	10.3
2 channels, 4 transactors	11.6	17.9	17.6
2 channels, 8 transactors	11.6	18.6	18.5
4 channels, 2 transactors	12.3	15.7	15.7
4 channels, 4 transactors	12.3	19.4	19.4
4 channels, 8 transactors	12.3	19.4	19.4

Table 6: Single tracker copy accelerator results (bytes/cycle)

	2 blocks	8 blocks	32 blocks
1 channel, 2 transactors	6.3	6.2	6.1
1 channel, 4 transactors	9.1	9.0	9.1
1 channel, 8 transactors	10.3	10.3	10.3
2 channels, 2 transactors	10.3	10.1	9.4
2 channels, 4 transactors	16.0	16.1	16.1
2 channels, 8 transactors	16.4	18.6	18.7
4 channels, 2 transactors	15.1	12.4	13.6
4 channels, 4 transactors	17.7	18.6	17.9
4 channels, 8 transactors	17.7	20.4	20.3

Table 7: Two tracker copy accelerator results (bytes/cycle)

	2 blocks	8 blocks	32 blocks
1 channel, 2 transactors	6.9	7.2	7.2
1 channel, 4 transactors	9.8	9.6	9.4
1 channel, 8 transactors	10.6	10.6	10.7
2 channels, 2 transactors	9.5	9.4	9.5
2 channels, 4 transactors	15.1	12.9	11.4
2 channels, 8 transactors	19.2	18.1	16.6
4 channels, 2 transactors	13.5	10.6	10.4
4 channels, 4 transactors	19.7	16.6	16.6
4 channels, 8 transactors	19.9	20.5	19.7

Table 8: Four tracker copy accelerator results (bytes/cycle)

the transfer size reaches 64 KB, the accelerated implementation is faster than the baseline implementation. As expected, most of the transfer time for the accelerated implementation is spent in the initial call to mlock() and munlock(). This is likely due to extra book-keeping done in these system calls. For instance, mlock() causes the OS page cache to be drained, and munlock() requires a second walking of the page table in order to unpin the page. There may be a faster alternative for forcing a page into memory that could lead to a performance

benefit for the copy accelerator at smaller transfer sizes.

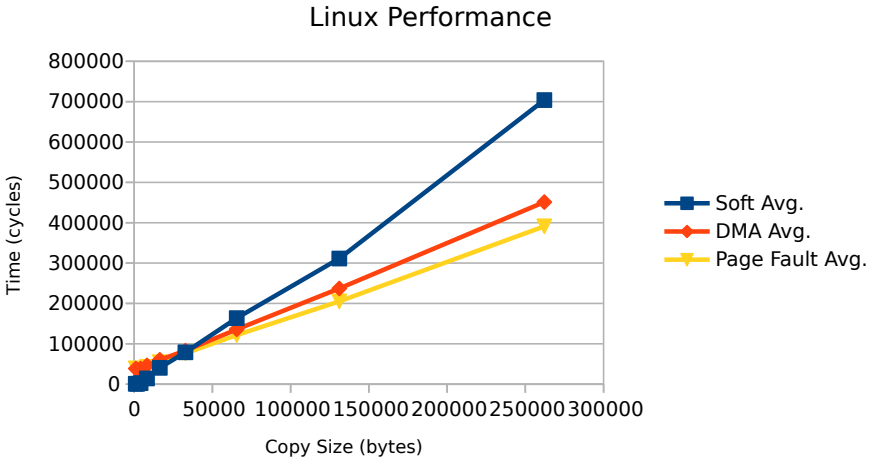

Figure 6: Linux microbenchmark results

5.3 Cycle Time and Area

Our synthesis run showed that a Rocket CPU with copy accelerator could meet a clock rate of 50 MHz. The copy accelerator was not on the critical path. Previous tapeouts of the Rocket CPU on more modern processes were able to achieve a 1 GHz clock rate, so we are confident that the copy accelerator would be able to achieve a similarly high clock rate.

As for area, the copy accelerator takes up approximately 4% of the total area of the tile. This is slightly larger than the amount of space taken up by just the integer/floating point pipeline. This is what we expected, since the size of the backend buffer is the same as the combined size of the integer and floating point register files. The majority of the space in the tile is taken up by the L1 data and instruction caches. The copy accelerator is much smaller than Hwacha, which we have previously found to take up approximately 50% of the tile area.

6 Discussion and Future Work

The results of our evaluation show that the memory copy accelerator confers a significant performance benefit over the basic Rocket core. The memcpy() accelerator can match the performance of the Hwacha vector co-processor while using an order of magnitude less area. This makes it ideal for systems which do significant data processing (such as warehouse-scale computers) but do not require the floating-point or integer computation features of Hwacha.

Our memory copy accelerator is also much easier to program than traditional DMA engines. Since the accelerator can take virtual addresses and translate them without intervention from the CPU, there is no need to specially allocate memory for the copy operation. This also allows the accelerator to be programmed almost entirely from userspace. The application using the accelerator only needs to invoke the operating system kernel if memory

must be paged in. We did not need to write any additional kernel code to run the accelerator in Linux.

There is still one major limitation in the current design. Since the CPU can only synchronize with the accelerator by waiting for the entire transfer to finish, it is difficult to write software that overlaps data movement and computation. An application could achieve some overlap through double-buffering. For instance, the application could start a transfer for one set of data while concurrently performing computation on the set of data that was previously transferred. However, to make this efficient, the application programer would have to manually divide the work into sets so that the computation and data movement complete at approximately the same time. This may not be possible if the copying and computation progress at different rates.

Another way of achieving parallelism between accelerator and CPU would be through multitasking. When the application starts an accelerator transfer and waits for it to finish, it would be swapped out with a different application which then continues computing. Once the transfer completes, it would send an interrupt, causing the original requesting thread to be woken and rescheduled. The current accelerator design would not be able to support this sort of multitasking, since its TLB shares the same PTW as the CPU. If the page table base register gets swapped out in the context switch, the accelerator would no longer be able to correctly translate the virtual addresses for the ongoing transfer. One solution to this problem would be to give the accelerator a separate page table walker. The PTBR for this separate PTW would be sent to the accelerator bundled with the RoCC instruction. This way, the CPU could context switch to a different process while the accelerator continues processing the request. One potential drawback of this design is that it would no longer be possible to program the accelerator entirely from kernel space. The application would have to call into the kernel after every transfer request. The extra overhead from sleeping and waking the process may adversely affect the latency of short transfers.

For further evaluation of the accelerator design, we would like to measure how using accelerated copies in place of the standard software memcpy() would affect the overall performance of an existing application. Distributed computing frameworks like Hadoop and Spark are good candidates for such a benchmark, since they tend to move large blocks of data around (either within memory, to the disk, or over the network). However, this would require more engineering effort to add networking devices to the RocketChip generator and port the JVM to the RISC-V ISA.

We would also like to investigate using the copy accelerator in more heterogeneous memory systems. It would be interesting to see how the accelerator could improve systems with non-uniform memory access. An accelerated memcpy() could be used to speed up computation by moving data to a memory location physically closer to the CPU. The copy accelerator could also be used to move data between memory and peripheral devices like a disk or network controller.

The copy accelerator is slated for inclusion in our upcoming tapeouts. This will give us the opportunity to validate its design in working silicon. One of these tapeouts will include radio antennae and DSP blocks. We intend to use the copy accelerator as a general-purpose DMA engine for moving data between memory and the DSP buffers. This will allow us to investigate the how effective the copy accelerator is at improving IO performance.

References

[1] Krste Asanovic, 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. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[2] Henry Cook. *Productive Design of Extensible On-Chip Memory Hierarchies*. PhD thesis, EECS Department, University of California, Berkeley, May 2016.

[3] F. Duarte and S. Wong. Cache-based memory copy hardware accelerator for multicore systems. *IEEE Transactions on Computers*, 59(11):1494–1507, Nov 2010.

[4] Svilen Kanev, Juan Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *ISCA '15 Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 158–169, 2014.

[5] W. Su, L. Wang, M. Su, and S. Liu. A processor-dma-based memory copy hardware accelerator. In *2011 IEEE Sixth International Conference on Networking, Architecture, and Storage*, pages 225–229, July 2011.

[6] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down cache: a virtual memory management technique for zero-copy communication. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 308–314, Mar 1998.

[7] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. Designing efficient asynchronous memory operations using hardware copy engine: A case study with i/oat. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007.

[8] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016.