Managing On-Chip Memory Hierarchies

Sagar Karandikar, Albert Magyar, and Howard Mao
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
{sagark, magyar, zhemao}@eecs.berkeley.edu

***Abstract*—A variety of optimizations can be used to improve the performance of memory systems without requiring significant modifications to the memory hierarchy. We implement three such optimizations both in isolation and combination to explore the performance characteristics of each for different workloads. Our optimizations consist of a vector runahead unit that performs prefetches in a decoupled vector processor, a DMA engine that can perform software-controlled prefetches, and a software-managed local store. Furthermore, we compare pseudo-LRU and random replacement policies with our other optimizations. We implement our designs in RTL, provide area estimates, and perform cycle-accurate simulations using a simulated memory hierarchy validated against a Samsung Exynos SoC.**

I. Introduction

In this work, we analyze the impact of several on-chip memory system features on streaming workloads. Standard memory hierarchies handle these workloads poorly due to long cache refill latencies from compulsory misses. In the past, there have been many attempts to either increase the performance of caches or reduce the overhead of hardware management. Traditional hardware prefetchers [**?**] operate by speculatively requesting data from higher levels of the memory hierarchy when it is possible to pattern-match the stream of requested addresses. This has the ability to reduce or eliminate the latency contribution of cache misses for some workloads. However, such a prefetching scheme may end up polluting the cache with data that will never be used. In our work, we implement an alternative form of prefetching, a vector runahead unit (VRU) for the Hwacha decoupled vector processor [**?**]. The VRU looks ahead in the instruction stream for vector loads and stores. It can then send prefetches for these memory locations before the vector processor executes the instructions. Because many common workloads do not exercise predication or consensual branches on Hwacha, these prefetches are non-speculative in most cases.

In addition to the VRU, we implemented virtual local store (VLS), a feature which allows a portion of the L2 cache to be partitioned off for use as a software-managed local store. Data stored in VLS will not be subject to eviction by the cache's replacement policy and thus is guaranteed to stay in the L2 cache until the VLS allocation is released. VLS provides many of the benefits of traditional scratchpad memories without the burden of having to manage data movement between the scratchpad and backing memory.

While VLS takes care of allocating a scratchpad memory, a secondary piece of hardware is necessary to quickly fill the VLS with data instead of relying on the standard word-size load and store instructions. To provide this functionality, we implemented a DMA engine which can perform strided, segmented transfers between two regions of memory as well as a strided, segmented software-controlled prefetch, which simply moves data up the memory hierarchy. Besides its use in concert with VLS, the DMA engine can be used as a software-managed prefetcher to replicate some of the functionality of the VRU.

We evaluated the performance of these features and investigated how they interact by running a set of streaming benchmarks with a special focus on a repeated SAXPY benchmark. Initially, we found that VRU and DMA prefetching both led to an improvement in SAXPY performance. However, a later optimization in the Hwacha processor allowed SAXPY to use the full cache bandwidth without prefetching, so the two prefetching schemes no longer improved performance for that particular benchmark. On benchmarks that are incapable of saturating the memory hierarchy on their own, the VRU and DMA engine continue to provide a performance increase.

VLS improved SAXPY performance across the board. However, much of the benefit came from preventing unnecessary conflict misses caused by the random replacement policy. This led us to implement Pseudo-LRU replacement in our L2 cache. We found that, with Pseudo-LRU, the system can reach a level of performance close to that of VLS.

II. Related Work

The decoupled design of the Hwacha vector processor and the resulting ability to prefetch vector memory operations using a vector runahead unit is motivated by earlier work on refill/access decoupling by Batten *et al.* [**?**].

Our desire to explore the use of VLS and DMA is motivated by the Leverich *et al.* paper [**?**] comparing cache-based memory systems to memory systems using explicitly-managed local stores, which the paper terms "streaming memories". Leverich found that streaming memories confer some advantage for benchmarks with very little data re-use, but deemed that the benefits were not worth the large overhead of having to explicitly manage all data movement between the local store and backing memory. However, we believe that VLS can capture most of the benefits of streaming memories without burdening the programmer with explicit management of the memory hierarchy.

Our implementation of VLS is based on the description of the system given in a tech report by Cook *et al.* [**?**]

TABLE I
BASELINE SYSTEM CONFIGURATION

Parameter		Value
Cache line size		32B
Scalar Processor	Cores	1
	Frequency	1000 MHz
L1I$, L1D$	Size	16kB
	Associativity	4
	Replacement	random
Hwacha	Lanes	1
	Mixed-precision	Yes
L2$	Size	256kB
	Banks	8
	Associativity	8
	Replacement	random
DRAM	2-channel, 933 MHz LPDDR3	

III. BASELINE SYSTEM DESIGN

Our baseline design is a RISC-V-based SoC generated by the Rocket Chip SoC generator. The SoC includes Rocket, a single-issue in-order 64-bit RISC-V core, and Hwacha, a decoupled vector coprocessor. Rocket and Hwacha are connected by the RoCC coprocessor interface, which allows Rocket to send custom instructions to Hwacha for execution. The core is backed by a cache-coherent memory system with a multi-banked L2 cache and multi-channel DRAM interface. The full system architecture is shown in Figure **??**.

The Hwacha coprocessor is a vector processor with a multi-banked vector register file and several pipelined execution units, including an integer ALU and an FMA unit supporting double-, single-, and half-precision computation.

One key difference between the Hwacha processor and other vector processors is that the control flow of the processor is

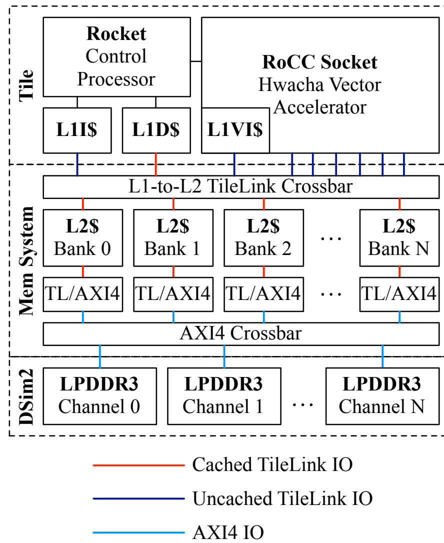


Fig. 1. Baseline System Architecture

[Control Thread]

```
1  saxpy:
2    vsetcfg ...
3    vmcs vs0, f1
4  stripmine:
5    vsetvl t0, a0
6    vmca va0, a2
7    vmca va1, a3
8    vf saxpy_vfb
9    slli t1, t0, 2
10   add a2, a2, t1
11   add a3, a3, t1
12   sub a0, a0, t0
13   bnez a0, stripmine
```

[Vector Thread]

```
1  saxpy_vfb:
2    vlw vv0, va0
3    vlw vv1, va1
4    vfma vv1, vs0, vv0, vv1
5    vsw vv1, va1
6    vstop
```

Fig. 2. **SAXPY kernel mapped to Hwacha architecture** `a0` holds the length `n`, `f1` holds the scalar `A`, `a2` holds a pointer to the float array `X`, and `a3` holds a pointer to the float array `Y`.

decoupled from the vector execution. Control instructions such as setting scalar registers, address registers, and the vector length register come into Hwacha from Rocket via a command queue. The instructions for vector computation come from a separate instruction stream pulled in from memory through a dedicated vector instruction cache.

Figure **??** shows example code for Hwacha, which implements a single-precision $aX + Y$ (SAXPY) operation. The code is split into a control thread (Figure **??**) and vector thread (Figure **??**). The control thread contains both standard RISC-V instructions and Hwacha control instructions. The control instructions first set the scalar and address registers in Hwacha using the `vmcs` and `vmca` instructions, then enters a strip-mining loop which sets the vector length register using the `vsetvl` register and starts the vector thread using the `vf` instruction.

The `vf` instruction tells the vector execution unit to begin executing vector instructions at a particular address. These instructions load vector length sections of X and Y into two separate vector registers. It then performs an FMA, multiplying one register with a scalar and adding the result to the other vector before storing it back to the second vector register. The result is then stored back to memory. Once this computation is completed, the vector thread executes the `vstop` instruction, which causes the vector thread to suspend execution and wait to be restarted by the next `vf` instruction from the control processor.

The `vf` instruction is non-blocking. Once the instruction is sent to the vector execution unit, the control processor can continue executing, thus queuing up more `vf` commands. This makes sure that both neither the control processor nor the vector processor goes idle when there is still work to be done. The control processor need not wait for the vector processor to finish its computation, and the vector processor is not stuck waiting for the control processor to supply more work.

IV. VECTOR RUNAHEAD UNIT

A. VRU Design

The Vector Runahead Unit (VRU), shown in Figure **??**, takes advantage of the decoupled nature of the Hwacha archi-

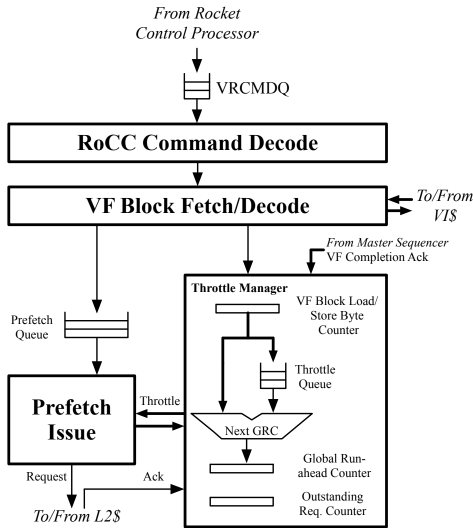


Fig. 3. VRU Implementation

```
1  innermost:
2      # Send addresses of vectors in B to
3      # Hwacha address registers
4      vmca    va4,a5
5      add     a5,t1,a5
6      vmca    va5,a5
7      add     a5,t1,a5
8      vmca    va6,a5
9      add     a5,t1,a5
10     vmca    va7,a5
11     # Load values from A matrix
12     # into Rocket regs
13     ld      s6,0(a4)
14     ...     # 14 more load instructions
15     ...     # here for vs2 ... vs15
16     ld      s5,24(t0)
17     # Copy values from A matrix to
18     # Hwacha scalar regs from Rocket
19     vmcs    vs1,s6
20     ...     # 14 more vmcs instructions
21     ...     # here for vs2 ... vs15
22     vmcs    vs16,s5
23     # Execute vf block
24     vf      0(t3)
25     ...
```

Fig. 4. Control Thread for **DGEMM kernel mapped to Hwacha architecture** (Innermost Loop Body)

tecture to hide memory latency and maximize functional unit utilization. Unlike out-of-order machines with SIMD that rely on the reorder buffer for decoupling and GPUs which rely on multithreading, the Hwacha design is particularly amenable to prefetching without relying on large amounts of state. In Section **??**, we discuss future improvements to enhance this decoupling further.

The VRU snoops on the command queue between Hwacha and the Rocket control processor. It receives the current vector length from `vsetvl` commands as well as addressing information from `vmca` commands, which it stores in an internal copy of the vector address register file. Upon receiving a `vf` command, the VRU fetches instructions from Hwacha's L1 instruction cache and decodes unit-strided load and store instructions. Using the previously collected address information along with the vector length, the VRU issues prefetching commands directly to the L2, in anticipation of loads and stores issued by the vector unit. Unlike in other machines, these prefetches are in most cases non-speculative. Since the address registers and vector length cannot be changed by the worker thread, the VRU will be certain what data is being fetched at each vector load and store instruction.

Efficiently using L2 tracking resources and managing the runahead distance are critical to balancing latency-hiding with allowing the rest of Hwacha to make forward-progress at a reasonable pace. We limit the VRU to using at most one-third of the outstanding access trackers, since in the unit strided case, the VRU's prefetch blocks are twice as large as the execution unit's loads and stores.

In managing the runahead-distance of the VRU, the controller must avoid two extremes. A VRU that runs too close to real-time execution risks invoking a performance penalty. This penalty arises not only from the obvious inability to hide latency, but also because the VRU wastes L2 tracking resources and creates a hotspot around one bank of the L2 cache. A VRU that runs too far ahead of real-time execution has the potential to remove items from the L2 that are in-use or that have been prefetched but not yet used.

To prevent the VRU from running too close to the execution units, we ignore a small number of vector fetch blocks at startup. We observe that sacrificing the prefetch of the loads and stores from one or two initial vector fetch blocks greatly increases the ability of the VRU to runahead in the steady state. To prevent the VRU from running too far ahead of the execution units, one can implement a throttling scheme that counts the total number of bytes of loads and stores that the VRU has decoded but that have not yet been encountered by the execution units. In a vector processor like Hwacha, this scheme is hindered by conditional execution of loads and stores in vector fetch blocks using predication. Our scheme ensures that the counts in the VRU's throttle mechanism are synchronized at the end of each vector fetch block, regardless of the presence of unexecuted loads and stores due to predication and branches. In our scheme, the VRU maintains a queue containing individual load/store byte counters for each vector fetch block that the VRU has seen, but that has not been acknowledged by the execution units. A global counter is also incremented by this per-block count of bytes whenever the VRU finishes decoding a vector fetch block. When the execution units complete the execution of a vector fetch block, an acknowledgement is sent to the VRU, which pops an entry off of the load/store byte count queue and decrements the global load/store byte counter by the appropriate amount. This global counter is then used to throttle the runahead distance of the VRU.

B. Hwacha CMDQ Compression

One significant result of our evaluation of the VRU is the realization that the command queue that decouples Rocket

and Hwacha is a limiting factor for runahead distance in many benchmarks. Consider the code in Figure **??**, which constitutes the body of the inner loop of an optimized `dgemm` benchmark for Hwacha, that achieves 93% of peak FLOPs with the aid of the prefetcher. This benchmark performs a blocked matrix multiply of the form $C = A * B$. In an ideal case, this innermost loop should contain only a `vf` command, which would result in a command queue containing only `vf` commands in the steady state, allowing the prefetcher to runahead significantly. However, as Figure **??** shows, the programmer naturally needs to supply new addresses and scalar values for each vector fetch block that is executed. Accounting only for the extra commands in the inner loop itself, for each `vf` block executed, there are 16 scalars and 4 vector addresses copied to Hwacha. Because this queue contains 64-bit addresses per-entry, its size must be relatively small (32 entries in the current implementation). Thus, the queue length is clearly a limiting factor of the potential amount of runahead. In general, there are three "setup" commands that reduce the amount of potential runahead: `vsetvl`, `vmcs`, and `vmca`. In the following sections, we suggest methods to reduce the number of these commands issued.

1) Compressing `vsetvls`: As a preliminary step, we implemented an optimization to reduce the number of `vsetvl` commands in the queue. A common pattern of code with machines like Hwacha is the use of a set-vector-length instruction in each iteration of the stripmining loop to indicate to the vector unit the amount of data that is left to process. However, for most iterations of the loop, the vector length used is simply the max vector length supported by the vector machine. Therefore, in the queue processing stage, we track the current vector-length that is in-use. If we detect a `vsetvl` command that would leave the vector-length in-use unchanged, we do not dispatch it to the Hwacha or VRU queues. As expected however, this only reduces one setup command per vector fetch block and thus our preliminary evaluation showed a negligible improvement across a set of `*axpy` and `*gemm` benchmarks.

2) Compressing `vmcas`: Another potential optimization is the reduction of `vmca` commands by employing an auto-increment mechanism. This would take advantage of the fact that in many cases, there is a pattern to successive addresses sent to particular address registers in Hwacha. For example, in the case of `dgemm`, `vmca` commands in the inner loop supply addresses to an address register that are $4 * vectorlength$ apart on each successive iteration. With the addition of special-case `vmcs` instructions that indicate the need to auto-increment, Hwacha could automatically perform this operation on successive vector fetch blocks after receiving the base address through a single set of `vmca` commands during the first iteration.

3) Compressing `vmcses`: One final optimization is the reduction of `vmcs` commands in the queue. A common pattern of utilizing `vmcs`es is shown in the `dgemm` benchmark. This consists of performing a load on the control processor and then issuing a `vmcs` to move the loaded data to Hwacha's scalar registers for use in computation. If we instead shift the responsibility of loading this data to Hwacha, we can use the addressing pattern to reduce the amount of information sent to Hwacha through the queue. In the case of `dgemm`, the ability to perform a segmented-strided vector load and stripe the loaded data across the scalar registers would reduce the number of `vmcs`es to one per `vf` command in the steady state. Using address auto-increment as described in the previous section with this optimization would reduce this to zero `vmcs`es per `vf` command in the steady state.

V. VIRTUAL LOCAL STORES

Previous work [**?**] has shown that many streaming workloads can exhibit better cache performance when features such as hints, pinning, and streaming prefetching are added. However, this adds complexity, along with a potential point of software control through instructions like Prepare For Store. Instead, a more general solution for enhancing the memory access time of workloads that challenge hardware-managed caches is to allow the use of software-managed local stores [**?**]. While local stores typically take the form of additional, explicitly addressed memories on a chip, it is possible to combine them with caches to allow the programmer to flexibly use a memory in either an implicitly or explicitly addressed form. One specific mechanism for flexible on-chip memory hierarchies is Virtual Local Stores (VLS) [**?**], a proposed addition that adds a way-based partitioning scheme to an on-chip cache in order to overlay a software managed scratchpad, in concert with a lightweight translation mechanism to virtually address data that is held in the local store.

A. ISA Extension

The RISC-V VLS extension divides the functionality into two parts: a lightweight mechanism for translating addresses within a hart's VLS range and a mechanism for pinning physical address segments in a cache. Used in concert, these allow for virtually-addressed regions of memory to enjoy both protection and software management of locality.

At any given time, a hart may have exactly one active VLS region. Since VLS space is requested by the user process for buffers where great locality of reference is expected, a separate, lightweight mechanism is provided to translate a virtual address when it falls within the hart's active VLS region. This translation step is base-and-bound, with the extent of the virtual region being established by the hart's `vlsvbase` and `vlssize` CSRs. Addresses falling in that region will be translated using the `vlspbase` register.

$$paddr_{vls} = \texttt{vlspbase} + vaddr_{vls} - \texttt{vlsvbase}$$

On a particular machine supporting the RISC-V VLS extension, it will be possible to support 'pinning' some number of physical address segments. This pinning is not a guarantee that accesses to these addresses will never miss, as it only establishes a contract that standard cache replacement will not evict the data. These pinned segments are a resource managed

by the system; therefore, properties of the machine related to physical segment pinning must be exposed to the supervisor.

The VLS translation mechanism can be completely described using per-thread state, so it is configured using CSRs. In contrast, the physical address pinning modifies the operation of the on-chip memory hierarchy. This configuration can produce performance effects that are visible beyond a particular thread, so it is more naturally controlled through memory-mapped configuration registers. For each cache supporting VLS, a set of registers will be demarcated within the memory map specified by the hardware device tree. The lowest word address will map to `nvlssegs`, a read-only value describing the number of VLS allocation segments. This value will then be followed by the appropriate number of allocation descriptors, each consisting of a `pbase` and `size` field. By writing these registers, the hardware abstraction layer can provide a binary interface for configuring a flexible number of VLS allocations co-resident in one cache.

B. Implementation

In order to perform concrete evaluations, a concrete implementation of VLS is provided as part of the L2 cache of the Rocket Chip generator. As shown in Figure **??**, the VLS is managed through way-based allocations in a VLS manager. An N-way cache has slots for $N-1$ allocations, each with an individual physical base address and zero to $N-1$ allocated ways. The VLS manager is integrated into the replacement strategy of the cache, allowing the VLS allocation state to influence the placement of blocks in the cache. In particular, by exercising control over VLS block placement, the VLS manager can maintain a direct mapping of each VLS block into a particular set and way. This behavior is enabled by the way-based allocation strategy, which factors heavily into the RTL-level implementation of the manager.

In order to set up allocations, software routines will perform MMIO to the VLS allocation segment descriptors. In Rocket Chip, this required the creation of a NASTI slave interface for the L2 metadata. In practice, this is implemented as an interconnect network mirroring traffic to an MMIO slave in the VLS manager of each bank. This configuration state adds minimal overhead to the size of the cache. Furthermore, the VLS translation will be set up using the appropriate CSRs, which are tightly integrated into the TLB of the Rocket pipeline.

Upon receiving a block address from the inner levels of the memory system, the L2 metadata array passes the address to the VLS manager. The VLS manager uses a set of parallel base and size checks to determine if the address falls within a VLS allocation. If it does not, the metadata is accessed just as with a normal cache lookup. If it does, the access will be based on the offset of the block from the base address of its allocation.

By exercising control over VLS block placement, the manager can maintain a direct mapping of each VLS block into a particular set and way; in particular, since allocations must be cache-way-size-aligned, the offset of a VLS block from the base address of its allocation can be used to find the *home way* of the block. By maintaining one key invariant, the VLS system can guarantee that no other way will contain the VLS block and that it will not be evicted by an access to any other block while the allocation holds.

TABLE II
CACTI-BASED ENERGY AND AREA FOR 45NM LOP L2 DATA AND METADATA SRAMS

SRAM	Read dynamic energy (pJ)	Area (mm^2)
32kB 6T, 256b ports	17.1	0.317
2048B 8T, 128b ports	3.08	0.0310
256B 8T, 16b ports	0.344	0.00212

TABLE III
READ ENERGY FOR VLS AND CACHE ACCESSES WITH SPLIT AND UNIFIED METADATA

Operation	Total dynamic E/op (pJ)	Relative E/op
Split metadata cache read	19.9	0.99
Split metadata VLS read	17.5	0.87
Contiguous metadata any read	20.2	1.0

Invariant: under a fixed VLS allocation state, a line from a given (way, set) pair is evicted if and only there is a miss to the block with the appropriate set index in the region of a VLS allocation mapped to that particular way.

C. Partial Metadata Reads

As a product of the VLS system invariants, it is always possible to perfectly predict which way (if any) will hold a given block of VLS data. Therefore, an enhanced metadata array implementing per-way metadata read enables was added to the L2 cache. This provides the ability to save energy by avoiding read operations to ways that need not be checked.

In order to provide a preliminary evaluation before pushing the design through a commercial process, the split and unified metadata arrays were modeled using the CACTI memory model [**?**]. This model provides rough approximations of area and dynamic energy per read operation for arbitrary SRAM parameters. Table **??** shows the output of the CACTI model, while Table **??** shows the relative dynamic read energies of VLS and cache accesses using the outputs of the model. Under the CACTI model, the split metadata arrays show some degree of promise given the 13% reduction in energy in VLS accesses. Although the model is of low fidelity, this indicates that the modification shows sufficient promise to be evaluated using a real, commercial process.

VI. DESIGN OF THE DMA ENGINE

A. The DMA ISA Extension

In order to accelerate data movement between VLS allocations and other parts of memory, we designed a direct memory access (DMA) unit which can copy or prefetch data into the L2 cache at high bandwidth. The DMA unit is exposed to software as a set of custom instructions. The first instruction allows the user to set the values of four shape registers, which control the

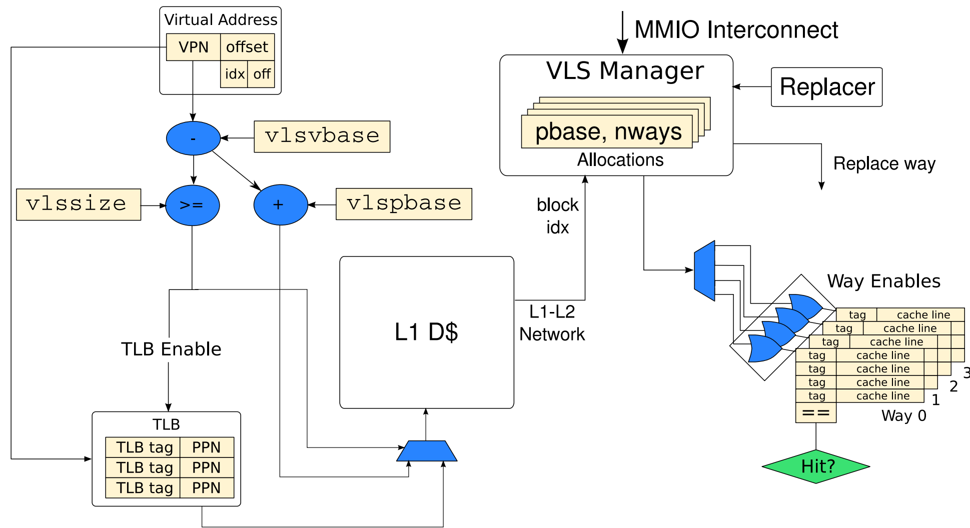


Fig. 5. Graphical depiction of VLS accesses

shape of the data being copied. These shape registers are the number of segments in the transfer, the size of each segment, the stride between source segments, and the stride between destination segments. This strided, segmented copy function is mainly useful for copying sub-sections of matrices.

After setting up the shape registers, the CPU can initiate the actual transfer by executing the "copy" instruction, which takes a source and destination address and performs the data transfer according to the setting of the shape registers.

Besides the copy operation, which moves data from one location to another, the DMA system also provides a software-controlled prefetch operation. This does not move data to a new location, but rather prefetches it from main memory to the L2 cache (if the data is not already resident in cache). The prefetch operations are also strided and segmented.

B. Implementation

The DMA system is composed of two distinct parts. The first is a DMA client, which resides on each CPU core as a co-processor. This client handles virtual address translation and segmentation. It then sends requests for contiguous transfers to a DMA engine located next to the L2 cache. The DMA engine consists of multiple DMA trackers, each of which can handle an independent DMA transaction. For copy operations, the tracker requests one or two cache blocks from memory at a time and stores them in a buffer. It then aligns the data in the buffer and stores it to the L2 cache. For prefetch requests, the DMA tracker simply sends block prefetch requests to the L2. This operation can run at a much higher bandwidth than the copy operation since it does not need to buffer any data itself.

VII. Better Replacement Policies: Pseudo-LRU

After early evaluations were performed, it became obvious that the random replacement policy in the L2 cache was crippling performance in many benchmarks. In particular, many of

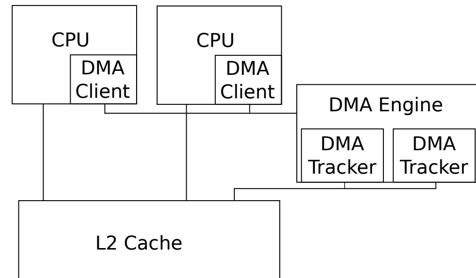


Fig. 6. DMA system design

the advantages of using VLS arise from the ability to defeat sub-optimal replacement policies. While it is quite easy to construct kernels with unit stride memory access patterns that produce poor results under random replacement, comparisons are much more valuable when made against a realistic set of baseline machine parameters. Therefore, a more effective pseudo-least recently used (PLRU) replacement policy became a critical part of the design space for the memory system experiments.

Tree-PLRU is a well-known replacement policy that stores a marshalled representation of a decision tree for each set in a set-associative cache, as shown in Figure **??**. For an N-way cache, this tree has N leaf nodes (each corresponding with a way) and $N-1$ internal nodes. Each internal node contains a state bit indicating which child is less recently used, allowing the evicted way to be found by traversing the tree. When an access hits in the cache, the internal nodes on the path from the hit way to the root are marked to point off the hit path.

In practice, Tree-PLRU is fairly efficient in logic and in state for moderately associative stores. Since only the internal nodes carry state, an N-way cache requires $N-1$ state bits per set, along with a single shared implementation of the update algorithm, which is relatively small for $N \in \{4, 8\}$ after logic minimization. When adding this replacement policy to the L2

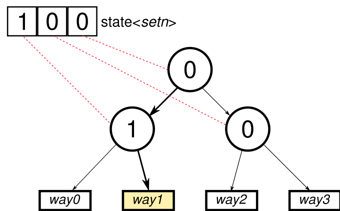


Fig. 7. Tree-PLRU replacement policy

cache from the reference baseline design point in Table ??, 7 bits of replacer state are added to each of the sets in each of the banks. Although these would likely be implemented as flip-flops to allow decoupling of replacer updates from metadata updates, this still represents only a modest increase in utilization relative to the 136 bits of metadata that are already held with each set.

Adding Tree-PLRU to the Rocket Chip generator required the creation of a new class hierarchy of stateful L2 replacers conforming to the pipelining of replacer accesses and updates in the L2 metadata unit. The state resides in a synchronous-read dual port RAM of 128x7b for each bank, and the pipelined replacement policy does not affect the critical path.

VIII. Evaluation Results

A. Simulated Memory System

In order to provide reasonable DRAM performance in simulation, we utilize DRAMSim2 [?] in our simulations. We supplied DRAMSim2 with the timing parameters of a Micron LPDDR3 part matching those of the dual-channel 933MHz LPDDR3 on the Samsung Exynos 5422 SoC, which contains an ARM Mali-T628 MP6 GPU. We use `ccbench` to empirically validate that our simulated memory hierarchy is similar to that of the Exynos 5422 SoC.

The `ccbench` benchmarking suite [?] contains a variety of benchmarks to characterize multi-core systems. We use `ccbench`'s `caches` benchmark, which performs a pointer chase to measure latencies of each level of the memory hierarchy. In unit-stride mode, each pointer in the array of pointers points to the next contiguously placed pointer in memory. In cache-line stride mode, each pointer points to another pointer a cache-line size away in memory to circumvent spatial prefetching. In random-stride mode, each pointer points to a random pointer in the array to avoid both stride and stream prefetching. The size of the array of pointers can be varied to exercise differing levels in the memory hierarchy.

Figure ?? compares the performance of our cycle-accurate simulated memory hierarchy against the Exynos 5422 using the `caches` benchmark in `ccbench` On the simulated RISC-V Rocket core, `ccbench` measures cycles, which we normalize to nanoseconds by assuming the 1 GHz clock of previous silicon implementations of Rocket cores [?]. On the Exynos 5422, `ccbench` measures wall-clock time to produce our results.

This baseline comparison highlights two important features that validate our experiments. Firstly, while the L1 and L2 cache sizes differ between the Rocket core and the Exynos 5422, the L1 and L2 caches have similar latencies in terms of processor clock cycles. With both a 1 GHz Rocket core and a 2 GHz ARM Cortex-A15, the L1 hit latency is approximately 4 cycles and the L2 hit latency is approximately 24 cycles. Secondly, both the simulated LPDDR3 used in our experiments and the LPDDR3 in the Exynos 5422 achieve similar latencies of approximately 110 ns.

The only significant latency difference between the memory system of Rocket Chip and that of the Exynos 5422 is exposed once the array size exceeds the size of the L1 or L2 caches. This discrepancy can be explained by the presence of a streaming prefetcher present in the Cortex-A15, which can automatically prefetch both unit-strided and non-unit-strided loads and stores [?].

TABLE IV
Area Results for Optimizations

	VRU	DMA	Rocket-Chip
Area (mm^2)	0.00792	0.0121	1.62
% of Rocket Chip	0.5	0.7	100.0

B. VLSI Results

The Synopsys ASIC CAD toolflow (Design Compiler, IC Compiler) was used to map the Chisel-generated Verilog to a standard cell library and memory-compiler-generated SRAMs in a widely used 28nm technology, using 8 of 10 metal layers for routing. We obtained area estimates as shown in Table ??. As expected, both the VRU and DMA Engine represent a very small fraction of the area.

C. Evaluating VRU and Replacement Schemes

The use of VRU or the choice of replacement scheme is transparent to software, so we could run the existing Hwacha benchmark suite with these hardware changes to evaluate their effect on performance. Figure ?? shows the number of cycles it took to complete several benchmarks using VRU or not using VRU and using random replacement or PLRU replacement. As expected, Hwacha with VRU enabled consistently does better than without VRU, and PLRU replacement consistently does better than random replacement.

D. Evaluating DMA Prefetching and VLS

Using DMA prefetching or VLS requires software modifications, so we decided to focus on a single benchmark for evaluating the performance effects of these two features. The benchmark we chose was a repeated SAXPY. In this variant of SAXPY, we accumulate multiple X vectors into Y. So if regular SAXPY is the operation.

$$Y = aX + Y$$

Repeated SAXPY is the operation

$$Y = aX_0 + aX_1 + ... + aX_n + Y$$

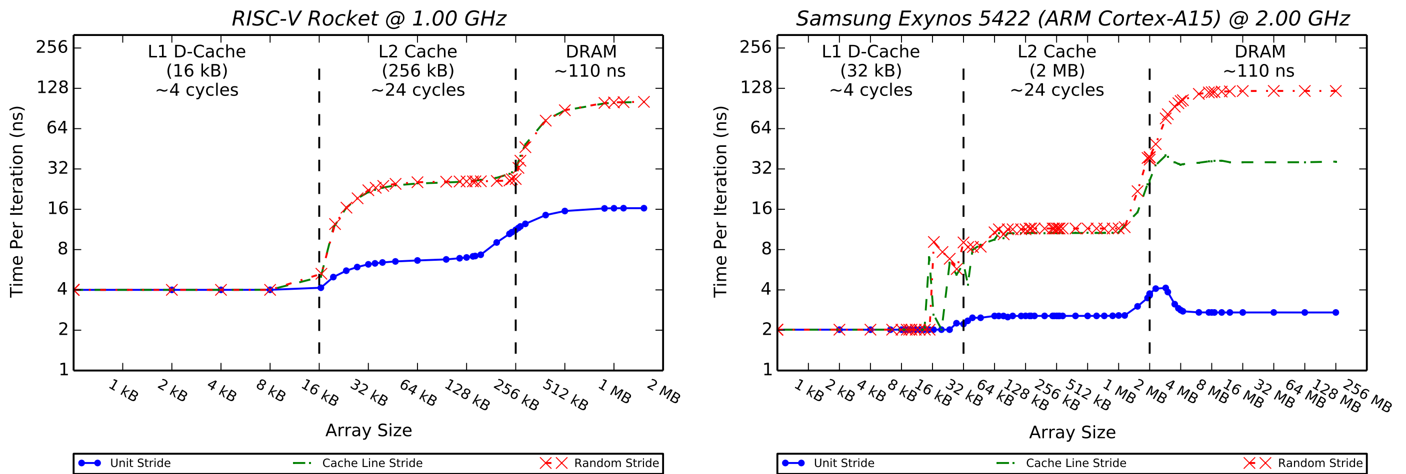


Fig. 8. `ccbench` "`caches`" Memory System Benchmark

For this benchmark, we compared three replacement schemes: VLS with random replacement, normal random replacement, and pseudo-LRU replacement. For the VLS case, we pinned only the output Y in a VLS segment, since it is accessed repeatedly throughout the benchmark, whereas the X vectors are accessed only a single time.

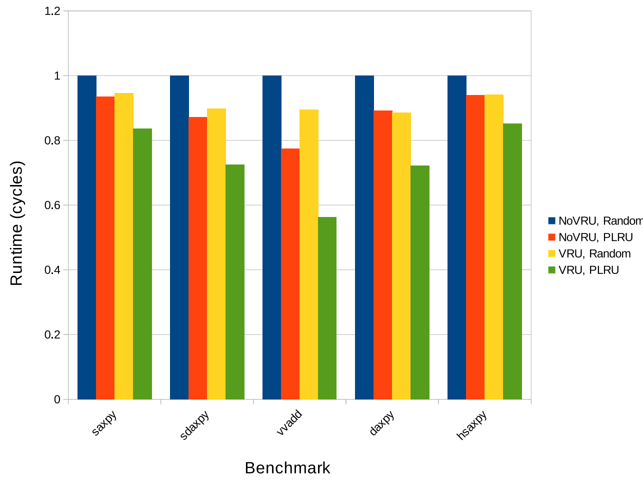
We compared four prefetching schemes: no prefetching, DMA prefetching, VRU prefetching, and DMA and VRU prefetching combined. While the latter scheme may seem redundant, since the VRU and DMA unit are prefetching the same blocks, we include it in this report as the results we received were somewhat unexpected. The results of running repeated SAXPY with the various combinations is shown in Figure **??**.

In all prefetching cases, simple random replacement performs the worst. Blindly replacing ways leads to a lot of unnecessary evictions and conflict misses. VLS performs the best out of the three. Fixing Y in the cache is a big win, since we access that array multiple times, and we do not want the replacement policy to evict those blocks in favor of data from X, since those are only accessed once. As expected Pseudo-LRU replacement works quite well, and is generally approaches the performance of VLS, with the added benefit that no software modifications need be made.

As far as prefetching goes, any form of prefetching improves the performance above the baseline. The VRU performs slightly better than software-controlled DMA prefetching. However, the DMA prefetching code wasn't particularly optimized. It simply prefetches one vector fetch block ahead at the beginning of the strip-mining loop. With some more tuning, the DMA prefetcher might be able to beat the VRU. However, this need for manual performance tuning is the main pitfall of software prefetching. Surprisingly, DMA prefetching combined with the VRU produces better performance than either prefetching scheme by itself. One would think that this shouldn't be the case, since they are prefetching the same data. To figure out why this was the case, we looked at the waveforms of the test runs and the L2 performance counters. From the waveforms, we found that the DMA prefetcher was fetching one block ahead of the VRU. So we hypothesized that, while fetching block N, the VRU sometimes evicts data from block $N+1$ that is being fetched by the DMA prefetcher. In the next phase, the VRU refetches the data that was evicted. This effectively gives two chances to make sure the data is in cache before the processor requests it. This hypothesis was validated when we checked the performance counters and saw that the DMA+VRU version had both more hits and more misses in the L2. Indeed, the performance counters shown in Figure **??** demonstrate that the extra performance comes at the price of significantly increased L2 cache traffic and consequently energy usage.

In addition to repeated SAXPY, we also ran a streaming histogram benchmark, which runs through an array of integers from 0 to 999 and counts the frequency of each integer. The bin counts are updated using an indexed vector atomic add operation. For the VLS tests, the bin counts are pinned with VLS. The results of this benchmark are shown in Figure **??**. As before, both DMA and VRU in isolation do better than the baseline, with VRU being marginally faster than DMA. However, unlike SAXPY, the case with DMA and VRU combined does not perform faster than VRU in isolation. It does slightly worse than VRU alone and slightly better than DMA alone.

These benchmarks come with a big caveat, which is that they were done using a slightly older version of the Hwacha processor. The newest changes increase the performance of the processor so that the regular vector memory unit can saturate the memory bandwidth without using prefetching. As a result, for SAXPY, which has a low computation to memory usage ratio, prefetching no longer leads to any performance gains. However, we think these results are still somewhat interesting, since they give an indication of what performance would be like given a more compute-heavy benchmark.



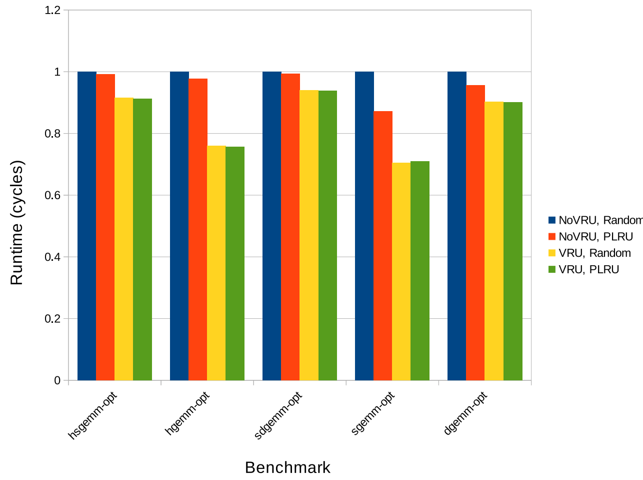


Fig. 9. Old Hwacha results

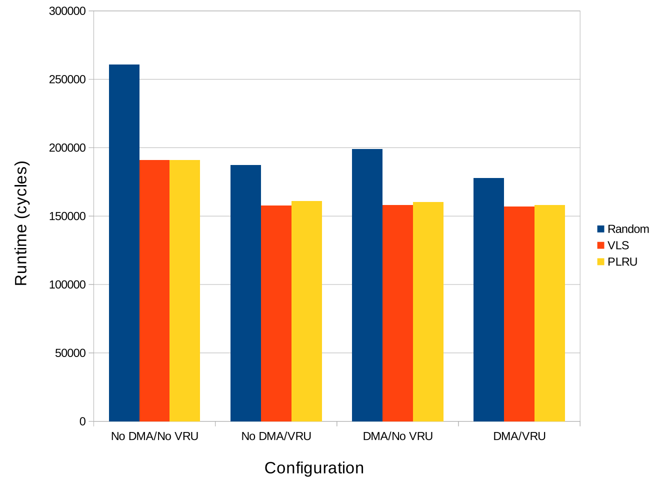


Fig. 10. Repeated SAXPY results

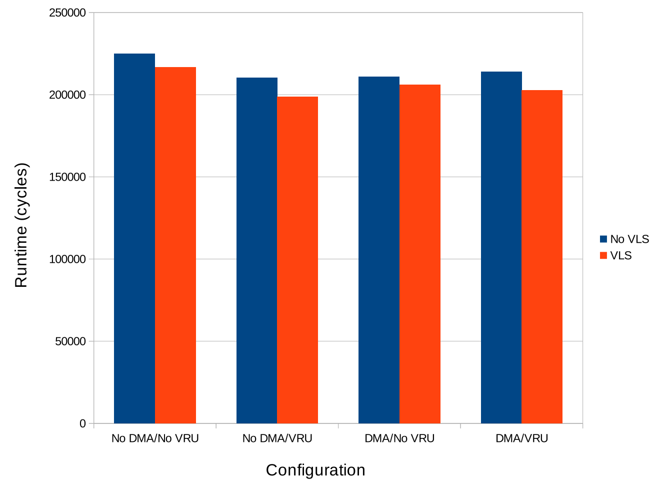


Fig. 11. Streaming histogram results

E. Evaluating VRU and Replacement w/ Improved Hwacha

With the improved Hwacha execution unit, the results look like those given in Figure **??**. For all of the *AXPY benchmarks and vector-vector add, enabling the VRU causes Hwacha to perform worse. As mentioned above, this is because these benchmarks do little computation relative to the amount of data fetched, and there is very little data re-use. As a result, the vector execution unit by itself can fill up the memory bandwidth, so the only effect the prefetch requests have is to eat into bandwidth that would otherwise be used to actually fetch the data.

For the *GEMM benchmarks, there is considerably more computation and data re-use, so the vector execution will not fill up the memory bandwidth. As a result, prefetching from the VRU will still lead to performance gains, since it can prefetch ahead during the periods when the execution unit is not making memory requests itself.

PLRU still performs better than random on all the *AXPY benchmarks. For *GEMM, when VRU is disabled, the PLRU performs about as well or slightly better than random replacement. This is because the matrices we use have dimensions which are multiples of the cache line size. As a result, set conflicts are more frequent that in *AXPY. Since PLRU only ensures that the most recently used way is not evicted, it does not handle frequent set conflicts well. When VRU is enabled, PLRU either does about the same or slightly worse than random. This might be a pathological case for PLRU. Since the most recently accessed way will be the prefetched line, it makes the line that is currently being accessed by the execution unit more likely to be evicted.

IX. Conclusion

The most important thing we learned from this project is that it is important to start from a good baseline. Vector code which does a lot of data movement and minimal computation should be able to saturate the memory bandwidth without prefetching. In this case, prefetching will not be useful. However, for

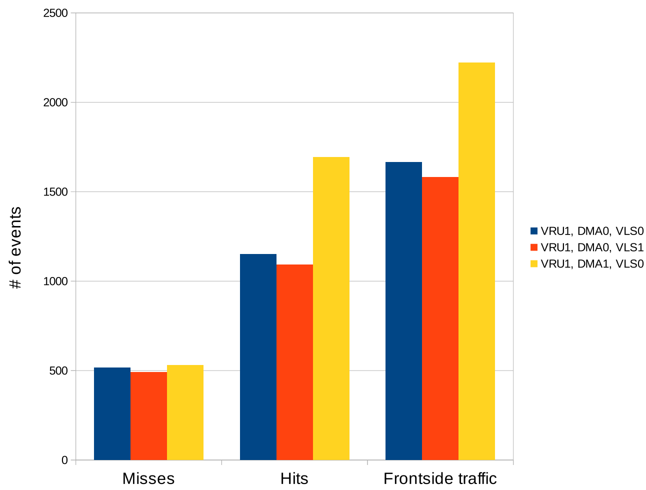


Fig. 12. Performance counter values for cache traffic in repeated SAXPY inner loop

problems which are memory-bound but cannot saturate the memory bandwidth, prefetching is generally useful.

We also found that tuning performance parameters is quite time-consuming. Tuning prefetching is especially difficult, as it is highly dependent on the workload. With software prefetching, the programmer must painstakingly profile how much and how far ahead to prefetch. Tuning hardware prefetching is also time-consuming, except the work must be done by the hardware designer, not the software programmer. In a way, this may be more difficult. For general-purpose hardware, the designer will not be able to predict all of the different workloads that users will run. This suggests the need for a more adaptive hardware prefetcher, which can adjust its own parameters to fit the currently running workload.

Finally, we learned that when investigating features for reducing conflict misses, it is important to ensure that the features are not simply compensating for problems with the replacement policy. A pseudo-LRU replacement policy is not terribly difficult to implement (though somewhat expensive in terms of added hardware) but confers significant benefits over random replacement.

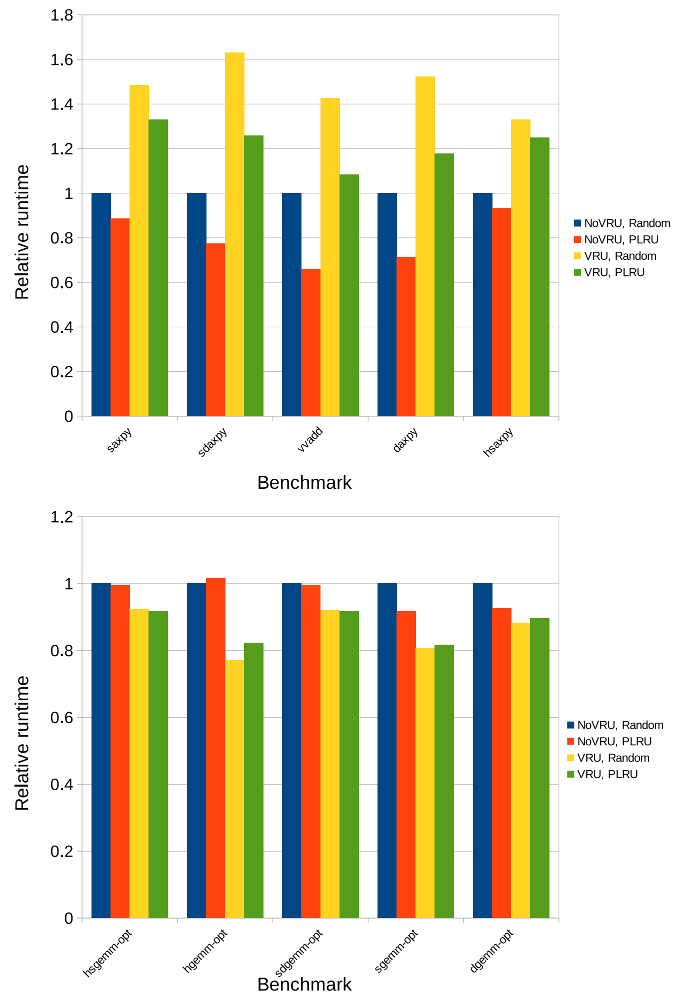


Fig. 13. New Hwacha results