Appears in the Seventh Annual Workshop on the Interaction amongst Virtualization, Operating Systems, and Computer Architecture (WIVOSCA 2013), at the 40th International Symposium on Computer Architecture (ISCA-40)

# A Case for OS-Friendly Hardware Accelerators

Huy Vo, Yunsup Lee, Andrew Waterman, Krste Asanović
University of California, Berkeley
{huytbvo, yunsup, waterman, krste}@eecs.berkeley.edu

## Abstract

Modern SoCs make extensive use of specialized hardware accelerators to meet the demanding energy-efficiency requirements of demanding applications, such as computer graphics and video encoding/decoding. Unfortunately, the state of the art is a sea of heterogeneous fixed-function processing units wired together in an ad-hoc fashion, with dedicated memory spaces and a wide variety of host-accelerator synchronization mechanisms. This cumbersome approach complicates acceleration of a mix of multi-programmed applications running on a conventional operating system, and adds considerable communication overhead that reduces achievable speedups on a wide range of applications. We propose that accelerators should adopt a more standardized OS-friendly interface, to ease integration and improve performance on a wider range of code. Our framework standardizes the host-accelerator communication interface, provides a memory consistency model, and specifies the minimal requirements for virtual memory support. To evaluate the feasibility of our proposal, we conduct a case study in which we modify an existing data-parallel accelerator. When the integrated accelerator and processor system is pushed through TSMC's 45nm process, we observe that the overhead is only 1.8% in area and 2.5% in energy, illustrating that the overhead in building OS-friendly accelerators can be minimal.

## 1. Introduction

Transistor density improvements are continuing, but energy scaling is lagging behind due to threshold-voltage scaling limitations. This has led to the increasing popularity of specialized hardware accelerators [16] for a wide variety of applications since these accelerators are much more energy-efficient than general-purpose processors. Modern SoCs, for example, tend to integrate numerous accelerators [1] in order to better carry out their myriad required tasks. These accelerators usually have little to no interaction with the general-purpose operating system, making it difficult to share the accelerators in a fine-grained way across multiple time-shared applications. To address this problem, we propose a framework for building OS-friendly hardware accelerators.

In Section 2, we present the general framework we propose for attaching OS-friendly hardware accelerators. In Section 3 we conduct a case study in which we augment an existing Hwacha data-parallel accelerator with an OS-friendly interface to allow close integration with a general-purpose CPU. We chose to examine a data-parallel accelerator because it has similar functionality to a GPU. Then in Section 3.3, we push our combined CPU and data-parallel accelerator system through an ASIC toolflow. We report area and energy numbers, showing that our OS-friendly interface to the parallel accelerator introduces very little overhead. We discuss related work in Section 4 before concluding.

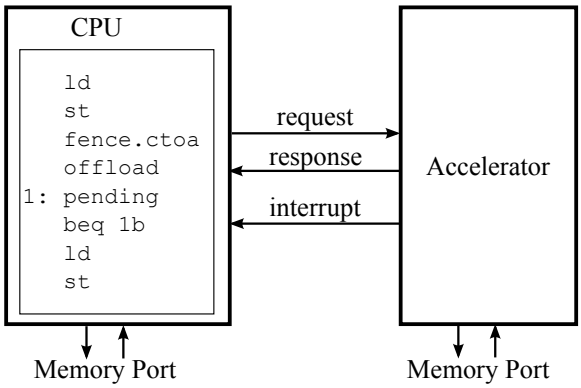



Figure 1: **Accelerator Interface Specification.**

## 2. A Framework for OS-Friendly Hardware Accelerators

In this section we present a framework for building OS-friendly accelerators, suitable for use with multi-programmed applications running on a general-purpose operating system. Our framework specifies a generic connection interface, a memory consistency model, and requirements for virtual-memory support, all of which will together standardize the interactions between the CPU and the accelerator, thus allowing integration between a general-purpose processor and any accelerator regardless of the processor or accelerator's implementation.

**Request/Response Interface.** Figure 1 shows a generic interface for integrating a control processor with an accelerator. The majority of communication between the CPU and the accelerator will use the request-response interface. The CPU pushes work to the accelerator using the request in-

terface. If the accelerator is unable to accept work, then the CPU must block until the accelerator is ready to accept work from the CPU. The accelerator uses the response interface to return any results produced from the work received. If the CPU expects a response value from the accelerator, then the CPU must block when it executes an instruction that reads the value until there is a valid response from the accelerator on the response interface.

Inserting a command queue into the accelerator can reduce the amount of time the processor spends stalling on the request interface. Whenever the CPU pushes work to the accelerator, the work is enqueued onto the command queue if there is room. Whenever ready, the accelerator dequeues work off the queue. The interface will look the same (the CPU still sees a request-response interface), but the processor is able to run ahead and push work to the accelerator until the command queue fills up. The accelerator must have logic to peek at the CPU request port to capture high priority CPU requests that require immediate responses, such as interrupt cause requests and memory fence operations.

Our Hwacha data-parallel accelerator uses a command queue to allow the strip-mining loop on the CPU to run ahead and enqueue work from multiple iterations. This is particularly useful when pushing vector loads since Hwacha has a vector prefetcher that looks at the CPU request interface for incoming vector memory commands that can be prefetched [4].

**Memory Consistency Model.** Accelerators that share memory space with the CPU require additional memory consistency semantics. To this end, the memory consistency model specifies the use of a fence instructions to order the memory operation of the control processor and accelerator. The fence.ctoa instruction ensures that the accelerator sees all the memory operations committed by the CPU. The example code sequence in Figure 1 shows a possible use case of the fence.ctoa instruction. In this example, the CPU wants to push some work to the accelerator and it wants the accelerator to see the results that it had previously committed to memory. The CPU thus inserts a fence.ctoa after the store and before pushing work to the accelerator. The fence.ctoa instruction would then block the CPU until all in flight loads and stores are committed. Then when the CPU pushes work to the accelerator, the accelerator is guaranteed to see the results that the CPU has committed. The CPU follows a different procedure if it wants to see results that the accelerator commits. In this scenario, the processor would execute the loop in Figure 1. The pending instruction is pushed to the accelerator through the request interface. The accelerator will then put a 0 or 1 on the response interface depending on whether or not it has work. The subsequent branch instruction will read the register and loop if there is pending work in the accelerator. Repeatedly executing this loop can be energy inefficient, so as an optimization the accelerator can choose to not respond to the pending request for some number of cycles. This will cause the CPU to block on the branch instead of needlessly executing the loop. Once the accelerator has written a 0, the processor can read results that the accelerator has committed to memory.

**CPU Interrupting the Accelerator.** Occasionally, the CPU might need to stop and reschedule the process running on the accelerator (due to an expired time quota for example). The CPU can use the request interface to stop the accelerator and push an address that the accelerator can then use to save its execution state out to memory. To make sure that the accelerator has finished saving its execution state, the CPU can use the pending loop that we describe previously. When the CPU wants to restart the accelerator at a later point, it can read the saved state from memory and push it back into the accelerator though the request interface.

**Accelerator Interrupting the CPU.** The accelerator signals an interrupt whenever it needs assistance from the CPU. The accelerator might want the CPU to execute some code on its behalf, in which case it would alert the CPU by raising the interrupt line in Figure 1. Whenever the CPU is ready to take the interrupt, it will save its current execution state and jump into an interrupt handler to take the interrupt. The CPU can figure out the interrupt cause by using the request-response interface. If the CPU wants to use the accelerator in another process while it handles the interrupt, the CPU can have the accelerator save its execution state out to memory using the process discussed above.

**Virtual Memory Support.** In a system that supports virtual memory, the CPU, which sees only virtual addresses, can push either physical or virtual addresses to the accelerator. Pushing physical addresses to the accelerator requires more work on the CPU side in order to offload work to the accelerator since the CPU has to perform the address translation. Pushing virtual addresses requires less work on the CPU side but the accelerator is now responsible for address translation. The accelerator can use its own TLB to speed up translation. If an address translation fails, the accelerator can use the interrupt interface described above to request help from the processor.

We have the CPU push virtual addresses since it is not always possible to know all the addresses that an accelerator will touch for a given task. Even in the case when we do know the addresses, pushing physical addresses presents a problem as the command queue in the accelerator can represent committed architectural state. If the OS swaps out the current process, the virtual to physical address mapping can change which would invalidate the committed physical address.

## 3. Case Study: An OS-Friendly Data-Parallel Accelerator

In this section we make a case for OS-friendly hardware accelerators. We carefully discuss the necessary changes to

```
for(int i = 0; i < n; i++)
  A[D[i]] = B[E[i]] * C[i];
```

(a) Vectorizable C loop

```
 1 loop:
 2  setvl vlen, n
 3  vld vc, Cptr
 4  vld vd, Dptr
 5  vld ve, Eptr
 6  vldidx vb, ve, Bptr
 7  vmul vt, vb, cv
 8  vsdidx vt, vd, Aptr
 9  add Cptr, Cptr, vlen
10  add Dptr, Dptr, vlen
11  add Eptr, Eptr, vlen
12  sub n, n, vlen
13  bneq n, 0, loop
14 done:
```

(b) Traditional Vector

```
 1 loop:
 2  setvl vlen, n
 3  vld vc, Cptr
 4  vld vd, Dptr
 5  vld ve, Eptr
 6  vfetch utcode
 7  add Cptr, Cptr, vlen
 8  add Dptr, Dptr, vlen
 9  add Eptr, Eptr, vlen
10  sub n, n, vlen
11  bneq n, 0, loop
12  j done
13 utcode:
14  add e, e, Bptr
15  ld b, e
16  mul t, b, c
17  add d, d, Aptr
18  sd t, d
19  stop
20 done:
```

(c) Vector-Thread

Figure 2: **Vector Code Example.** Vectorizable C code compiled for the traditional vector architecture and the vector-thread architecture.

our Hwacha data-parallel accelerator. We then show that these changes introduce very little overhead.

### 3.1 The Baseline Hwacha Data-Parallel Accelerator

We first describe the baseline microarchitecture of our Hwacha vector-threaded (VT) data-parallel accelerator in detail. We present only a brief overview of VT machines here before diving into the details of our microarchitecture, but a fuller survey of data-parallel accelerators is available in [12]. In a VT architecture, the control thread (CT) is responsible for configuring and managing the data-parallel accelerator as well as pushing work to the data-parallel accelerator. The microthreads ($\mu$T) execute the work that they receive from the CT. The CT can push work through the use of a vector-fetch instruction. When the CT executes this instruction, it sends the PC of the start of a scalar instruction stream to the data-parallel accelerator. The $\mu$Ts will then start fetching and executing instructions from this PC until they reach a stop instruction.

Figure 2 shows an example vectorizable C loop compiled for both a traditional vector and a VT architecture. In a traditional vector architecture, the CT fetches and executes all of the instructions in Figure 2b. Instructions 3–8 of Figure 2b, when executed, will push work to the data-parallel accelerator. The remaining instructions comprise the stripmining loop. Compare the compiled traditional vector code to the compiled vector-thread code in Figure 2c. In this example, the CT will fetch and execute instructions 2-12. The major difference between the code in Figure 2b and Figure 2c is that CT does not explicitly push the vector gather-scatter and vector multiply instructions to the data-parallel accelerator. Instead, instruction 6 in Figure 2c will push the start address (`utcode` at line 13 in this example) of a code block meant to run on the data-parallel accelerator. The accelerator will then start fetching instructions at this address until it hits the stop instruction. Notice that the instructions in the `utcode` section appear to be scalar instructions, but the accelerator applies them to each $\mu$Ts.

Figure 3 shows the block diagram for the Hwacha data-parallel accelerator. The control thread is mapped to the CPU while the $\mu$Ts are mapped to Hwacha. Hwacha is meant to be decoupled from the control processor and thus has a command queue [8]. This allows the strip-mining loop on the CPU to run ahead and enqueue work from multiple vector-fetch commands. Whenever ready, the issue unit dequeues a vector-fetch command and starts fetching instructions from Hwacha's instruction cache at the PC indicated by the vector-fetch command. In addition to vector fetches, Hwacha also understands vector memory operations such as vector loads and vector stores. These instructions make it possible to hide memory latencies since the CPU can prefetch the data as it pushes the instruction to Hwacha.

Hwacha has a single lane which consists of 8 256$\times$64b register banks with 1 read port and 1 write port. We use SRAMs to build the register file and opt for the more area-efficient banked register file as opposed to a monolithic register file with many read and write ports. All of the registers corresponding to a $\mu$T are grouped into the same register

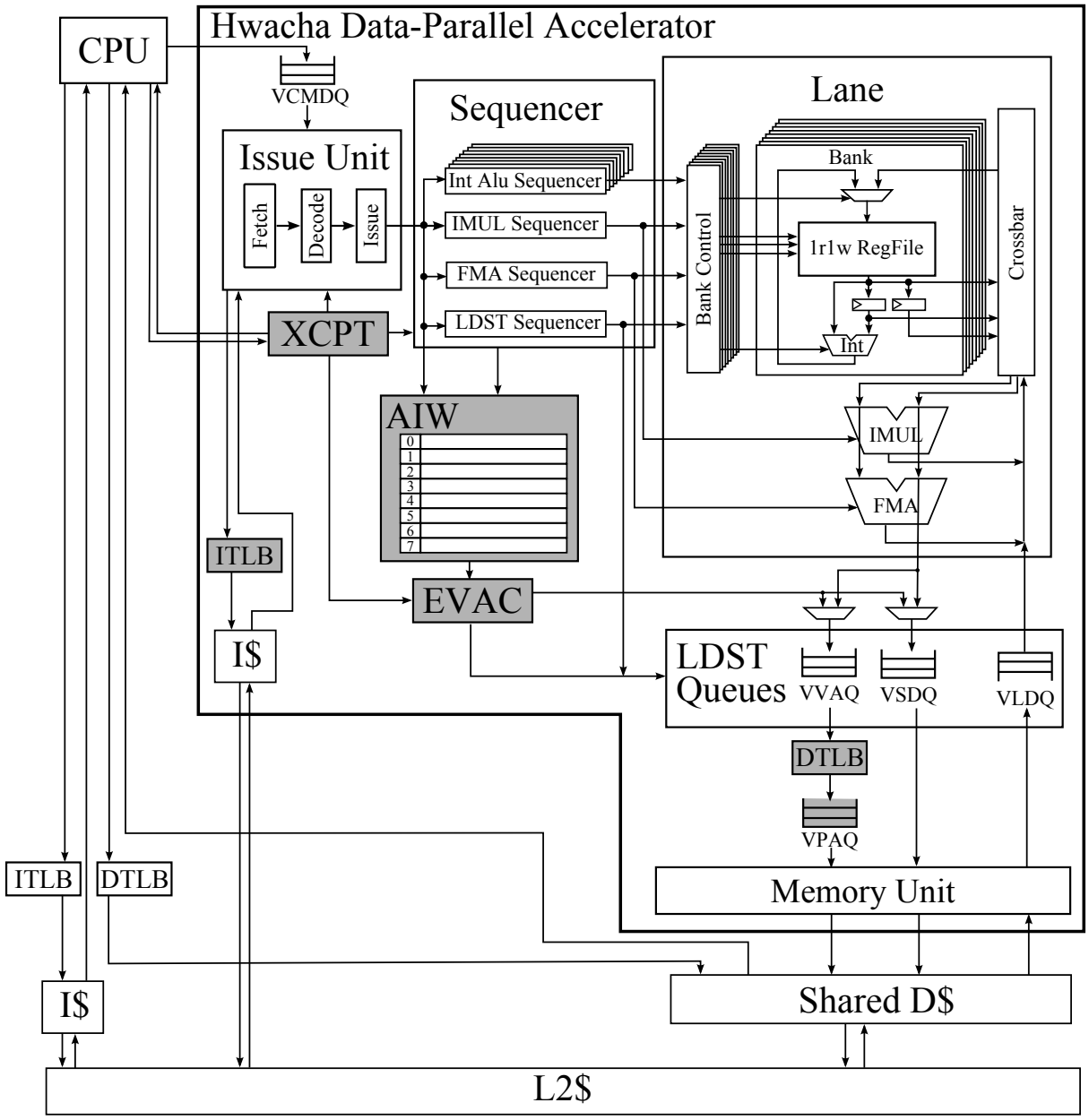



Figure 3: **Hwacha Microarchitecture.** AIW = active instruction window, D$ = data cache, EVAC = evacuator, FMA = fuse multiply-add, I$ = instruction cache, L2$ = level two cache, VCMDQ = vector command data queue, VLDQ = vector load data queue, VPAQ = vector physical address queue, VSDQ= vector store data queue, VVAQ = vector virtual address queue, XCPT = exception handling logic, EVAC = state machine that saves out the architectural state.

bank and the $\mu$T are striped across the 8 banks. Using the code from Figure 2c as an example and suppose we had a `vlen` of 64, there would be 64 total $\mu$Ts. Each $\mu$T would have its own set of registers ($b, c, d, e$ and $t$ in this example). All of the registers for $\mu$T 0 would live in register bank 0, all of the registers for $\mu$T 1 would live in the first bank, and so on. There are 8 integer ALUs connected directly to each of the register banks. The lane also has long-latency functional units (integer multiplier and single/double-precision floating-point fused multiply-adders). The data crossbar is responsible for sending data between the long-latency functional units and the register file.

The sequencer converts instructions it receives from the the issue unit into $\mu$operations, which are a sets of control signals, for each $\mu$T in the vector. As an example, instruction 14 in Figure 2c is a two-operand ALU instruction. The sequencer will turn this instruction into two sets of $\mu$op: a read $\mu$op and a read-execute-write $\mu$op. The sequencer systolically sends these $\mu$ops down the lane, accessing a different register bank every cycle as it iterates through the $\mu$T. It is the responsibility of the issue unit to not push instructions into the sequencer that would cause structural hazards such as bank conflicts (e.g., pushing two back-to-back two-operand ALU instruction would cause a conflict since the first instruction would read its second operand while the second instruction would be reading its first operand from the same bank).

The LDST sequencer, LDST queues, and memory unit coordinate data movements between the register file and the data cache. The LDST sequencer transforms memory instructions into a generate address $\mu$op and a read $\mu$op (for stores) or a write $\mu$op (for loads). For stores, addresses and data are buffered up into the VVAQ and the VSDQ. The memory consumes the address and data whenever possible. For loads, data is buffered up into the VLDQ until there is enough data. Once the appropriate amount of data returns from the memory system, the LDST sequencer pushes a write $\mu$op down the lane to writeback the data from the VLDQ into the register file. As an example, let us look at the load instruction on line 15 of Figure 2c. The LDST sequencer will first issue a read $\mu$op to read out the addresses contained in register $e$ into the address queue. Eventually, the memory unit will consume these addresses and produce the corresponding load data which it will enqueue into the load data queue. Once enough data has come back from the memory system (one word for each bank in this implementation), the LDST sequencer will issue a write $\mu$op to write back data from the VLDQ into the register file.

### 3.2 An OS-Friendly Hwacha Accelerator

In order to improve integration with a conventional OS running on a general-purpose CPU, we augment Hwacha to support virtual memory and handle restartable exceptions. The highlighted parts in Figure 3 show all the necessary augmentations to the datapath. Since Hwacha now sees virtual addresses, the addresses sitting in the VVAQ must be sent out to the TLB for translation. Once they are translated, they are enqueued into the VPAQ. Once the VPAQ buffers up enough addresses, the sequencer will issue the appropriate read or write $\mu$op.

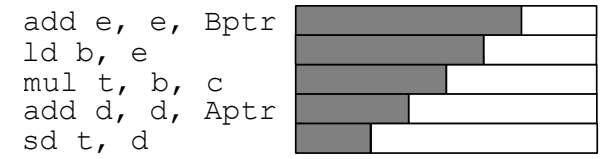



Figure 4: **Active Instruction Window.** The resulting instruction window at a certain point in time when executing the code in Figure 2. This example uses a vlen of 64. A grayed out box indicates that corresponding $\mu$T has committed that instruction.

We also augment Hwacha to handle restartable exceptions to allow for multiprogramming. Whenever the machine is busy, the sequencer will have a set of instructions in flight, as shown in Figure 4. We refer to this as the active instruction window. The AIW in Figure 3 is the hardware encoding of this structure. Every issued instruction is allocated a slot in the AIW. A slot consists of the PC of the instruction as well as the number of $\mu$Ts that have committed that instruction. Recall that the sequencer systolically iterates through each $\mu$T of the vector, pushing down the appropriate $\mu$op. We consider a $\mu$T to have committed an instruction once the sequencer has pushed down the final $\mu$op corresponding to that instruction. When this happens, the AIW is updated accordingly. Once all the $\mu$Ts have committed that instruction, the instruction is removed from the AIW.

Assume the code in Figure 2c will fill up the AIW as shown in Figure 4. Since instruction 14 is fetched and issued first, it fills up the first slot in the AIW. Instruction 14 continues to make progress as the issue unit fetches and issues the other instructions, filling up the other slots in the AIW. The AIW is updated whenever a $\mu$T commits an instruction (indicated by the gray box in Figure 4.

The AIW along with the register file encapsulates the execution state of Hwacha. On an exception, the XCPT block halts the vector machine. The EVAC block then writes out the contents of the AIW out to memory. Once that is finished, the CPU then issues vector stores to save out the register file. The CPU can execute a special instruction to read out the exception cause from Hwacha. If Hwacha took a recoverable exception, its state can be restored as follows. The CPU first issues vector loads to restore the register file. Next the CPU tells Hwacha to start fetching instructions at the address of the earliest instruction in the saved AIW. The CPU will also send the amount of progress that the instructions have made. When the instruction is issued, only the $\mu$Ts which have not yet committed that instruction will execute it.

To see how this works, we will assume that Hwacha took an exception when executing the code in Figure 2c (the load on line 15 failed a translation, for example) and that the AIW looks like Figure 4. Hwacha will first raise a flag to indicate that it wants to take an exception. The CPU will then give Hwacha an address to save its state. Hwacha will then write out the AIW state to this address. For example, it will write the PC of the instruction at line 14 and the number of $\mu$Ts that have committed the instruction. The same is done for each of the remaining instructions in the AIW. Next, the CPU issues vector stores to save the register state. Finally, the CPU will ask Hwacha for the exception cause so it can run the appropriate exception handler. On recovery, the CPU issues vector loads to restore register file state. Next, the CPU reads the saved Hwacha state and sees that the instruction at line 14 needs to be reissued. It pushes the address of the instruction as well as the progress that the instruction has made to Hwacha. Hwacha will fetch and issue this instruction and every instruction following it along with the progress information that it receives from the CPU. The sequencer and AIW will see that non-zero progress has been made. For the sequencer, this means that the add on line 14 will only issue $\mu$ops for $\mu$Ts that have not yet committed the instruction. For the AIW, the add instruction will occupy the first slot as before, but the progress count will start at however many $\mu$Ts have committed the instruction instead of 0.

### 3.3 Area and Energy Results

To quantify the area and energy overhead of adding virtual memory and restartable exception support to the data-parallel accelerator, we wrote RTL for the processor in Chisel HDL [3] and pushed the design through a Synopsys-based ASIC toolflow. This section describes the target machine in more detail, and the hardware and software infrastructure we used to generate accurate estimates for silicon area and energy, and finally discusses the results.

**Target Machine.** We built the Hwacha data-parallel accelerator described in the previous section alongside a 5-stage in-order decoupled RISC-V Rocket CPU [17]. The CPU has a 16KB 2-way set-associative L1 instruction cache with an 8-entry TLB, and the Hwacha data-parallel accelerator has an 8KB direct-mapped L1 instruction cache with a 2-entry TLB. The CPU and the accelerator share a 32KB 4-way set-associative L1 data cache. Both the CPU and the accelerator have a private 8-entry data TLB. The primary caches are all backed by a unified 256KB 8-way set-associative L2 data cache.

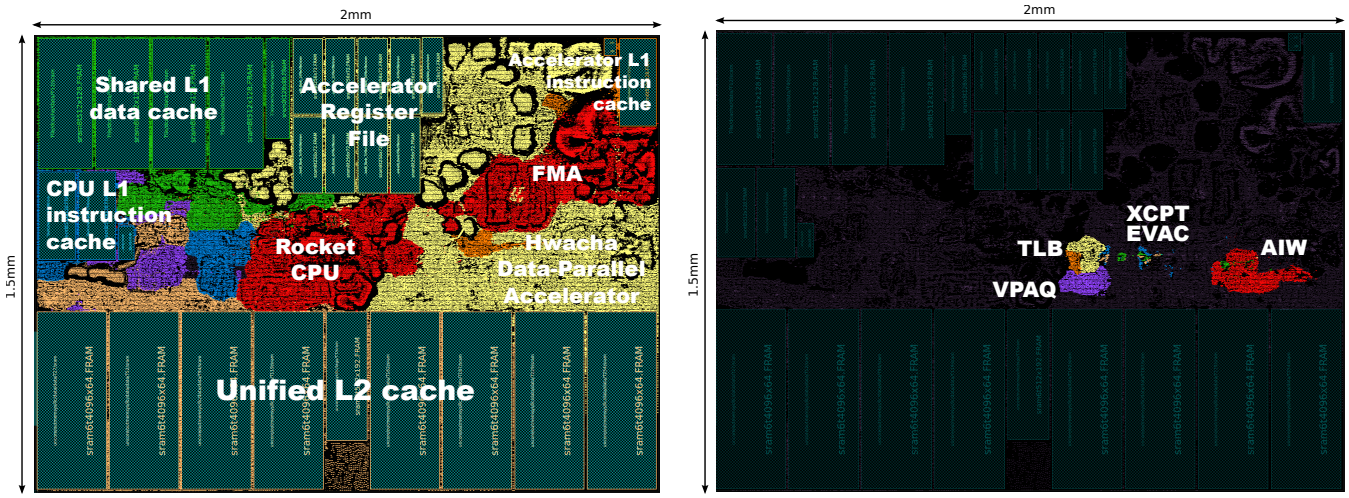
**Hardware Toolflow.** We targeted TSMC's 45nm GP CMOS library using a Synopsys-based ASIC toolflow using methodologies described in [6, 12]. We use VCS for simulation, Design Compiler for synthesis, IC Compiler for place-and-route, and PrimeTime for power analysis. We make use of Design Compiler's retiming functionality to retime our long-latency functional units. The vector register file, instruction caches, and data cache are implemented with SRAMs. We do not have access to a memory compiler for our target process, so we model SRAMs by creating abstracted black-box modules, with area, timing, and power models generated by CACTI [13]. We simulate the post place-and-route gate-level Verilog netlist with the logic and wire delay information generated from the layout to capture accurate switching activities. We combine this activity count with the layout information to generate detailed power/energy numbers. Figure 5a shows the ASIC layout from IC Compiler.

| Module Hierarchy | Area ($mm^2$) | | Power (mW) | |
|---|---|---|---|---|
| Total | 2.156 | | 81.10 | |
| Tile | 0.978 | 100.0% | 69.80 | 100.0% |
| Tile/CPU | 0.105 | 10.7% | 16.00 | 22.9% |
| Tile/CPU/ICache | 0.092 | 9.4% | 3.48 | 4.9% |
| Tile/CPU/FPU | 0.064 | 6.5% | 7.13 | 10.2% |
| Tile/CPU/FPU/DFMA | 0.025 | 2.6% | 2.21 | 3.2% |
| Tile/CPU/FPU/SFMA | 0.008 | 0.8% | 2.49 | 3.6% |
| Tile/Hwacha | 0.422 | 43.1% | 37.50 | 53.7% |
| Tile/Hwacha/ICache | 0.044 | 4.5% | 1.46 | 2.1% |
| Tile/Hwacha/Lane | 0.274 | 28.0% | 22.30 | 31.8% |
| Tile/DCache | 0.357 | 36.5% | 13.20 | 18.9% |
| L2$ | 1.147 | | 6.78 | |
| Overhead | | | | |
| Tile/Hwacha/ITLB | 0.0025 | 0.2% | 0.21 | 0.3% |
| Tile/Hwacha/DTLB | 0.0033 | 0.3% | 0.22 | 0.3% |
| Tile/Hwacha/AIW | 0.0077 | 0.8% | 0.69 | 1.0% |
| Tile/Hwacha/EVAC | 0.0008 | 0.1% | 0.07 | 0.1% |
| Tile/Hwacha/XCPT | 0.0008 | 0.1% | 0.12 | 0.2% |
| Tile/Hwacha/VPAQ | 0.0034 | 0.3% | 0.44 | 0.6% |

Figure 6: **Area and Power Breakdown.** Note that area numbers only account for the actual area used for the standard cells and SRAM macros. We used a 70% utilization factor for place-and-route, so to calculate the actual silicon area, multiply the area number by 10/7 to calculate the actual silicon area. To calculate energy numbers multiply 804us, since the sgemm benchmark ran for 803,664 cycles at 1GHz. Percentages are calculated with respect to the Tile which consists of CPU+Hwacha.

**Software Toolchain.** We use our RISC-V GCC (4.6.1) cross compiler and the Three Fingered Jack (TFJ) system [15], an in-house loop vectorizer to map microbenchmarks, kernels, and applications to our CPU and Hwacha data-parallel accelerator. For this case study, we use a 64 $\times$ 64 single-precision floating-point matrix multiplication routine that runs on the vector unit.

**Area Overhead.** Figure 5b highlights the modules we added to the Hwacha data-parallel accelerator in order to support virtual memory and restartable exceptions. Table 6 shows the post-PAR area results of the processor. The area




(a) Rocket CPU/Hwacha Data-Parallel Accelerator

(b) Overhead

Figure 5: **VLSI Layouts.** The scales are listed in the layout. This design meets timing at 1GHz. The Rocket CPU, the Hwacha data-parallel accelerator, register file, L1 instruction caches, L1 data cache, and the unified L2 cache are highlighted. The modules we added to support virtual memory and restartable exceptions are shown on the right.

overhead of the added modules (TLB+AIW+EVAC+XCPT+ VPAQ) turns out to be only 4.4% of the total data-parallel accelerator area, 1.8% of the CPU + data-parallel accelerator area, or 0.9% of the total area including the L2 cache.

**Power/Energy Overhead.** Table 6 also shows the power breakdown of various modules. The power/energy overhead of supporting virtual memory and restartable exceptions are only 4.7% of the total power/energy consumption of the data-parallel accelerator, 2.5% of the CPU + data-parallel accelerator, or 2.3% of the entire chip.

## 4. Related Work

Early vector machines such as Cray-1 [14] required all pages it accessed to be pinned in physical memory, due to the difficulty of implementing precise exceptions or restartable exceptions. The IBM Vector Facility supported restartable exceptions by limiting the machine to only one vector instruction in execution at a time [7]. Asanović [2] proposed a decoupled vector pipeline design that issues vector instructions to the vector datapath only when all addresses from previous vector memory instructions are known to not cause an exception. Espasa et al. [9] and Kozyrakis [11] renamed vector destination registers to implement precise exceptions. Once a vector instruction faults, the destination registers of all subsequent vector instructions are rolled back to the previous mapping to maintain preciseness. Since the Hwacha data-parallel accelerator eschews vector register renaming, it must allow partial completion of more than one vector instruction (see Figure 4), at the expense of larger architectural state.

Hampton [10] presented software restart markers as a foundation to handle exceptions in parallel architectures. The compiler is responsible for delimiting the program into idempotent regions. Once an exception occurs, the operating system will simply resume execution from the beginning of the faulting region. Although this approach has very low implementation overhead, it is constrained by the ability of the compiler to statically determine the idempotency of a region, and can hence have large execution overheads on some codes.

Our proposal is most similar to the DEC Vector Vax [5] design, which provided the OS with an opaque microarchitectural state save and restore mechanism, and also provided fence instructions to synchronize vector unit execution with the scalar processor.

## 5. Conclusions

In order to remain successful, hardware accelerators must seamlessly integrate with general-purpose CPUs. We describe a general framework for building OS-friendly hardware accelerators, consisting of a generic connection interface, a memory consistency model, and requirements for virtual-memory support.

Our case study details the augmentations we made to our Hwacha data-parallel accelerator in order to integrate it into general-purpose systems. We push our combined system through an ASIC toolflow to extract area and power/energy numbers. Our results show that there is very little overhead (1.8% overhead in area and 2.5% overhead in energy over the combined CPU and Hwacha) in making Hwacha OS-friendly. In future work, we plan to build a library of general-purpose hardware accelerators using our proposed framework to allow integration with conventional operating systems.

## 6. Acknowledgements

Research partly funded by DARPA Award Number HR0011-12-2-0016. The content of this paper does not necessarily reflect the position or the policy of the US government and no official endorsement should be inferred.

## References

[1] AnandTech. *LG Optimus 2X & NVIDIA Tegra 2 Review*.

[2] K. Asanović. *Vector Microprocessors*. PhD thesis, EECS Department, University of California, Berkeley, 1998.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12.

[4] C. Batten, R. Krashinsky, S. Gerding, and K. Asanović. Cache refill/access decoupling for vector machines. In *37th International Symposium on Microarchitecture*, pages 331–342, Portland, OR, December 2004.

[5] D. Bhandarkar and R. Brunner. VAX vector architecture. In *ISCA-17*, 1990.

[6] H. Bhatnagar. *Advanced ASIC Chip Synthesis Using Synopsys® Design Compiler® Physical Compiler® and PrimeTime®*. Springer, 2001.

[7] W. Buchholz. The IBM System/370 vector architecture. *IBM Systems Journal*, 25(1):51–62, 1986.

[8] R. Espasa and M. Valero. Decoupled vector architectures. In *Proc. 2nd High Performance Computer Architecture Conf.*, pages 281–290. IEEE Computer Society Press, Feb 1996.

[9] R. Espasa, M. Valero, and J. E. Smith. Out-of-order vector architectures. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, 1997.

[10] M. Hampton. *Reducing Exception Management Overhead with Software Restart Markers*. PhD thesis, Massachusetts Institute of Technology, 2008.

[11] C. Kozyrakis. *Scalable Vector Media-processors for Embedded Systems*. PhD thesis, EECS Department, University of California, Berkeley, 2002.

[12] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *ISCA*, 2011.

[13] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, 2009.

[14] R. M. Russell. The Cray-1 computer system. *Communications of the ACM*, 21(1):63–72, Jan. 1978.

[15] D. Sheffield, M. Anderson, and K. Keutzer. Automatic generation of application-specific accelerators for fpgas from python loop nests. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012.

[16] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, S. Bryskin, J. Lugo-Martinez, S. Steven, and M. B. Taylor. Conservation cores: Reducing the energy of mature computations. In *Architectural Support for Programming Languages and Operating Systems*, Asplos '10.

[17] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Number UCB/EECS-2011-62.