The Accelerator Store framework for high-performance, low-power accelerator-based systems

Michael J. Lyons† (mjlyons@eecs.harvard.edu), Mark Hempstead‡ (mhempstead@coe.drexel.edu),
Gu-Yeon Wei† (guyeon@eecs.harvard.edu), and David Brooks† (dbrooks@eecs.harvard.edu)
†Harvard University, ‡Drexel University

Abstract—Hardware acceleration can increase performance and reduce energy consumption. To maximize these benefits, accelerator-based systems that emphasize computation on accelerators (rather than on general purpose cores) should be used. We introduce the "accelerator store," a structure for sharing memory between accelerators in these accelerator-based systems. The accelerator store simplifies accelerator I/O and reduces area by mapping memory to accelerators when needed at runtime. Preliminary results demonstrate a 30% system area reduction with no energy overhead and less than 1% performance overhead in contrast to conventional DMA schemes.

1 Introduction

Tightening power budgets and performance limits have led to a surge of interest in hardware accelerators. Many processors currently use these specialized circuits to compute specific workloads – the iPhone processor's HD video engine [6] is one such example. Designers rely on specialization to increase logic efficiency, which improves energy consumption and performance by 100-500x compared to general purpose cores [3]. Accelerators achieve these gains at the expense of flexibility, limiting each accelerator to the workloads it was designed for. As a result, accelerated systems must utilize several hardware accelerators to target the diverse set of workloads typically handled by processors. Each accelerator must be small since many accelerators will be needed, and must communicate with other accelerators efficiently to prevent I/O complexity and bottlenecks. Unfortunately, today's accelerators meet neither criteria: most contain large memories and rely on complex DMA I/O schedules. To address these challenges, we introduce the "accelerator store," a shared memory structure that simplifies accelerator I/O and reduces chip area.

We developed the accelerator store to take advantage of hardware acceleration's improved energy and performance. Diminishing threshold voltage reductions limit the amount of logic that a processor can simultaneously power, resulting in growing regions of unpowered logic known as "dark silicon" [10]. Dark silicon threatens general purpose (GP) multicore processor performance by limiting the number of active cores, in conflict with multicore's thirst for additional active logic. Although accelerated systems are subject to the same power limits as GP-CPUs, improved logic efficiency means that accelerated systems can use less energy for the same computations. We envision accelerated systems will contain several accelerators, but only turn on the accelerators designed for the current workload. This approach allows systems to use growing transistor budgets to add several accelerators, while powering the subset designed for current workloads. As a result, accelerated systems will only power highly efficient and lower energy logic, and therefore are less likely to encounter dark silicon problems when compared to GP approaches.

This multi-accelerator vision requires a new architecture that favors accelerators over general purpose logic whenever possible. Toward this vision, we introduce the *accelerator store*, an efficient memory structure designed to simplify accelerated system design and reduce chip area. Current DMA approaches follow a predetermined sequence of data transfers between accelerators and system memory: a GP-CPU core decides the location and size of transfers in advance, and accelerators can only request the next transfer in the sequence. In accelerator store-based systems, accelerators decide which blocks of memory to access and how much data to transfer on demand. This increased flexibility lets accelerators tackle unpredictable workloads without GP-CPU involvement. The accelerator store achieves this by offering random access and FIFO interfaces, the most common memory primitives used in ASIC and FPGA toolkits.

The following design aspects of the accelerator store reduce accelerator area demands and simplify accelerator I/O with minimal impact on performance and energy:

- **Reduced memory buffers:** Accelerators can directly access data in the accelerator store without large buffer memories required by DMA.
- **Memory reuse:** The accelerator store dedicates memory to running accelerators. When an accelerator finishes and turns off, the accelerator store rededicates the memory to other running accelerators. This requires less memory than present approaches which provision memory to accelerators individually.
- **Simple accelerator I/O:** FIFOs provide a simple mechanism for exchanging data between accelerators. In contrast to FIFOs, DMA requires accelerators to agree on a common data structure or a GP-CPU to translate between structures.
- **Reduced GP-CPU energy:** FIFOs also allow accelerators to communicate with little or no GP-CPU assistance. This lets GP-CPUs stay in low-power sleep modes for more time.
- **Reduced memory energy:** The accelerator store monitors each of its SRAM memory blocks and turns off any that are unused (SRAMs must be powered if they contain data, otherwise they will lose state).
- **Low performance overhead:** Although a centralized structure may suggest high performance overheads, early results show the accelerator store reduces performance by less than 1% using memory selection techniques discussed in Section 3.2.

In the remaining sections, we describe the accelerator store design and how accelerators use the accelerator store to access shared memory (Section 2), analyze private memories in eleven accelerators to estimate potential area reductions and performance pitfalls (Section 3), and achieve a 30% area reduction with no impact on energy and a performance impact of less than 1% running on a simulated prototype system (Section 4).

2 Accelerator store design

Accelerator store systems contain several accelerators, one or more GP-CPU cores, and an accelerator store (Figure 1).

Manuscript submitted: 13-Aug-2010. Manuscript accepted: 19-Sep-2010. Final manuscript received: 28-Sep-2010

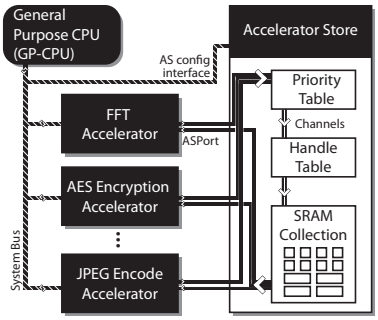

Fig. 1. **Example accelerator store processor**

These systems rely on accelerators to process most workloads, whereas GP-CPU cores provide system oversight and perform rarely executed operations (those not worth accelerating). The system's accelerator store manages accelerator data, and provides each accelerator with an *accelerator store port (ASPort)* to access this data. Accelerators use dedicated ASPorts rather than a single shared bus to avoid contention, since accelerators often require significant bandwidth when accessing memory.

Data access requests from accelerators traverse through ASPorts and continue through the accelerator store's three major components: the *priority table* for arbitrating memory requests from accelerators, the *handle table* for translating these requests from accelerators into SRAM accesses, and the *SRAM collection* for storing data. In cases when the SRAM collection does not have enough storage capacity, accelerator state can be temporarily paged out from the accelerator store to system memory.

We connect each component as Figure 1 shows, and discuss the design of each component in Section 2.2. First we explain the accelerator store's *handle* abstraction for shared memory.

2.1 Accelerator store handles

Accelerators cannot access accelerator store memory directly. Instead, the memory must be allocated to a handle, which represents a region of shared memory. Accelerators can then access the memory region by specifying the handle's corresponding *handle ID* number. Software running on a GP-CPU core is responsible for creating handles in the accelerator store and distributing the corresponding handle IDs to the relevant accelerators. Currently, the programmer must specify the maximum size of memory allocated to each handle statically at compile time. We are investigating future tools and runtime systems for the accelerator store that would support dynamic memory allocation and mapping.

During computation, accelerators send information to each other using FIFO handles. For example, if an application wishes to compress and encrypt data, it will configure a compression accelerator to send its resulting compressed data to the encryption accelerator. The software creates a FIFO handle and gives the handle ID to both accelerators during configuration. The compression accelerator pushes results into the FIFO handle, and the encryption accelerator retrieves the compressed data by popping it from the FIFO handle. In this way, FIFO handles allow accelerators to exchange data without any prior knowledge about each other's design or data structures. In addition to FIFO handles, the accelerator store supports random access (RA) handles which uses addresses rather than push and pop operations to access data.

Handles provide several benefits to accelerator based systems. First, handles provide memory protection – accelerators can only access a handle's memory if given its handle ID. Second, handles declare the mapping of shared physical memories to accelerators. This information includes the location and amount of SRAM memory mapped to a handle. Third, handles enable automatic SRAM VDD-gating. Without handles, the accelerator store cannot know which data is valid and cannot safely turn off any SRAMs.

TABLE 1
Accelerator memory compositions

Accelerator	Function	Memory area
AES	Data encryption	40.3%
JPEG	Image compression	53.3%
FFT	Signal processing	48.6%
Viterbi	Convolutional coding	55.6%
Ethernet	Networking	90.7%
USB (v2)	Peripheral bus	79.2%
TFT Controller	Graphics	65.9%
Reed Solomon Decoder	Block coding	84.3%
UMTS 3GPP Decoder	Turbo coding	89.2%
CAN	Automotive bus	70.0%
DVB FEC Encoder	Video error correction	81.7%
Average		69.0%

2.2 Accelerator store components

The following accelerator store components make the handle abstraction possible:

2.2.1 Priority table

The priority table arbitrates all handle requests from accelerators. Each accelerator communicates with the accelerator store via an ASPort, but the accelerator store may not be able to satisfy requests from every port at every cycle. The priority table selects a subset to satisfy using a priority-ordered list of ASPorts. Starting with the highest priority ports, it accepts as many requests as it can satisfy in a cycle. The priority table can be modified at runtime – this allows software to prevent starvation and provide bandwidth management by regularly adjusting the priority table.

2.2.2 Handle table

The handle table stores each handle's configuration and uses this information to translate handle requests from accelerators into SRAM accesses. To explain it's functionality, we offer an example: when an element is to be popped from a FIFO, the handle table receives a handle request containing the FIFO handle ID. The ID leads to the handle configuration, which includes the FIFO handle's address in SRAM memory and the head pointer (the next element to pop). The handle table uses this information to identify the word in SRAM corresponding to the pop request, and passes this information to the SRAM collection. Meanwhile, the handle table updates the head pointer.

Handles can be modified, created, or deleted at runtime, allowing software to reuse memory with different accelerators and to use sophisticated dynamic memory allocation schemes.

2.2.3 SRAM collection

The accelerator store contains several SRAMs to increase VDD gating opportunities. If the accelerator store contained one valid word and used one large SRAM, the entire large SRAM would need to remain on. By instead using many small memories, the accelerator store can VDD-gate all but one SRAM to reduce leakage power.

The SRAM collection contains a mix of 2KB and 4KB memories, since measurements of UMC 130nm SRAMs indicated these memories provided the best balance between VDD gating granularity and addressing logic overhead. To achieve these VDD-gating benefits, the accelerator store turns off all SRAMs not mapped to handles. It also uses head and tail pointers to identify and VDD-gate SRAMs mapped to FIFO handles that do not contain valid data.

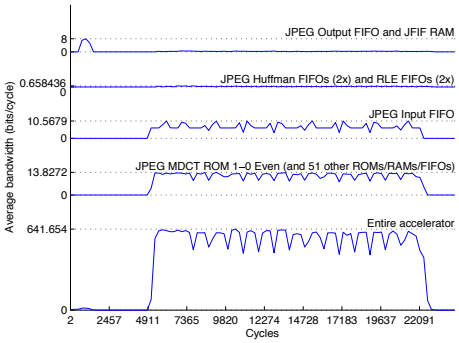

Fig. 2. **JPEG accelerator memory profiles** Access patterns for each accelerator memory category are shown during compression of a 640x8 pixel image. Memories within each category are similar and omitted for space.

3 Characterizing Accelerator Memory

Centralizing data is key to the accelerator store's area reductions, but must not introduce significant contention. Contention concerns are especially relevant for accelerators, since dedicated memories are frequently touted as a primary source of performance improvement. We first examine the contents of eleven accelerators, and find each contains significant amounts of memory which can be reduced using the accelerator store. We then focus on throughput patterns of memories in four accelerators to ensure area savings does not come at the expense of performance.

3.1 Accelerator composition & private memories

Memory centralization achieves significant area reductions through memory reuse only if accelerator area is dominated by memory. To evaluate the opportunity for memory area reductions, we analyze the composition of eleven accelerators from OpenCores [1] and Xilinx [12] (Table 1). ASIC area measurements are obtained using synthesis results from Design Compiler when generic RTL is available. ASIC area measurements for FPGA-specific RTL is obtained by adjusting Xilinx ISE synthesis results using known scaling factors [5].

Table 1 shows that an average of 69% of accelerator area is consumed by memory – a large potential for area reductions. We must ensure these reductions are possible without creating significant contention.

3.2 Accelerator memory utilization

To understand the characteristics of accelerator memory use, memories in four accelerators (JPEG, AES, Viterbi, FFT) were instrumented to record all accesses while processing test workloads. Results for the JPEG accelerator are shown in Figure 2; Viterbi, AES, and FFT show similar access characteristics but are omitted for space.

Results indicate many memories have varying throughputs and are used for varying lengths of time. The first three categories of JPEG SRAMs in Figure 2 are well suited for transplant to the accelerator store. All of these memories have a sizable capacity, and are used infrequently or utilize little throughput. The fourth category of JPEG SRAMs contains many small ROMs, RAMs, and FIFOs that also have the highest throughput. This fourth memory category is poorly suited for the accelerator store because their small size will result in little area reduction and large throughput will add significant performance loss due to contention.

This accelerator memory area characterization demonstrates that significant memory area reductions and low performance overheads are possible in principle, provided memories are intelligently selected for the accelerator store. The following section evaluates the accelerator store's ability to achieve these goals while powering a security application.

4 Evaluation

This section evaluates performance overheads, energy overheads, and area savings while running a prototype embedded security application on a simulated accelerator store system. The security application listens for suspicious activity and records surveillance photos if alerted using three accelerators from Table 1 (FFT, JPEG, and AES). The application also uses three peripherals designed for the accelerator store (ADC, digital camera, and SD flash memory). The application detects suspicious activity by acquiring audio signals from a microphone, analyzing them with the FFT, and checking if a frequency response corresponding to suspicious activity occurred (glass breaking or dog barking). If so, the camera takes pictures every second, compresses the photos using the JPEG accelerator, encrypts the JPEGs with the AES accelerator, and writes the encrypted photos to the SD flash card. All accelerators and peripherals maintain input and output FIFOs in the accelerator store, and some maintain internal FIFOs and memories in the accelerator store as well. Virtually all computation is executed on accelerators rather than GP cores.

4.1 Cycle accurate simulation

Cycle accurate simulation of the accelerator store consists of two parts: simulating the accelerator store logic, and reproducing accelerator memory accesses.

The accelerator store portion of the simulator's timing model is based on an RTL implementation, designed in Bluespec HLL [2] and compiled to RTL. Accelerator store power and area estimates are obtained by synthesizing the RTL in Design Compiler for the UMC 130nm GP process.

The simulator models accelerators by replaying traces of memory accesses for each memory moved to the accelerator store. Power consumption due to wire interconnect is from previous estimates [4]. We conservatively assume interconnect wires are the maximum length ($chip\ width + height$).

Cycle accurate accelerator timing is maintained by ensuring the time between accelerator memory accesses is unchanged. For example, if an accelerator store access stalls due to contention, the simulator assumes that the accelerator stalls until the access completes. This is a conservative performance estimate, and assumes that all memory accesses are dependent on all prior memory accesses. Actual accelerators may not exhibit this behavior and may yield better than simulated performance. Accelerator area and energy values are Design Compiler estimates for the UMC 130nm GP process.

4.2 Selecting memories to move into the accelerator store

To balance memory size and minimize contention, memories with the largest values for the memory selection metric $\frac{memory\ capacity}{average\ throughput}$ are moved from the private accelerator memory to the accelerator store. This includes all input and output FIFOs, as well as several large random access and internal FIFO memories as discussed in Section 3.2.

4.3 Performance & energy overhead

The accelerator store's performance overhead is first evaluated by tuning two parameters: shared memory percentage and internal bandwidth (Figure 3). Shared memory percentage refers to the amount of accelerator memory located in the accelerator store, rather than private accelerator memory. Systems with 0% shared memory only use private memories, whereas 100% shared memory systems only use shared accelerator store memory. Memory is moved from private to shared in order of the memory selection metric described in Section 4.2: memories

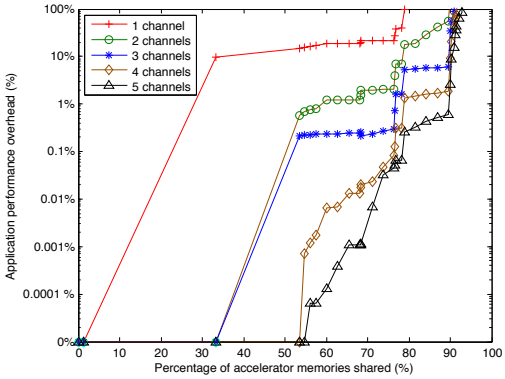

Fig. 3. **Performance overhead** Memories are added in order of memory selection metric. Channels represent maximum access requests from accelerators per cycle.

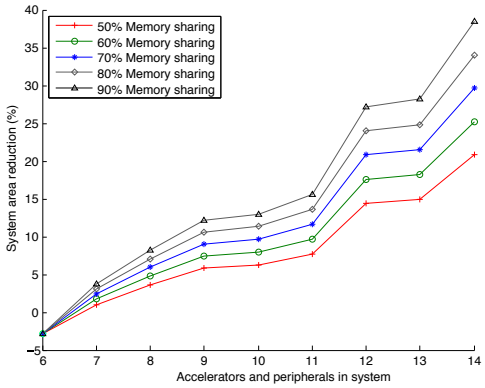

Fig. 4. **System area savings** System area includes accelerator store and accelerators from Table 1. GP-CPU area is considered insignificant compared to total accelerator area.

with the highest ratio of capacity to throughput are selected first (on the left side of the plot).

Internal bandwidth is measured in *channels*, defined as the number of handle requests the accelerator store can process per cycle. Increasing channels decreases contention, but increases energy and area. The number of channels must be chosen to balance these overheads.

Results show the accelerator store can share a majority of accelerator memory with a limited number of channels and low performance degradation. Systems should not share all accelerator memories: even five channels cannot keep performance overheads below 100% when all memories are shared. Sharing large memories and FIFOs, and keeping small lookup memories private results in significant memory sharing with insignificant performance overheads. In this approach, a three channel system maintains 70% of all accelerator memory in the accelerator store with less than 1% impact on performance.

Energy consumption overhead is manageable as well. The accelerator store introduces an 8% system energy consumption overhead when ignoring VDD-gating features. However automatic VDD-gating reduces energy consumption by roughly 8%, effectively canceling out energy overheads. Although accelerators could implement memory VDD-gating without the accelerator store, we have never seen an accelerator that implements this feature. Further, supporting automatic VDD-gating in the accelerator store requires only one set of gating logic rather than replicating copies in each accelerator.

4.4 Area reduction

The accelerator store's ability to reduce memory area (by eliminating I/O buffer memories and dedicating memory to accelerators at runtime) assumes that a significant portion of accelerators will be unused at any given time. If the system only contains accelerators that will be in use simultaneously, there is no opportunity for reusing memory. Such a processor containing only the six accelerators and peripherals used by the security application would incur a 3% system area overhead (Figure 4) compared to a system without an accelerator store. However, we believe designers are unlikely to fabricate a processor for a single application and will include a set of commonly used accelerators that applications can choose from. In this scenario, the processor is more likely to contain additional accelerators, such as those listed in Table 1. In such a configuration, the accelerator store reduces system area by 30%.

5 Related Work

Multi-component memory sharing has been explored but without the full features of the accelerator store. SoCDMMU [11] provides hardware support for malloc() for multicore SoCs but does not support FIFOs, automatic VDD-gating, or handles. Smart memories [8] features a GP-CPU in multiple tiles that interact with memory, but does not target accelerators or support automatic VDD-gating. Memory sharing within functional units [9] and server blades [7] has been investigated as well, although these approaches are not designed for accelerators and do not support automatic VDD-gating, handles, or FIFOs.

6 Conclusion

We introduced the accelerator store, a shared memory framework for accelerator-based systems. We described the accelerator store's handle-based approach to sharing accelerator memory, characterized accelerator memory to gauge the potential for area reduction and performance degradation, and achieved system area reductions of 30% with less than 1% performance impact and no additional energy in a simulated prototype.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. IIS-0926148. The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

References

[1] "Opencores," http://www.opencores.org.
[2] Arvind *et al.*, "High-level synthesis: An essential ingredient for designing complex ASICs," in *ICCAD 2004*.
[3] R. Hameed *et al.*, "Understanding sources of inefficiency in general-purpose chips," in *ISCA 2010*.
[4] R. Ho *et al.*, "High speed and low energy capacitively driven on-chip wires," *JSSC*, vol. 43, no. 1, pp. 52–60, Jan. 2008.
[5] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *FPGA 2006*.
[6] A. lai Shimpi, "The iPhone 3GS hardware exposed and analyzed," http://www.anandtech.com/show/2782.
[7] K. Lim *et al.*, "Disaggregated memory for expansion and sharing in blade servers," in *ISCA 2009*.
[8] K. Mai *et al.*, "Smart memories: A modular reconfigurable architecture," in *ISCA*, May 2000.
[9] A. Meixner and D. J. Sorin, "Unified microprocessor core storage," in *CF 2007*.
[10] R. Merritt, "ARM CTO: power surge could create 'dark silicon'," *EE Times*, 2009.
[11] M. Shalan and V. J. Mooney, "A dynamic memory management unit for embedded real-time system-on-a-chip," in *CASES 2000*.
[12] Xilinx, "Xilinx IP center," http://www.xilinx.com/ipcenter.