The Accelerator Store: A Shared Memory Framework For Accelerator-Based Systems

MICHAEL J. LYONS, Harvard University
MARK HEMPSTEAD, Drexel University
GU-YEON WEI and DAVID BROOKS, Harvard University

This paper presents the many-accelerator architecture, a design approach combining the scalability of homogeneous multi-core architectures and system-on-chip's high performance and power-efficient hardware accelerators. In preparation for systems containing tens or hundreds of accelerators, we characterize a diverse pool of accelerators and find each contains significant amounts of SRAM memory (up to 90% of their area). We take advantage of this discovery and introduce the accelerator store, a scalable architectural component to minimize accelerator area by sharing its memories between accelerators. We evaluate the accelerator store for two applications and find significant system area reductions (30%) in exchange for small overheads (2% performance, 0%–8% energy). The paper also identifies new research directions enabled by the accelerator store and the many-accelerator architecture.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) Systems*

General Terms: Design, Performance, Management

Additional Key Words and Phrases: Hardware acceleration, memory systems, shared memory, low power

ACM Reference Format:
Lyons, M. J., Hempstead, M., Wei, G.-Y., and Brooks, D. 2012. The accelerator store: A shared memory framework for accelerator-based systems. ACM Trans. Architec. Code Optim. 8, 4, Article 48 (January 2012), 22 pages.
DOI = 10.1145/2086696.2086727 http://doi.acm.org/10.1145/2086696.2086727

1. INTRODUCTION

Increasing performance has always driven computer architecture. But for over a decade, power consumption has been architecture's greatest obstacle. Architects once relied on faster clocks, deeper pipelines, and more complex prediction to improve performance. But by the early 2000's, rocketing power consumption rendered the old architectural techniques unusable. The multicore age was born.

Transistor sizes continued dropping, but without additional power additional transistors were useless. So, for even greater power reductions, architects created designs that lowered clock frequencies. The result: processors containing multiple, identical simpler cores, each using significantly less power. Individually, these cores no longer

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 program. The authors also acknowledge the support of the National Science Foundation (award no. CCF-0926148). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.
Author's address: M. J. Lyons, Harvard University, 33 Oxford St., Cambridge, MA; email: mjlyons@eecs.harvard.edu.

DOI 10.1145/2086696.2086727 http://doi.acm.org/10.1145/2086696.2086727

kept up with 1990's performance improvements, but together, the processor's cores could achieve compelling results. The transition to multicore was, and continues to be, difficult for software developers; to obtain multicore performance, programmers must parallelize code to simultaneously run on multiple cores. And as architects pile on more cores, software must be parallelized even more. But escalating core counts have led to a potentially deeper problem, dark silicon.

Unfortunately, the benefits of frequency reduction tapered off; eventually, power and performance scale at the same rate. Emerging subthreshold approaches can reduce power through very slow clock speeds, but lead to long latencies and depend on extensive software parallelization. So without other means to reduce power, architects must turn off cores to stay within power budgets. These *dark silicon* transistors are unused and therefore wasted. Yet accepting that some dark silicon is unavoidable opens another avenue for performance improvement, hardware accelerators.

In the dark silicon era, processors contain more cores than can be powered simultaneously; there is no reason to create more copies of the same core design than can be powered. Including a heterogeneous mix of core designs allows the processor to turn on the most efficient cores for current workloads, and only the least efficient cores will be dark at any given time.

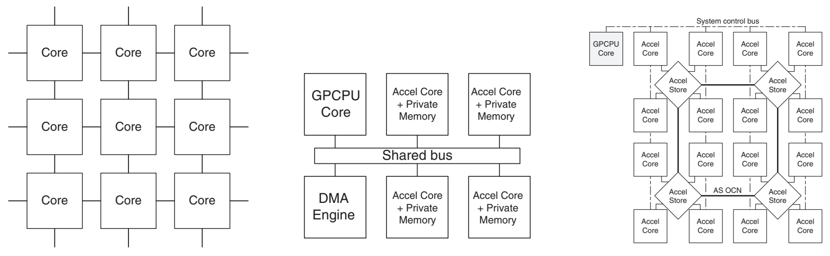
Hardware accelerator cores represent the frontier of customization-fueled performance: for maximum efficiency, each accelerator implements a single algorithm, such as AES encryption or JPEG compression. Efficiency gains result in significant performance improvement: Previous work has shown 100x increases [Hameed et al. 2010]. By building processors containing many accelerators, not just identical copies of the same general purpose core, architects can again achieve 90's performance gains. The key is taking advantage of, rather than falling victim to, dark silicon.

In order for many-accelerator systems to be a viable performance-enhancing solution, it is important to first understand characteristics of several accelerators and develop a flexible framework that ties them together. This paper presents the accelerator store (AS), a shared-memory framework, which allows for efficient allocation and utilization of resources in many-accelerator architectures. This paper expands on preliminary work [Lyons et al. 2010] with a deeper examination of accelerator memory characteristics, the addition of a scalable distributed architecture, a discussion of the AS/software interface, and additional evaluations of energy consumption and a new server workload.

To successfully design the AS and efficiently support many-accelerator systems, we must first develop a deep understanding of the accelerators. Section 2 surveys eleven commercial and open source accelerators, revealing that generic SRAM memories consume 40 to 90% of the area in each accelerator. This study categorizes these private accelerator memories into four categories based on capacity, bandwidth requirements, and latency sensitivity. We find that in many cases large private SRAM memories embedded within accelerators also tend to have modest bandwidth and latency requirements. Sharing these SRAM memories between accelerators would reduce the amount of on-chip memory through amortization, thereby shrinking total processor area. This area reduction can be used for cost savings or to place additional accelerators for even greater diversity and performance improvements. A well-architected accelerator store along with careful selection of shared SRAM memories leads to very little overhead compared to private-memory based systems.

To efficiently share memory between accelerators and optimize many-accelerator systems, Section 3 presents the accelerator store's design. The accelerator store supports ways to combine multi-core and customized logic as illustrated in Figure 1.

—*Support for shared SRAM memories.* The AS allocates memory to accelerators on an as-needed basis.



(a) Traditional multi-core (b) Traditional SoC (c) Many-accelerator

Fig. 1. Comparison of architecture styles.

—*Centralized model/decentralized scalability.* The AS presents a centralized view of shared accelerator memories to keep interactions with software and accelerators simple, but is physically distributed. A many-accelerator system can use multiple ASes, with a group of accelerators clustering around each AS as shown in Figure 1(c). The multi-AS design can scale to hundreds of accelerators.
—*Fine-grained accelerator support.* The AS's decentralized implementation supports accelerators in greater numbers than the handful found in today's SoCs. Many-accelerator systems may not just add new accelerators, but also decompose accelerators into multiple fine-grained accelerators. These finer-grained accelerators implement more commonly used algorithms and are less application specific.
—*Flexible abstraction.* The accelerator store's refined accelerator and software interfaces minimize overheads and open many research avenues.

Section 4 evaluates an RTL implementation of the accelerator store in multiple systems, demonstrating that area reduces by 30% while maintaining customized logic's superior performance and energy. We show that performance and energy overheads due to added memory latency and contention are minor. Section 5 presents related work, and Section 6 concludes the paper and presents future research directions.

2. ACCELERATOR CHARACTERIZATION

To design an efficient many-accelerator architecture, we must first analyze the accelerators that comprise many-accelerator systems. We begin with an analysis of several commercial and open-source accelerators and find generic SRAM memories consume between 40% to 90% of each accelerator's area. SRAMs are the biggest consumer of accelerator area, but amortizing memory through sharing will greatly reduce this cost. Some accelerator memories are better shared than others, so we characterize memory access patterns in four accelerators and analyze these memories based on capacity, bandwidth, and sensitivity to memory latency. This analysis results in a simple methodology to select which accelerator memories to share and which memories should remain private.

2.1. Accelerator Composition Characterization

We first analyze the composition of several open-source and commercially developed accelerators when synthesized for ASIC fabrication. These accelerators implement algorithms from several widely used domains, including security, media, networking, and graphics. We could not directly obtain composition data with a standard ASIC toolchain because many accelerators utilized FPGA-specific logic blocks. Instead, we synthesized each accelerator using the Xilinx ISE 10.1 FPGA synthesis toolchain to obtain FPGA memory and logic statistics. We then applied previously measured scaling factors [Kuon and Rose 2006] to obtain ASIC composition figures. The analysis shown

Table I. Accelerators Studied

Accelerator	Function	Area used by SRAM memory
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 convolutional coding	89.2%
CAN	Automotive bus	70.0%
DVB FEC Encoder	Video broadcast error correction	81.7%
Average		69.0%

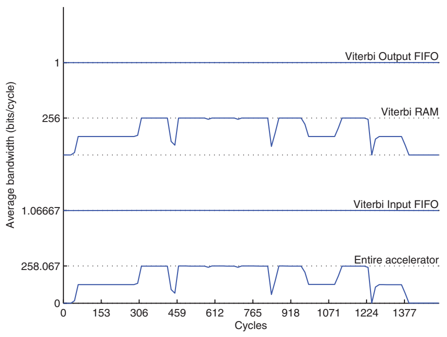
in Table I demonstrates that SRAM memories consume an overwhelming amount of area in all accelerators, up to 90%.

Memory is therefore the best target for area optimization. We use the large numbers of accelerators in the many-accelerator system to our advantage and reduce area by sharing memories between accelerators. At any given time, some accelerators will be actively processing and turned on, while most accelerators will be dormant and VDD-gated off. These off accelerators continue consuming ASIC chip area without providing any function to the system. Unlike private accelerator memories that must be provisioned at circuit design time, shared accelerator memory can be dynamically assigned to accelerators at runtime. Via sharing, memory can be effectively provisioned to accelerators when needed without the power, performance, and cost overheads of reconfigurable logic. This approach reduces the area cost of accelerator memory from the sum of each accelerator's memories to the much lower sum of memories used by accelerators at any point in time.

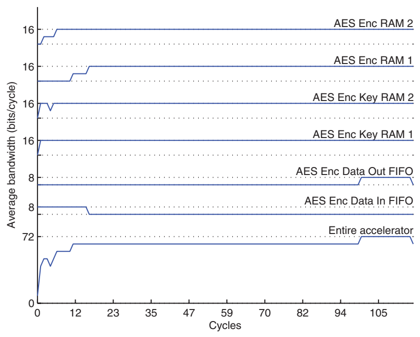
Shared memory also creates new memory reductions by eliminating over-provisioning and by merging redundant memories. Accelerator designers do not always know how their accelerators will be used and over provision memory to add flexibility. For example, a 1024 point FFT may also support 256 and 512 point FFTs. Although the core computation between any of these FFTs is the same butterfly operation, the 1024 point memory requires 4x of the memory used by the 256 point FFT. In another light, using the over-provisioned 1024 point FFT to perform a 256 point FFT wastes 75% of the FFT's memory. Although a smaller FFT would be more efficient in this case, the processor may be used for other applications requiring a larger FFT or the application may switch between FFT configurations. Memory sharing eliminates over-provisioning by assigning memory as required by the accelerator at runtime.

Merging memories with memory sharing reduces area costs and redundant transfers. Most accelerators contain memories for storing input and output data. Outputs from one accelerator often become inputs for another accelerator, and output memory and input memory store the same data. Sharing can merge these two memories into one shared memory, reducing memory area in half. Memory merging also reduces data transfers because copies between the two unshared memories are no longer required.

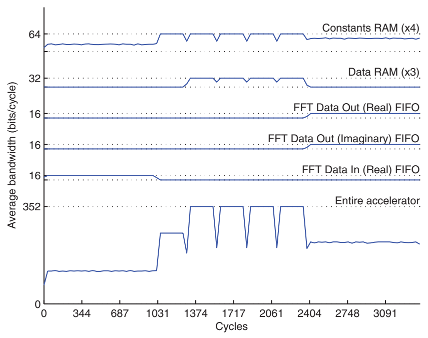
During early memory sharing experiments, we initially shared all private accelerator memories. We quickly found that our indiscriminate approach saved a large amount of chip area but incurred significant performance overheads. A small portion of accelerator SRAM memories were not amenable to sharing due to extremely high bandwidth requirements or sensitivity to memory latency. We characterized SRAM memories from four accelerators to identify memories that can be shared effectively and memories



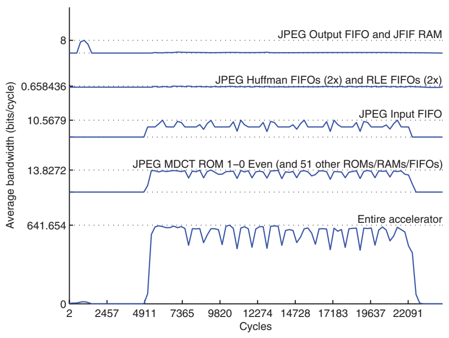
(a) Viterbi decoder



(b) AES-128 encrypter



(c) 1024 point, 16-bit FFT



(d) JPEG encoder, one horizontal line

Fig. 2. Total memory bandwidth utilization of several accelerators. Plots indicate one round of computation. Labels on the Y-axis indicate maximum binned bandwidth for each memory.

that should remain private for significant area reductions and minimal performance overheads.

2.2. Memory Access Pattern Characterization

To determine which attributes result in poorly performing shared memories, we characterized the access patterns of each memory using RTL for the first four accelerators in Table I (Viterbi, AES, FFT, and JPEG). Because the function and design of these four accelerators vary significantly, we ensure that the AS will provide high performance shared memory for accelerators in all domains.

We first instrumented each accelerator's RTL to record every memory access. We then executed test workloads for each accelerator to determine the average bandwidth, maximum bandwidth, and bandwidth variation for every memory over full workloads. Bandwidth use for each memory is shown in Figure 2.

We also analyzed each memory's degree of dependency, a measure of latency sensitivity. Sharing memories adds more logic and latency into each memory access. It is therefore important to ensure shared memories are latency insensitive. A memory is latency sensitive and highly dependent if the results of previous memory accesses contain information necessary to perform the next access. Highly dependent memories with long access latencies exhibit poor performance because logic must stall while waiting for memory accesses to complete. A memory used for linked list traversal exhibits high dependency because the first node must be completely read from memory to determine the memory address of the next node to read. Conversely, memory accesses from non-dependent memories do not depend on previous accesses to complete and can be planned in advance. Therefore, non-dependent memories are insensitive to increased memory latency and amenable to memory sharing. FIFOs are inherently non-dependent because access order is fixed by design (first in, first out). Many random access memories are non-dependent or can be designed non-dependent by pipelining memory accesses.

We use capacity as well as the above dependency and bandwidth characteristics to place each accelerator memory into one of four categories: inter-accelerator FIFO, intra-accelerator FIFO, large internal RAM, and small lookup table.

Inter-accelerator FIFOs are used to load data into the accelerator for processing or load data out after processing. Accelerators typically communicate with the system over shared busses, which are subject to arbitration delays for every transfer. Larger transmissions are more efficient: Reducing the number of transmissions by increasing transmission size results in fewer arbitration delays. These large transmissions are buffered in the inter-accelerator memories, so these memories must be large enough to store the large transmissions. Inter-accelerator memories are typically sized in the kilobytes, and we assume these memories are 2 KB for accelerators that do not define inter-accelerator memory size.

Inter-accelerator FIFOs exhibit bursty bandwidths, using their full bandwidth at the beginning or end of an operation when data is streamed in or out. These memories spend many cycles completely dormant after transfers until the next batch of data is ready. This behavior is shown for every accelerator's input and output FIFOs in Figure 2.

Inter-accelerator FIFOs do not introduce additional memory dependencies and can be easily pipelined. These FIFOs do not need to follow a strict schedule, provided input FIFOs are filled before the next operation begins and output FIFOs are unloaded before the next operation ends. This timing can be relaxed further if the FIFOs are sized to handle datasets for multiple operations. Inter-accelerator FIFOs make ideal candidates for memory sharing. They consume large amounts of accelerator area, so sharing these memories results in significant area savings. Inter-accelerator FIFOs have relatively light bandwidth requirements. These memories are insensitive to the increases in access latency that memory sharing would introduce because they do not require strict schedules. Sharing inter-accelerator FIFOs also adds the unique advantage of merging input and output FIFOs between accelerators. Intra-accelerator FIFOs feature attributes similar to inter-accelerator FIFOs, though not to the same magnitude. Some accelerators consist of several stages, each resembling a small accelerator. Intra-accelerator FIFOs are used to connect these stages and build up larger accelerators consisting of several steps. For example, the JPEG accelerator uses FIFOs to connect Huffman encoding, run-length encoding, and other stages. Accelerators use FIFOs rather than direct connections so stages can be designed independently and to ease timing complexities between stages. Intra-accelerator FIFOs are therefore resilient to memory access latencies and non-dependent. Unlike inter-accelerator FIFOs, intra-accelerator FIFOs are not all large and only some will be worth sharing. As shown in Figure 2, bandwidth variations are much smaller in intra-accelerator FIFOs and result in fewer idle periods. Some accelerators require large amounts of bandwidth, and others need only a small trickle (Huffman encoder, RLE). The low bandwidth memories are also the largest, resulting in the greatest area savings and lowest performance impact.

Deciding to share intra-accelerator FIFOs is a more subjective choice than for inter-accelerator FIFOs. Both FIFO types are non-dependent, but intra-accelerator FIFOs may be too small or require too much bandwidth to make sharing worthwhile. Therefore, a bandwidth- and capacity-based analysis is necessary before considering intra-accelerator FIFOs for sharing.

The third memory type we identify is large internal RAMs. These memories are typically sized in the kilobytes and are often used for values that rarely change. For example, the FFT stores constant coefficients for its butterfly operation in four 2 KB SRAMs as seen in Figure 2(c). Other memories are used as internal buffers for values that must be written out of order (JPEG JFIF RAM, FFT data RAM). In most cases these memories are non-dependent because access addresses are predictable, and requests to these memories are easy to pipeline. Bandwidth requirements for these memories also tend to be low and bursty. Internal RAMs are large, usually exhibit low or no dependence, and require little bandwidth. Therefore, large internal RAMs are well suited for memory sharing.

The final memory we consider is small RAM and ROM lookup tables (LUTs). These are the smallest memories, usually about 100 bytes. These memories typically require high bandwidths and are often used to determine control flow, making pipelining difficult. Sharing small RAM/ROM LUTs provides little benefit due to their small size and performance overheads.

2.3. Shared Memory Selection Methodolgy

Ultimately, the selection of which memories to share should maximize shared area and minimize performance overheads. Large, low-bandwidth, and non-dependent memories are ideal for sharing. At first glance, inter-accelerator FIFOs, large internal RAMs, and some intra-accelerator FIFOs are the best candidates for sharing. To formally decide which accelerator memories to share, we suggest the following methodology.

(1) Calculate the average bandwidth of each memory for all accelerators, assuming the accelerator is always processing (no idle periods).
(2) Calculate and sort each memory by memory size/bandwidth. This figure provides a balance between maximizing memory sharing and minimizing performance overheads due to contention.
(3) Pick memories with the largest memory size/bandwidth and weed out highly dependent memories. These memories strike a balance between area savings and low performance overheads.

This methodology balances area savings and performance overheads effectively. This approach is also application independent, since we assume each accelerator is always active and processing at maximum ability. Optimizing for each accelerator's worst-case bandwidth requirements prevents the system from incurring prohibitive performance overheads regardless of the applications running on the processor. We demonstrate this methodology's success for multiple applications in Section 4.

When we apply the above methodology to the memories contained by the four characterized accelerators, we find that inter-accelerator FIFOs, large internal RAMs, and the larger intra-accelerator FIFOs are the most shareable. As Figure 3 shows, larger memories also tend to use less bandwidth. The figure is sorted from most sharable (large size, low bandwidth) on the left to least sharable (small size, high bandwidth) on the right. Although the correspondence is not absolute, we see that the memories providing the biggest sharing benefits also result in the lowest bandwidth demands. This factor and our previous analysis showing the largest memories are the least dependent means that we can obtain near-maximum memory sharing with minor performance overheads.

In this section, we characterized several accelerators and demonstrated that memory sharing provides a significant opportunity to slash on-chip area in many-accelerator

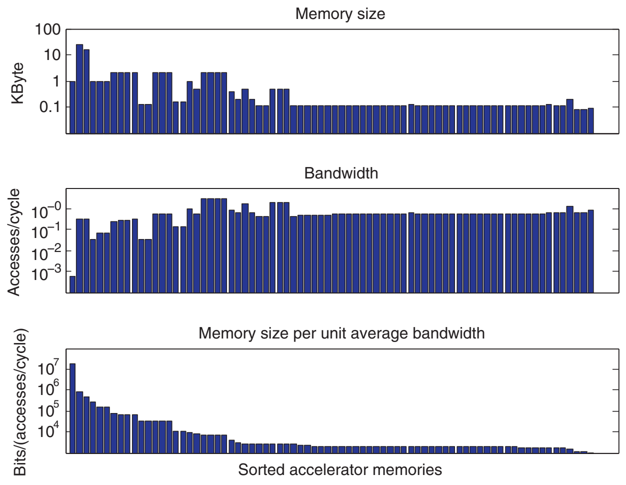


Fig. 3. Accelerator SRAM memories sorted by memory size per bandwidth.

systems. We also introduced a methodology for selecting shared memories that results in near-maximum area savings and minor performance overheads. In the following section we describe our architectural framework for sharing memories.

3. ACCELERATOR STORE DESIGN

This section describes the accelerator store (AS), our architectural framework to support memory sharing in many-accelerator systems. The accelerator store contains SRAMs for accelerators to store internal state or to communicate with other accelerators. The accelerator store manages how this shared memory is allocated and provides accelerators with access to shared memory. In this section, we describe AS features, the AS's architecture, using multiple ASes for scalability, the accelerator/AS interface, and the software/AS interface.

3.1. Accelerator Store Features

Handles are integral to the accelerator store's ability to share memory with accelerators. A handle represents a shared memory stored in the accelerator store, similar to the way a file handle represents a file in the C programming language or a virtual address space represents a region of memory used by an application. Each accelerator store can contain several shared memories and uses handles to keep track of these memories. To create a shared memory in the accelerator store, the system adds a handle with the shared memory's configuration and a corresponding handle ID (HID) is returned. The system passes this handle ID to an accelerator, giving the accelerator access to the shared memory. Accelerators can use multiple handles if more than one shared memory is needed by retaining HIDs for each handle. The system can pass the HID to multiple accelerators, so the accelerators can exchange data through shared memory. Accelerators include the HID when sending access requests to the accelerator store to access shared memory. When shared memory is no longer needed, the system can remove it by deleting its corresponding handle from the accelerator store and informing accelerators that the matching HID is invalid. Specifics about how the accelerator store manages handles are given in Section 3.2, and details about how system software can configure handles are discussed in Section 3.5.

Support for handles also allows the accelerator store to emulate multiple types of memories.

—*Random access (RA)* memories allow accelerators to read or write data at any location in the shared memory. RA memories are useful for representing shared internal RAMs.
—*FIFOs* can only put data in or get data out, in first in/first out order. FIFOs are useful for representing shared inter- or intra-accelerator queues. Support for FIFOs provides a simpler alternative to DMA controllers for exchanging data between accelerators.
—*Hybrid* memories combine RA and FIFO types, and support reads, writes, puts, and gets. Hybrid addressing maps address 0 to the head value (next value out) and addresses increase moving toward the tail value. Unlike puts or gets, hybrid reads and writes do not add or remove values. Hybrid memories are useful for operations that stream in blocks of data that are used out of order.

The accelerator store keeps leakage power low by VDD-gating unused SRAMs. Initially, all physical memory in the accelerator store is turned off, keeping leakage to a minimum. SRAM memory must be kept on to retain data, so a memory is turned on once it holds any value. Unfortunately, current SRAMs must be completely on or off, and VDD-gating part of an SRAM is problematic. To keep leakage power low, each accelerator store contains several 2KB and 4KB SRAMs rather than one large SRAM. This design allows the accelerator store to save power by turning off SRAMs that do not contain valid data at a finer granularity. If the accelerator store used only one large SRAM, the entire memory would be forced to turn on, even if the SRAM only stored one word.

Once a word is written, shared random access memories must remain on until the memory's handle is removed from the accelerator store. The accelerator store VDD-gates shared FIFO and hybrid memory more aggressively because it can automatically identify memory that does not contain valid data. Once an accelerator gets a value from a shared FIFO, the accelerator store knows that value is no longer stored in the FIFO. Large FIFOs may span multiple SRAMs in the accelerator store, and the accelerator store will turn off any SRAMs allocated to a FIFO if the SRAMs do not contain valid FIFO data.

To prevent accelerators from accidentally destroying handles mapped to other accelerators, only the General-purpose CPU (GPCPU) can add or remove handles. In most cases, this scheme limits accelerators to modifying handles to which they are mapped. Of course, if GPCPU software incorrectly maps handles to accelerators or accelerators cannot be trusted to access the handles to which they are mapped, accelerators can make unauthorized handle accesses. If more security is necessary, a list of allowed accelerators can be added to each handle in the handle table, blocking accelerators from making unauthorized handle accesses.

3.2. Architecture of the Accelerator Store

The accelerator store depends on several elements as shown in Figure 4. A GPCPU first configures the accelerator store and accelerators over the system bus and AS management interface. Accelerators can then use their AS ports to send access requests to the AS's priority table. The priority table selects as many requests as there are channels to send them. The channels send these requests to the handle table which translates the request into a physical address for the AS's SRAMs. The channels transmit the physical address requests to the SRAMs and perform the memory access. The channels finally relay the access result back to the accelerator via the channel, priority table, and finally the AS port.

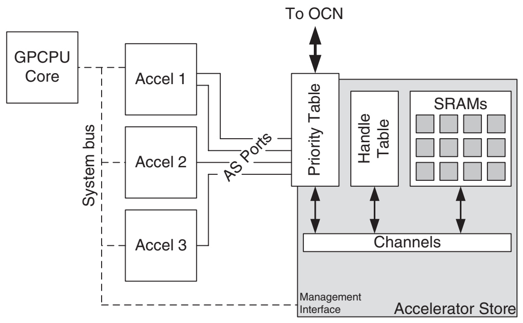


Fig. 4. Accelerator store design.

Table II. Handle Table Layout

HID	Alloc (y/n)	Type	Start Addr	Mask (Size)	Head Offset	Tail Offset	Full (y/n)	Trigger
0	Y	RAND	0x0600	0xFF00 (256)	X	X	X	X
1	Y	FIFO	0x0000	0xFC00 (1024)	0x0081	0x0081	Y	224
2	Y	FIFO	0x0400	0xFE00 (512)	0x0010	0x0004	N	0 (off)
3	N	X	X	X	X	X	X	X
...								

The accelerator store relies on many small SRAMs to increase VDD-gating opportunities as previously described. Measurements of SRAMs fabricated in a commercial 130nm process revealed that 2KB and 4KB SRAMs provided the best balance between VDD-gating granularity and arbitration overheads.

Handle Table. The handle table maintains each shared memory in the AS by storing the handles for each active shared memory (Table II). The handle table stores sixteen handles by default, although this number can be changed if the accelerator store is expected to simultaneously share more than sixteen memories. Each handle includes several pieces of information, including the shared memory's size, which SRAMs store the shared memory, as well as FIFO-specific settings. After the system configures handles for each shared memory, the handle table's primary function is to translate access requests from accelerators to the AS's SRAMs. For RA shared memory requests that contain a handle ID and offset address, the handle table obtains the SRAM physical address by looking up the handle's starting address and adding it to the request's offset address. This operation is similar to the virtual to physical address mapping performed by MMUs. The translation for FIFO and hybrid shared memories is similar, but uses the head offset for gets and the tail offset for puts.

The handle table also enables the already discussed SRAM VDD-gating. An SRAM can be VDD-gated off if no handles are mapped to the SRAM, the SRAM contains an untouched RA memory, or maps the SRAM to a FIFO but does not contain valid data. The handle table continually monitors changes to each handle, waiting for one of the above conditions to be true for each SRAM. If so, the handle table signals the relevant SRAMs to turn off and keeps SRAM leakage power to a minimum.

The handle table's trigger allows workloads to be batched, in order to maximize VDD-gating energy savings and to reduce AS contention. Each FIFO handle maintains a trigger value; when the number of elements in the FIFO reaches this value, the AS raises an interrupt in the GPCPU. The GPCPU can then turn on an accelerator to consume the FIFO's data. Setting the trigger value close to the size of a handle's SRAM can reduce leakage power and contention. Rather than turning all accelerators on, which results in underutilized SRAMs and high competition for AS resources, the trigger allows accelerators to only turn on when a batch of workloads is available, and without turning on additional SRAMs. In addition, fewer accelerators are on at any given time, resulting in fewer simultaneous AS requests, lowering contention and increasing performance.

Channels. All shared memory requests from accelerators are transmitted over channels. Each channel can carry one shared memory access per cycle, so ASes servicing accelerators with higher bandwidth needs should provision additional channels. However, additional channels require extra arbitration in the accelerator store and this extra logic will require additional area and power. In addition, additional channels will add more wires and increase dynamic power. If many channels are required, we instead suggest many-accelerator systems distribute multiple accelerator stores throughout the system as described in Section 3.3. In the distributed AS model, each accelerator store offers a few channels to provide a good balance between performance and overheads, which we analyze in Section 4.

Priority Table. The priority table controls which shared memory accesses are completed if too many accelerators contend for channels. Each accelerator has at least one port for communicating with the accelerator store and may have multiple ports for increased bandwidth. Each of these ports is identified by a port ID. The system configures the priority table by assigning a priority to each port ID, so that some ports have priority over others. Assuming the accelerator store has n channels, the priority table will select up to n memory requests from the ports with the highest priorities. This approach can be used to insure that time-sensitive operations take precedence over operations with flexible timing requirements. The priority table can be modified at runtime, so round-robin and other arbitration schemes can be implemented in software.

Management Interface. The accelerator store also features a management interface accessible over the system bus, allowing the system to configure the handle table and priority table dynamically at runtime. The management interface is memory mapped, so software can use memory load and store instructions to add or remove shared memories in the handle table or modify arbitration settings in the priority table.

3.3. Distributed Accelerator Store Architecture

We expect the number of accelerators in many-accelerator systems to reach tens or hundreds, and we expect the accelerator store to grow accordingly. The number of channels and ports required to support hundreds of accelerators will not scale within a single acclerator store; instead, we propose designing systems with multiple accelerator stores, as shown in Figure 1(c). Each accelerator's port will be directly connected to a single AS, and it will primarily use this AS to keep access latencies low. As a result, the system topology will consist of several clusters of accelerators, each cluster surrounding a single AS.

An accelerator may need to communicate with ASes outside of its cluster in some cases. This may happen if the accelerator's primary AS is fully allocated or if the accelerator must communicate with an accelerator tied to a different primary AS. In these cases, the accelerator will utilize an on-chip network (OCN), allowing accelerator stores to communicate directly. OCNs have been well studied in the network on-chip (NoC) community [Benini 2002; Pratim 2005] and we do not attempt to duplicate their work here. Several OCN topologies can be used to connect the distributed accelerator stores, but we anticipate that a grid topology will result in the best scalability.

Although OCNs will introduce additional latency, they will not result in significant performance overheads. We only share latency insensitive memories as discussed in Section 2, so the additional OCN latency will not cause stalls or noticeably affect performance. Further, we anticipate that accelerators will use their primary AS most of the time and will occasionally use other ASes when communicating with accelerators in other clusters. Today's SoCs use high-latency DMA transfers to exchange data between accelerators, so the OCN will not introduce any new high-latency accesses.

Communicating with distributed ASes is simple from the accelerator's viewpoint. To each accelerator, the system contains one AS. We achieve this abstraction by preassigning handle IDs to each AS at circuit design time. For example, the first AS would contain HIDs 0 through 15, the second AS would contain HIDs 16 through 31, and so on. To access a shared memory in the first AS, an accelerator would simply use a HID from 0 to 15, regardless of the accelerator's primary AS. Each accelerator store maps HIDs to ASes and will forward access requests over the OCN if the request does not match the primary AS's HIDs. Therefore, the software compiler or dynamic allocation libraries are responsible for assigning AS handles to accelerators in the same cluster whenever possible.

3.4. Accelerator/Accelerator Store Interface

Each accelerator communicates with the accelerator store over ports as previously mentioned. Accelerators may contain one port or add additional ports to increase bandwidth.

Each port carries three types of messages. First, the accelerator store sends access requests to the accelerator store. An access request contains the type of request (read, write, get, put) and a HID. If the access is a read or a write, the request will also contain an address offset. The accelerator store may not be able to satisfy a request in certain cases and will send the accelerator an access reply. For example, an accelerator may have tried to do a FIFO get on an empty FIFO. If such an error occurs, the AS will send an access reply with an error code describing the problem. Finally, the AS sends an access response back to the accelerator when the access completes. If the access was a read or a get, the access response will contain the accessed data. If the access is a write or a put, the access response simply indicates the access completed.

We designed the accelerator store to efficiently support memory sharing in a many-accelerator system. The AS is the result of many design choices necessary to keep overheads low and minimize complexity for accelerator designers. However, some changes within accelerators are necessary to use AS memories. Accelerator designers will need to decide which memories are worth sharing, as shown in Section 2. Sharing FIFOs should be a simple process, since FIFOs are already designed to be latency insensitive. Sharing large RAMs will require memory access logic to be pipelined, which introduces small design complexities in most cases. Pipelining these memory requests is much simpler than pipelining in CPU datapaths, since these accesses cannot cause branch mispredictions or pipeline flushes.

3.5. Accelerator Store Software Interface

Software is a critical element in any computer system and the many-accelerator architecture is no different. The accelerator store fully exposes the handle table and priority table so that more complex shared memory allocation and scheduling schemes can be built up without complex hardware additions. Moreover, we introduce a bridge accelerator that allows software to modify the contents of shared memory in the accelerator store. The accelerator store's approach to software creates new research opportunities in system software, which we now identify.

The handle table allows an open approach toward shared memory allocation by completely exposing the table to software. Currently, the application software developer manually allocates shared memory to accelerators, creating handles and mapping their HIDs to accelerators whenever needed. In the future, we believe the number of accelerators in many-accelerator systems will scale upward, and manual allocation will become too complex. Instead, we believe software compilers and dynamic memory allocation software libraries will solve this problem.

Compilers can improve memory allocation using an automated static allocation system. Metadata can be written into software so that the compiler knows each accelerator's shared memory requirements. For example, the AES encryptor requires two 2KB FIFOs to operate. Instead of forcing the software designer to allocate these two FIFOs in the AS's handle table, the compiler could do this automatically through macros or programming language primitives.

Ultimately, dynamic memory allocation will provide the most robust approach to allocate shared memories. Similar to malloc(), we anticipate that software libraries can keep track of the contents of each accelerator store and allocate shared memory on demand at runtime. This approach uses shared memory more efficiently, because the system will have a better understanding of what memory is available at runtime.

The priority table's full software accessibility can enable more complex scheduling schemes as well. By default, the priority table arbitrates by always picking certain ports over others. To implement a round-robin scheme, the priority table is rotated periodically, moving the lowest priority port to the highest priority slot and moving the other ports down one slot. This gives each accelerator equal time as the top priority slot and eliminates any contention-related starvation. An real-time scheduler could also be implemented by coordinating the priority table with software. If a certain accelerator needs guaranteed access to the accelerator store for a fixed period of time, software could place that accelerator's port at the highest priority for that period of time. Although the priority table's initial configuration is simple, software can add more complex and robust schedulers.

The many-accelerator system contains one or more GPCPUs, which may wish to access shared memories in the AS. To enable this, we introduce the bridge accelerator which bridges the GPCPU's system bus with the AS. The bridge accelerator is memory mapped on the system bus, enabling the GPCPU to access shared memory in the accelerator store by writing commands over the bus. The bridge accelerator will replay these access requests to the AS. From the AS's viewpoint, the bridge accelerator is no different from any other accelerator and can be prioritized in the priority table. When the access completes, the accelerator store will return the access response, and the bridge will relay the result back to the GPCPU.

The bridge scheme seperates AS memory from the GPCPU's system memory. Future systems could integrate these SRAM memories in the same address space, but doing so is beyond the scope of this paper. Such a change would require managing simultaneous accesses to the same SRAM from both accelerators and the GPCPU, as well as complications to SRAM VDD-gating and caching.

The accelerator store is carefully designed to make sharing memory as simple as possible and keep performance and energy overheads minimal. In the following section we evaluate the accelerator store's ability to satisfy these goals.

4. ACCELERATOR STORE EVALUATION

This section evaluates the accelerator store's ability to reduce area while keeping performance and power overheads low for two applications. The first application is representative of embedded systems and utilizes several accelerators processing a highly serial dataflow. An alternative application representing a server workload utilizes several JPEG encoders operating in parallel.We first present the implementation of our accelerator-based model before evaluating the benefits and overheads of sharing accelerator memories.

4.1. Accelerator-Based System Model

We model six accelerators in our application: AES, JPEG, FFT, a digital camera, an ADC, and a flash memory interface. The first three accelerators are derived from real accelerators obtained from OpenCores and Xilinx in Table I. We instrumented each SRAM memory in the accelerator to record every read, write, put, or get at every cycle. The resulting traces allow us to simulate the accelerator access patterns with contention and latency by playing back the memory access traces in order and maintaining the accelerator's timing between accesses. We first optimistically assume that the accelerator store can satisfy accesses in a single cycle.

After we optimize the accelerator store configuration to minimize contention, we demonstrate the system's access latency tolerance by increasing the accelerator store's access latency up to 50 cycles. Although we assume that each memory inside the accelerator is non-dependent as described in Section 2, we still treat accesses between accelerators to be dependent. As an example, consider the case where accelerator A wishes to send data to accelerator B. Accelerator A must first completely write its data to shared memory before accelerator B can attempt to read it. To ensure this dependency, we stall accelerator B until the write completes.

RTL implementations for the remaining three accelerators were not available, so we manually generated memory access traces via known device characteristics. The camera is modeled after a synthetic camera found in the JPEG testbench, the ADC will write one 16-bit word at 44.1 KHz, and the flash accelerator writes data in 2 KB pages at 6 Mbps, typical settings for an MMC/SD flash card SD Association.[1]

We have implemented a per-channel implementation of the accelerator store in Verilog RTL. The design was synthesized with Design Compiler and placed-and-routed in Encounter for a commercial 130nm process. The AS requires an area overhead of 80,000 μm^2, equivalent to 1% of the area used by the six accelerators from the embedded application.

Our simulation model consists of three stages. Scheduling is done first to decide when each accelerator will begin and complete an operation. This step is done without considering contention, as if the accelerators do not share memory. Memory access overlay is performed second, to copy the memory access traces obtained from real accelerators into the schedule. This step is also completed as if no memory sharing is possible. Stalling replay, the final step, considers contention due to shared memory. The access logs are replayed for each accelerator and accelerators may stall if more requests are made than accelerator store channels are available. Each accelerator tracks its own time in addition to the system time; accelerators may only increment their cycle count if all pending accelerator store requests have been satisfied. Accelerators are also able to fast forward past cycles in which the accelerator would be turned off. This feature models the situation where an accelerator stalled for a few cycles, but finished its operation and can catch up to the rest of the system.

We are unable to evaluate systems with multiple accelerator stores due to the difficulty of obtaining a large number of distinct accelerators. Instead, we demonstrate one accelerator store/accelerator cluster, and artificially increase memory access latency to simulate the additional latency the on-chip network would add.

4.2. Embedded Application

The first application is designed to demonstrate a typical workload for a mobile, embedded device. This application combines the six accelerators described previously to implement a security monitoring system. The ADC samples a microphone at 44.1 KHz, enough to accurately capture frequencies audible to the human ear. These audio samples are processed by the FFT (1024 point, 16-bit, radix-4) every 1024 samples. The resulting frequency response is checked by the GPCPU to look for a specific frequency, such as a car horn or glass breaking. If such an event is detected, a camera will take a

[1]http://www.sdcard.org/developers/overview/speed_class/.

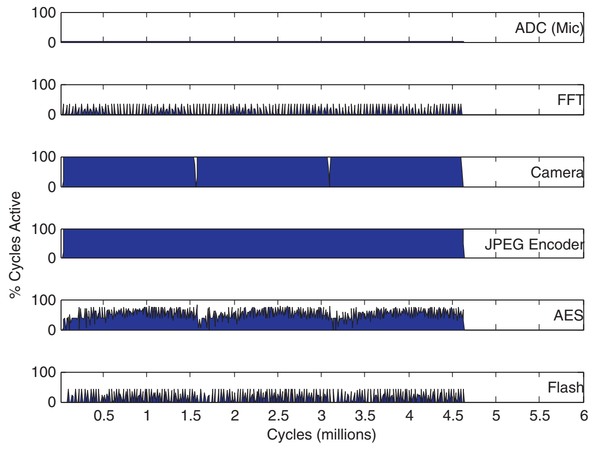


Fig. 5. Embedded application accelerator activity.

picture once per second. The resulting image is compressed via the JPEG accelerator, then encrypted by the AES accelerator. The encrypted JPEG photos are finally stored in the flash accelerator.

4.2.1. Embedded Application Performance. To model the most contentious workload possible, we reduced the frequency so all accelerators can perform their tasks while meeting the timing guarantees described above. The JPEG accelerator was the limiting accelerator, so we ran the clock frequency just fast enough to compress one 640x480 photograph per second (1.53 MHz). Although this clock frequency may seem slow, recall that accelerators can compute far more per cycle than a corresponding GPCPU. Each accelerator was checked for activity at every cycle and put into one of 100 bins. Figure 5 demonstrates how often each accelerator was actively performing work during each of the cycle bins. The JPEG, AES, and camera accelerators are active all or most of the time, and the FFT and flash memory are moderately active. The ADC samples rarely by comparison.

Contention can be kept low despite utilizing six accelerators simultaneously (many at a high duty cycle). Figure 6 shows the twenty most shareable accelerator memories (large memory, low bandwidth, non-dependent) as sorted in Figure 3. The most shareable memories are drawn bottom-to-top, and results are grouped into 100 bins and averaged. This figure demonstrates that most of the memories selected by the memory size/bandwidth metric require low bandwidth, and the remaining few are quite large (JPEG input FIFO, camera output FIFO). We can also see that including at least the top 15 memories would result in low bandwidth contention (we would only need two channels) and a large percentage of accelerator memory sharing (76%). Up to roughly 25 memories can be shared before performance becomes unacceptable for any number of channels, verifying our memory selection methodology from Section 2.3.

Although it would not be feasible to fabricate a chip specifically for this embedded application due to design costs, we wanted to gauge how effective our sorting and memory selection algorithm would be if performed for our application rather than our application-blind approach. Even if we sort memories based on their application bandwidth needs and chose these memories based on application contention plots, we saw a negligible reduction in contention and only 2% improvement in memory savings.

The number of accelerator store channels (number of simultaneous accesses) to provision is highly dependent on the processor's design goals. As Figure 7 shows, additional

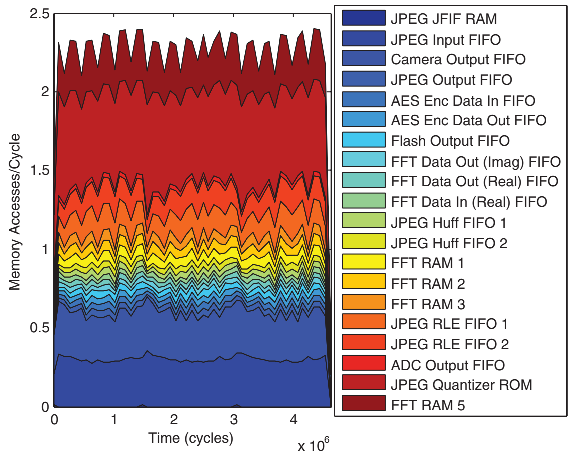


Fig. 6. Embedded application "top 20 to share" memory bandwidth.

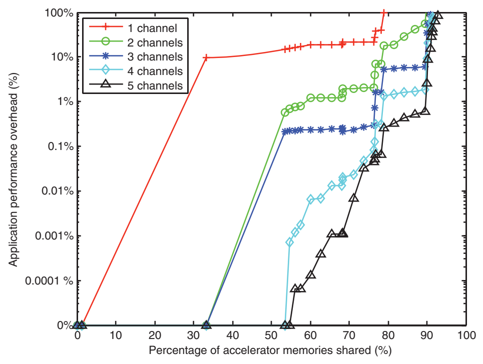


Fig. 7. Embedded application contention performance overhead.

channels can significantly alleviate performance overheads. Most applications will add additional channels because the area overhead of additional channels is low. Each additional channel requires only 1% of the total system area. If fewer channels are used, the processor will need to operate at an increased frequency (and voltage), resulting in a cubic increase in power. For most systems, this power increase would be unacceptable, though some embedded sensing applications with low duty cycles may not have these performance concerns. For this reason, we would recommending provisioning at least three channels for all but the simplest processors.

4.2.2. Distributed Embedded Application Performance. To test the system's latency sensitivity introduced by the distributed AS OCN, we increased the latency of each AS access up through 50-cycles. Because far fewer distinct accelerators were available than the number we expect to find in a many-accelerator system, we use a 50-cycle latency to reflect unusually high delays to model such a many-accelerator system. The 50-cycle latency incorporates delays due to inter-cluster network routing and contention, delays turning on VDD-gated SRAMs, and delays from AS arbitration.

To demonstrate why 50 cycles represents an unusually high delay in a many accelerator system, we consider a system consisting of 25 AS clusters, shown in Figure 8. The clusters are connected in a 5x5 network-on-chip (NoC) grid. Assuming each cluster

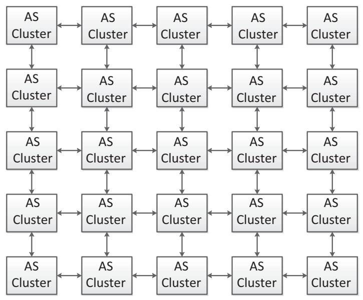


Fig. 8. Distributed AS. Each AS cluster in the example system contains an AS and several accelerators.

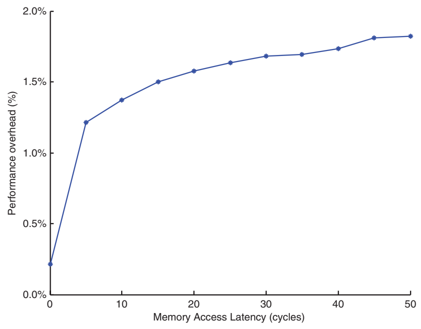


Fig. 9. Embedded application access latency and contention performance overhead.

resembles the single cluster embedded system previously modeled, each will contian roughly six active accelerators and several others turned off. Therefore, this example system contains 150 active accelerators and many more turned off.

A 50-cycle AS access latency is an unusually high delay in this example system. Assuming each hop between clusters requires a cycle, a delay found in commercially available processors [Gwennap 2011], the worst-case round-trip path requires 16 cycles. Using HSIM simulations of 2KB and 4KB SRAM HSPICE models, we measured worst-case SRAM power-on times of 20.5 ns or 21 cycles when operating at a 1 GHz clock frequency. Of the 50-cycle latency, the remaining 13 cycles are budgeted for AS arbitration (up to 2 cycles) and contention within the NoC. Note that we expect most AS accesses to require fewer cycles since most accesses will not travel the maximum path or require SRAMs to power on. Rather, the 50-cycle latency demonstrates the distributed accelerator store design performs well, adding up to a 2% performance overhead under unusual cases (Figure 9).

These low performance overheads are possible because of careful selection of shared memories, described in Section 2.3. If all accelerator SRAM memories were shared in the AS, performance overhead would likely exceed 100% due to data dependency delays. Instead, we select memories with preset access patterns to share in the AS, such as I/O buffers and FIFOs. These memories are the majority of accelerator SRAM area, and their access patterns are known in advance and can be pipelined. As a result, the performance overhead for accessing these memories remains below 2%, even for many-accelerator systems.

4.2.3. Embedded Application Power. The accelerator store takes several steps to achieve a low-power overhead. To measure accelerator power as well as accelerator store overheads and benefits, we model several factors.

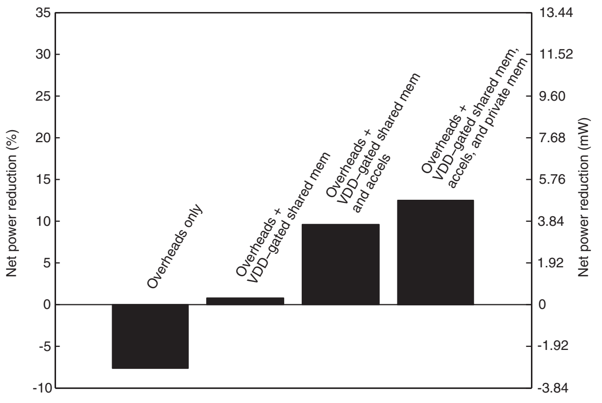


Fig. 10. Embedded app power breakdown.

—*Accelerator and AS power (active and leakage).* We estimate leakage power using area to power ratios derived from previously synthesized logic for the a commercial 130nm process.
—*Wire activity.* We conservatively assume all wires connecting accelerators to the accelerator store are the system's length (2.354 mm). Using known wire characteristics [Ho 2008], we estimate 0.49 pJ/bit/mm at 1.2 V operating voltage.
—*Automatic AS SRAM VDD-gating.* We assume VDD-gated memories consume negligible power.
—*Accelerator VDD-gating.* Sharing memory via the AS can make accelerators easier to VDD-gate as well.

The accelerator store was configured with three accelerator store ports and 15 shared memories as suggested in Section 2. The embedded application is executed at a maximum workload for roughly three million cycles.

The accelerator store is designed to keep power costs low. As shown in Figure 10, the overheads of the accelerator store add an additional 8% to the total system power cost. The majority of this overhead (5.24%) is incurred by the accelerator store arbitration logic. Additional wire power (2.38%) and accelerator stalling (0.14%) use measurably less power.

The accelerator store is also able to reduce power consumption through aggressive VDD-gating. Although each accelerator could implement VDD-gating individually, we have never seen an accelerator that VDD-gates its memory or logic. The accelerator store makes VDD-gating SRAMs guaranteed and automatic. Shared memory automatic VDD-gating reduces power by 8.44% by VDD-gating the 24.78KB of SRAMs that are temporarily unused on average. This may make VDD-gating accelerator logic easier as well, since 76% of accelerator memory is shared in the accelerator store and no longer private. An additional 8.81% of power can be trimmed by gating accelerator logic and leaving the remaining private memories on, or 11.76% of system power can be saved by VDD-gating accelerator logic and private memories.

Leakage power is a growing concern as fabrication technologies continue to shrink. ARM has noted that low-leakage technologies that SoC designers relied on will no longer be the magic bullet when entering 45nm technologies, stating, "Whatever transistor is used, leakage management is a significant challenge that must be addressed," and noting 20% increases in performance will result in 1000% increases in leakage [Muller 2008]. Low-leakage processes also require more active power, and judicious use of VDD-gating may be a better alternative to low-leakage flavors by keeping

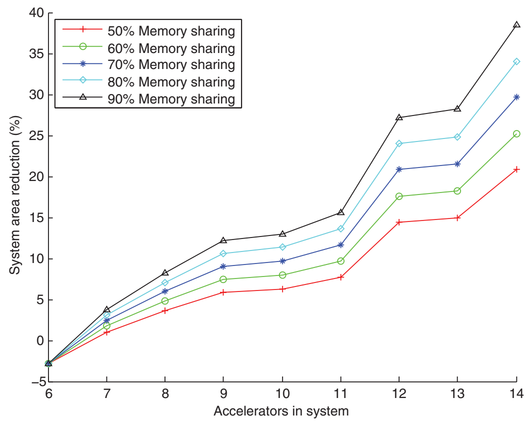


Fig. 11. Embedded app area reduction.

leakage and active power low. Therefore, the accelerator store's support for automatic VDD-gating will become more critical in the future.

4.2.4. Embedded Application Area. By following the guidelines proposed in this paper, the processor can share 76% of its accelerator memory and keep power low, all with a minor 2% impact on performance. Until now, we have only considered the accelerators in use. As Figure 11 demonstrates, memory sharing translates into significant memory savings. The figure starts with six accelerators and a three channel accelerator store, corresponding to the case where all accelerators and all memories are in use, including private and shared memories. In this case, there is no memory savings from unused memory amortization, and a small area overhead of 3% resulting from the accelerator store area overheads. If we consider a system with more dark accelerators, our area savings grow large quickly. We add unused accelerators to the system as arranged in Table I. If we assume 70% of accelerator memory can be shared on average, we can reduce total system area by 30% (this area includes SRAM memory, accelerator logic, and accelerator store logic overhead). Area savings will grow further if we include larger regions of dark accelerators.

4.2.5. Embedded Application Software Implementation. The embedded application is easy to implement in software using the accelerator store. The first step is to allocate memory for the accelerators as needed. This is done by modifying the handle table through memory mapped I/O. For example, a 4 KB FIFO should be allocated for the ADC to put audio samples and for the FFT to get samples for frequency analysis. Although each frequency analysis requires 2 KB of data, creating a 4 KB FIFO allows the ADC to record samples even when the FFT has not finished computation. Note that this single allocation is much easier than existing buffer designs, which would require statically sized buffers in the ADC and on the FFT, and routine DMA operations to transfer data between the two. Using the accelerator store, we could later decide to do a 512 point FFT and reduce our memory buffer size to 2 KB. This software-based memory resizing would not be possible under current designs, since private buffers must be statically allocated at circuit design time.

Second, we must configure the priority table through memory mapped I/O. Memory for recording audio samples and photographs receives a high priority to ensure sampling is performed at strict time intervals. Compression and encryption are less time sensitive, so memories used for these tasks receive lower priorities.

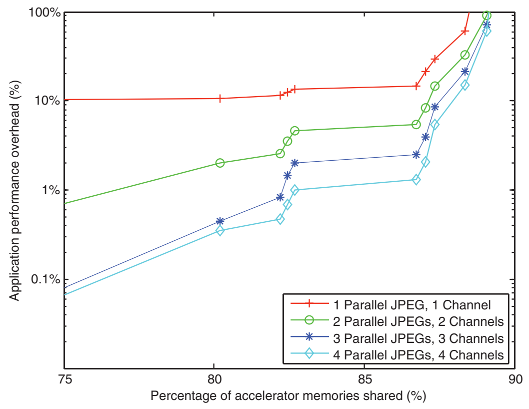


Fig. 12. JPEG server application performance overhead.

Third, software needs to initialize interrupts and implement interrupt handles for events occurring during accelerator operation. For example, the handle table should be configured to trigger an interrupt when the number of samples stored in the ADC FIFO reaches 1024, enough to perform an FFT. A short interrupt handler would turn the FFT on, pass the ADC FIFO's handle to the FFT (so it knows where to read samples from), and activate the frequency analysis operation.

Software design for accelerator-based tasks in the many-accelerator framework is simple. Most of the computation is performed by accelerators, so software takes on an accelerator management role.

4.3. Server Application

The sample application representing a server workload is highly parallelized, unlike the serial workflow found in the embedded application. These server application features multiple JPEG encoders compressing separate images in parallel. This application could be used by Facebook or other photo Web sites, which must compress or resize and recompress many pictures in parallel.

We sweep the level of parallelism from one JPEG encoder (no parallelism) to four JPEG encoders (4x parallelism) to examine the accelerator store's performance in a server-class application. Each JPEG's workload is staggered, started slightly after each other to prevent simultaneous bandwidth surges. As Figure 2(d) shows, the JPEG accelerator's bandwidth requirements are highly variable.

We anticipate the server application will require more channels than the embedded application due to the increased use of the JPEG encoder. The JPEG encoder is quite demanding and requires the most bandwidth of the investigated accelerators. Therefore, we provision one channel for every JPEG encoder operating in parallel. We anticipate processors designed for server workloads would provision more channels than processors for embedded applications.

Including more copies of the accelerator in parallel results in better performance, as shown in Figure 12. If we had one JPEG encoder with one port, the performance overhead would be 10% for any significant amount of memory sharing due to the JPEG's bandwidth demands. By staggering each JPEG's execution, each JPEG accelerator rarely demands maximum bandwidth at the same time and the increased bandwidth is amortized between the JPEGs. As a result, the 4x JPEG configuration is able to share more than 85% of its memory with less than 1% performance overhead. This result shows that it is best to include many accelerators under the accelerator store to amortize channels and improve performance.

5. RELATED WORK

As mobile computing has become pervasive, SoCs have been developed to provide general purpose and application-specific processing on one platform. In one example system, separate scratch pad memories are included within a security accelerator that must be managed by the application programmer or software library [Arora et al. 2006]. Brick and mortar assembly techniques could provide a low-cost approach to piece together accelerator-based architectures without requiring a new lithographic mask for each collection of accelerators [Kim et al. 2007]. This work does not address the need for dynamic allocation of memory for each accelerator, nor does it specify an interface to communicate between accelerators without copying data between accelerators. Programming environments for heterogeneous systems have been proposed which could be applied to the accelerator store [Linderman et al. 2008]. These library-based frameworks are used to map general purpose code down to graphics accelerators.

Supporting advanced memory management in hardware for accelerator-based architectures has been explored by a few groups, but without the full features provided by the accelerator store. An SoC dynamic memory management unit (SoCDMMU) has been proposed for multicore SoCs that provides support for malloc() in hardware [Shalan and Mooney 2002]. This system does not support advanced structures (FIFOs, hybrids), automatic VDD-gating, or handles. Smart memories features a general purpose microcontroller in multiple tiles that interact with memory [Mai et al. 2000]. Unlike smart memories, the accelerator store targets heterogeneous accelerated systems and includes support for automatic memory VDD-gating.

Sharing memory within functional units in general purpose processors has been investigated [Meixner and Sorin 2009]. This work optimizes shared memory between GPCPU blocks (L1, BTB, etc.) using two tiered caching. Disaggregated Memory shares memory between server blades [Lim et al. 2009]. In contrast, our work introduces novel features designed for accelerators including FIFO support, improved inter-accelerator communication by eliminating redundant copies, automatic VDD-gating, configurable priorities, and highly configurable memory management. Our work also demonstrates accelerator memory access patterns and the accelerator store's success for these needs.

6. CONCLUSION

This paper presented the many-accelerator architecture, a design approach combining the scalability of homogeneous multi-core and SoC's high performance and power-efficient hardware accelerators. In preparation for systems containing tens or hundreds of accelerators, we characterized a diverse pool of accelerators and found each contained a significant amount of SRAM memory (up to 90% of their area). This motivated us to create the accelerator store, a scalable architectural component to minimize accelerator area by sharing memories between accelerators. We evaluated the accelerator store for two distinct applications and found significant system area reductions (30%) in exchange for small overheads (2% performance, 8% to −8% energy). The paper also identified new research avenues including dynamic shared memory allocation, software assisted prioritization, and scheduling optimizations to further reduce area and improve performance.

REFERENCES

Arora, D., Raghunathan, A., Ravi, S., Sankaradass, M., Jha, N. K., and Chakradhar, S. T. 2006. Software architecture exploration for high-performance security processing on a multiprocessor mobile SoC. In *Proceedings of the ACM/IEEE 43rd Design Automation Conference*.

Benini. 2002. Networks on chips: A new SoC paradigm.

Gwennap, L. 2011. Adapteva: More flops, less watts. Tech. rep. The Linley Group.

Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B., Richardson, S., Kozyrakis, C., and Horowitz, M. 2010. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*.

Ho, R. 2008. High speed and low energy capacitively driven on-chip wires. *IEEE J. Solid-State Circuits*.

Kim, M. M., Mehrara, M., Oskin, M., Austin, T., Kim, M. M., Mehrara, M., Oskin, M., and Austin, T. 2007. Architectural implications of brick and mortar silicon manufacturing. *ACM SIGARCH Comp. Architect. News*.

Kuon, I. and Rose, J. 2006. Measuring the gap between FPGAs and ASICs. In *Proceedings of FPGA'06*.

Lim, K., Chang, J., and Mudge, T. 2009. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of ISCA'09*.

Linderman, M. D., Collins, J. D., Wang, H., and Meng, T. H. 2008. Merge: a programming model for heterogeneous multi-core systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*.

Lyons, M., Hempstead, M., Wei, G., and Brooks, D. 2010. The Accelerator Store framework for high-performance, low-power accelerator-based systems. *IEEE Comput. Architec. Lett. 9,* 2.

Mai, K. Paaske, T., Jayasena, N., Ho, R., Dally, W, J., and Horowitz, M. 2000. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of ISCA*.

Meixner, A. and Sorin, D. J. 2009. Unified microprocessor core storage. In *Proceedings of CF'07*.

Muller, M. 2008. Embedded processing at the heart of life and Style. In *Proceedings of the IEEE International Solid-State Circuits Conference.* (Digest of Technical Papers).

Pratim, P. 2005. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Trans. Computers*.

Shalan, M. and Mooney, V. I. 2002. Hardware support for real-time embedded multiprocessor system-on-a-chip memory management. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES'02).*

Received July 2011; revised October 2011; accepted December 2011