A Low-Cost Memory Interface for High-Throughput Accelerators

Jing Huang [123] Yuanjie Huang [12] Olivier Temam [4]
Paolo Ienne [5] Yunji Chen [1] Chengyong Wu [1]

[1] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing China*
[2] *University of Chinese Academy of Sciences, Beijing, China*
[3] *Loongson Technology Corporation Limited, Beijing, China*
[4] *INRIA, Saclay, France*
[5] *EPFL, Switzerland*

{huangjing,huangyuanjie,cyj,cwu}@ict.ac.cn, paolo.ienne@epfl.ch, olivier.temam@inria.fr

Abstract

Heterogeneous multi-cores, a mix of cores and accelerators, are becoming prevalent. These accelerators are designed for both speed and energy improvements, and thus, they increasingly come with a large number of load/store ports for achieving a high degree of parallelism. However, beyond GPGPUs, accelerators such as ASICs and CGRAs are increasingly capable of accelerating computations with irregular control flow and memory accesses; as a result, such accelerators need to be plugged to caches instead of scratchpads, and few studies focus on accelerator-to-cache interfaces. The main existing alternative are Load/Store Queues (LSQs) traditionally used to connect superscalar processors to caches and memory, but in the context of accelerators, they are overkill and could significantly reduce the area and power benefits of accelerators. Moreover, we show that they are just not fit for accelerators plugged to multi-banked caches.

In this article, we propose a fast accelerator-to-cache interface with a moderate area and power footprint compared to LSQs, even for a large number of load/store ports. For that purpose, we introduce a set of low-overhead techniques for ensuring in-order delivery of requests to/from cache banks. We synthesize and layout at 65nm the design of both our interface and an LSQ specially adapted to accelerators for a fair comparison. We find that our interface achieves on average 78% of the performance of an LSQ using only 16% of the area and 24% of the power.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ESWEEK'14, October 12 - 17 2014, New Delhi, India
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3050-3/14/10...$15.00.
http://dx.doi.org/10.1145/2656106.2656109

1. Introduction

Due to energy constraints [17, 8], there is a growing consensus that architectures will be evolving towards heterogeneous multi-cores composed of a mix of cores and accelerators. The micro-architecture community has initially focused on GPGPUs because they are the form of accelerators closest to traditional multi-cores, but increasingly specialized architectures are being considered, especially ASICs [12, 20] and reconfigurable architectures [9, 13]. While these different accelerator propositions can achieve very significant energy and/or performance gains, the corresponding studies and designs are focused on the computational aspects of the accelerator, less so on its interface with the cache or memory system.

However, there are several reasons why accelerator memory interface should receive greater attention: (1) unlike for most GPGPU applications, ASICs and CGRAs can be used to map applications with complex control flow, but the resulting memory access patterns can be very *irregular*, so that such accelerators cannot be plugged to traditional scratchpads, they must be plugged to *caches*, just like processors (a typical system organization would be processors and accelerators each plugged to private L1s, with shared L2s), (2) as accelerators reduce the energy spent in computations, the fraction of energy spent accessing memory will comparatively increase, a kind of Amdahl's law effect on energy, and (3) one of the key assets of accelerators is their reduced area, so one should take care that this area advantage is not outweighed by an over-sized memory interface.

However, accelerator-to-cache interfaces are not straightforward to design for at least three reasons. (1) A cache can induce memory requests ordering and dependence issues that accelerators are ill-equipped to handle. (2) Conversely, an accelerator can have significant bandwidth requirements with a far higher number of memory ports than a traditional processor [13, 24, 9], so that simply scaling up a processor LSQ is not a reasonable solution. (3) But an inefficient interface can partly wipe out the area and power benefits of the accelerator, and/or induce an excessive area/power overhead, incompatible with the usually small size and multiplicity of accelerators.

While there is a broad literature on using DMAs for handling the regular memory accesses of accelerators directly plugged to an embedded or main memory [2], there are few existing options for connecting a multi-port accelerator to a multi-banked cache in order to handle irregular memory accesses. In fact, the only existing option for handling the complex access ordering issues raised by such a multi-port architecture (whether processor or accelerator) connected to a multi-banked cache (which can return data out-of-order) remains the Load-Store Queue (LSQ) used in OoO processors. While acceptable for embedded and high-performance cores, an LSQ is an exceedingly costly solution for small-footprint accelerators.

In this article, we introduce a generic interface design between multi-port accelerators and multi-banked caches which can accommodate a large number of ports, and which can operate at the high clock frequency required by high-throughput accelerators. This interface can achieve 78% of the performance (in number of cycles) of a comparable LSQ (same number of ports), with only 24% of the power and 16% of the area footprint on average . The key principle of our approach is to ensure in-order processing of requests throughout the accelerator-to-cache interface which both avoids the load-/store dependence issues normally addressed by LSQs, and resolves the accelerator-specific in-order addressing/delivery constraints. At the same time, we implement this in-order management of requests without resorting to the costly buffers and comparators traditionally found in LSQs. In order to obtain reliable power, area and cycle time comparisons, we implement both our interface called AINT (*Accelerator INTerface*) and a specially adapted LSQ called ALSQ (*Accelerator LSQ*) down to the layout at 65nm, using standard CAD tools.

In summary, the contributions of this article are the following:

• Investigating the design issues of the special, but increasingly important, case of accelerators plugged into caches.

• Proposing a low-cost accelerator-to-cache interface alternative at a fraction of the cost of LSQs.

• Providing a detailed design, down to the layout.

• Quantifying the area and power costs of accelerator-to-cache interfaces with respect to existing accelerators.

In Section 2 we motivate the need for developing a specific accelerator-to-cache interface and we introduce the ALSQ baseline, in Section 3, we introduce the design of our AINT interface, in Section 4 we introduce the experimental methodology, we compare AINT and ALSQ in Section 5, and we describe the related work in Section 6.

2. Motivation

In this section, we explain both the functional and cost reasons why a different kind of interface must be introduced to connect accelerators to caches. We start by presenting examples of typical accelerators we target.

Accelerator	Type	Area (mm^2)	Power (W)	Clock (ns)	# load ports	# store ports
CGRA	Reconfigurable	1.63	0.35	4.80	16	16
ML ASIC	Machine-Learning	1.73	1.02	24.97	90	10
H264 ASIC	Video encoding	0.54†	0.39†	2.30‡	10	11

†*Extrapolated from 90nm to 65nm.*

‡*Original clock at 90nm.*

Table 1: *Description of accelerators.*

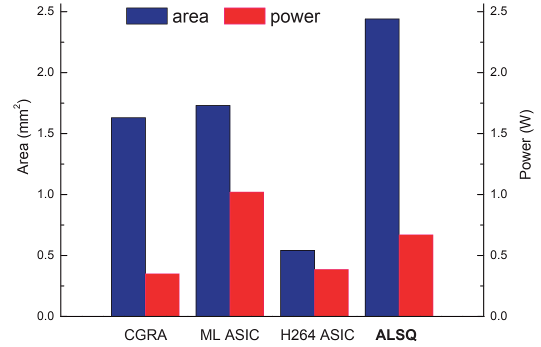

Figure 1: *Accelerators vs. LSQ area and power (ALSQ, 16 load ports, 16 store ports).*

2.1. Example Accelerators

Three example accelerators include a CGRA accelerator (Coarse-Grain Reconfigurable Array) [13], i.e., a word-level configurable circuit which is considered a more likely reconfigurable accelerator candidate for heterogeneous multi-cores than standard bit-level FPGAs, an ASIC for H264 video encoding [12], and an ASIC for machine-learning tasks [7]. We only gathered accelerators for which a layout was available (for reliable power and area measurements), we provide their characteristics in Table 1, and their area and power in Figure 1. Note that the CGRA and ML ASICs were designed at 65nm like the different interfaces of this article, but the H264 ASIC was designed at 90nm; we made a rough extrapolation by dividing area and power by the transistor size reduction factor, i.e., $\left(\frac{90}{65}\right)^2$; we left the clock unchanged.

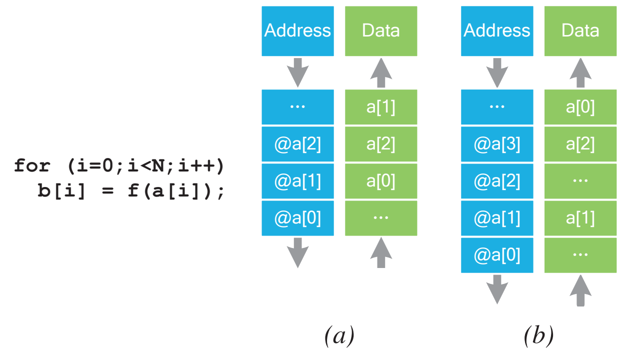

Figure 2: *Requests ordering issues in accelerators: (a) out-of-order delivery, (b) delayed delivery.*

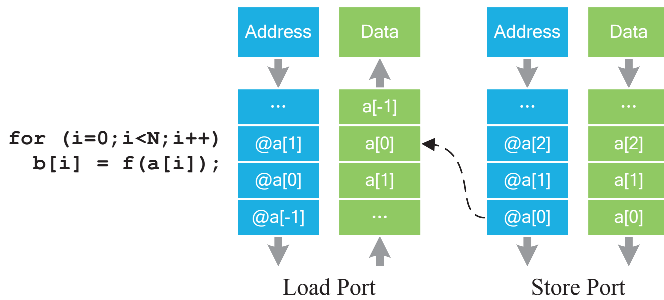

Figure 3: *Typical Read-After-Write (RAW) dependence.*

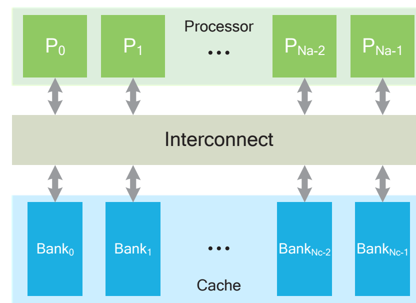

Figure 4: *Many ports to many banks.*

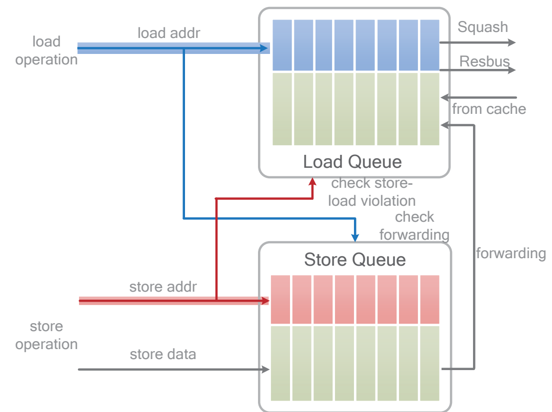

Figure 5: *Typical LSQ Structure.*

2.2. Push vs. Pull Addressing

One fundamental difference between how a processor and an accelerator deals with memory is that the former uses load-/store instructions, i.e., it *pulls* data from memory, while an accelerator sends memory requests (via internal address generation and/or DMA) and it expects all data to arrive to its ports in the proper order, i.e., data is *pushed* to the accelerator. So plugging an accelerator to a cache creates novel challenges as requests can come from the cache, and thus be pushed to the accelerator, out of order leading to functionally incorrect execution.

For instance, consider the simple case of a transformation applied to each individual element of an array where the input array data does not come back in the request order, see Figure 2(a).

Even if an interface provides some means to keep track of the order of data, it may fail to detect that a data has been skipped (because it was delayed), or it may detect that too late in the execution. Consider again the example of Figure 2(b) where $a[0]$ and $a[2]$ are received in-order but $a[1]$ is delayed.

A processor avoids such issues because it keeps track of instruction order (either because it uses in-order execution or through a reorder buffer) and of the data address associated with each load instruction, so if instruction I_0 requested the data at address A_0, it won't receive the data at address A_1 requested by instruction I_1, whatever the incoming order of the data at addresses A_0 and A_1. However, many accelerator fabrics do not even have a notion of instruction to maintain such an order.

2.3. Load/Store Dependences

Just as important is the classic issue of load/store dependences. Consider the textbook Read-After-Write case of Figure 3. A modern out-of-order processor will handle the dependence between $a[i-1]$ and $a[i]$ via its load/store queue (LSQ).

Unlike a processor, an accelerator may have a large number of load/store ports because it is the hardware instantiation (whether it is custom or configurable) of an algorithm with possibly many load/store instructions (especially if unrolling is used in the hardware algorithm to achieve higher ILP), see Table 1. The load/store instructions are typically implemented as hardware load/store ports, roughly one per load/store instruction of the software implementation [5, 24, 9, 13].

The overall structure of these many ports connected to multiple banks via an interconnect, can be construed as closer to a multi-processor than a processor, see Figure 4. However, a multi-processor can tolerate *per node* ordering, while an accelerator requires ordering both per "node" (for an accelerator, a node is a port) and across "nodes", see Section 2.2. So, currently, the only satisfying way for resolving such dependences is to resort to an LSQ.

2.4. Relative LSQ Cost and Scalability

The purpose of this section is twofold: (1) to assess the relative cost of an LSQ with respect to a typical accelerator, (2) to evaluate the scalability of an LSQ for a number of ports commensurate with the requirements of an accelerator.

IVM LSQ. In order to assess the relative cost of an LSQ, we have first extracted the LSQ developed as part of the synthesizable superscalar processor developed at UIUC, called IVM (Illinois Verilog Model) [25]. The IVM is a typical LSQ de-

LSQ	Area (mm^2)	Power (W)	Latency (ns)	# load/store ports	# banks
IVM	0.33	7.28e-02	1.88	2	2
ALSQ	0.21	6.72e-02	1.25	2/1	2

Table 2: *Characteristics of the IVM LSQ, and our own LSQ for accelerators (ALSQ) for similar ports/bank configuration.*

signed for a 4-issue processor; entries are pre-allocated when load/store instructions pass through the decode stage; the IVM can process 2 load/store instructions at the same time in the execute stage, and up to 6 load/store instructions can be retired simultaneously in the commit stage, see a block diagram in Figure 5. Both the load and store queues have 16 entries. The IVM LSQ implements load-store forwarding (a load address is compared to all addresses in the store queue), store-load violation (a store address is compared to all addresses in the load queue). The area and power of that LSQ at 65nm are reported in Table 2. The buffers and the large comparators required for these operations correspond to a significant fraction of the IVM LSQ area.

ALSQ. The IVM LSQ only contains two load/store ports, but a modern accelerator can contain a large number of load/store ports, as shown in Table 1. Most LSQs, like IVM, are designed for a small number of simultaneously memory accesses. While IVM is a thorough design, the RTL cannot be easily extended to the large number of ports of accelerators, because it is meant to be connected to a processor, and a lot of the dimensions (such as the number of ports) are related to processor characteristics (such as the issue width). Moreover, it is slightly unfair to directly compare it against accelerators because it contains a number of features and design decisions which are not needed by (or relevant to) accelerators. For instance, it is decomposed into three pipeline stages which would create significant overhead in the accelerator context and which are, in fact, largely irrelevant: in the first pipeline stage, the LSQ is accessed by the processor decode stage to pre-allocate entries in the load/store queues, then it is accessed again once the instruction is ready (to put the load/store instruction in the appropriate queue, and to perform the load-store forwarding and store-load violation checks), and accessed one last time when the load/store instruction retires/commits. Another example is that upon store/load violation, the LSQ directs the processor to flush the load queue and re-execute the load operation, a feature (and the associated control signals) which is obviously not necessary in our case.

Therefore, we decided to recode the RTL of an LSQ from scratch with none of the aforementioned features, though we tried to stay close to the design of the IVM LSQ whenever appropriate. Since this LSQ is dedicated to accelerators, we called in ALSQ (*Accelerator LSQ*). This LSQ supports store/load forwarding, load/store order-violation detection and in-order store commit. We use ALSQ as a more scalable baseline to later compare against our own interface. The characteristics of ALSQ are shown in Table 2 after layout at 65nm (as for the IVM LSQ; same library and VT mode, see Section 4). Note that, in spite of a best effort on the layout of the IVM LSQ, its cycle time is 1.88 ns vs. 1.38 ns for ALSQ; the difference is largely due to a different splitting of the LSQ tasks: the IVM LSQ both puts the load/store instruction in queues and performs the checks in the second pipeline stage, while ALSQ splits both tasks in two separate cycles.

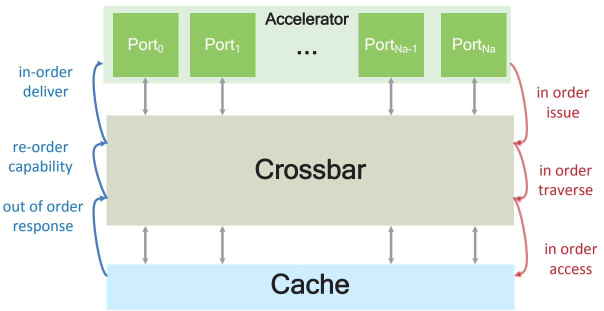

Figure 6: *Guaranteeing in-order delivery at every step of the interface (N_a accelerator ports, N_c cache ports).*

Accelerators vs. ALSQ We can now compare the accelerators area/power of accelerators and ALSQ. The largest configuration of ALSQ that we have placed and routed has 16 load ports, 16 store ports (and it is designed to communicate with a cache containing 16 ports), and it is the one used in Figure 1. Note that this number of ports is sufficient for both the CGRA and the H264 ASIC, but it is still insufficient for the ML ASIC (90 load ports). Still, we can see that the ALSQ area and power are already similar if not larger than those of the different accelerators. In other terms, the accelerator-to-cache interface already corresponds to an overhead of the order of 100%, i.e., it will roughly double the area and power footprint of the accelerator, and thus it will significantly reduce the potential benefit of introducing accelerators.

2.5. Summary

In summary, when plugging an accelerator to a cache, there is a need to address issues typically resolved by an LSQ, as explained in Section 2.3, but the cost of an LSQ with a large number of ports is unreasonably high, as explained in Section 2.4. Moreover, accelerators operate differently from processors, as explained in Section 2.2, so an LSQ cannot even serve as an accelerator-to-cache interface as is.

3. Accelerator-to-Cache Interface

In order to resolve both the data dependence issues normally solved by LSQs, and the aforementioned accelerator-specific ordering issues, we implement an accelerator-to-cache interface which guarantees the in-order delivery of data to the accelerator, without the overhead of processor instructions, with only a moderate performance penalty compared to an LSQ, and with a significantly lower area and power, a tradeoff in line with typical accelerators objectives. The principle of the interface is to guarantee request order every step of the way. Let us consider the general structure of an accelerator-to-cache interface, see Figure 6: we need to guarantee request order at the level of load/store ports when the request is issued, during the crossbar traversal, at the level of cache ports, and on the backward path from cache ports to load/store ports. We consider each step in the sections below and introduce the corresponding interface design component.

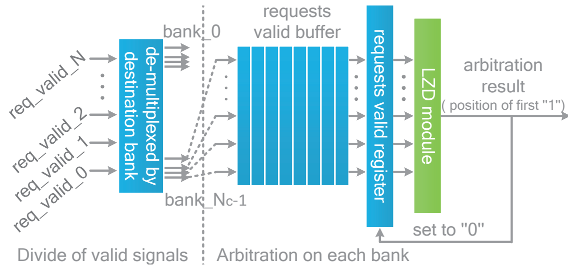

Figure 7: *Arbitration logic.*

3.1. In-Order Issue

If the crossbar cannot process a load/store request immediately after it has reached the load/store port (e.g., all crossbar ports busy), the request order can be altered.

The counter is incremented every cycle where at least one load/store request is received by a port, and used to timestamp (tag) all load/store requests issued by the accelerator at that cycle. Then, even if the crossbar stalls due to contention at the cache ports, or lower in the memory hierarchy, a load/store ports-level arbitration logic can ensure that requests are sent to the crossbar in the same order they arrived at the load/store ports.

The crossbar contains N_a inputs (number of load/store ports) and N_c outputs (number of cache ports). If $N_c < N_a$, we need to select the N_c load/store ports among N_a with the oldest global counter timestamp. We implement the selection as follows. Firstly, the N_a valid signals of the requests are divided into N_c groups according to their destination cache ports. Then, instead of an actual counter, we introduce a buffer (FIFO) for each group where each entry contains N_a bits, see Figure 7 ($N = N_a - 1$ in the figure). Each load/store port contains a bit signaling that the port has a request at the *current* cycle; this bit is reset every cycle. The bits of the N_a ports are all pushed simultaneously into the buffer every cycle, forming the content of one entry; then they are rest. The oldest non-empty entry of the buffer is used to select which ports will issue to the buffer at a given cycle. So the buffer entries actually form the different ticks of a request counter; we figuratively represent the corresponding order at the bottom of each entry in Figure 7 (in practice, these numbers are not stored, they simply correspond to the physical buffer entries which are permanently rotated); all ports requests within one entry have been issued by the accelerator at the same time. We add an entry register after the RAM, see Figure 7, where buffer control logic pushes the oldest non-empty RAM entry every cycle. This also avoids repeated RAM accesses (and saves energy) in case of crossbar contention, where a buffer entry cannot be consumed in one cycle, and it has to be read several times. The depth of the buffer is related to contention tolerance: the deeper the buffer, the more entries can be pushed by the accelerator to the buffer, even when the crossbar is experiencing contentions; this, in turn, avoids to stall the accelerator. We empirically found that a 16-entry buffer can accommodate contentions found in accelerators such as the 32-port (16 load ports, 16 store ports) CGRA described in Section 2.4; we use that number throughout our interface configurations though, in practice, it should be adapted to the crossbar dimensions.

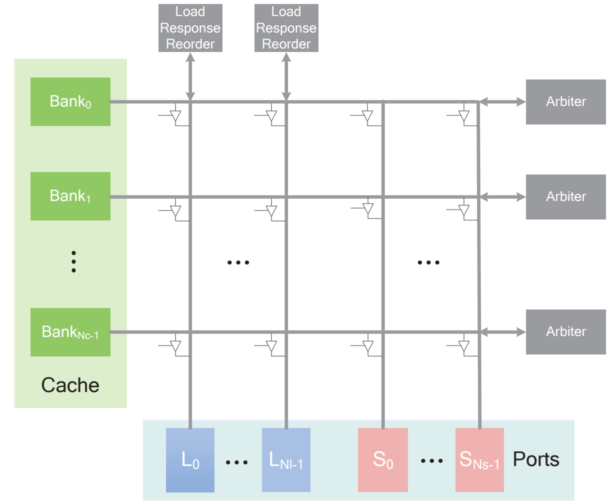

Figure 8: *Load/store ports to cache banks crossbar.*

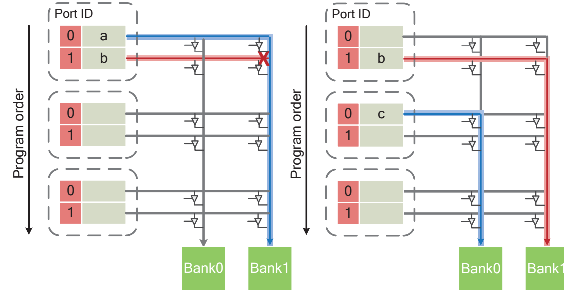

Figure 9: *Crossbar contention due to cache bank conflict, b is stalled.*

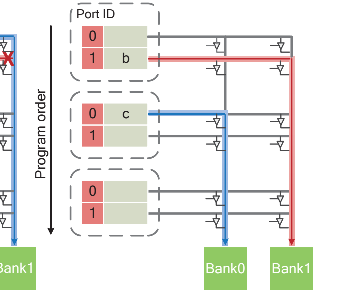

Figure 10: *Out-of-order delivery (b and c arrived at different cycles in the ports, yet are sent at the same cycle).*

3.2. In-Order Crossbar Traversal

Even though the crossbar has N_a inputs on the forward path, there is still a need for arbitration beyond the request order mentioned in Section 3.1. When two load/store ports requests need to access the same cache bank, they would induce a contention within the crossbar, see Figure 9. In most crossbar designs, such contentions are mitigated by introducing buffers within the crossbar. In order to reduce the crossbar cost and latency, and to avoid creating additional requests ordering issues within the crossbar, we delegate contention avoidance to the arbitration logic: the arbitration logic only issues sets of requests to the crossbar that will not conflict at the level of cache banks. As a result, the interface allows *partial* out-of-order requests delivery to the cache, i.e., strict order for a given bank, but out-of-order requests are possible across banks, see the example of Figure 10.

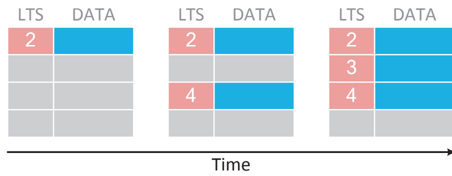

Figure 11: *Out-of-order incoming load requests at one port.*

In order to implement bank contention avoidance, we split the arbitration logic into N_c arbiters, like the one in Figure 7, one per cache bank. Each bank arbiter only selects *one* request every cycle, i.e., one per bank. The arbiter logic goes through the different bits of the buffer entry, one per cycle; in order to quickly find the first non-zero bit in the entry, we add a Leading-Zero Detection logic in the arbiter, see Figure 7. After a bit is selected, it is reset. Once all bits in an entry are 0, the buffer is shifted.

One cycle is used for the arbiter to select which port will issue to which bank, and one more cycle is used to pass the address (and data in case of store) request to the corresponding cache bank. We have implemented an interface with up to 32 load ports, 16 store ports and 32 banks at 65nm, and we have empirically observed that the arbitration logic can be performed in 1.09 ns, well above the frequency of the accelerators in Table 1, and in line with the high-throughput accelerators we are targeting. For even larger configurations (more ports and/or more banks) or accelerators with a high clock frequency, the traversal may require more than 1 cycle, in which case, we can pipeline the arbitration logic. We leave this case for future work as the number of ports we can tackle, and the clock frequency at which we can operate with the current interface are already sufficient for a broad range of accelerators.

3.3. Ordering Incoming (Load) Requests

Since any accelerator guarantees correct execution if it receives data at each port in the proper order, there is no need for maintaining global program order as an LSQ would do, the interface should only ensure that data is delivered back in-order to the load ports. Even though the forward path is designed to guarantee in-order requests at the level of each cache bank to avoid load/store dependence issues normally solved by an LSQ, cache misses can naturally skew the order in which requests are sent back to the load ports, see an example in Figure 11; it can also happen that two banks send a request back to the same load port at the same time.

So we need to keep track of *port order*. For that purpose, each request issued by a load port is augmented (tagged) with a local id, see Figure 12, which can be construed as a *local timestamp*, except that we rotate through a limited set of values (we used 16 different local ids/timestamps after exploration, i.e., 4 bits). This tag is passed throughout the interface. On the way back to the port, we use this tag to know how to sort the incoming requests and whether the port must still wait for a missing request. Exactly how we sort the incoming requests

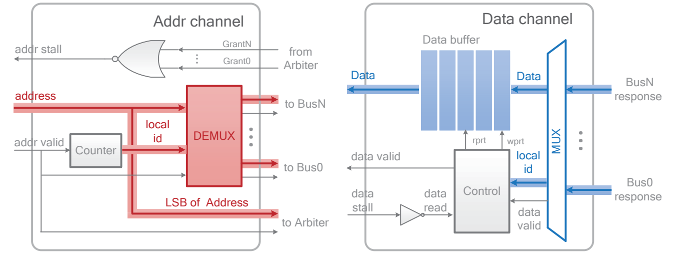

Figure 12: *Load port.*

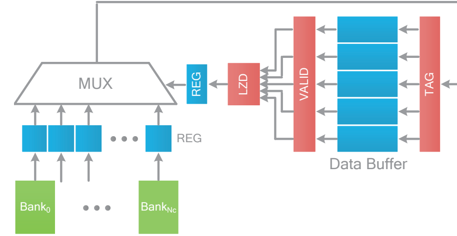

Figure 13: *Backward path arbiter (one per port).*

is explained in Section 3.4.2.

3.4. Load Port FIFOs: Buffering and Pre-Allocation

3.4.1. Buffering Incoming Requests Besides ordering incoming requests, a load/store queue plugged into an accelerator would also fulfill a second role: buffering the incoming data before it is consumed by the accelerator. In fact, a central LSQ-like buffering organization is both costly and detrimental for performance because of the time required to read the data from the LSQ and to pass it to the accelerator, and because multiple such requests could occur simultaneously, requiring again multi-ported RAMs.

Since, in the multi-port in-order delivery accelerator context, there is no need for centralizing all the load/store information in a single buffer/queue, we implement one single-port buffer in each load port, see Figure 12. For example, Park et al. [18] propose to split the load/store queue into several segments in order to reduce its complexity.

3.4.2. Pre-Allocation When a load issues a request to the interface, it first pre-allocates an entry in the aforementioned load port buffer. This serves several purposes.

Implementing incoming load requests order. When a load request comes back from the cache, we use the tag (local id) mentioned in Section 3.3 as a buffer index to determine where the data should be stored. Since several requests can come back from the cache banks at the same time, we need to arbitrate at the level of each port, in a dual spirit of the forward path (arbitrate at the level of each bank). However, the arbitration task is more simple because the arbiter does not need to preserve the order at which banks send the requests back, since that order may not correspond to the program order. Instead, it is the tag which contains the local order information. The load port maintains one register which is the id of the oldest (in

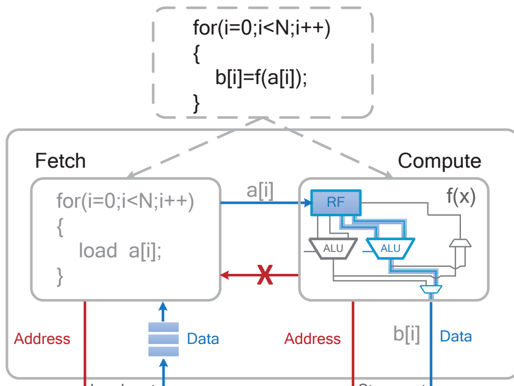

Figure 14: *Example of decoupled fetch/execution in accelerators.*

the local FIFO order) buffer entry which has issued its request but is still waiting for the corresponding response, using again LZD logic, and that register is used as the select bits of an $N_c \times 4 - 4$ multiplexer, see Figure 13; we also add a register before the multiplexer to store the incoming bank request.

Throttling down data fetch. Another fundamental difference between an accelerator and a processor, not yet touched so far, is that the fetching of data can be entirely *decoupled* from the consumption of data. Consider the example of Figure 14; on a CGRA, for instance, the address generation part and the computational part (corresponding to function f) would typically be implemented as two separate sub-circuits, the only connection between the two being the output of the load port. As a result, the address generation sub-circuit would greedily fetch data from memory, possibly at a rate not compatible with the memory bandwidth or the ability of the accelerator to consume the data. Therefore, there is a need to throttle down data fetching. However, excessively throttling down would jeopardize the accelerator ability to process data at the maximum speed.

A tradeoff consists in *pre-allocating* the data in the aforementioned load port buffer. This pre-allocation has several benefits: when all buffer entries have been pre-allocated, the fetch part can be stalled, but the buffer contains enough upcoming data that the accelerator itself won't stall. Moreover, the pre-allocation actually simplifies the ordering of incoming requests mentioned in the above paragraph. Instead of an arbitration logic to sort incoming requests according to their local timestamp, a simple tag check (corresponding to the local timestamp) is sufficient to decide where the incoming data should be stored, see Figure 12. Finally, this pre-allocation can leverage the buffer introduced in the load ports in Section 3.3; the overhead is small as it corresponds to a tag in each buffer entry for storing a local counter timestamp.

Disrupting other nodes. Another, more classic reason, for pre-allocating load requests is to avoid clogging the lower levels of memory in case the accelerator cannot accept all the data it has requested; this, in turn, can disrupt other nodes of a heterogeneous multi-core.

Bench -mark	Hot function	Bench -mark	Hot function
gzip	`deflate`	vpr	`try_route`
mcf	`refresh_potential`	gcc	`cse_insn`
crafty	`Evaluate`	parser	`power_prune`
perlbmk	`Perl_sv_update`	gap	`CollectGarb`
bzip2	`generateMTFValues`	twolf	`acceptt`

Table 3: *The hot functions of the SPEC2000 INT benchmarks.*

4. Methodology

Tools. We extracted the RTL of IVM from the Verilog version of the superscalar processor developed at UIUC [25]. We wrote from scratch the Verilog of ALSQ. We also wrote the Verilog version of our own interface, called AINT. Both ALSQ and AINT are configurable, and we generated the Verilog, and did the layout for each configuration. We synthesized all RTL codes using the Synopsys Design Compiler and the TSMC 65nm library GP (General Purpose), HVT (High Voltage Threshold, which corresponds to a low-power option). We then did the layout of the different designs using the Synopsys IC Compiler. We used Synopsys VCS and PrimeTime to get the power and power measurements after layout. Power measurements not derived from benchmarks correspond to a 50% activity.

While we performed functional validation using the netlists after place and route (waveforms), we built C models of both ALSQ and AINT in order to obtain cycle time measurements on large benchmark traces.

Baseline. In order to validate our baseline, i.e., ALSQ, we performed simulations to check the functional correctness of ALSQ, and we also compared the area and power of both LSQs for the configuration provided by IVM, i.e., 4 load ports, 2 store ports, 2 banks, see Table 2. ALSQ is slightly smaller and more power efficient than IVM, not because of any design optimization but simply because of the features we stripped out of IVM, see Section 2.4. Similarly, the total time required to process a load/store request is significantly smaller because ALSQ has a single pipeline stage vs. three in IVM.

Functionally, ALSQ and AINT almost have the same inputs and outputs, except for the `seq-id` field in ALSQ which identifies the individual load/store request order. Unlike the global request counter which is incremented every cycle, the `seq-id` counter is incremented with every load/store request.

Benchmarks. We extracted traces from 10 SPECINT2000 benchmarks on a Core i5 platform running CentOS 6.3 listed in Table 3; the programs are compiled with GCC4.4.7. The traces correspond to the hottest function of each benchmark, since this is the most likely part to be mapped on an accelerator. We decided to use SPECINT benchmarks for two reasons: because the three accelerators we considered, i.e., CGRA, H264 ASIC, ML ASIC, see Section 2.4, only

Configuration	# load ports	# store ports	# banks
l2s1b4	2	1	4
l4s2b8	4	2	8
l16s8b16	16	8	16

Table 4: *Interface configurations used for comparing AINT and ALSQ.*

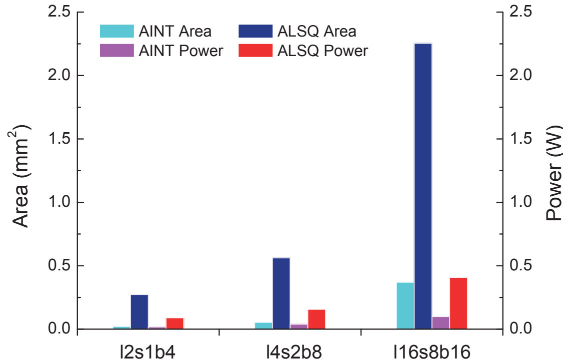

Figure 15: *Area and power of different ports/banks configurations.*

use fixed-point computations, and because we wanted a mix of regular streaming-like computations and complex memory accesses in order to exercise the memory interfaces. As can be seen in Table 1, high-throughput accelerator designs can have 10 load ports/10 store ports or even (much) more. We considered interfaces with up to 32 load ports and 16 store ports. In order to exercise such interfaces, we simulated the benchmarks using the technique proposed by Postiff et al. [19] in order to extract the high degree of parallelism known to exist in SPECINT benchmarks, and which can effectively be realized in practice with appropriate compilation techniques [4].

5. Experiments

5.1. Area, Power, Clock

In this section, we compare the area, power and cycle time of ALSQ and AINT after layout at 65nm. We consider three different combinations listed in Table 4. We report both area and power in Figure 15; note that we keep the depth of the different buffers used in both AINT and ALSQ constant (32-entry buffer in ALSQ; the depth of the different AINT buffers can be found in Section 3). We can observe that, as the number of ports/banks increases, AINT is significantly smaller and more power efficient than ALSQ. This is due to several features of ALSQ which scale poorly: especially the queue buffers which must be multi-ported, and the comparison logic for load-store forwarding and store-load violation.

In Figure 16, we examine in more details the impact of the number of banks for both power and area, for the same number of ports (4 load ports, 2 store ports). We can see again that the size and power consumption of ALSQ increases much faster than that of AINT as the number of channels to memory, and thus the number of requests that must be served simultaneously, increases.

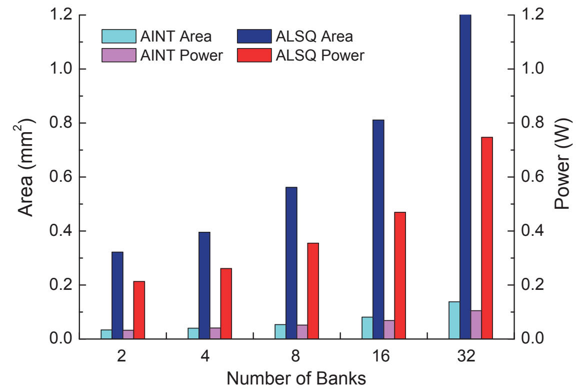

Figure 16: *Area and power as a function of the number of banks (4 load ports, 2 store ports).*

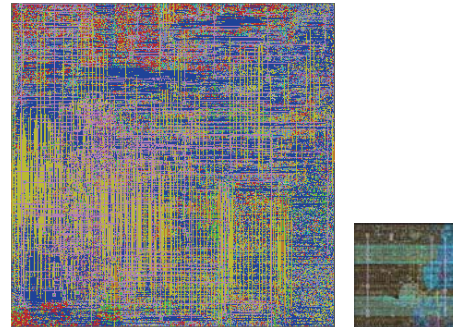

Figure 17: *ALSQ vs. AINT layout (4 load ports, 2 store ports, 4 banks); image size ratio corresponds to area size ratio.*

Finally, in Figure 17, we provide a snapshot of the two interfaces in that same configuration (4 load ports, 2 store ports) and 4 banks after layout; the relative snapshot sizes correspond to the relative interface areas.

5.2. Scalability

We have placed and routed AINT for all the following configurations: n load ports ($2 \leq n \leq 32$), $\frac{n}{2}$ store ports, and b banks ($2 \leq b \leq 32$). The corresponding area and power surfaces are shown in Figure 18 and 19 respectively. Even with 32 load ports, 16 store ports and 32 banks, the area of AINT is just below 1 mm^2, and the power is 0.69 W, i.e., less than 1/2 the

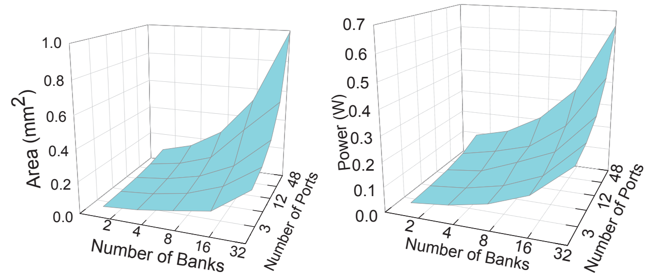

Figure 18: *Area of AINT for a range of configurations*

Figure 19: *Power of AINT for a range of configurations*

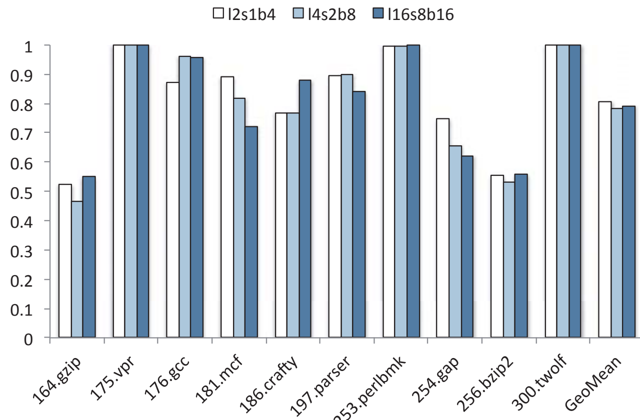

Figure 20: *Slowdown of AINT w.r.t. ALSQ for three configurations.*

Configuration	Time (ns) ALSQ	Time (ns) AINT
l2s1b4	1.34	0.58
l4s2b8	1.51	0.83
l16s8b16	2.14	1.00

Table 5: *Cycle time of AINT and ALSQ (65nm).*

area and 2/3 the power of ALSQ with 16 load ports, 8 store ports and 16 banks.

5.3. Execution Time

The main drawback of AINT with respect to ALSQ is that it does not implement load-store forwarding, and as a result, its performance, expressed in number of cycles, is lower than LSQs, as shown in Figure 20 on a set of benchmarks. These results are pessimistic for AINT because we have assumed the maximum possible throughput, i.e., as many requests as the number of ports every cycle, and perfect caches (no misses). Requests can be delayed due to contention at the level of cache banks, and load-store forwarding reduces the number of requests ALSQ effectively has to send to the cache banks, and thus, the port occupation is lower and there are fewer bank contentions. Even though the slowdown can reach 0.47, the average slowdown remains reasonable, varying between 0.78 for l4s2b8 to 0.81 for l2s1b4 (l16s8b16 exhibits an intermediate average slowdown of 0.79). Moreover, the design of AINT allows to achieve a critical path about 2x smaller than that of LSQ, see Table 5, and the corresponding lower cycle time can fully compensate for the performance difference.

6. Related Work

Due to growing energy constraints, an increasing number of research groups are investigating accelerator designs. For instance, Venkatesh et al. [24] propose a grid of small-scale and specialized cores, each capable of efficiently executing a subset of functions, and the union of which can have a significant application scope. Fan et al. [9] and Yehia et al. [26] propose two different approaches for designing accelerators intermediate between ASICs and reconfigurable circuits. Other types of accelerators include dedicated functions, such as software-defined radio [16], H264 video encoder [12], machine-learning [20], etc. Some have attempted to find common kernels among many programs such as Clark et al. [5] who have proposed a loop accelerator capable of efficiently executing a broad set of loop body patterns, while others propose to generate accelerators [15]. Next to these more or less specialized accelerators, there is a large body of research on configurable accelerators, either FPGAs, or CGRAs [23, 13].

However, most of these research works focus on the design of accelerators themselves, rather than their connection to memory. Many of these accelerators, especially the more regular or streaming oriented ones, e.g., [5, 26], propose to connect to memory via a set of streaming buffers. Sometimes, the connection to memory is voluntarily and explicitly ignored [12, 20] for understandable reasons of separation of concerns. Beyond the aforementioned streaming buffers, there are few existing options for connecting accelerators to memory. The most classic approaches rely on a combination of DMAs and scratchpads [2], and many accelerators have integrated efficient internal DMA controllers or scratchpads [21, 10]. Some more sophisticated versions include scalable DMAs capable of managing multiple streams [6], perform some degree of prefetching for multi-ported streaming accelerators [11], or adjusting the layout of data in memory for achieving high bandwidth. However, because connecting accelerators to caches is at the crossroad of two architecture domains (SoC design and high-performance micro-architectures), few research studies focus on this issue. Igehy et al. [14] uses a reorder buffer for incoming memory requests which can arrive out of order; however the issue of ordering multiple concurrent streams of requests present in a crossbar and multiple cache banks is not considered.

Naturally, there is a very large body of research works on the interactions between processors and caches, and it is not possible to mention even a representative subset here. A significant fraction has focused on load/store queue design, though the goals were different. For instance, Park et al. [18] proposed to reduce the load/store queue complexity by splitting it into several segments. However, regardless of splitting criterion, each LSQ segment still requires non-trival area/power to support OoO issue. Sha et al. proposed a store queue index prediction technique to simplify the forwarding logic in load-/store queues [22]; though more simple, such a forwarding technique can still not be easily grafted onto our interface.

Another large and partially relevant body of work exists in the domain of multi-processors or vector processors, and their interaction with multi-banked memories, including bank contention issues [1, 3]. Not only the consistency constraints of our interface are different from the context of multi-processors + multi-banked memory, but some of these techniques may turn out to be complementary to AINT; our goal was not to reduce bank contentions, rather to find a lightweight micro-

architecture approach for ensuring correct execution in the presence of multiple ports.

7. Conclusions and Future Work

This article is focused on the rarely addressed, albeit increasingly important, issue of connecting accelerators to caches. Heterogeneous multi-cores propose to combine a relatively uniform template of nodes connected together via caches and a NoC, with an increasingly diverse set of accelerators. In order to minimize the burden of the accelerator designer, it is important to find a simple, generic and lightweight method for connecting accelerators to caches. Because caches induce access ordering issues that accelerators are not designed to cope with, reusing processor LSQs is not an option. Moreover, we show that LSQ designs scaled up to accommodate the number of ports of high-throughput accelerators would induce an unreasonable overhead, partly voiding the area and power benefits of accelerators. Consequently, we investigate an alternative method for connecting accelerators to caches. The principle is to guarantee in-order processing of accelerator requests all the way to the cache (and back for load requests). We propose an interface design that does not require the large multi-ported buffers or comparators of LSQs. While imposing in-order access induces a slowdown of 0.78 to 0.81 on average, we also show that the corresponding interface design requires only 16% of the area and 24% of the power of an LSQ. Moreover, the lower cycle time of the interface can compensate for the additional cycle count. We synthesize and then perform the layout of multiple configurations of the proposed interface at 65nm, and of the LSQ baseline for the sake of comparison.

The accelerator model and the traffic patterns are still simple in this work. As our future work, we consider to integrate our interface with state-of-the-art accelerators and evaluate its performance with more practical traffics.

8. Acknowledgment

We thank the anonymous reviewers for their helpful comments and insight suggestions. This work is partially supported by the NSF of China (under Grants 61100163, 61133004, 61222204, 61221062, 61303158), the 863 Program of China (under Grant 2012AA012202), the Strategic Priority Research Program of the CAS (under Grant XDA06010403), and the 10,000 talent program.

References

[1] D. H. Bailey, "Vector computer memory bank contention," *Computers, IEEE Transactions on*, vol. 100, no. 3, pp. 293–298, 1987.

[2] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory," in *Proceedings of the tenth international symposium on Hardware/software codesign - CODES '02*. New York, New York, USA: ACM Press, May 2002, p. 73.

[3] G. E. Blelloch, P. B. Gibbons, Y. Matias, and M. Zagha, "Accounting for memory bank contention and delay in high-bandwidth multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 8, no. 9, pp. 943–958, 1997.

[4] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August, "Revisiting the Sequential Programming Model for Multi-Core," in *International Symposium on Microarchitecture*. Portland, Oregon: IEEE, Dec. 2007, pp. 69–84.

[5] N. Clark, A. Hormati, and S. Mahlke, "Veal," in *International Symposium on Computer Architecture*, Beijing, Jun. 2008, pp. 389–400.

[6] D. Comisky and C. Fuoco, "A Scalable High-Performance DMA Architecture for DSP Applications," p. 414, Sep. 2000.

[7] Z. Du, A. Lingamneni, Y. Chen, K. V. Palem, O. Temam, and C. Wu, "Leveraging the Error Resilience of Machine-Learning Applications for Designing Highly Energy Efficient Accelerators," in *Asia and South Pacific Design Automation Conference*, 2014.

[8] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, Jun. 2011.

[9] K. Fan, M. Kudlur, G. S. Dasika, and S. A. Mahlke, "Bridging the computation gap between programmable processors and hardwired accelerators," in *HPCA*. IEEE Computer Society, 2009, pp. 313–322.

[10] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias, "An integrated hardware/software approach for run-time scratchpad management," in *Proceedings of the 41st annual Design Automation Conference*, 2004.

[11] S. Girbal, O. Temam, S. Yehia, H. Berry, and Z. LI, "A memory interface for multi-purpose multi-stream accelerators," in *International conference on Compilers, architectures and synthesis for embedded systems*. New York, New York, USA: ACM Press, Oct. 2010, p. 107.

[12] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *International Symposium on Computer Architecture*. New York, New York, USA: ACM Press, 2010, p. 37.

[13] Y. Huang, P. Ienne, O. Temam, and C. Wu, "Elastic CGRAs," in *International Symposium on Field-Programmable Gate Arrays*. Monterey: paper under submission, 2013.

[14] H. Igehy, G. Stoll, and P. Hanrahan, "The design of a parallel graphics interface," in *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '98. New York, NY, USA: ACM, 1998, pp. 141–150.

[15] V. Kathail, "Creating power-efficient application engines for SoC design," Synfora Inc., Tech. Rep., 2005.

[16] Y. Lin, H. Lee, M. Woh, Y. Harel, S. A. Mahlke, T. N. Mudge, C. Chakrabarti, and K. Flautner, "SODA: A Low-power Architecture For Software Radio." in *ISCA*, 2006, pp. 89–101.

[17] M. Muller, "Dark Silicon and the Internet," in *EE Times "Designing with ARM" virtual conference*, 2010.

[18] I. Park, C.-l. Ooi, and T. N. Vijaykumar, "Reducing design complexity of the load/store queue," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03)*, 2003.

[19] M. A. Postiff, D. A. Greene, G. S. Tyson, and T. N. Mudge, "The limits of instruction level parallelism in spec95 applications," *SIGARCH Comput. Archit. News*, vol. 27, no. 1, pp. 31–34, Mar. 1999.

[20] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," in *International Symposium on Computer Architecture*, 2013.

[21] S. S., L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proceedings of the conference on Design, automation and test in Europe*, 2002.

[22] T. Sha, M. M. Martin, and A. Roth, "Scalable store-load forwarding via store queue index prediction," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, 2005.

[23] B. D. Sutter, P. Raghavan, and A. Lambrechts, "Coarse-Grained Reconfigurable Array Architectures," *Elements*, no. 1, 2010.

[24] G. Venkatesh, J. Sampson, N. Goulding-hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "QsCORES : Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores Categories and Subject Descriptors," in *International Symposium on Microarchitecture*, 2011.

[25] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *Dependable Systems and Networks, 2004 International Conference on*. IEEE, 2004, pp. 61–70.

[26] S. Yehia, S. Girbal, H. Berry, and O. Temam, "Reconciling specialization and flexibility through compound circuits," in *International Symposium on High Performance Computer Architecture*. Raleigh, North Carolina: Ieee, Feb. 2009, pp. 277–288.