Efficient Data Streaming with On-chip Accelerators: Opportunities and Challenges

Rui Hou[1], Lixin Zhang[3], Michael C. Huang[2,4], Kun Wang[1], Hubertus Franke[2], Yi Ge[1], and Xiaotao Chang[1]

[1]IBM China Research Laboratory,
Email:{hourui, wangkun, geyi, changxt}@cn.ibm.com
[2]IBM T.J. Watson Research Center,
Email: {mchuang, frankeh}@us.ibm.com
[3]National Research Center of High Performance Computers,
Institute of Computing Technology, Chinese Academy of Sciences
Email: zhanglixin@ict.ac.cn
[4]University of Rochester,
Email: michael.huang@rochester.edu

Abstract

The transistor density of microprocessors continues to increase as technology scales. Microprocessors designers have taken advantage of the increased transistors by integrating a significant number of cores onto a single die. However, a large number of cores are met with diminishing returns due to software and hardware scalability issues and hence designers have started integrating on-chip special-purpose logic units (*i.e.*, accelerators) that were previously available as PCI-attached units. It is anticipated that more accelerators will be integrated on-chip due to the increasing abundance of transistors and the fact that not all logic can be powered at all times due to power budget limits. Thus, on-chip accelerator architectures deserve more attention from the research community.

There is a wide spectrum of research opportunities for design and optimization of accelerators. This paper attempts to bring out some insights by studying the data access streams of on-chip accelerators that hopefully foster some future research in this area. Specifically, this paper uses a few simple case studies to show some of the common characteristics of the data streams introduced by on-chip accelerators, discusses challenges and opportunities in exploiting these characteristics to optimize the power and performance of accelerators, and then analyzes the effectiveness of some simple optimizing extensions proposed.

1 Introduction

Continued technology scaling has enabled modern chips to integrate a very significant amount of transistors on a single die. How to translate this vast amount of transistors into end performance with good energy efficiency is a central goal for modern microprocessor designers and researchers in the architecture community. Chip-multiprocessors (CMP) are the current main approach to utilizing the increasing transistor budget. For instance, many recently announced mainline processors, such as Intel's Nehalem, AMD's Opteron 6100 series, and IBM's Power7, have put eight or more identical general-purpose cores on a single chip. However, simply replicating the same type of core on a processor chip has met with diminishing returns in both performance and efficiency.

As an alternative, processor designers have started adopting heterogeneous designs that incorporate processing units other than general-purpose cores into a die. Such a processing unit can be a light-weight core (*e.g.*, the SPU in IBM's Cell processor [6]) or a domain-specific accelerator (*e.g.*, the Crypto engine in Sun UltraSPARC T2 [15]). Heterogeneous designs with multiple types of general-purpose cores have been gaining wider popularity in the literature. However, heterogeneous designs with general-purpose cores and special-purpose accelerators received less attention even though there are already multiple commercially available products (Table 1).

By targeting specific, well-defined functionalities, an accelerator uses hard-wired implementations that avoid the overheads of general-purposes architectures. As a re-

sult, an accelerator can improve performance and energy efficiency by orders-of-magnitude over a typical general-purpose core. At the same time, their special-purpose nature seems to defy generalization, and make research of broadly applicable solutions difficult. In practice, however, there are ample opportunities for the general research community. In particular, we have observed that typical accelerator tasks demonstrate predictable, well-behaved streaming memory access patterns. As designers incorporate more accelerators into existing architectures with general-purpose cores, and reuse the general-purpose coherence and communication substrates, there are clear opportunities to better match the access behavior and the underlying hardware support. We also observed that the streaming data account for a considerable fraction of total on-chip data traffic in real-world deployments. Therefore, effective optimizations will have a significant impact on performance and efficiency. Yet, to the best of our knowledge, little is done in this area in both commercial products and academic research work.

This paper attempts to generalize a popular on-chip accelerator design Section 2) and uses it as a platform for case studies of the data traffic of on-chip accelerators. In particular, we describe the common characteristics of the data access patterns of the accelerators (Section 3); point out both opportunities and practical challenges in addressing deficiencies in existing designs (Section 4) and hopefully attract more academic research in this important area. We also discuss our own proposals of optimization (Sections 5); present some experimental results (Sections 6); discuss related work (Sections 7); and conclude with a petition for more research work in this area (Section 8).

2 Overview of On-Chip Accelerators Architecture

Accelerators come in many different flavors. Table 1 lists a few existing or upcoming processors with on-chip accelerators [7, 13, 15, 17]. For example, Intel's EP80579 (Tolapai) combines an *IA Complex* (*i.e.*, x86) with an *Acceleration and Networking I/O Complex*. It targets network security and IP telephony applications. Its accelerators share the physical address space with the IA complex for low overhead data transmission. Sun's UltraSPARC T2 and Rainbow Falls both integrate a cryptography accelerator into each core.

We base our study on a generalized version of the IBM PowerEN™ architecture (formerly known as wire-speed processor) [5, 9]. PowerEN™ represents a generic processor architecture in which processing cores, hardware accelerators, and I/O functions are closely coupled in a system-on-chip. It is designed to work at the "edge

Vendor	Processor Name	Accelerators
Intel	EP80579	Crypto, Network offloading engine
Sun	UltraSPARC T2	Bulk encryption, Secure hash, Elliptic Curve Cryptography (ECC)
Sun	Rainbow Falls	Kasumi bulk cipher, Hash (SHA-512)
RMI	XLP	Network offloading engine, Security Acceleration Engine Compression engine
IBM	PowerEN™ (formerly known as Wire-Speed Processor or WSP)	Crypto, Compression, XML parsing, Pattern matching, Network offloading engine
IBM	Z-series Processor	Crypto, Compression

Table 1. Accelerators in commercial processors.

of the network" and operate at "wire speed" (*i.e.*, speeds in which data are transmitted over the network). Its first implementation combines 16 multi-threaded IBM PowerPC cores with special-purpose accelerators optimized for packet processing, security, pattern matching, compression, XML parsing, and high-speed networking. Its applications include edge-of-network processing, intelligent I/O devices in servers, network attached appliances, and streaming applications.

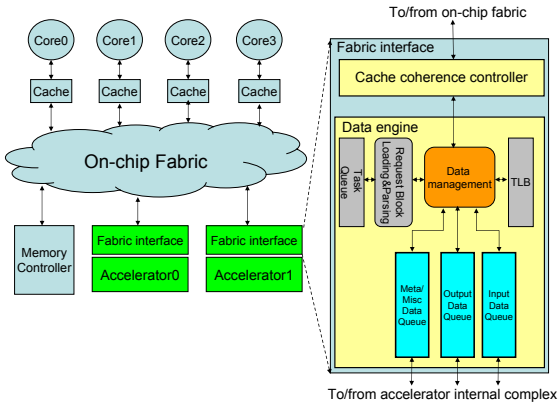

Figure 1. On-chip accelerator architecture.

The right-hand half of Figure 1 contains the generalized accelerator architecture used in our study. It shows a modular design where all cores and accelerators use the same coherence module to interface with the on-chip fabric. The central control unit of the accelerator is the *data engine*, which includes a task queue, a data management module, three data queues, and an address translation unit. It behaves in a way similar to the DMA engines widely used in the memory and I/O devices. The three data queues carry the input data, output data, and meta/miscellaneous data respectively. The *data management module* is responsible for issuing data access requests and managing the data queues.

To enable an accelerator to tap into the high-bandwidth and low-latency on-chip cache-to-cache data transfer fabric, a typical design handles data requests

from an accelerator just like data requests from a (general-purpose) core. As a result, an accelerator can directly access data from on-chip caches, but pays the penalty of going through all the steps required by the coherence protocol. To better support accelerators' traffic, chip designers have added some specialized functionalities like partial cache line accesses and cache injections. For example, PowerEN™ supports cache injection, where data from accelerators can be directly injected into the caches rather than being written to the local memory and then being fetched to the caches upon subsequent demand misses.

To use an accelerator, software needs to initialize the source and destination buffers, build a request block that contains relevant information, and activate the accelerator by sending the request block or a pointer to the request block to the accelerator. Upon receiving the request, an accelerator inserts it into the *accelerator task queue*. The task queue enables multiple outstanding tasks. Once a task is selected for processing, the accelerator parses the request block, performs various functions on the specified input data, and posts results to the specified output buffer.

3 Data Access Patterns of Accelerators

Due to its special-purpose nature and the simplicity of its hardware and software, an accelerator often exhibits more predictable memory access patterns than a general-purpose core. In this section, we first describe three different types of accelerators and their memory access patterns, and then extract the common characteristics of the memory access streams of those accelerators.

3.1 Case Studies

Case study 1: Crypto accelerator

A crypto accelerator performs compute-intensive encryption/decryption operations. Its typical application uses a very small amount of meta data (usually less than 200 bytes), easily captured in the meta data buffer. As a result, almost all of its data requests issued to the on-chip fabric are for the input/output data. Figure 2 shows the memory address trace of one task that decrypts incoming network data. The trace was taken from our full-system simulator described in Section 6. The x-axis marks time in an increasing order when memory requests were issued by the accelerator. The figure clearly shows two distinct un-interleaved streams, one for loading the input data and one for storing the output data.

Case study 2: Decompression accelerator

The Decompression accelerator in our study adopts the implementation proposed by Yu et al. [19], which uses only 2KB hardware buffer essentially as a (very effective) cache of the 32KB history buffer needed by the algorithm. Occasional access to history data not captured in the hardware buffer results in extra requests disrupting the otherwise perfectly sequential input stream as shown in Figure 3.

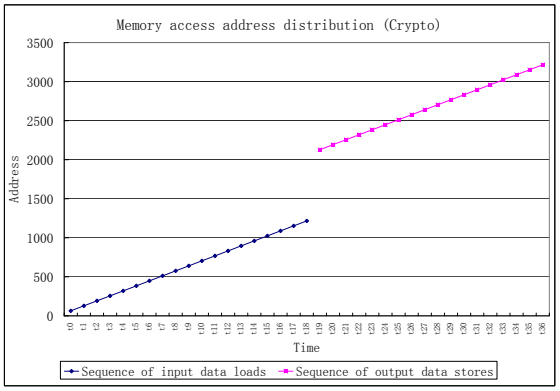

Figure 2. Address trace from a crypto task.

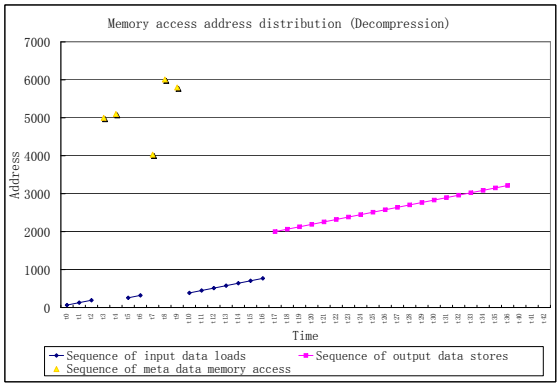

Figure 3. Address trace from a decompression task.

A pattern matching accelerator exhibits similar access patterns with those of a compression accelerator, as a pattern matching algorithm requires a set of rules to be applied on the input data. When the rule set exceeds the capacity of the data buffer in the accelerator, accesses to the rule set will interleave with those to the input data. On the other hand, we have seen application developers tuning the rule set to make it fit into the buffer available in the hardware.

Case study 3: Network offload accelerator

The integrated network offload engine, or the HEA (Host Ethernet Adapter), provides the packet processing framework optimized for the PowerEN™ multicore environment. A HEA application typically deals with a limited amount of meta data. Its primary data flow includes reading the outgoing packets from the caches or memory

in the egress phase, and injecting the incoming packets into the caches in the ingress phase. Because the size of a typical network packet is smaller than 2KB and thus can be contained within a 4KB page, a packet is typically stored in contiguous physical addresses. Consequently, data accesses coming out of the HEA are mostly consecutive. Additionally, each HEA supports four 10Gb/s serial interfaces. When the four HEA serial interfaces all operate at 10Gb/s, the HEA can consume a significant fraction of the on-chip bus bandwidth.

In chips optimized for networking environments, such as IBM PowerEN™and Intel's EP80579, a network offload engine is often integrated with a crypto engine and/or a pattern matching engine, because encryption/decryption and pattern matching are some of the most common operations performed on the incoming network packets. For instance, the IPSec protocol stack requires encryption and hash operations on the data packets in both egress and ingress phases [8].

3.2 Common Characteristics

We can conclude from the discussion above that most streams to/from an accelerator are perfectly sequential and smaller than a 4KB page, and load streams and store streams are distinctive.

Another feature not shown in the discussion above is that a stream often gets all lines within the stream from the same data source (*e.g.*, a particular L2 cache) and puts them to the same destination. This feature is largely due to the programming model used for accelerators. In a typical use of an accelerator, the entire source buffer is initialized by one software thread and the entire destination buffer is set up by one software thread. (In many cases, it is the same thread that sets up the source and destination buffers. However, it can also be two different threads exchanging data through the accelerator.)

A typical accelerator is fully pipelined to handle each incoming data block in a fixed number of cycles. Once it passes the initial stage and reaches the steady state, it issues requests at the speed it processes data, *i.e.*, at constant time intervals.

In summary, accelerator-related data streams have the following characteristics: ① uninterrupted or slightly interrupted sequential streams; ② distinctive load streams and store streams; ③ relatively constant timing intervals between consecutive requests; and ④ short streams ranging from a few cache lines to no larger than a 4KB page.

4 Opportunities and Challenges

In current practice, accelerators are generally incorporated into processor chips as an add-on component with minimum changes to the existing on-chip cache coherence and communication substrates. These substrates are not yet optimized to deal with accelerators, which can prevent accelerators from reaching their full potentials.

Take coherent data transport of an accelerator data stream in a snooping CMP for example.[1] While conceptually straightforward, transporting a single cache line incurs a whole set of "handshake" actions. It first triggers tag look-ups in all caches participating in the snooping (usually to all the last level caches) and the on-chip memory controller (in order to search for conflicts with outstanding memory requests). After all tag look-up responses are combined a decision is made as to which unit provides the requested cache line of data. The decision is then passed to the units that participated in the snooping phase. The chosen unit then performs a cache or DRAM access to get the cache line and initiates a data transfer request to the on-chip fabric. The data transfer request first goes through two arbitration processes: one for the incoming data port of the destination and one for the data bus from the source to the destination. Upon successful arbitration, the data transfer starts after a fixed number of cycles.

As Section 3 shows, an accelerator typically streams many consecutive cache lines. These lines are commonly located in the same unit. Conceivably, they can be transported under a single set of handshake actions, amortizing the cost and reducing resource occupancy all around. Below, we discuss the opportunities and challenges in more detail.

Coherence When a stream of n cache lines requested by an accelerator indeed resides in the same cache, the first snoop can reveal that and save the m participating units from the rest of the $n-1$ snoops. The savings in energy and in the occupancy of the tag arrays are clear and can be significant. However, this requires non-trivial extension to the overall coherence protocol. First, a coherence action needs to specify the number of lines desired. Second, each controller needs the capability to walk through the cache and figure out the number of consecutive lines it can provide. Finally, the logic that combines responses needs to detect corner cases in which the action would revert to basic line-based transport.

Data transfer fabric A cache line transfer request must go through one or more arbitration phases regardless of whether the underlying on-chip fabric is a broadcast bus, or a segmented ring, or a packet-switched network-on-chip. To overcome the arbitration delay, an on-chip data

[1] While coherence is not always required – some DMA engine requires software to flush all data out of the caches and then use backdoor or non-coherent requests to move data between the memory and the target unit – in most CMPs, cache-to-cache coherent data transfer is much more efficient.

fabric normally supports pipelined transfer and multiple outstanding requests to maximize the utilization of the data transfer channel.

As with the case of snooping, combining multiple consecutive lines into one transport action amortizes the overhead of arbitration and other setup operations and potentially allows much higher data channel utilization. There are a number of options to support bulk data transfer.

First, the same underlying fabric can be designed to accept more granule sizes of data transfer. In the case of a network-on-chip design for example, this means accommodating longer packets that contains more flits. This reduces the overhead related to routing and virtual channel allocation, but increases buffering complexities. For fabrics such as (segmented) buses, we can allocate a longer period of time for transfer, but need to carefully consider the issues of fairness and response time.

Another option is to build a separate, dedicated link for bulk data transfer. Given the predictability of the accelerator data streams, there are ample opportunities for latency tolerance. Therefore this data link can be simpler in architecture (*e.g.*, a basic bus) and made more energy-efficient by sacrificing transport latency.

Streaming access in memory hierarchy Conventional memory hierarchy is designed to work at the granularity of cache lines. For every cache line in a stream, it checks the tag, reads the data, and then updates the tag and LRU bits. Additional support is necessary to allow efficient bulk data handling. On the other hand, there are opportunities such as that the tag and LRU bits of the lines in a stream have a high probability of being identical. How to effectively exploit properties likes this remains to be seen.

Even though at the DRAM macro level, it is natural to support streaming from an open DRAM page, the common practice of the memory controllers in commercial systems is to use a close-page policy. Furthermore, some systems like Power7 intentionally interleave consecutive lines across different memory channels/ranks/banks or caches for better load balancing. Such interleaving makes it easier to achieve better load balance in the memory system, without affecting long data streams often seen in the commercial and scientific workloads. But for a short stream such as the one used by an accelerator, it is often not long enough to have two lines from the stream mapped to the same DRAM bank. How to design a memory system that can balance the needs of traditional applications and the need of accelerator applications is part of future research.

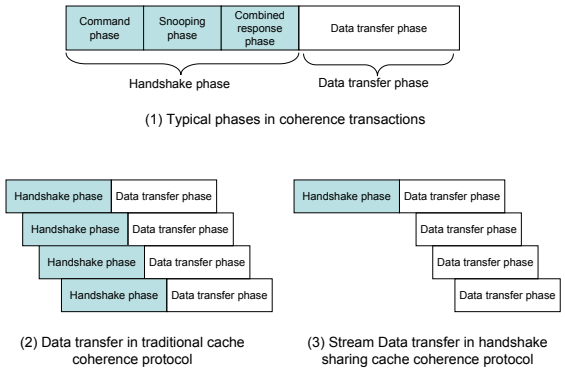

Figure 4. Coherence snooping action comparison.

5 Support for More Efficient Data Streaming

This section presents our initial exploration of improving data streaming within the CMP chip. In particular, we propose two extensions to the existing chip architecture: (1) *optimistic handshake sharing*, in which we use as few handshake actions as possible to set up the transfer of a stream; and (2) *data location speculation*, in which the location of an input data stream is predicted to further minimize handshake cost.

5.1 Optimistic Handshake Sharing

Optimistic handshake sharing attempts to eliminate unnecessary coherence snoop operations by exploiting the fact that an accelerator-requested data stream can be typically supplied from a single data source as discussed before. Figure 4-(1) recapitulates the actions described in Section 4 for a coherent data transfer. The transaction starts with a *command phase*, in which the request is sent to all snooping units. During the subsequent *snooping phase*, all snooping units perform tag lookup and respond to the snoop request. After that, in the *combined response phase*, the responses are combined to determine the data source/destination and the corresponding units are notified. These first three phases are collectively called the *handshake phase*. After the handshake, the actual data transfer takes place.

Based on this protocol, the transfer of a consecutive stream of data involves multiple (overlapping) handshake phases (as Figure 4-(2) shows) even when all the lines are in the same cache. The basic idea of optimistic handshake sharing is to allow subsequent requests in a stream to "share" the handshake effort of the first request and bypass the actual handshake actions and avoid unnecessary resource consumption. Let us consider reading a stream first and discuss the difference of writing a stream later.

When the accelerator starts a task by parsing the request block, it obtains the starting address and the length

of input data. With optimistic handshake sharing, the data engine issues a special *stream request* using the starting address and the length of the stream. Note that length information does not necessarily involve extra bits in the hardware interface. In our test systems, the accelerator driver always starts an accelerator stream at a cache line boundary. Thus, several least significant bits of the request address are always zero and can thus be re-purposed to indicate whether the request is a stream request and if so, the number of cache lines in the stream. With 128B cache lines, we have 7 extra bits for encoding, enough to describe streams of 4KB page size or smaller. Longer streams will be broken down at the page boundaries.

For implementation simplicity, the snooping logic handles the stream request just like a normal request. In other words, only the starting address of the stream is being considered in the snooping phase. At the end of the handshake phase, one cache is selected to provide data for the first cache line. In our empirical observation, the cache providing the first cache line typically has many, if not all, cache lines in the stream. By ignoring the rest of the stream in the snooping phase, we are optimistically relying on the cache to supply the entire stream.

Once the selected cache sends out the requested line, it activates the state machine to continue with the next line and repeats the process until the entire stream is supplied or the next line is not present in the cache. In the latter case, a special message is used to notify the accelerator. The accelerator can then start a new stream access request, picking up from where the last stream stops.

After issuing a stream request, the accelerator no longer needs to continuously issue individual line requests. However, it continues to reserve slots in its input buffers for the data replies. When the input buffers are exhausted, the data supplying cache's request to send more cache lines will be rejected (at the time of bus arbitration) and retried later.

Handling an output data stream injected directly into a cache has three differences from handling an input stream. First, the destination of the output data stream may be contained in the request block. In this case, the accelerator directly sends output data to the destination without going through a handshake phase even for the first line in the stream. Second, for some applications like compression/decompression, the length of the output stream is unknown at the beginning. In this case, we use a special flag attached to the last store request of the stream to inform the receiving cache of the completion of the current stream. Third, the receiving cache must make sure it already owns the line before it can accept a store that has not gone through the handshaking phase. In a MOESI protocol, it means the line must already be in an "M" or "E" state. This is mostly true in systems like PowerEN™ where the software uses special instructions to zero out entire cache lines to obtain exclusive ownership of a line without actually moving data. In systems where the output stream does not reside in the cache, the effect of optimistic handshaking sharing will be limited.

5.2 Data Location Speculation

Data location speculation attempts to further reduce the setup overhead of stream transfer by predicting which unit can supply the data and thus avoid unnecessary snooping of unrelated units. Since the snooping overhead is already reduced because of optimistic handshake sharing, speculating the data location has measurable benefits only for shorter streams, which are common in some networking processing applications.

We use a simple mechanism that predicts the core sending the task request to be where the data is located. To understand why such a simple prediction works well in the common case, recall that an application goes through three phases to launch a task on an accelerator: (1) initializing the input data and building the request block; (2) sending the request block to the accelerator via special instruction or memory-mapped I/O request; and (3) waiting for the completion signal. It is thus not a coincidence that the core sending the task to the accelerator still has the input data in its caches. The accelerator can simply predict the core that initiates the accelerator task as the source location for the input data. The ID of that core is readily available as it is part of the tag in the coherence transaction mentioned above (step 2).

To implement data location speculation, we extend the on-chip fabric to support snoop-less load requests that directly go from the requester (the accelerator) to the destination (the cache predicted to have the input data). If the predicted cache does not contain the requested data, a special NACK is sent back to the requesting accelerator. The accelerator will then resort to normal (snooping) requests. Such a miss occurs due to normal eviction of the data or thread migration (after initializing the data structure but before sending the task request to the accelerator). Our experience shows that this kind of migration is rare.

It is worth noting that the actual implementation is non-trivial. It first requires a filtering mask in the unit that broadcasts requests to all snooping units on the chip. The filtering mask indicates which units should pay attention to and which units should ignore the request. Additionally, each unit needs a piece of logic that processes the filtering mask for each incoming request and another piece of logic to generate the filtering mask for an outgoing request. The generation of the filtering mask is directly on the critical path and could add one fabric cy-

cle (in our case equivalent to two processor cycles) to the snooping path.

This extension is similar to many proposals found in literature where a minor benefit requires a minor, but non-trivial hardware change to achieve. From a practical standpoint, ideas such as this may not be adopted despite their advertised benefits.

6 Experimental Analysis

6.1 Methodology

We have built a full-system simulator for the development, analysis, and optimization of commercial applications to be run on PowerEN™. It is based on Mambo [2], and it includes function-accurate and cycle-accurate models for the crypto, compression, and pattern matching. Table 2 lists the major parameters of the simulated system. The analysis presented in this section is mostly based on the crypto accelerator. We have designed micro-benchmarks, where their input data sets mimic the behavior of real applications, and developed them using the PowerEN™SDK.

Parameter	Configuration
Coherence caches (L2 cache) Cache line size	4*2M Bytes 128 Bytes
Snooping queue in L2 cache	8 entries
Crypto accelerator	load queue : 16 entries Store queue : 12 entries Meta/Misc data queue: 16 entries Peak performance : 1/8X, 1/4X, 1/2X, 1X, 2X, 4X bus bandwidth
Cache to accelerator data access latency	100 cycles
Off-chip memory latency	300 cycles

Table 2. Simulation parameter configurations.

6.2 Cache Tag lookups

Figure 5 shows the numbers of L2 cache tag lookups performed by snoop operations. All numbers are normalized to the baseline architecture (conventional MOESI snooping protocol). Optimistic handshaking sharing reduces the number of cache tag lookups by up to 96.8% (31X reduction) over the conventional protocol. Each input stream used in the test contains 23 cache lines. The reduction in the number of tag lookups can be even more significant if the stream becomes longer or the chip integrates more on-chip snooping caches. The addition of location speculation can further reduce the number of tag lookups by up to 99.2% (125X reduction).

6.3 Bus Traffic

Figure 6 shows the normalized snooping traffic for various protocols, further broken down to different types (data transfer, normal requests, and retries etc). For simplicity, we count one snoop request as 6 transactions (1 command transaction, 4 partial response transactions due to 4 L2 caches, and 1 combined response transaction), and one cache line needs 4 data transactions. Note that one request results in many transactions depending on the real hardware implementation. Once again, the proposed design can eliminate most of the snoop requests in the baseline design, resulting in significantly reduced snoop traffic.

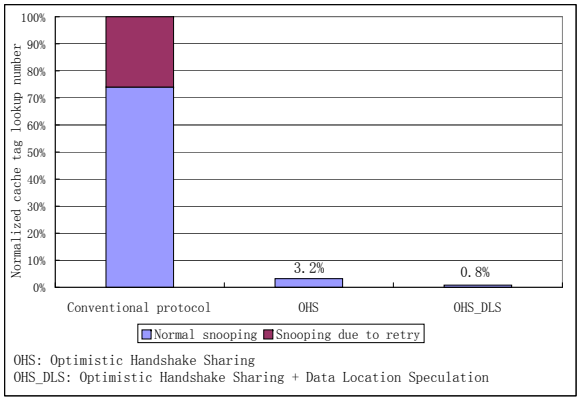

Figure 5. Cache tag lookup comparison.

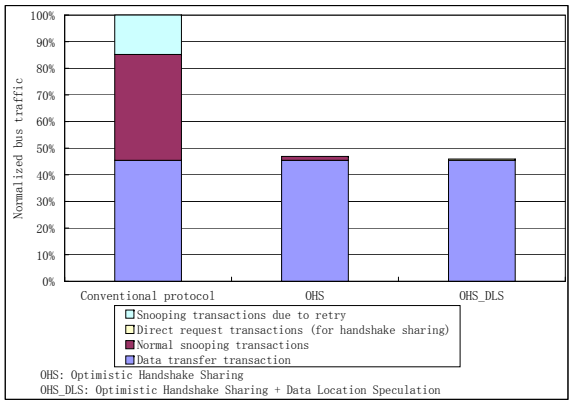

Figure 6. Bus snooping traffic comparison.

6.4 Execution Time

Figure 7 shows the execution time of a single accelerator task with the baseline protocol and the optimistic handshake sharing protocol. The x-axis is the processing capability of the accelerator relative to the available bandwidth. A value of 1/2 means that the accelerator processes data in half of the rate the on-chip fabric could provide. Not surprisingly, when there is sufficient bandwidth (1/8 to 1/2 cases), the proposed optimizations have little performance implications. To our surprise, it does not exhibit the same trend when the data bandwidth matches with the accelerator processing rate.

Further investigation reveals that a cache does not have a snoop queue large enough to hold a sufficient number of overlapping requests that would fully utilize the available bandwidth between the cache and the accelerator. By eliminating a majority of snoop requests, the handshake sharing protocol enables the cache to pipeline data onto the on-chip fabric without bubbles, resulting in a 13.7% performance gain. When the available bandwidth is smaller than the processing rate of the accelerator, the proposed design improves the performance up to 23% by offering better utilization of the on-chip fabric.

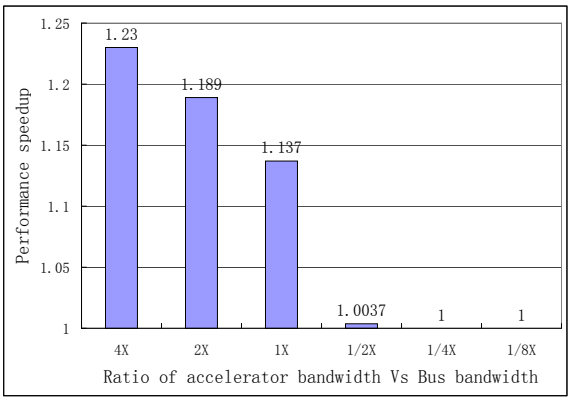

Figure 7. Sensitivity to different data bandwidth.

One of the main benefits of the proposed design is the reduction in power consumption. Unlike the performance models, we have not been able to validate the power models against the real hardware, thus we are unable to present power/energy numbers of different designs in this paper. However, it is our belief that the reduction of the power consumed by snoop operations related to the accelerators is proportionally to the performance numbers presented above.

7 Related Work

A few major microprocessor vendors, along with some standards organizations, have started developing standards for the communication interface and programming models of the accelerators. The examples include Intel's QuickAssist [16], AMD's Torrenza [4], and the Tightly Coupled Accelerators (TCAs) by AMD and HP.

Snoop filtering techniques were studied to eliminate unnecessary cache tag accesses for energy savings. For example, JETTY [12] uses an exclusive table to track the subsets of lines that are not cached and an inclusive table for the superset of the lines that are cached. These two tables are small, cache-like structures. A bus snoop request first probes these two tables for snoop filtering. IBM Blue Gene/P also implements stream registers and snoop caches to eliminate the vast majority of useless invalidations [14].

A destination-set prediction scheme was proposed to avoid indirect forwarding in directory protocol or to eliminate unnecessary snoops by dynamically capturing the sharing behavior of data access [1, 10].

Coarse-grain coherence protocols utilize dedicated hardware buffers to monitor the coherence status of large regions of memory, and use that information to avoid unnecessary broadcasts [3, 11, 18, 20].

These techniques are all optimized for cache-to-cache or core-to-core data movements. And in essence, they reduce the snooping related energy consumption by utilizing hardware structures smaller than the cache tag arrays for filtering or prediction or tracking information. While they can help an accelerator's data streams to some extent, they neither completely solve the problems that an accelerator's data streams have, nor fully exploit the unique feature of accelerator streams. First, maintaining the extra structures introduces additional hardware complexity and incurs additional power consumption. Second, these techniques often rely on history information tracking and require a training phase for accurate filtering or prediction. However, accelerator streams may be too short to pass the training phase, rendering these structures less useful.

8 Conclusion and Future Work

As more special-purpose processing units are being integrated onto microprocessor chips, much research remains to be done to optimize such architectures. This paper hopes to serve as a case study for illustrating the challenges and opportunities in such architectures and demonstrating the potentials of possible optimizations.

This paper focuses on the streaming patterns exhibited by a few on-chip accelerators. We have studied three types of accelerators, deducted four common patterns in data streams used by these accelerators, depicted the efficiency of supporting such streams with a traditional snooping-based cache coherence protocol, proposed two extensions for a improved design, and evaluated the performance of the proposed design quantitatively using a full-system simulator.

Our activity has thus far been limited to the IBM PowerEN^{TM}architecture. Many other on-chip accelerator architectures are possible targets for the same exploration and optimizations. Many other aspects of on-chip accelerators can also be explored and improved. Accelerator architecture design and optimizations are an important area that we hope will attract a lot of talent in the research community.

Our future work has two major aspects. First, we will continue to investigate the memory access behaviors of different types of accelerators. Second, we plan to extend this investigation from accelerator to other types of on-chip modules, including on-chip DMA controller, on-chip memory controller, as well core-to-core communications etc. We expect the insights obtained from these investigation to motivate more architectural innovations.

Acknowledgements

We would like to express our thanks to the IBM PowerEN™team. We would also like to thank Yu Zhang from IBM China Research Lab, and Jian Li from IBM Austin Research Lab, for the technical discussions and valuable comments on earlier stages of this work. We also wish to thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] E.E. Bilir, R.M. Dickson, H. Ying, M. Plakal, D.J. Sorin, M.D. Hill, and D.A. Wood. Multicast snooping: a new coherence method using a multicast address network. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 294–304, 1999.

[2] Patrick Bohrer, James Peterson, Mootaz Elnozahy, Ram Rajamony, Ahmed Gheith, Ron Rockhold, Charles Lefurgy, Hazim Shafi, Tarun Nakra, Rick Simpson, Evan Speight, Kartik Sudeep, Eric Van Hensbergen, and Lixin Zhang. Mambo: a full system simulator for the PowerPC architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):8–12, 2004.

[3] J.F. Cantin, M.H. Lipasti, and J.E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 246–257, June 2005.

[4] AMDs Holistic Vision for the Future: Torrenza Enables Platform Innovation. `http://www.instat.com/promos/07/dl/IN0703889WHT_DahenUd9.pdf`.

[5] H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson. Introduction to the wire-speed processor and architecture. *IBM Journal of Research and Development*, 54(1):3:1–3:11, January 2010.

[6] H. Peter Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proceedings of International Symposium on High-Performance Computer Architecture*, volume 0, pages 258–262, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[7] Intel. `http://www.intel.com/design/intarch/ep80579/index.htm`.

[8] IPSec. `http://datatracker.ietf.org/wg/ipsec/`.

[9] C. Johnson, D.H. Allen, J. Brown, S. Vanderwiel, R. Hoover, H. Achilles, C.-Y. Cher, G.A. May, H. Franke, J. Xenedis, and C. Basso. A wire-speed Power™ processor: 2.3GHz 45nm SOI with 16 cores and 64 threads. In *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC)*, pages 104–105, February 2010.

[10] M.M.K. Martin, P.J. Harper, D.J. Sorin, M.D. Hill, and D.A. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 206–217, June 2003.

[11] A. Moshovos. RegionScout: exploiting coarse grain sharing in snoop-based coherence. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 234–245, June 2005.

[12] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. JETTY: filtering snoops for reduced energy consumption in SMP servers. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 85–96, February 2001.

[13] RMI XLP Series processor. `http://www.rmicorp.com/assets/docs/XLP832_Product_Guide.pdf`.

[14] V. Salapura, M. Blumrich, and A. Gara. Design and implementation of the Blue Gene/P snoop filter. In *Proceedings of the 14th IEEE International Symposium on High Performance Computer Architecture*, pages 5–14, February 2008.

[15] M. Shah, J. Barren, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Sana, D. Sheahan, L. Spracklen, and A. Wynn. UltraSPARC T2: A highly-treaded, power-efficient, SPARC SOC. In *Proceedings of IEEE Asian Solid-State Circuits Conference*, pages 22–25, November 2007.

[16] Intel QuickAssist Technology. `http://www.intel.com/technology/platforms/quickassist/`.

[17] C.F. Webb. IBM Z10: The Next-Generation Mainframe Microprocessor. *Proceedings of IEEE Micro*, 28(2):19–29, March 2008.

[18] T.F. Wenisch, S. Somogyi, N. Hardavellas, Jangwoo Kim, A. Ailamaki, and Babak Falsafi. Temporal streaming of shared memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 222–233, June 2005.

[19] H. Yu, H. Franke, G. Biran, A. Golander, T. Nelms, and B.M. Bass. Stateful hardware decompression in networking environment. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 141–150, New York, NY, USA, 2008. ACM.

[20] J. Zebchuk, E. Safi, and A. Moshovos. A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 314–327, December 2007.