Memory-Side Acceleration for XML Parsing

Jie Tang, Shaoshan Liu, Zhimin Gu, Chen Liu, Jean-Luc Gaudiot

Memory-Side Acceleration for XML Parsing

Jie Tang[1], Shaoshan Liu[2], Zhimin Gu[1], Chen Liu[3] and Jean-Luc Gaudiot[4]

[1]*Beijing Institute of Technology, Beijing, China*
[2]*Microsoft, Redmond, WA*
[3]*Florida International University, Miami, Florida*
[4]*University of California, Irvine, California*

tangjie.bit@gmail.com,shaoliu@microsoft.com,zmgu@x263.net,chen.liu@fiu.edu, gaudiot@uci.edu

Abstract. As Extensible Markup Language (XML) becomes prevalent in cloud computing environments, it also introduces significant performance overheads. In this paper, we analyze the performance of XML parsing, identify that a significant fraction of the performance overhead is indeed incurred by memory data loading. To address this problem, we propose implementing memory-side acceleration on top of computation-side acceleration of XML parsing. To this end, we study the impact of memory-side acceleration on performance, and evaluate its implementation feasibility including bus bandwidth utilization, hardware cost, and energy consumption. Our results show that this technique is able to improve performance by up to 20% as well as produce up to 12.77% of energy saving when implemented in 32 nm technology.

1 Introduction

One of the main challenges in cloud computing environments is data exchange between different platforms. XML is emphasized for its language neutrality and application independency, and thus it has been adopted as the data exchange standard in cloud computing environments. When data is in XML, thousands of existing software packages can handle that data. Data in XML is universally approachable, accessible, and usable. Although XML exhibits many benefits, due to its verbosity and descriptive nature, XML parsing has incurred heavy performance penalties [1, 2]. Studies have shown that servers spend a significant portion of their execution time on processing XML documents. A real world example would be Morgan Stanley's Financial Services system, which spends 40% of its execution time on processing XML documents [3]. This is only going to get worse as XML data get larger and more complicated. Generally, in cloud computing environments, XML parsing is memory and computation intensive, it consumes about 30% of processing time in web service applications [4], and has become a main performance bottleneck in real-world database servers [5].

To improve performance of XML processing, many have proposed computation-side acceleration techniques. In this paper, we find out that memory accesses actually incur significant performance overheads in XML parsing. Therefore, different from previous studies which focus on computation acceleration, we propose to accelerate

XML parsing from memory side. Unlike computation acceleration, which has a strong dependency on the parsing model, memory-side acceleration is generic and can be effective across all parsing models. We believe the combination of computation-side and memory-side acceleration will largely relieve the performance pressure incurred by XML parsing. In our vision, cloud computing services can be hosted by one many-core chip, such as Intel's SCC chip [6]. Within this many-core chip, we should have at least one core act as the Data Exchange Frontend (DEF), which is dedicated to and optimized for XML parsing; and this DEF core should incorporate special instructions for computation-side acceleration as well as dedicated prefetchers for memory-side acceleration.

Within this context, we aim to answer three questions in this paper: first, what is the performance bottleneck of XML parsing? Second, can memory-side prefetching techniques improve the performance of XML parsing? Third, is it feasible to implement these techniques in hardware? The rest of this paper is organized as follows: in section 2, we review XML parsing techniques and related research work; in section 3, we discuss the methodology of our study; in section 4, we study the performance of XML parsing under native and managed environments; in section 5, we aim to answer the first question by evaluating the performance of XML parsing and identify the performance bottleneck of XML parsing; in section 6, we aim to answer the second question by delving into memory-side acceleration of XML parsing; In section 7, we aim to answer the third question by studying the implementation feasibility of the memory-side acceleration. At the end, we conclude and discuss our future work.

2 Background

In this section, we review XML parsing techniques as well as related studies on software and hardware acceleration of XML parsing.

2.1 XML Parsing Techniques

Based on how data is processed, there are two categories of XML parsing models: event-driven parser and tree-based parser. Event-driven parsers first parse the document, and then through callbacks, they notify client applications about any tag they find along the way. It transmits and parses XML infosets sequentially at runtime. As a result, Event-driven parsers don't cache any information and have an enviably small memory footprint. However, it does not expose the structure of the XML documents, making them hard to manipulate. Furthermore, according to how events are delivered, event-driven model can be divided into two classes: pull parser and push parser. In pull parsing, clients pull XML data when it is needed. In push parsing, an XML parser pushes XML data to the client as new elements are encountered. Simple API for XML (SAX) [7] is the industry standard for push based event-driven model. As shown in upper part of Figure 1, SAX processes the XML document and then pushes the XML information into Application in terms of SAX Events.

On the other hand, tree-based parsers read the entire content of an XML document into memory and create an in-memory tree object to represents it. Once in memory,

DOM trees can be navigated freely and parsed arbitrarily for the duration of the document processing, providing maximum flexibility for users. However this flexibility pays great costs of a potentially large memory footprint and significant processor requirements. Document Object Model (DOM) [8] is the official W3C standard for tree-based parser. As shown in bottom part of Figure 1, DOM parser processes XML data, creates an object-oriented hierarchical representation of the document and offer the full access to the XML data.

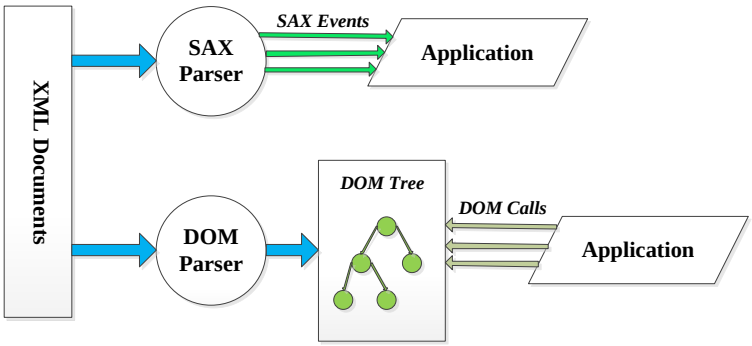

Figure 1: SAX and DOM Parsing Flow

2.2 Related Research Work

There have been several proposals on mitigating the performance overheads of XML processing. In software community, several researcher groups employed the concept of binary XML to avoid performance bottleneck of XML parsing [1, 9, 10]. Specifically, VTD-XML parser [10] parses XML documents and creates 64-bit binary format VTD records. However, the major shortcoming of this approach is that the parsed binary data can't be used by other XML applications directly. On the other hand, some researchers focus on parallelizing the parsing process. Pre-scanning based parallel parsing model [11] builds a skeleton of the XML document to guide partitioning of the document for data parallel processing. Also, in [12] researchers exploited the parallelism by dividing XML parsing process into several phases, so that they can schedule working threads to execute each parsing phase in a pipeline model. In addition, Parabix employs the SIMD capabilities of commodity processors to process multiple characters at the same time [13].

In the hardware community, based on the profiling analysis, researchers incorporate new instructions with special hardware support to speedup certain frequently-used operations of XML parsing [14]. In [15], researchers presented a technique to automatically map regular expressions directly onto FPGA hardware and implemented a simple XML parser for demonstration. Their technique could be not sufficient to solve all problems since XML syntax rule is not a regular language. XOE [16] use an Offload Engine to accelerate XML document parsing. Some fundamental parsing functionality like tokening is offloaded to XOE. XPA [17] is another XML Parsing Accelerator implemented on FPGA capable to do XML well-formed checking, schema validation, and tree construction. It can reach up to1 Cycle-Per-Byte throughput for XML parsing. Nevertheless, their design works only for tree-based parsers. As we show in the following sections, memory access is one of the major bottlenecks of XML parsing, thus we may be able to generate extra

performance gain from memory-side acceleration. In addition, unlike computation-side acceleration that targets specific parsing model, memory-side acceleration is generic and can be effective regardless of the parsing model.

3 Methodology

In this section, we discuss our methodology to study the performance of XML parsing, the effectiveness of memory-side acceleration, and implementation feasibility.

3.1 XML Parsers and Benchmarks

In order to make fair comparison, we choose XML parser implementations of both event-driven and tree-based model from Apache Xerces [18]. Apache Xerces provides SAX and DOM XML parsers, and it has implementations of these two models in both native (C++) and managed (Java) environments. This allows us to perform a thorough study to understand the performance of SAX and DOM models in different execution environments. As for inputs to the XML parsers, we have selected seven real world XML documents with varying sizes (from 1.4 KB to 113 MB) and complexities as input data and they are listed in Table 1. Specifically, *personal-schema* is a very simple document with flat structure, thus the parsing process is straightforward; on the other hand, *standard* is a long document with deep structures, which complicates the parsing process.

Table 1: benchmarks

Name	Size (KB)	Description
long	*65.7*	*sample XML SOAP file*
mystic-library	*1384*	*Information of library books*
personal-shema	*1.4*	*personal information data*
physics-engine	*1171*	*configuration data for physics simulation*
resume_w_xsl	*51.8*	*personal resume*
test-opc	*1.8*	*xml test file for web services gateway*
Standard	*113749*	*bank transaction records*

3.2 Prefetchers

In this study, we evaluate how different prefetching techniques impact the performance of XML parsing. In order to make a comprehensive investigation, we have selected eight hardware prefetchers, which utilize different techniques and algorithms. We summarize these prefetchers in Table 2: *cache hierarchy* indicates the coverage of the prefetching, which means if the prefetching is applied at L1 cache, L2, cache, or both; *prefetching degree* suggests whether the aggressiveness of the prefetcher is statically or dynamically adjusted. Usually, the dynamic prefetching degree can adapt itself to the phase change of the application so as to produce more efficient prefetching; *trigger L1* and *trigger L2* respectively show the trigger set for covered cache hierarchy respectively, in this case *demand access* stands for access requests from upper memory hierarchy regardless whether it is a miss or hit and *N/A* means no prefetching is applied. Since *demand access* trigger set contains more

opportunity to invoke the prefetching, it always yields more aggressiveness. Besides, all selected prefetchers can filter out redundant access requests.

Table 2: Summary of Prefetchers

	Cache Hierarchy	Prefetch Degree	Trigger L1	Trigger L2
n1	*L1 & L2*	*dynamic*	*Miss*	*Access*
n2	*L1*	*Static*	*Miss*	*N/A*
n3	*L1 & L2*	*dynamic*	*Miss*	*Miss*
n4	*L1*	*Static*	*N/A*	*N/A*
n5	*L2*	*Static*	*N/A*	*Miss*
n6	*L1 & L2*	*dynamic*	*Miss*	*Miss*
n7	*L2*	*Static*	*Miss*	*Access*
n8	*L2*	*Static*	*N/A*	*Access*

The aggressiveness of the prefetching is the co-production of all these four metrics and prefetching algorithms. The first prefetcher *n1* can tolerate out of order memory accesses by making prefetching based on the recent memory access pattern. The second prefetcher *n2* exploits various localities in both local and global cache-miss streams, including global strides, local strides and scalar patterns. A multi-level prefetching framework is applied in *n3*: it uses a sequential tagged prefetcher at L1 cache and either an adaptive prefetcher or a sequential tagged prefetcher at L2 cache. With the observation that memory accesses often exhibit repetitive layouts spanning large memory region, *n4* is the optimized implementation of Spatial Memory Streaming (SMS) including a novel mechanism of pattern bit-vector rotation to reduce SMS storage requirement. Combining the storage efficiency of Reference Prediction Tables and high performance of Program Counter/Delta Correlation (PC/DC) prefetching, *n5* can substantially reduce the complexity of PC/DC prefetching by avoiding expensive pointer chasing and re-computation of the delta buffer. The sixth prefetcher *n6* applies a hybrid stride/sequential prefetching schema at both L1 and L2 cache levels. Metrics such as prefetcher accuracy, lateness and memory bandwidth contention are fed back to adapt the aggressiveness of prefetching. By understanding and exploiting a variety of memory access patterns, *n7* combines global history buffer and multiple local history buffers to improve the coverage of prefetching. Finally, *n8* is a stream-based prefetcher with several enhancement techniques including constant stride optimization, noise removal, early launch of repeat stream and dead stream removal.

3.3 Performance and Memory Modeling

To study the performance of the memory-side acceleration, we utilize CMP$IM [19], a binary-instrumentation based cache simulator developed by Intel. CMP$IM is able to characterize cache performance of single-threaded, multi-threaded, and multi-programmed workloads. The simulation framework models an out-of-order processor with the basic parameters as outlined in Table 3.

Table 3: Simulation Parameters

Frequency	1 GHz
Issue Width	4
Instruction Window	128 entries
L1 Data Cache	32KB, 8-way, 1cycle
L1 Inst. Cache	32 KB, 8-way, 1cycle
L2 Unified Cache	512 KB, 16-way, 20 cycles
Main Memory	256 MB, 200 cycles

To understand the implementation feasibility of these memory-side accelerators, we also study the energy consumption of these designs. To model the energy consumption of these prefetchers, we utilize CACTI [20], an energy model which integrates cache and memory access time, area, leakage, and dynamic power. Using CACTI, we are able to generate energy parameters of different storage and interconnect structures implemented in different technologies. Note that the overall system energy consumption consists of two sources: static power and dynamic power. Static power is generated by the leakage current of the transistors, and it persists regardless of whether the transistors are actively switching or not. On the other hand, dynamic power is incurred only when the transistors are actively switching. In this paper, we use CACTI to model both static and dynamic energy to evaluate the implementation feasibility of memory-side accelerators.

4. Native vs. Managed Execution

In this section, we study the performance of XML parsing in both managed and native environments. We executed XML parsers on a dual-core machine running at 2.2 GHz and used the Intel Vtune analysis tool [21] to capture the overall execution time. The results are shown in Figure 2, in which we take the performance of native execution as the baseline. The x-axis shows the seven benchmarks and the y-axis shows the percentage of the excess execution time incurred by the managed layer (in this case JVM). It is obvious that when parsing with SAX model, managed execution produces high performance overhead. For instance, when parsing *test-opc* and *mystic-library*, the managed middle layer contributes 41.67% and 38% performance overhead respectively. Even in the best case, *long*, the middle layer still incurs 20.73% performance overheads. The situation is even worse when using DOM parsing model. Even the best case has incurred 25.93% performance overheads. In the worst case, *test-op*, it incurs up to 52.08% performance degradation.

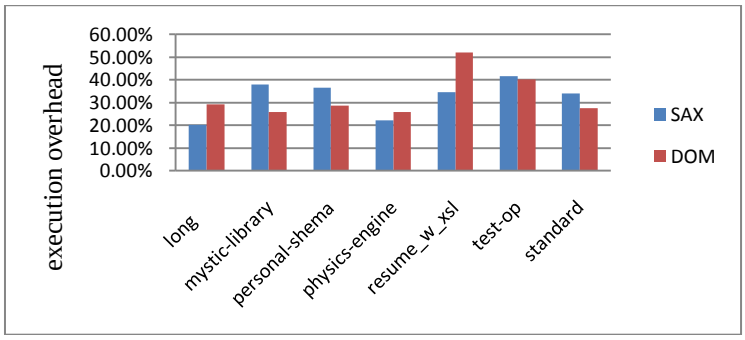

Figure 2: Managed Execution vs. Native Execution

Although managed environment is able to reduce development time, in cloud computing environments, XML parsers reside at the data exchange frontend and many other components in the system may have dependency on the outputs of the XML parsers. Therefore, the performance overheads incurred by the managed layer would largely hinder the performance of the whole system. This result indicates that managed execution of XML parsing is not suitable in cloud computing environments and we focus on native execution of XML parsing in the rest of this paper.

5 Performance Analysis of XML Parsing

In this section we aim to determine the performance bottleneck of XML parsing by studying the throughput of XML parsing at different parts of the system, including network data exchange, disk I/O, and memory accesses. Figure 3 shows the data flow of XML parsing: first, data is loaded from either network or local hard disk. Then, data flows into the memory subsystem: main memory, L2 and L1 caches. At the end, the processor fetches data from cache and performs the actual XML parsing computation.

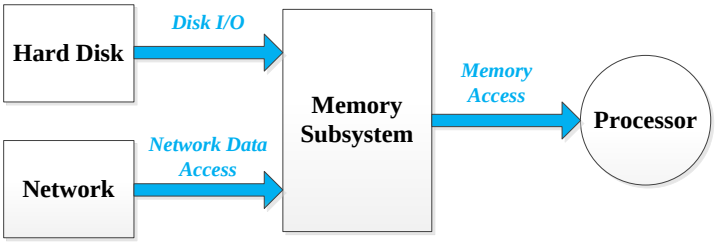

Figure 3：Data flow of XML data parsing

5.1 Network Data Exchange

Some previous work has demonstrated that network data exchange would incur significant performance overheads [23, 27]. In this subsection, we measure the data exchange throughput of different cloud data services. In Table 4，we summarize our measurements of two popular categories of cloud data services: *Content Distribution*

Network (CDN) and *Cloud Storage*; and for each category, we measured the data exchange throughput of four service providers. Note that CDN services contain several copies of data in network to maximize bandwidth, whereas Cloud Storage services provide online storage where people can require their storage capacity for their data hosting needs. On average, the data exchange throughput to CDN services can reach 29.26 Mb/s. When employing the CDN service provided by Amazon CloudFront, the rate can reach 48.85 Mb/s. On the other hand, the average data exchange throughput to Cloud Storage services is 12.56 Mb/s and its best case, provided by Amazon S3 – US East, can reach 21.8 Mb/s. In our experiment, our machine contains a 100 MB/s Network Interface Card and network it connected to has a bandwidth limit of 100Mb/s, which is far greater than the throughput provided by cloud data services. That indicates the network I/O interface is not fully utilized, and thus the network interface is not likely to be the bottleneck of XML parsing operations.

Table 4: Cloud Service Data Rate

CDN Service			***Cloud Storage Service***	
Provider	*Rate (Mb/s)*		*Provider*	*Rate (Mb/s)*
Akamai CDN	27.50		Amazon S3 - US East	21.80
Amazon CloudFront	48.85		Amazon S3 - US West	10.31
Cotendo CDN	27.72		Azure-South Central US	6.97
Highwinds CDN	12.97		Nirvanix SDN	11.17
AVG	29.26		AVG	12.56

5.2 Disk data loading

In order to study the disk I/O throughput, we used the XPerf Performance Analyzer tool [26] to capture disk I/O throughput when running the XML parsers using the *standard* benchmark, and the collected results are shown in Figure 4: the x-axis shows the execution timeline in seconds and the y-axis shows the amount of triggered disk I/O during the execution. The gray curve overlaid on top of the bar diagram shows the CPU usage information. The peak of the curve means the CPU is fully utilized. Figure 4 shows that most of the time disk I/O is in the underutilized state; that is to say that the I/O subsystem rarely needs to reach its full capacity. Besides, when looking into the overlaid CPU usage curve, it shows that most of the time the CPU is fully utilized, running at 100%. Once in a while, the CPU utilization drops down, probably due to high-latency memory accesses. It indicates that in this case most of the time CPU is fed with enough data from the I/O subsystem. Based on these observations, it can be concluded that disk I/O is also not likely to be the bottleneck of XML data processing. We also ran the XML parsers with other benchmarks shown in Table, and the results were similar.

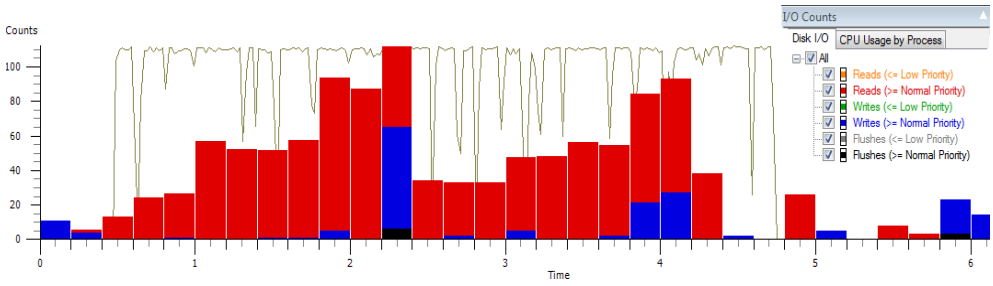

Figure 4: Disk I/O counts and CPU Usage

5.3 Data loading from memory side

Finally, we studied the overhead produced by memory data loading stage. Here, memory data loading refers to the data flow starting from main memory, going through each cache layer and finally fetched into CPU. Making a comparison, we measured the CPI (cycle per instruction) of *Speed-test*, which is a computation intensive CPU stress test application with negligible memory access; and we measured the CPI of native XML parser using the *standard* benchmark, which is the large XML document with a lot of memory accesses. The CPI of *Speed-test* is 0.80. Using the SAX parser, the CPI of *standard* is 1.27, which introduces nearly 50% overheads compared to *Speed-test*; using the DOM parser, the CPI of *Standard* becomes 1.42, which nearly doubles that of *Speed-test*. In addition, when using other benchmarks shown in Table 1, we obtained similar performance data as that of *standard*.

As a further validation, we measured the miss count per kilo instructions (MPKI) of both L1 and L2 cache layers, which are nearly 10 and 2 respectively. That is to say that for every 1000 instructions there comes about ten L1 and two L2 cache misses. The large number of cache misses mainly contributes to the CPI increase of XML parsing. Compared with the CPI of *Speed-test*, the extra cycles consumed by XML parsing may indicate that memory data loading stage incurs a significant amount of overhead to the execution.

5.4 Summary

In summary, the results from the previous three subsections show the following: first, network I/O throughput can easily reach over 15 MB/s, and this is far below the 100 MB/s network bandwidth limit, showing that network I/O is far from being stressed and network data exchanging is not likely to be the bottleneck of XML parsing. Second, our experiment results show that the disk I/O subsystem is under-utilized most of the time, which means disk data loading of XML data parsing is within the coverage of disk I/O subsystem and cannot be the bottleneck of execution as well. At last, comparing CPI data of XML parsing workloads and a CPU stress test, we have found that in some case the CPI of XML parsing almost double that of the CPU stress test. Upon further analysis, we have found that the high cache miss rate on both L1 and L2 caches is the main contributor to this CPI increase. These results indicate that the performance bottleneck of XML parsing is memory data loading stage; in other word, the overheads introduced from memory subsystem really hit the pain point of

XML data parsing. Therefore, in order to speed up the XML parsing execution, it is imperative to turn around the focus of acceleration and reduce the overheads incurred by the memory subsystem.

6 Memory-Side Acceleration

We have identified that memory accesses impose significant overheads in XML parsing workloads. Similarly, a study released by Intel verifies that memory accesses contribute to more than 60% execution cycles of the whole parsing process [22]. Furthermore, another empirical study done by Longshaw *et al.* has shown that loading an XML document into memory and reading it prior to parsing may take longer than the actual parsing time [24]. Consequently, instead of optimizing specific computation of parsing model, we explore acceleration from memory side; that is to say, accelerate the XML data loading stage.

Table 5 summarizes the reduction of cache misses as a result of applying the prefetchers (please refer to section 3.2 for the details of the prefetchers). Note that different prefetchers may target different cache levels; in this table, we show the cache miss reduction of the lowest level cache that the prefetcher is applied to. For example, n1 is applied to both L1 and L2 caches, we show the cache miss reduction of L2 cache; n2 is applied to only L1, so we show the cache miss reduction of L1 cache. The results indicate that prefetching techniques are very effective on XML parsing workloads, as most prefetchers are able to reduce cache miss by more than 50%. In the best case, n3 is able to reduce L2 cache miss by 82% in SAX and 85% in DOM.

In Figure 5, we show how the cache miss reduction translates into performance improvement on SAX parsing: it shows the performance of the eight prefetchers (n1-n8) as well as the average performance. The X-axis lists the seven benchmarks we used and the Y-axis shows the percentage of performance improvement (in terms of execution time reduction). The results indicate that prefetching techniques are able to improve SAX parsing performance by up to 10%. For instance, on average, the parsing time of *personal-schema* has been reduced by 7.24%. Even in the worst case, *standard*, prefetchers are still able to reduce execution time by 3%. Looking into each prefetching technique, we observe that n3 shows greatest power in improving the performance by 2.58% to 9.72% across different benchmarks. This is because n3 is the most aggressive prefetcher and covers both L1 and L2 cache level, thus resulting in the best average performance.

Similarly, Figure 6 summarizes the performance impact of prefetching on DOM parsing. The results indicate that prefetching techniques are able to improve DOM parsing performance by up to 20%. For instance, when averaging the results, memory-side acceleration produces 13.74% execution cycle reduction for *mystic-library*. It is obvious that the most effective prefetcher is still n3: even in the worst case, n3 can still reduce execution time by 6%. Note that different from SAX parsing, DOM must construct inner data structure in memory for all elements. The bigger the document is the more space it would consume, and the more cache miss it would induce. As a result, large sized benchmarks such as *mystic-library*, *physics-*

engine and *standard* can get a higher performance gain from memory side acceleration from 7.65% up to 13.75%. These results confirm that memory-side acceleration can be effective regardless of the parsing models.

Table 5: Cache Miss Reduction

	n1	n2	n3	n4	n5	n6	n7	n8
SAX	*0.69*	*0.43*	*0.82*	*0.82*	*0.51*	*0.4*	*0.73*	*0.77*
DOM	*0.77*	*0.52*	*0.85*	*0.85*	*0.61*	*0.52*	*0.77*	*0.84*
Cache Level	*L2*	*L1*	*L2*	*L1*	*L2*	*L2*	*L2*	*L2*

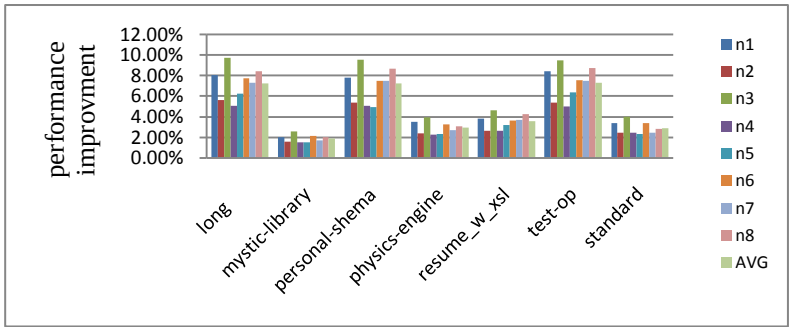

Figure 5: Performance improvement for SAX parsing

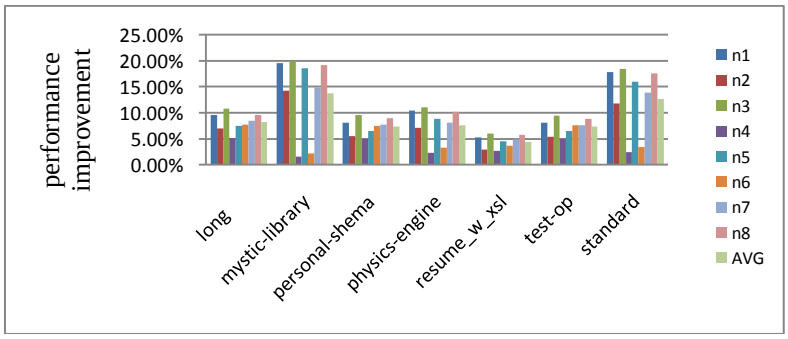

Figure 6: Performance improvement for DOM parsing

7. Implementation Feasibility

By now we have shown that memory-side acceleration can significantly improve XML parsing performance. However, the conventional wisdom is that prefetching requires extra hardware resource, competes for limited bus bandwidth, and consumes more energy. Thus, many would argue that it is not worthwhile to implement memory-side accelerators for XML parsing. In this section, we address these doubts

by validating the feasibility of memory side acceleration in terms of bandwidth utilization, hardware cost and energy consumption.

7.1 Bandwidth Utilization

Contention for limited bus bandwidth often leads to serious performance degradation. Prefetching techniques result in extra bus traffic and thus require extra bus bandwidth. If the application itself has used up all the bus bandwidth, then the contention brought in by memory-side acceleration might hinder rather than improve performance. Hence, we study the bandwidth utilization of XML parsing workloads and the results are summarized in Table 6. The results show that bus bandwidth utilization without prefetching is far away from exhaustion. On average, bus utilization for SAX and DOM parsing are only 3.72% and 5.51% respectively. This indicates the performance of XML parsing is hurt by the latency but not the throughput of memory subsystem, and thus confirming that prefetching would be effective.

Table 6: bandwidth consumption without prefetching

	SAX	DOM
long	*4.09%*	*5.55%*
mystic-library	*4.38%*	*7.46%*
personal-shema	*4.94%*	*6.31%*
physics-engine	*4.07%*	*6.08%*
resume_w_xsl	*0.97%*	*1.03%*
test-opc	*1.01%*	*5.25%*
Standard	*6.58%*	*6.89%*

7.2 Hardware Cost and Energy Consumption

We summarize the hardware cost of the eight prefetchers in Table 7. On average, these prefetchers require about 28000 bits of memory space. For instance, n6 consists of a 14080 bits L1 prefetcher, a 4096 bits L2 prefetcher and eight 20 bits counters producing 32416 bits hardware cost. All of their hardware cost is below 4KB, which is not a significant amount of hardware resource in modern high-performance processor design.

Table 7: hardware cost of prefetcher

n1	*32036 bits*	*n2*	*20329 bits*	*n3*	*20787 bits*	*n4*	*30592 bits*
n5	*25480 bits*	*n6*	*32416 bits*	*n7*	*30720 bits*	*n8*	*32768 bits*

Next we study how these memory-side accelerators impact system energy consumption. Using our simulation framework consisting of CMP$IM and CACTI, we can generate energy parameters of different storage and interconnect structure implemented in different technologies. Here, we focus on the implementation with 32 nm technologies and the results are summarized in Figures 7 and 8. In these Figures, we select energy consumption with no prefetching as our baseline, thus a positive number indicates that the prefetcher consumes extra energy, and a negative number indicates otherwise. Note that in 32 nm technology, static energy is comparable to dynamic energy [25]. The prefetchers generate extra memory requests and bus transactions, thus adding dynamic energy consumption. On the other hand,

prefetchers accelerate XML parsing execution, resulting in reduction of static energy consumption. If the static energy reduction surpasses the dynamic energy addition, then the prefetcher results in overall system energy reduction.

As shown in Figure 7, in SAX parsing, most prefetchers lead to more energy consumption: It is due to the increase of dynamic energy dissipation coming from excess memory accesses incurred by prefetching. Nevertheless, looking into details, n5 always leads to energy efficiency, resulting in 1% to 4.5% energy saving across the benchmarks. Similarly, n1 results in energy saving in about half of the cases. This is because n1 and n5 are relatively conservative prefetching techniques: they either prefetch at only one cache level or prefetch a small amount data each time.

In Figure 8, we summarize how acceleration impacts energy consumption in DOM parsing. Identical with Figure 7, n5 is still the most energy efficient prefetcher which archives 12.77% energy saving in *mystic-library*. Even when running its worst case, *resume_w_xsl*, n5 can still reduce overall energy by almost 3%. Different from the results in SAX parsing, most prefetchers become energy-efficient in many cases due to their ability to further reduce execution time in DOM parsing. Note that static energy is the product of static power and time, since static power is constant, by reducing execution time, we can reduce static energy as well.

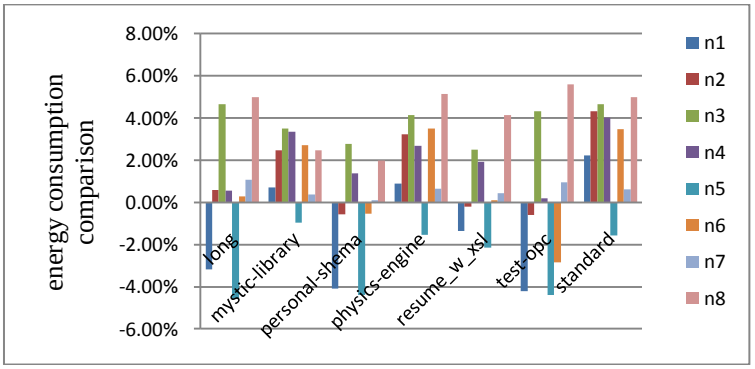

Figure 7: Energy consumption of SAX parsing

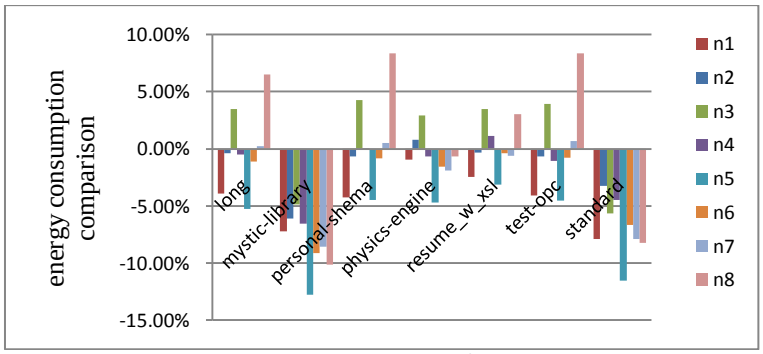

Figure 8: Energy consumption of DOM parsing

8. Conclusions

XML has been adopted in cloud computing environments to address the data exchange problem. While XML has brought many benefits, XML parsing also imposes heavy performance penalties. As XML processing often resides at the data exchange frontend, different parts of the system would have dependency on the outputs of XML parsing, making it critical to system performance. While previous research work has focused on computation acceleration of XML parsing, we have identified memory access as one of the performance bottlenecks. Motivated by this finding, in this paper, we have done a study on the effectiveness and feasibility of memory-side acceleration of XML parsing. The results are encouraging, we have demonstrated that memory-side acceleration is effective regardless of the parsing model and is able to reduce cache miss by up to 80%, which translates into up to 20% of performance improvement.

On implementation feasibility, we have identified that XML parsing performance is hurt by the latency but not by the throughput of the memory subsystem, thus verifying that memory-side acceleration is not likely result in resource contention. In addition, we have shown that the memory-accelerators require an insignificant amount of extra hardware resources, and more importantly, in many cases they are indeed able to reduce the overall system energy consumption. These results confirm that memory-side acceleration of XML parsing is not only effective but also feasible.

In our vision, cloud computing services can be hosted by one many-core chip, and within this many-core chip, one or more cores act as the Data Exchange Frontend (DEF). Our next step is to incorporate memory-side and computation-side accelerations into the DEF cores and evaluate its performance in many-core environments.

Acknowledgements

This work is supported in part by the National Science Foundation under Grant No. CCF-1065448, the National Natural Science Foundation of China under Grant No. 61070029, as well as the China Scholarship Council. Any opinions, findings, and conclusions as well as recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

1. K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of soap performance for scientific computing. In Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11, 2002
2. M. R. Head, M. Govindaraju, R. van Engelen, and W. Zhang. Grid scheduling and protocols—benchmarking XML processors for applications in grid Web services. In SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, page 121, New York, NY, USA, 2006. ACM Press.
3. P. Apparao, R. Iyer, R. Morin, N. Naren, B. Mahesh, D. Halliwell, and W. Steiberg, "Architectural characterization of an XML-centric commercial server workload," in 33rd International Conference on Parallel Processing, 2004
4. P. Apparao and M. Bhat. A detailed look at the characteristics of XML parsing. In BEACON '04: 1st Workshop on Building Block Engine Architectures for Computers and Networks, 2004

5. M. Nicola and J. John, “XML parsing: A threat to database performance,” in Proceeding of the 12th International Conference on Information and Knowledge Management, 2003
6. Intel Single-Chip Cloud Computer, http://techresearch.intel.com/ProjectDetails.aspx?Id=1
7. SAX Parsing Model: http://sax.sourceforge.net
8. W3C, “Document object model (DOM) level 2 core specification.” http://www.w3.org/TR/DOM-Level-2-Core
9. K. Chiu, T. Devadithya, W. Lu, and A. Slominski. A binary XML for scientific applications. In Proceedings of e-Science 2005. IEEE, 2005.
10. XimpleWare, “VTD-XML: The Future of XML Processing,” (accessed 10 Mar 2007), http://vtdxml. sourceforge.net.
11. W. Lu, K. Chiu, Y. Pan, A Parallel Approach to XML Parsing,In Proceedings of The 7th IEEE/ACM International Conference on Grid Computing, Barcelona, Spain, Sept 2006
12. Michael R. Head and Madhusudhan Govindaraju. "Approaching a Parallelized XML Parser Optimized for Multi-Core Processor”. SOCP’07, June 26, 2007, Monterey, California, USA. ACM
13. R. D. Cameron, K. S. Herdy, D. Lin, High Performance XML Parsing Using Parallel Bit Stream Technology, In Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research, Ontario, Canada, Oct 2008
14. L. Zhao, L. Bhuyan, Performance Evaluation and Acceleration for XML Data Parsing, In Proceedings of the 9th Workshop on Computer Architecture Evaluation using Commercial Workloads, Texas, USA, 2006
15. J. Moscola, J. W. Lockwood, Reconfigurable Content-based Router using Hardware-Accelerated Language Parser, In theACM Transactions on Design Automation of Electronic Systems, Vol.13, 2008
16. B. Nag, “Acceleration techniques for XML processors,” in XML Conference & Exhibition, November 2004.
17. Zefu Dai, Nick Ni, Jianwen Zhu. A 1 Cycle-Per-Byte XML Parsing Accelerator. FPGA’10, 2010
18. Apache Xerces: http://xerces.apache.org/index.html
19. A. Jaleel, R. S. Cohn, C. K. Luk, and B. Jacob. CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator. In MoBS, 2008
20. P. Shivakumar, N.P. Jouppi, CACTI3.0: an integrated cache timing, power, and area model, WRL Research Report 2001
21. Intel Vtune, http://software.intel.com/en-us/intel-vtune/
22. XML Parsing Accelerator with Intel® Streaming SIMD Extensions 4 (Intel® SSE4), December 15, 2008. http://software.intel.com/en-us/articles/xml-parsing-accelerator-with-intel-streaming-simd-extensions-4-intel-sse4/
23. S. Lee and W.W. Ro, “Accelerated Network Coding with Dynamic Stream Decomposition on Graphics Processing Unit”, The Computer Journal
24. A. Longshaw, “Scaling XML parsing on Intel architecture,” Intel Software Network Resource Center, November 2008. http://www.developers.net/intelisnshowcase/view/537
25. Power vs. Performance: The 90 nm Inflection Point, http://www.xilinx.com/publications/archives/solution_guides/power_management.pdf
26. Windows Performance Analysis Tool, http://msdn.microsoft.com/en-us/performance/cc825801
27. K. Park, J-S Park, and W.W. Ro, “On Improving Parallelized Network Coding with Dynamic Partitioning”, IEEE Transactions on Parallel and Distributed Systems, Vol. 21, No. 11, pp. 1547-1560, Nov. 2010