1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

# Acceleration of XML Parsing Through Prefetching

Jie Tang, Shaoshan Liu, Chen Liu, Zhimin Gu, and Jean-Luc Gaudiot, *Fellow, IEEE*

**Abstract**—Extensible Markup Language (XML) has become a widely adopted standard for data representation and exchange. However, its features also introduce significant overhead threatening the performance of modern applications. In this paper, we present a study on XML parsing and identify that memory-side data loading in parsing stage incurs a significant performance overhead, at the same level as computation. Hence, we propose memory-side acceleration with the incorporation of data prefetching techniques, which can be applied on top of computation-side acceleration to speed up the XML data parsing. To this end, we study the impact of our proposed scheme on the performance and energy consumption and find out that it is able to improve performance by up to 20% as well as produce up to 12.77% of energy saving when implemented in 32nm technology. In addition, we implement a prefetcher on FPGA platform in an effort to evaluate its implementation feasibility in terms of area and energy overhead.

**Index Terms**— XML Parsing, Prefetching, Hardware Acceleration.

For Peer Review Only

## 1 Introduction

Extensible Markup Language (XML) is emphasized for its language neutrality, application independency and flexibility, and thus has been adopted as the standard in data exchange and representation. For example, the International HapMap project uses XML schemas to represent the common patterns in human DNA sequence [6]. Although XML is prevalent with many benefits, due to its verbosity and descriptive nature, XML parsing has introduced heavy performance overhead [1, 2]. Generally, XML parsing is both memory and computation intensive. It consumes about 30% of processing time in web service applications [4], and has become a major performance bottleneck in database servers [5]. 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 even worse as XML dataset get larger and more complicated.

To improve the performance of XML processing, most existing proposals are dedicated to make acceleration from computation side. However, in this paper, we demonstrate that memory access acceleration is equally (if not more) important compared to computation acceleration. Therefore, different from previous computation acceleration studies, we propose to accelerate XML parsing from the memory-side with the incorporation of data prefetching techniques. Unlike computation-side acceleration, which has a strong dependency on the parsing model, memory-side acceleration is generic and can be applied irrelevant of the parsing model underneath. In addition, its combination with computation-side acceleration will largely relieve the performance pressure incurred by XML parsing.

Note that our long-term goal is to develop a dedicated XML parsing core, which consists of a computation accelerator and dedicated prefetching logics. With the computation accelerator, we can develop specialized instructions to accelerate the computation of XML parsing; with the dedicated prefetching logics, we would avoid data pollution from other workloads, and keep feeding data to the computation accelerator. This core should eventually be one of multiple specialized cores in a heterogeneous many-core chip, and acts as the Data Exchange Frontend (DEF), efficiently (in terms of power and performance) parsing the incoming XML data, and then passing the output to other cores for further processing.

To this end, we address the following questions in this paper: first, is memory access overhead indeed the same performance bottleneck for XML data parsing as that from computation? Second, how much can memory-side prefetching techniques improve the performance of XML parsing? Third, is it feasible to implement these techniques in hardware considering its energy and hardware cost?

The rest of this paper is organized as follows: before answer those three questions, in Section 2, we first review related research work on XML parsing techniques and prefetching techniques. In Section 3, we describe the methodology of our study. Then in Section 4, we present a performance study of XML parsing under both native and managed environments. To answer the first question, in Section 5, we evaluate the performance of XML parsing and identify the performance bottleneck. To answer the second question, in Section 6, we delve into our proposal:


- *Jie Tang is with the School of Computer, Beijing Institute of Technology. E-mail: tangjie.bit@gmail.com*
- *Shaoshan Liu is with Microsoft. E-mail: shaoliu@microsoft.com*
- *Chen Liu is with the Department of ECE. Florida International University, E-mail: cliu@fiu.edu*
- *Zhimin Gu is with the School of computer, Beijing Institute of Technology. E-mail: zmgu@x263.net*
- *Jean-Luc Gaudiot is with the Department of EECS, University of California, Irvine. E-mail: gaudiot@uci.edu*

memory-side acceleration of XML parsing. We propose to use hardware prefetching to break the performance bottleneck on the memory side and messure what the improvement is. To answer the third question, in Section 7, we study the implementation feasibility of the memory-side acceleration. As a further verification, in Section 8, we make a case study by implementing one prefetching engine on FPGA platform and exploiting its hardware overhead as well as energy consumption. Finally, we conclude and discuss our future work in Section 9 and Section 10 respectively.

## 2 Backgrounds

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

### 2.1 The XML Parsing Process

XML parsing is a process that scans through the input XML documents, breaks them into small elements, and builds corresponding inner data representation. It is a pre-requisite for any processing of an XML document because an XML document has to be parsed before any other operations can be performed. However, XML parsing is also very expensive due to the high overhead incurred by both computation and memory access.

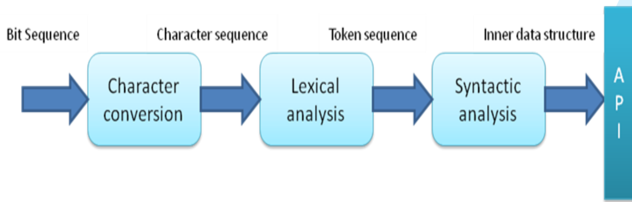


Figure1: XML Parsing Process

Usually, XML data parsing consists of three steps: *character conversion*, *lexical analysis* and *syntactic analysis*, as shown in Figure 1. The first parsing step, *character conversion*, converts a bit sequence from source XML document into the character sets that are host programming language dependent. The second parsing step, *lexical analysis*, partitions the character sets into subsequences called tokens, like *start element*, *text*, and *end element*. Each token is defined by a regular expression in the World Wide Web Consortium (W3C) XML specifications [8]. The third parsing step, *syntactic analysis*, verifies the structure of tokens by checking that they have been properly nested. It is usually implemented by *pushdown automaton* (PDA). After syntactic analysis, the PDA organizes tokens into different data representations available for subsequent accesses or modifications via various application programming interfaces (APIs) provided by different parsing models. The first two steps stay the same among different parsing models. However, the third step, *syntactic analysis*, exhibit variable behaviors when different parsing model is applied [6].

### 2.2 XML Parsing Modeling

Based on how inner data structure is represented, there exist two categories of XML parsing models: event-driven parser and tree-based parser.

On one hand, event-driven parser simply parses the document and associates any tag it finds along the way with corresponding event, including the start and end of the document, finding a text node, finding child elements, and hitting a malformed element. It transmits and parses XML infosets sequentially at runtime. The parser itself does not store any information of the XML document, so that the application can just access partial data before parsing is completed. As a result, event-driven parser has an enviably small memory footprint and low latency, making it suitable for streaming or forward-only applications. Event-driven model can be further divided into two classes: pull parser and push parser, according to the parser-application interaction. Simple API for XML (SAX) [7] adopts the push model, which uses callback functions to report all the events from the parser to the application. In contrast, StAX [34] adopts the pull model, in which clients pull XML data when it is needed so that it can skip uninterested events. As shown in upper part of Figure 2, SAX parses the XML document and then pushes the XML information into application in terms of SAX events.

On the other hand, tree-based parser reads the entire content of an XML document into memory and creates an in-memory tree structure to represents parent-child-sibling information. Only after parsing is complete, constructed trees can be navigated freely and parsed arbitrarily for the duration of the document processing, which makes this parser suitable for massive and frequent updates. This flexibility, however, comes at a great cost of potentially large memory requirement and significant access delay, especially when large document is processed. Document Object Model (DOM) [8] is the official W3C standard for tree-based parser. As shown in bottom part of Figure 2, DOM parser processes XML data, creates an object-oriented hierarchical representation of the document and offers the full access to the XML data. In this study, we focus on the two most popular parsing models, namely, SAX and DOM.

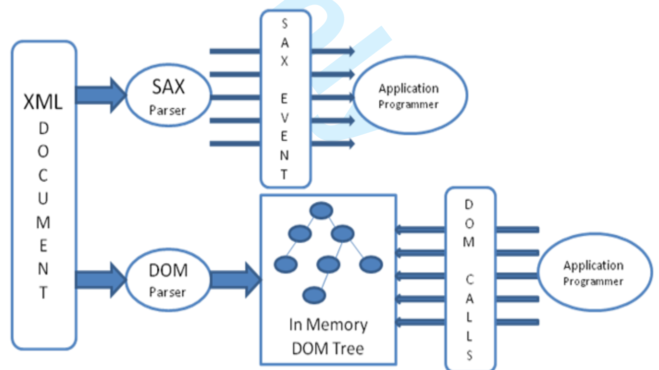


Figure 2: SAX and DOM Parsing Flow

### 2.3 Prefetching Techniques

Data prefetching has been proposed as a speculative technique to bridge the speed gap between CPU and memory subsystem. It alleviates the performance degra-

For Peer Review Only

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60

dation from the long-latency memory accesses by predicting the memory access pattern of the application and speculatively prefetching data that would be used in future computation. Considering that the CPU-memory performance gap is on the order of hundreds of processor clock cycles, prefetching is an attractive way to remove the affect of long latency memory accesses.

### 2.3.1 Classic Prefetching Algorithm

Prefetching tehniques has been well studied and lots of algorithms have been proposed. We list some classic prefetching algorithms below.

*Sequential prefetching* prefetchs the block or blocks that follow the current demanded block, and is fit for the programs with the consecutive memory access pattern [36, 37]. As an improvement, *Sequential tagged prefetching* [51] issues a prefetch upon a cache miss as well as when a prefetched block is referenced for the first time, thus it requires an extra bit per block to mark the prefetch state. The *Sequential prefetching* family increases the performance on a broad range of applications at a low cost, however, at the expsense of many useless prefetches.

*Stride prefetching* makes prefetch requestes according to the observed strides that separate memory addresses flow. Conventional *stride prefetching* uses a record table indexed by the program counter (PC) that associates strides to the loads following this kind of memory access pattern [38, 48]. If address *a* is referenced by a load that hits in the table, the matching entry indicates that the load is following a stride pattern, then prefetcher issues the request for addresses *a+s*, where *s* is the associated stride.

*Strem Prefetching* traces a sequence of nearby misses when their addresses follow the same positive or negative direction in a small memory region [39, 40]. In some design [39], there always exsites a streaming buffer to store the fetched data.

*Correlating prefetching* predicts future addresses from tables that record the past memory program behavior [49, 50]. Usually, it generalizes the stride table by registering the stream of addresses associated either to the load PC or to an address that misses in the corresponding cache level. In correlating prefetchers, mega-sized tables are needed to record enough information. Thus, it can provide good prefetch results to a broader range of applications. However, it is associated with considerable hardware cost.

### 2.3.2 Software Prefetching Vs. Hardware Prefetching

According to how prefetching is implemented, it can be classified into two classes: software prefetching and hardware prefetching.

- Software Prefetching

Software prefetching [35, 44, 46] need to introduce new prefetching instructions into the instruction set architecture (ISA), which could bring data at speicified memory addresses into cache. It is assisted by compiler algorithms to insert software prefetching instructions into proper places of the source code. In the preprocessing stage, compiler gets the global information about memory data access pattern, locates those data-sets that are lean towards cache misses and calculates the positions to insert the prefetching instruction. In Intel® Pentium® 4 processor, it enables using the four prefetch instructions introduced with Streaming SIMD Extensions (SSE). These instructions are hints to bring a cache line of data into various cache levels.

Since software prefetching gets the assistance from compiler or programmer, it can acquire a globle map of data accesses, handle irregular access patterns and make more precise prefetchings. However, the insertion of the prefetching instruction is statically determined so software prefetching can not adapt to the phase change of the application. Since new instructions need to be added, recompilation is required, so these do not benefit the scenarios where recompilation is inconvenient.

- Hardware Prefetching

Different from software prefetching that statically inserts prefetching instructions by compiler, hardware prefetching frees the need to expand instruction set architecture and frees the compiler from revising the source code of applications. It automatically determines the data accesses that might cause cache misses and then make pretching requestes. Its decision is based on the recorded history information so that it can adapt to the phase change of application. However, it must comsume extra hardware resource and is unable to gain a complete picture of the whole memory pattern. Therefore, it does not suit for the case of irregular data access and short arrays for the penalty of history start-up. In our study, we focus on hardware prefetching for its advantage of no revise of the source code.

## 2.4 Overview of Hardware Prefetcher

In this subsention, we present the system integration and detailed architecture of hardware prefetcher.

### 2.4.1 System Integration of Hardware Prefetcher

As a speculative techinique, hardware prefetching can locate in any cache layer. Since modern processor has taken multi-layer cache hierarchy to hide the long latency memory access, multi-level prefetcher may give more performance improvement, considering its effect in each level applied. For example, as shown in Figure 3, we can have a L1 cahce prefetcher which requests data in L2 cache into L1 cache, a last-level cache (LLC) prefetcher which prefetches the requested data in memory into last-level cache, and a memory prefetcher which communicates between memory and disk, reducing the time used for disk accesses. However, most modern processors perform out-of-order (OoO) execution, so that the latency of L1 cahce miss can be tolerated by the architecture improvement. Therefore, it is arguable to apply prefeting in L1 cache level. For the last level cache, its cache miss latency is determined by the time used for memory access, which is on the order of hundred of processor cycles nowadays. Here, the applying of prefetching can hide several long latency memory accesses, resulting considerable performance improvement, especially for memory-intensive applications. It also can be seen in Figure 3 that the prefetched data are usually sent into the low level

For Peer Review Only

cache directly. However, to reduce the chance of cache pollution, some researchers proposed to cache the prefetched data into an extra buffer [44, 45]. As a result, the competition for limited cache lines is alleviated at the cost of additional hardware overhead.

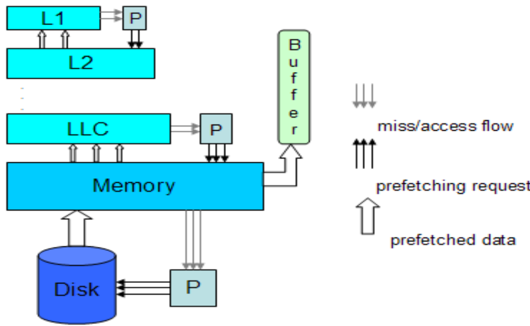


Figure 3: System Integration of Prefetcher

### 2.4.2 Architecture of Hardware Prefetcher

Hardware prefetcher utilizes hardware resources in order to load useful data ahead of time. As shown in left part of Figure 4, the flow of cache miss or cache access is passed into hardware prefetcher, which can be considered as the trigger set of the prefetching. Then prefetcher requests data through predicting future accesses and issuing prefetch requests into lower memory hierarchy. When request is fulfilled, those returned data are forwarded into upper hierarchy where cache flow is collected or additional prefetching buffer.

As in the right part of Figure 4, we can obtain the generic architecture of hardware prefetcher. The trigger set, regardless it is collected as cache access or cache miss, is passed into *history recorder* and corresponding information is captured based on the algorithm applied. Usually, the flow can be recorded by a particular program counter value, being the memory activities of the same instruction; or by a constant time slot, being the time-sliced memory behavior.

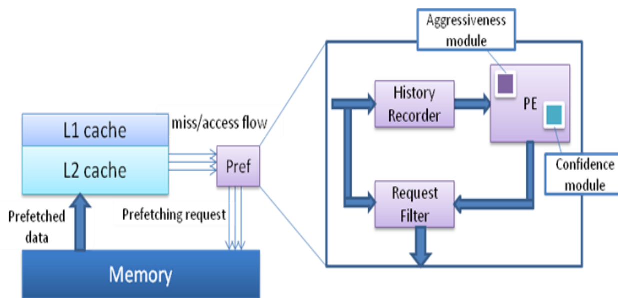


Figure 4: Architecture of Generic Prefetcher

*Prefetching engine* (PE) fetches the recorded information from the *history recorder* and makes prediction by considering memory behavior relations. There have been a lot of relations considered by different prediction algorithms, e.g., *Sequential Prefetching*, *Stride Prefetching*, *Stream Prefetching* and *Correlation Prefetching* as we mentioned earlier.

There are also some optional modules inside prefetching engine to control the precision and aggressiveness of the prefetcher. The aggressiveness module can be used to tune the number of triggered requests through optimizing the prefetching degree or prefetching distance. The confidence module can be used to diminish the opportunity of unsuccessful prefetching. It checks the confidence recording and excludes those requests whose confidence is below certain threshold. The combination of these optimization modules can produce benefits as well as side effects such redundant prefetchings. Since the coverage of prefetching can overlap with that of demand access and the adjacent prefetching may request data within the same block, there come several redundancies that might lead to fruitless accesses and bandwidth waste. Corresponding to that, the *request filter* component is introduced. When system bandwidth support is limited, most of those redundant accesses can be captured and filtered by the *request filter*, yielding less ineffectiveness.

## 2.5 Related Work on XML Acceleration

There have been several proposals on mitigating the performance overhead of XML processing.

In software community, several research groups employed the concept of binary XML to avoid performance bottleneck of XML parsing [1, 9, 10]. For example, VTD-XML parser [10] parses XML documents and creates 64-bit binary-format VTD records. However, its parsed binary data cannot be used by other XML applications directly. On the other hand, some researchers focus on parallelizing the parsing process with the presence of parallelism within modern processor or parsing itself. Prescanning-based parallel parsing model [11] builds a skeleton of the XML document to guide partitioning of the document for parallel data processing. Also, in [12] Head *et al.* 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 pipelined model. In addition, Parabix exploits the SIMD capabilities of commodity processors to process multiple characters simultaneously [13].

In the hardware community, based on the profiling analysis, Zhao *et al.* incorporated new instructions with special hardware support to spedup certain frequently-used operations of XML parsing [14]. In [15], Moscola *et al.* presented a technique to automatically map regular expressions directly onto FPGA hardware and implemented a simple XML parser for demonstration. However, since XML syntax rule is not a regular language. XOE [16] accelerates XML document parsing via offloading some fundamental parsing functionalities like tokening onto a special XOE engine. XPA [17] is another XML parsing accelerator implemented on FPGA, capable of performing XML well-formed checking, schema validation, and tree construction. It can reach up to 1 cycle-per-byte throughput for XML parsing. Nevertheless, their design works only for tree-based parsers. As we will show in the following sections, memory access is one of the major bottlenecks of XML parsing, thus we will be able to generate extra performance gain by memory-side accelera-

Peer Review Only

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

tion on top of all those computation-side acceleration proposed previously. In addition, unlike computation-side acceleration that can be only applied to specific parsing model, memory-side acceleration is generic and can be prevalently adopted to enhance the effectiveness of XML parsing, regardless of the parsing model underneath.

## 3 Methedology

In this section, we discuss our methodology to study the performance of XML parsing, covering the benchmarks we selected and the tools we employed.

### 3.1 XML Parsers and Benchmarks

In order to make fair comparison, we choose XML parser implementations of both event-driven and tree-based models from Apache Xerces [18]. Apache Xerces provides SAX and DOM XML parsers, and 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 of varying sizes (ranging from 1.4 KB to 113 MB) and complexities as input data, which are listed in Table 1. Specifically, *personal-schema* is a very simple document with flat structure, thus the parsing process is straightforward; on the contrary, *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 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 outlined in Table 2.

To understand the implementation feasibility of the memory-side accelerators, we need to model the energy consumption of these designs. For this purpose, we employ 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 power consumption consists of two components: static power and dynamic power. Static power is generated by the leakage current of the transistors, and it persists regardless of the swithing state of the transistors. 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 power to evaluate the implementation feasibility of memory-side accelerators.

Table 2: 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 |
| Frequency | 1 GHz |

### 3.3 XML Parsing Bottleneck Evaluation

Since XML data have been floated in cloud environment, with carefully designed study of the cloud data throughput, we can find whether the network data exchange stage determines the performance of XML parsing. To study the network performance, we measured the data exchange throughput of two different cloud data services: *Content Distribution Network* (CDN) and *Cloud Storage*, which are two popular categories of modern cloud data services.

To study the behavior in disk data loading stage, we make use of the Xperf Performance Analyzer tools [25]. Xperf belongs to Windows Performance Analysis Toolkit. It can be used to monitor system performance on Windows OS, as it is specifically designed to give a complete system-wide view of performance over long period of time. With the help of Xperf, we can capture disk I/O throughput and determine if it is the bottleneck in parsing execution.

### 3.4 FPGA based Case Study

To further study the feasibility of memory-side acceleration, we make a case study by implementing one accelerator design on a Xilinx Spartan-3 FPGA board [41]. In detail, we implemented the design in Verilog, synthesized the design, and then used the XPower Analyzer [42] to generate power consumption information. Since the design we choose to implement is the most complicated hardware prefetcher, with the FPGA implementation, we can get the upper limit of energy consumption, chip area overhead and hardware cost for memory-side acceleration.

For Peer Review Only

For Peer Review Only

## 4 NATIVE VS. MANAGED EXECUTION

In this section, we compare the performance of XML parsing under 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 5, 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 overhead. The situation is not so good either when using DOM parsing model. Even the best case *physics-engine* has incurred 25.93% performance overhead. In the worst case, *resume_w_xsl*, it incurs up to 52.08% performance overhead.

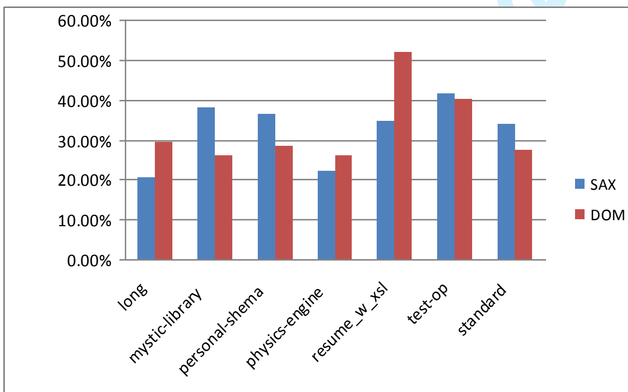


Figure 5: Managed Execution vs. Native Execution

Although managed environment is able to reduce development time, in a common application scenario, XML parsers reside at the forefront of the XML data processing and many other components in the system may have dependency on the outputs of the XML parsers. Therefore, the performance overhead incurred by the managed layer would severely hinder the performance of the entire system. In addition, in large-scale systems (such as those in cloud computing environments), this large performance overhead also leads to energy consumption overhead. This result indicates that managed execution of XML parsing is not suitable in large-scale computing environment (*e.g.* cloud computing environment), considering the significant overhead it introduced. Henceforth, for execution and energy efficiency, we focus on native execution of XML parsing in the rest of the paper.

## 5 PERFORMANCE ANALYSIS OF XML PARSING

In this section we aim at determining the performance bottleneck of XML parsing by studying the throughput of XML parsing at different stages of the information passing flow, including network data exchange, disk I/O, and memory accesses. Figure 6 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 computation.

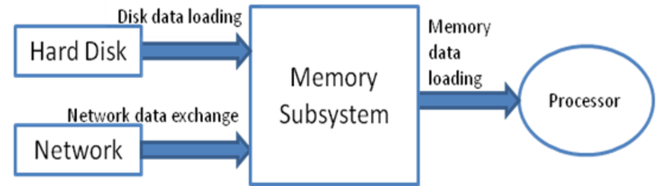


Figure 6: Data Flow of XML Parsing

### 5.1 Network Data Exchange

To study the network performance, we measured the data exchange throughput of different cloud data services. In Table 3，we summarize our measurements of two popular categories of cloud data services: *Content Distribution Network* (CDN) and *Cloud Storage*. For each category, we measured the data exchange throughputs from four different service providers. Note that CDN services contain several copies of data in the 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 of CDN services is around 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 of 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 the network it connects to has a bandwidth limit of 100Mb/s, which is far greater than the throughput provided by cloud data services. This 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 3: 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 |
| Average | 29.26 | Average | 12.56 |

### 5.2 Disk Data Loading

We used the XPerf Performance Analyzer tools [25] in order to capture disk I/O throughput when running the XML parsers using the *standard* benchmark. To get the precise I/O throughput data, we start from a clean environment and make sure the XML data was not already in the cache. The collected results are shown in Figure 7: the x-axis shows the execution timeline in seconds and the y-axis shows the amount of triggered disk I/O operations during the execution. The gray curve overlaid on top of the bar diagram indicates the CPU usage information. The peak of the curve means the CPU is fully utilized. Figure 6 shows that most of the time disk I/O is in underutilized state, which means the I/O subsystem rarely needs to reach its full capacity. Based on this observation, 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 from Table 1, and the results were similar.

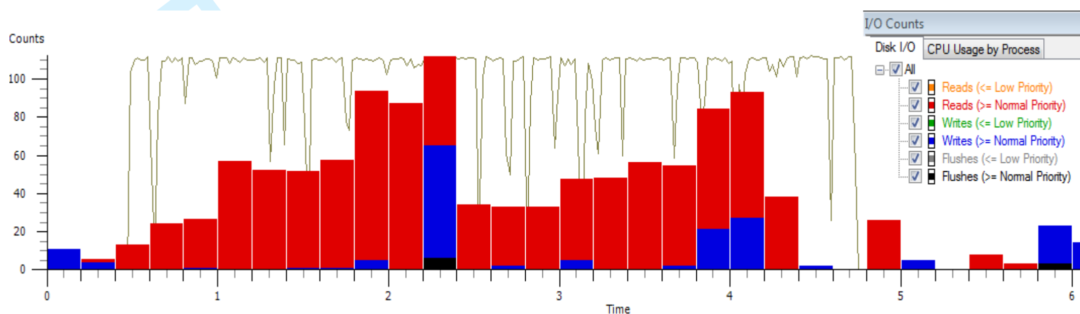


Figure 7: Disk I/O Counts and CPU Usage

### 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. In order to make a comparison, we measured the CPI (cycles per instruction) of *Speed-test*, which is a computation-intensive CPU stress test application with negligible memory accesses; 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 58.75% overhead compared to *Speed-test*; using the DOM parser, the CPI of *standard* becomes 1.42, which introduces 77.5% overhead compared to *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 for *standard*. This means for every 1000 instructions there come about ten L1 and two L2 cache misses. On the other hand, the *Speed-test* has a L1 MPKI of 0.051, and a L2 MPKI of 0.072. 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 exchange is not likely to be the bottleneck of XML parsing.
- Second, our experiment results show that the disk I/O subsystem is underutilized 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 with a CPU stress test workload, we have found that in some cases the CPI of XML parsing almost doubles 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.

Recall in Figure 6 that gives the data flow of XML parsing, we can draw the results that: the performance bottleneck of XML parsing is the memory data loading stage, rather than the disk data loading stage or the network exchange stage. In other words, the overhead introduced from memory subsystem really hits the weakness of the 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 overhead incurred by the memory subsystem.

## 6 Memory-Side Acceleration

We have identified that memory accesses impose significant overhead 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 even longer than the actual parsing time

For Peer Review Only

itself [23]. 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.

### 6.1 Prefetchers

In this study, we evaluate how different prefetching techniques behave as the memory-side accelerator to impact the performance of XML parsing. In order to make a comprehensive investigation, we have selected eight hardware prefetchers named *n1* through *n8*, which utilize different techniques and algorithms. We summarize these prefetchers in Table 4:

- *Cache hierarchy* indicates the coverage of the prefetcher, 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 prefetchings;
- *Trigger L1* and *trigger L2* respectively show the trigger set for covered cache hierarchy. 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 opportunities to invoke the prefetching, it always yields more aggressiveness as well as pollutions.

Besides, all the selected prefetchers can filter out redundant access requests.

TABLE 4: SUMMARY OF PREFETCHERS

| 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 itself. Usually, the more applied cache hierarchy, the deeper prefetch degree and the larger trigger set would lead to more aggressive prefetching. However, it always comes with cache pollution and other side effects. In the following, we introduce the eight selected prefetchers briefly.

The first prefetcher *n1* can tolerate out-of-order (OoO) memory accesses by making prefetching based on the recent memory access pattern [26]. The second prefetcher *n2* exploits various localities in both local and global cache-miss streams, including global strides, local strides and scalar patterns [27]. A multi-level prefetching framework is applied in *n3*, which uses a sequential tagged prefetcher at L1 cache and either an adaptive prefetcher or a sequential tagged prefetcher at L2 cache [28]. 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 [29]. Combining the storage efficiency of reference prediction tables (RPT) 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 [30]. 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 [31]. 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 [32]. 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 [33].

### 6.2 Performance Analysis

Table 5 summarizes the reduction of cache misses as a result of applying 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. The last row shows the lowest-level cache where prefetching is applied. For example, n1 is applied to both L1 and L2 caches, and 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 parser and 85% in DOM parser.

TABLE 5: REDUCTION IN CACHE MISSES

| | n1 | n2 | n3 | n4 | n5 | n6 | n7 |
|---|---|---|---|---|---|---|---|
| *SAX* | *69%* | *43%* | *82%* | *82%* | *51%* | *40%* | *73%* |
| *DOM* | *77%* | *52%* | *85%* | *85%* | *61%* | *52%* | *77%* |
| *Cache Level* | *L2* | *L1* | *L2* | *L1* | *L2* | *L2* | *L2* |

In Figure 8, we show how the cache miss reduction translates into performance improvement on SAX parsing: it shows the impact of the eight prefetchers (n1 - n8) as well as the average. 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

For Peer Review Only

up to 10%. For instance, the parsing time of *personal-schema* has been reduced by 7.24% on average. Looking into each prefetching technique, we observe that n3 shows greatest advantage in improving the performance, ranging from 2.58% to 9.72% across different benchmarks. This is because n3 is the most aggressive prefetcher and covers both L1 and L2 cache levels, thus resulting in the best average performance.

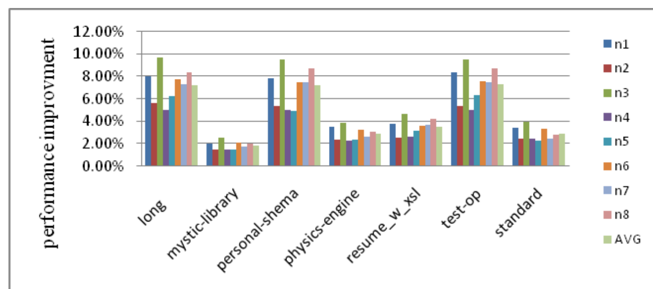


Figure 8: Performance Improvement for SAX Parsing

Similarly, Figure 9 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 of *resume_w_xsl*, n3 can still reduce execution time by 6%. Note that different from SAX parsing, DOM parsing must construct inner data structure in memory for all elements. The bigger the document, the more space it would consume and the more cache miss it would induce. As a result, large-size benchmarks such as *mystic-library*, *physics-engine* and *standard* can get a higher performance gain from memory-side acceleration, ranging from 7.65% up to 13.75%. These results confirm that memory-side acceleration can be effective regardless of the parsing models.

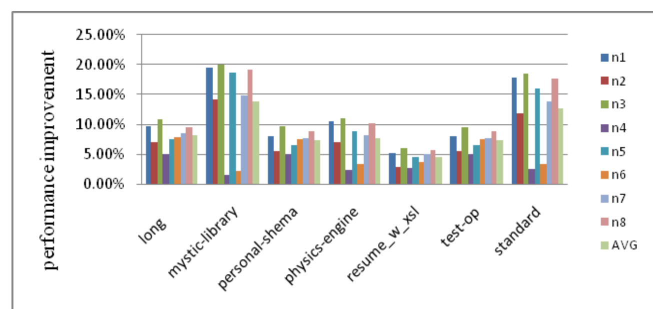


Figure 9: Performance Improvement for DOM Parsing

## 7 Implementation Feasibility

By now we have shown that memory-side acceleration can significantly improve XML parsing performance. However, conventional wisdom is that prefetching requires extra hardware resources, 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 regular memory traffic of 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. In order to validate whether this is the case or not, 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 bandwidth of memory subsystem, and thus confirming that prefetching would be effective.

Table 6: Bandwidth Consumption without Prefetching

| Benchmarks | 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 extracted the hardware cost information from the design documents of the eight prefetchers and summarized this information in Table 7. On average, these prefetchers require about 28,000 bits of memory space. For instance, n6 consists of a 14080-bit L1 prefetcher, a 4096-bit L2 prefetcher and eight 20-bit counters, producing a 32416-bit hardware cost. All of their hardware cost is less than or equal to 32 Kbits, which is not a significant amount of hardware overhead in modern high-performance processor design.

Table 7: Hardware Cost of Prefetchers

| Prefetchers | Hardware Cost (bits) |
|---|---|
| *n1* | *32036* |
| *n2* | *20329* |
| *n3* | *20787* |
| *n4* | *30592* |
| *n5* | *25480* |
| *n6* | *32416* |

eer Review Only

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

| *n7* | *30720* |
|---|---|
| *n8* | *32768* |

Next we study how these memory-side accelerators impact system energy consumption. When calculate the overall energy consumption, we sum up the energy comsumed by the memory system and the energy cosumed by the hardware prefetcher. As stated in Equations 1-3, energy can be classified into two categories: static energy and dynamic energy. The first one is the product of the overall execution time (t) and the static power consumption ($P_{static}$) of system; whereas the second component can be derived by multiplying the number of read/write accesses ($n_m$) and the energy dissipated on each access ($E'_m$). The static power is a constant across all the implementations so that the static energy is just determined by how prefetching can shorten the execution time.

$$(1)\quad E = E_{\text{dynamic-mem}} + E_{\text{static-mem}} + E_{\text{dynamic-}pref} + E_{\text{static-}pref}$$

$$(2)\quad E_{static} = P_{static} \times t$$

$$(3)\quad E_{dynamic} = n_m \times E'_m$$

Using our simulation framework consisting of CMP$IM and CACTI, we can generate energy parameters of different storage and interconnect structures implemented in different technologies. Here, we focus on the implementation with state-of-the-art 32nm technology and the results are summarized in Figures 10 and 11. In these figures, we use 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 32nm technology, static energy is comparable to dynamic energy [24]. 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 10, 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, when we look 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.

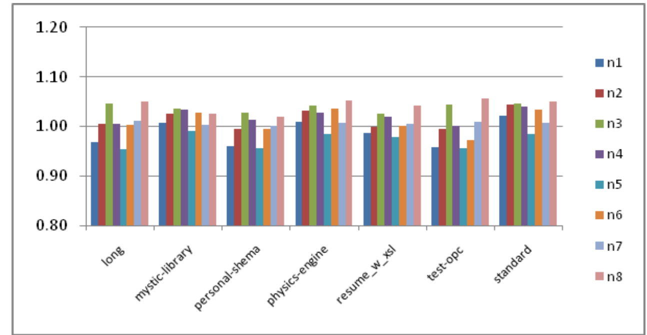


Figure 10: Energy Consumption of SAX Parsing

In Figure 11, we summarize how acceleration impacts energy consumption in DOM parsing. 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. Identical with Figure 9, n5 is still the most energy-efficient prefetcher which archives as high as 12.77% energy saving in *mystic-library*. Even when running its worst case of *resume_w_xsl*, n5 can still reduce overall energy consumption by almost 3%.

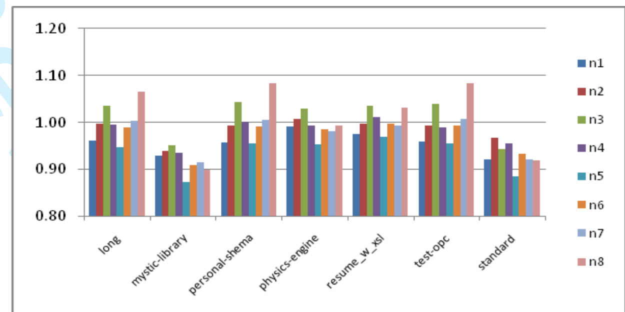


Figure 11: Energy Consumption of DOM Parsing

## 8 An Exemplary Hardware Implementation

The results from Section 6 indicate that *n3* provides the best performance as it being the most aggressive prefetcher. In this section we use *n3* as a case study to discuss the implementation details of using prefetchers to improve XML parsing performance.

Prefetcher *n3* is a multi-level prefetching framework that consists of six components: in L1 cache, it uses a sequential tagged prefetcher; in L2 cache, it implements a selective correlating prefetcher based on a Differential Finite Context Machine (PDFCM), which predicts the next occurrence by considering sequences of differences between consecutive addresses issued by the same memory instruction. To remove redundant memory accesses, in L1 cache it uses Miss State Holding Register (MSHR) structures to hold memory accesses to the same address. Similarly, in L2 cache, it uses a Prefetch Memory Address File (PMAF) structure, which is a FIFO structure similar to eliminate prefetching requests to blocks that have already

been issued to the next memory level. In addition, to control the aggressiveness of the L1 and L2 prefetch engines, it applies a degree controller at each cache level.

First, in order to find out the chip area overhead and the power overhead of this design, we implemented the *n3* design on a Xilinx Spartan-3 FPGA board and used the XPower Analyzer to generate power consumption information. The results show that the overall power consumption of this design is 32.6 mW. In Figure 12, we break down the power consumption by different hardware components: note that the clock is the highest power consumer, accounting for 68% of the total power consumption. The next biggest power consumer is the signals, consisting of 26% of the power consumption. On the other hand, DCM and Logic make up a small proportion of the total consumption, with each just accounting for 3% of the overall energy consumption.

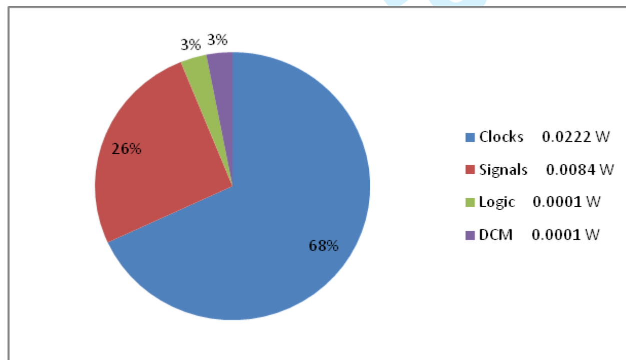


Figure 12: Power Consumption of the Prefetcher Implementation on FPGA

The next question we raised was how this power consumption data compared to that of a simple processor, and what the hardware overhead of this design is. In Table 8, we summarize the hardware cost and power consumption of *n3* prefetcher: the hardware utilization measures include the number of flip-flops (#FF), and the number of look-up tables (#LUT). For comparison purpose, we also present the resource utilization of the eMIPS processor, a simple MIPS in-order processor implementation [43]. The results show that eMIPS requires 17055 flip-flops, and 26106 look-up tables. Compared to the eMIPS processor, the hardware resource consumption of *n3* prefetcher consists of 582 flip-flops and 1005 look-up tables, which represents less than 5% hardware overhead. In terms of power consumption overheads, eMIPS consumes 356 mW whereas prefetcher *n3* consumes 32 mW, which represents 9% power consumption overhead. Note that at the first sight, 9% power consumption overhead may seem high, but eMIPS is a very simple in-order processor that runs at very low frequency (100 MHz). Comparing to commercial processors, such as Intel Xeon®, which consumes about 100 W of power, the power overhead of this prefetcher design is really negligible.

TABLE 8: HARDWARE COST AND POWER CONSUMPTION COMPARISON BETWEEN EMIPS AND PREFETCHER

| | # FF | #LUTs | Power (mW) |
|---|---|---|---|
| eMIPS | 17055 | 26106 | 356.36 |
| Prefetcher n3 | 582 | 1005 | 32.6 |

As mention above, prefetcher *n3* consists of prefetch engines at both L1 and L2 cache levels, and our implementation data indicates that the area overhead of the L1 prefetch engine is about one third of that of the L2 prefetch engine. In this part we try to understand whether it is worthwhile to implement prefetch engines at both L1 and L2 levels, or whether it is sufficient to implement a prefetch engine at only the last-level cache (L2 in this case).

In Figure 13, we compare the performance of the two-level (both L1 and L2) prefetcher versus that of the L2-only prefetcher. The y-axis shows the performance improvement achieved by the two-level prefetcher compared to the L2-only prefetcher. Across all benchmarks with both DOM and SAX parsers, two-layer prefetching outperforms the L2-only prefetcher, but providing only 1~2% performance gain. This is because the latency of L1 cache miss is several orders of magnitude smaller than that of memory access. When applying the two-level prefetcher, some L1 cache misses can be reduced, however, the impact on overall performance is small. In other words, we spend 1/3 more area budget to achieve only 1~2% of performance gain, which implies that it may not be a cost-effective option to implement the two-level prefetcher.

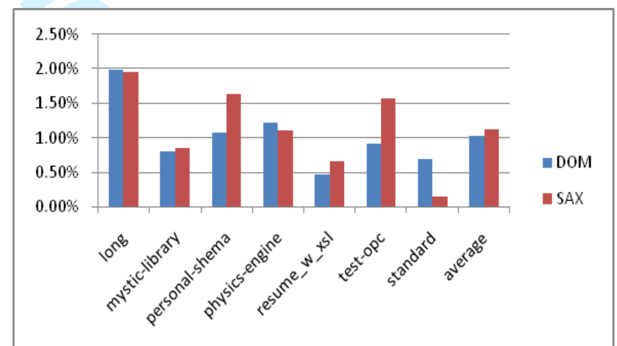


Figure 13: Performance Comparison of L2-only vs. Two-Level Prefetchers

In next step, we compare the energy consumptions as a result of applying the two-level prefetcher versus applying the L2-only prefetcher, and the data is presented in Figure 14. The y-axis shows the system energy overhead as a result of applying the L2-only prefetcher compared to that of applying a two-level prefetcher: a positive number represents that applying a L2-only prefetcher would consume more energy; whereas a negative number indicates otherwise.

For Peer Review Only

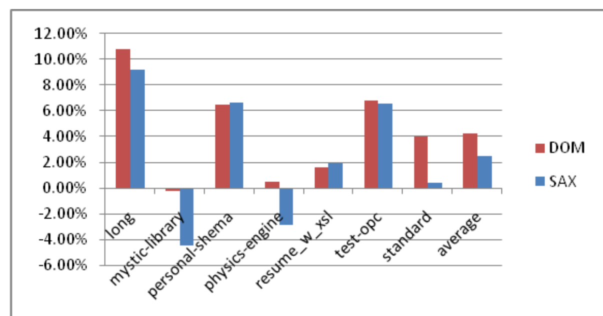


Figure 14: Energy Consumption Comparison of L2-Only vs. Two-Level Prefetchers

It is generally recognized that multi-layer mechanism might incur more energy consumption due to their increased hardware overheads. However, as shown in Figure 14: for most cases the two-layer prefetcher leads to energy efficiency, on average producing a 2.46% energy reduction in SAX parsing and 4.27% reduction in DOM parsing. For benchmark *long*, applying the two-level prefetcher introduces an energy saving of over 10%; and for *personal-schema* and *test-opc*, its improvement also achieves over 6%. Although a two-level prefetcher incurs 1/3 of area and power overhead compared to a L2-only prefetcher, since the power consumption of the prefetcher is negligible comparing to the power consumption of the system, and thus the power overhead of the two-level prefetcher over the L2-only prefetcher also becomes negligible. On the other hand, the two-level prefetcher provides 1~2% performance improvement, leading to reduction of system-level energy consumption, thus the two-level prefetcher design actually leads to system-level energy efficiency.

Also, it is worth noting that for some benchmarks, applying a two-layer prefetcher may lead up to 4% extra energy consumption. That is because multi-layer prefetching may also introduce cache pollution problems that incur excessive memory accesses, offsetting the static energy reduction from reduced execution time. To sum up, if energy efficiency was the optimization goal, we should use simple but aggressive prefetching engines, such as n5. Otherwise, if performance was the optimization goal, we should use complex but aggressive prefetching engines such as n3. Specifically, in n3, it is worthwhile to implement a multi-layered scheme as it provides marginal performance improvement and better energy efficiency.

## 9 Conclusions

Different from previous research work which focused on computation acceleration of XML parsing, we first identified memory access as one of the performance bottlenecks. We then proposed to make acceleration for XML parsing from memory side by improving its data loading performance. The results show that memory-side accelerators exhibit considerable effectiveness across existing parsing models. They are able to reduce cache misses by up to 80%, which translates into up to 20% of performance improvement.

We also verified the feasibility of our proposal by checking its implementation impact on bandwidth, energy consumption and hardware cost. The results show that memory-side accelerator can produce up to 12.77% of energy saving when implemented in 32nm technology. For the eight accelerators studied, all their hardware cost is within 32 Kbits which is a very small and reasonable overhead considering modern hardware budget. Regarding bandwidth, XML parsing performance is hurt by the latency but not by the throughput of the memory subsystem, thus confirming that memory-side acceleration will not likely result in resource contention of memory bus bandwidth. In conclusion, memory-side acceleration of XML parsing is not only effective but also feasible.

## 10 Future Work

The next step of this research project is to integrate memory-side and computations-side accelerators of XML parsing into a single core, and optimize its performance and power consumption. Then, ultimately, we are going to integrate this core onto many-core architectures to act as a Data Exchange Frontend (DEF).

Our ultimate goal is to build a heterogeneous many-core chip, which consists of two kinds of cores: general-purpose cores and specialized cores. The general-purpose ones take care of conventional general computing workload as well as control. The specialized ones are designed for some critical and commonly used functions, for example, XML parsing, garbage collection (GC), and memory encryption/decryption. With the specialized core design, we can achieve both performance and energy improvement.

## Acknowledgements

This work is supported by the National Science Foundation under Grant No. CCF-1065448, ECCS-1125762, 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 National Science Foundation.

## 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 20002

[2] M. R. Head, M. Govindaraju, R. van Engelen, andW. 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, and et al., "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

For Peer Review Only


[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] International HapMap Project: http://hapmap.ncbi.nlm.nih.gov/

[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] A. Longshaw, "Scaling XML parsing on Intel architecture," Intel Software Network Resource Center, November 2008. http://www.developers.net/intelisnshowcase/view/537

[24] Power vs. Performance: The 90 nm Inflection Point, http://www.xilinx.com/publications/archives/solution_guides/power_management.pdf

[25] Windows Performance Analysis Tool, http://msdn.microsoft.com/en-us/performance/cc825801

[26] Y Ishii, M Inaba, K Hiraki, "Access map pattern matching prefetch: Optimization friendly method". The 1st international Journal of Instructional Level Parallelism Data Prefetching Championship (DPC-1), 2009

[27] M Dimitrov, H Zhou, "Combining local and global history for high performance data prefetching", The 1st international Journal of Instructional Level Parallelism Data Prefetching Championship (DPC-1), 2009

[28] LM Ramos, JL Briz, PE Ibáñez, V Viñals, "Multi-level Adaptive Prefetching based on Performance Gradient Tracking", The 1st international Journal of Instructional Level Parallelism Data Prefetching Championship (DPC-1), 2009

[29] M Ferdman, S Somogyi, B Falsafi, "Spatial Memory Streaming with Rotated Patterns", The 1st international Journal of Instructional Level Parallelism Data Prefetching Championship (DPC-1), 2009

[30] M Grannaes, M Jahre, L Natvig, "Storage Efficient Hardware Prefetching using Delta Correlating Prediction Tables", The 1st international Journal of Instructional Level Parallelism Data Prefetching Championship (DPC-1), 2009

[31] S Verma, DM Koppelman, L Peng, "A Hybrid Adaptive Feedback Based Prefetcher", The 1st international Journal of Instructional Level Parallelism Data Prefetching Championship (DPC-1), 2009

[32] A Sharif, HHS Lee, "Data Prefetching Mechanism by Exploiting Global and Local Access Patterns", The 1st international Journal of Instructional Level Parallelism Data Prefetching Championship (DPC-1), 2009

[33] G Liu, Z Huang, JK Peri, X Shi, L Peng, "Enhancement for Accurate Stream Prefetching", The 1st international Journal of Instructional Level Parallelism Data Prefetching Championship (DPC-1), 2009

[34] STAX parsing model: http://jcp.org/en/jsr/detail?id=173

[35] D. Callahan, K. Kennedy, and A. Portereld. "Software prefetching". In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 40-52, April 1991.

[36] D.G. Perez, G. Mouchard, and O. Temam, "Microlib: A case for the quantitative comparison of micro-architecture mechanisms", In Proceedings of International Symposiums on Microarchitecture (MICRO), 2007.

[37] A.J. Smith, "Sequential Program Prefetching in Memory Hierarchies", IEEE Transactions on Computers, 11(12), Dec. 1978.

[38] J. Fu and J. Patel, "Stride directed prefetching in scalar processors", MICRO-25, 1992.

[39] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", in Proceedings of International Symposium on Computer Architectures (ISCA), 1990.

[40] Santhosh Srinath, Yale N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), 2007.

[41] Xilinx Spartan III http://www.xilinx.com/products/devkits/HW-SPAR3-SK-UNI-G.htm

[42] Xilinx XPower http://www.xilinx.com/products/design_tools/logic_design/verification/xpower.htm

[43] eMIPS: http://research.microsoft.com/en-us/projects/emips/default.aspx

[44] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu. "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching". In Proceedings of Microcomputing 24, 1991.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60

[45] A. C. Klaiber and H. M. Levy. "Architecture for software-controlled data prefetching". In Proceedings of the 18th Annual International Symposium on Computer Architecture, pages 43-63, May 1991.

[46] A. K. Porterfield. "Software Methods for Improvement of Cache Performance on Supercomputer Applications". PhD thesis, Department of Computer Science, Rice University, May 1989

[47] Intel Labs, "SCC Platform Overview," Intel Many-core Applications Research Community, Revision 0.75, September 2010.

[48] J.L. Baer and T.F. Chen. "An Effective On-chip Preloading-Scheme to Reduce Data Access Penalty". In *Int. Conf. on Supercomputing* (ICS) pp.176-186, 1991.

[49] A. Lai, C. Fide and B. Falsafi. Dead-Block Correlating Prefetchers". In Procs. of the *28th Intl. Symp. on Computer Architecture* (ISCA) pp. 144-154, 2001

[50] Mark J. Charney and Anthony P. Reeves. "Generalized correlation-based hardware prefetching". TR EECEG-95-1, School of Electrical Engineering, Cornell University, February 1995.

[51] Smith, A.J., "Cache Memories," *Computing Surveys*, Vol.14, No.3, September 1982, p. 473-530.

**Jie Tang** is a Ph.D. candidate in Beijing Institute of Technology, China. Her research interests include high performance computer architecture, cloud computing, and embedded system. Jie's Ph.D. thesis title is "Performance Acceleration and Energy Efficiency Mechanisms in Cloud Computing Environment", in which she studies the impact of hardware acceleration and prefetching techniques in enhancing both the performance and energy efficiency for cloud computing environment. During her Ph.D. study, Jie also worked as a visiting researcher in the Center for Embedded Computer Systems, University of California, Irvine. Jie holds a B.S. in Computer Science from National University of Defense Technology, China.

**Shaoshan Liu** is currently with Microsoft. He received Ph.D. in Computer Engineering, M.S. in Biomedial Engineering, M.S. in Computer Engineering, and B.S. in Computer Engineering, respectively in 2010, 2007, 2006, and 2005 respectively, all from the University of California, Irvine. His research interests include parallel computer architectures, embedded systems, runtime systems, as well as biomedical engineering.

**Chen Liu is** an assistant professor in the Department of Electrical and Computer Engineering at Florida International University, Miami, Florida. He received the B.E. degree in Electronics and Information Engineering from University of Science and Technology of China, Hefei, Anhui, China in 2000, the M.S. degree in Electrical Engineering from the University of California, Riverside in 2002 and the Ph.D. degree in Electrical and Computer Engineering from the University of California, Irvine in 2008, respectively. His research interests include multi-core multi-threading architecture, the interaction between system software and microarchitecture, power-aware many-core computing, hardware acceleration techniques and reconfigurable computing. He is a member of IEEE and IEEE Computer Society. He also served as the chair of Computer Society Chapter, IEEE Miami Section from 2010 to 2011.

**Zhimin Gu** is a professor of computer science at Beijing Institute of Technology. Prior to that he was visiting scholar of computer science at University of Birmingham in UK from 2003 to early period of 2004, and associate professor and post-doctorate researcher of computer science at Northwest Polytechnic University from 1997 to 1999. He received the BS in computer science from Shanxi University in 1985, and the MS in Computer science from Harbin Institute of Technology in 1991, and the PhD in computer science from Xian Jiaotong University in 1997.

**Jean-Luc Gaudiot** received the Diplôme d'Ingénieur from the École Supérieure d'Ingénieurs en Electrotechnique et Electronique, Paris, France in 1976 and the MS and PhD degrees in Computer Science from the University of California, Los Angeles in 1977 and 1982, respectively. He is currently a Professor and Chair of the Electrical and Computer Engineering Department at the University of California, Irvine. Prior to joing UCI in January 2002, he was a Professor of Electrical Engineering at the University of Southern California since 1982, where he served as and Director of the Computer Engineering Division for three years. He has also done microprocessor systems design at Teledyne Controls, Santa Monica, California (1979–1980) and research in innovative architectures at the TRW Technology Research Center, El Segundo, California (1980–1982). He consults for a number of companies involved in the design of high-performance computer architectures. His research interests include multithreaded architectures, fault-tolerant multiprocessors, and implementation of reconfigurable architectures. He has published over 170 journal and conference papers. His research has been sponsored by NSF, DoE, and DARPA, as well as a number of industrial organizations. In January 2006, he became the first Editor-in-Chief of IEEE Computer Architecture Letters, a new publication of the IEEE Computer Society, which he helped found to the end of facilitating short, fast turnaround of fundamental ideas in the Computer Architecture domain. From 1999 to 2002, he was the Editor-in-Chief of the IEEE Transactions on Computers. In June 2001, he was elected chair of the IEEE Technical Committee on Computer Architecture, and re-elected in June 2003 for a second two-year term.He is a member of the ACM, of the ACM SIGARCH, and of the IEEE. He has also chaired the IFIP Working Group 10.3 (Concurrent Systems). He is one of three founders of PACT, the ACM/IEEE/IFIP Conference on Parallel Architectures and Compilation Techniques, and served as its first Program Chair in 1993, and again in 1995. He has also served as Program Chair of the 1993 Symposium on Parallel and Distributed Processing, HPCA-5 (1999 High Performance Computer Architecture), the 16th Symposium on Computer Architecture and High Performance Computing (Foz do Iguaçu, Brazil), the 2004 ACM International Conference on Computing Frontiers, and the 2005 International Parallel and Distributed Processing Symposium.In 1999, he became a Fellow of the IEEE. He was elevated to the rank of AAAS Fellow in 2007.

For Peer Review Only