XML on a Chip?

XimpleWare

Introduction

XML (Extensible Markup Language) is a text-based semi-structured data/document format standardized by W3C in 1998 as the foundation for next-generation Internet applications. Since then XML was applied extensively to areas such as EAI, B2B, middleware, database and content management and has shown enormous amount of promise in enabling new applications as well as solving problems associated with older generations of proprietary technologies. However, many challenges remain that hinder the widespread adoption of XML. Problems such as processing model and performance are technical ones and finding solutions to them probably requires, more than anything else, technical innovation and engineering creativity. Although the concept of using custom hardware to improve performance is hardly new, one has to be very careful about the idea of processing XML in hardware. *The devil is in the details*: DOM/SAX processing models have many problems that prevent them from being implemented in hardware. The goal of this paper is threefold: (1) to examine root causes of XML performance issues in general and DOM's memory issue (2) to investigate the feasibility of moving XML processing on chip (3) to outline properties as well as potential applications of XimpleWare's patent-pending technology.

Technical Challenges of XML Processing

XML has several favorable attributes that distinguish it from other competing technologies. Programmers find XML easy to learn because it is human-readable. The downside, however, is that an XML document needs to be parsed for it to become machine-readable. Existing parsing libraries, such as Crimson and Xerces, don't offer very good performance even on a top-of-the-line general-purpose processor. Various research papers and publicly available benchmarks have shown that performance of XML middleware applications lags behind proprietary middleware, such as CORBA and RMI, by ***orders*** of magnitude.

Also programmers working with XML often face the dilemma of picking the right processing model. Generally people like DOM (Document Object Model) because it offers a tree view of the XML document and is a natural and easy way to work with XML. But DOM is slow and quite resource intensive, making it unsuitable for most high performance applications. SAX (Simple API for XML) is faster and consumes less memory, but doesn't provide much structural information of an XML document. As a result, programmers using SAX often have to manually maintain the state information, which can be quite slow and tedious for a complex XML document. In light of those issues, XML luminary James Clark, in a recent interview, points out that one of the challenges for XML is to "**Improve XML processing models.** Right now, developers are generally caught between the inefficiencies of DOM and the unfamiliar feel of SAX. An API that offers the best of both is needed."

Why is DOM resource intensive?

Current XML parsing inherits heavily from traditional text processing techniques invented decades ago for compiler design purposes, which have relatively low performance requirement because once an application has been compiled, it can run standalone without compiler. Also DOM is originally designed for HTML rendering in a web browser, which also isn't performance critical. In an environment where a constant stream of XML documents needs to be processed real-time, performance becomes a far greater concern. DOM is slow in a large part because it is resource intensive. A couple of factors contribute to DOM's memory inefficiency:

- **Overhead of allocating small memory blocks**
  The operating system uses a segment of process address space called "heap" for the purpose of dynamic memory allocation. To do that, OS pre-divides heap into linked lists of small fixed-size free memory blocks, also known as buckets. Any request for a small memory block will be assigned by OS a smallest pre-allocated block in the bucket that the fits the size of the request. For instance, a request to allocate a single-byte returns a 16-byte chunk (an 8-byte memory block plus 8 byte for boundary tags). When the OS has to allocate lots of small memory blocks, the overhead can become very significant.
- **Unnecessary de-coupling between a node object and its name**
  A node object is a small memory block containing a pointer to the node name in the form of a string object, which is another small block. The binding between node object and node name plays right into the weakness of the OS: It is like the overhead of small memory blocks isn't bad enough – DOM "knowingly" creates as many small blocks as possible to take advantage of the "overhead."

Limitation of General-purpose Processor Architecture

Normally, when people talk about general-purpose processors, they think of features that make a piece of fabricated silicon programmable to solve any computing task. One of them is the "sequential execution model" where a general-purpose processor does the computation by sequencing through a set of operations in time, that is, one at a time. Modern microprocessors also incorporate deep pipelines so multiple instructions can be assigned to different stages of the pipeline to improve computation throughput. A good example is Intel's Pentium 4, with its deep 20-stage pipeline. Moreover, modern microprocessors often employ a hierarchy of SRAM-based cache memory in order to bridge the performances gap between registers and main memory.

Nevertheless, XML parsing belongs to a class of computation tasks that doesn't benefit from those features; therefore, its performance usually isn't satisfactory. All those tasks have low data dependency and require same basic operations to be executed repeatedly on large amount of data. For them, a general-purpose architecture is not able to take advantage of the inherent parallelism because of its sequential nature. Failing to recognize such concurrency also has unwelcome consequences on the accuracy of branch prediction, especially with a deep-pipeline. Suppose the branch prediction unit makes a bad prediction and the CPU was just about to process the data in the last stage of the pipeline, it now has to flush the entire pipeline and start over again, losing valuable clock cycles in the process. Also

the memory hierarchy doesn't completely eliminate main memory access, which can cost over a hundred cycles in the latest Pentium IV processors.

XML on a chip: Does it make sense?

The answer is yes. Only done properly, though.

As such, XML parsing is a special case of layer-7 processing whose on-chip implementation allows concurrent operations to be executed in parallel with maximum efficiency. A parallel architecture also obviates the need for branch prediction, as the execution becomes deterministic. With the resulting improvement in efficiency, the custom hardware can achieve staggering performance gain when running at only a fraction of the clock frequency of a general-purpose microprocessor. A lower clock speed also closes the performance gap between on-chip memory and main memory, eliminating the penalty of cache-miss.

In the meantime, one has to be aware of the constraints and limitations of custom hardware when thinking of hardware-based XML parsing. It almost goes without saying that hardware will generate lots of bits that will be interpreted by application logic. Still, many questions remain. Any sensible technology should at least provide satisfactory answers to the following questions:

- ***Do these bits lose meanings after being transferred into system main memory?***
  Usually the custom hardware owns its own memory space, which is separate from the system main memory. This presents a big problem that disqualifies hardware implementation of many software-based algorithms**:** *pointers lose their meanings when transported into separate address spaces*. Notice that the same principle governs the inner workings of IPC (Inter-Process Communication) and RPC. Passing pointers among threads, on the other hand, is one notable exception because threads share the same heap.

- ***How (easy) does the application logic interpret these bits to achieve the purpose of parsing?***
  This is a critical question in the context of XML, since it isn't as obvious as SSL acceleration where the input and output of the acceleration are easy to understand. Also it touches on the most difficult aspect of the design architecture. Ideally, APIs built on top should offer **usability**, **performance** and **flexibility** so applications can unleash the power of XML to the fullest extent.

- ***What is the impact of the acceleration to the overall performance?***
  A good design should strive to accelerate as much of the task as possible. Suppose a design accelerates only 10% of the overall task by a factor of 1000, the overall performance gain is only a paltry 11%, essentially rendering the acceleration meaningless.

DOM on a chip

Because of its power and flexibility, DOM is a primary candidate for hardware acceleration. Yet there are several reasons, in our view, that make "DOM on a chip" difficult to realize:

- ***Custom hardware can't build hierarchical data structure.***
  Building a DOM tree requires OS' sophisticated memory allocation and management capabilities, as any part of DOM, such as DOM nodes and strings, is dynamically allocated at run time. In contrast, custom hardware manages memory statically—in other words; custom hardware dump bits into memory as if the memory is a large contiguous array, making it ill suited for building a DOM tree.

- ***Hierarchical data structure can't be directly transported across the memory boundary***.
  DOM's nodes are stitched together through the heavy use of pointers, which can't be transported across the memory boundary directly. A bad work-around requires a re-serialization of the data structure. Applications at the receiving end will have to perform parsing all over again, essentially defeating the purpose of high performance hardware parsing.

- ***Prohibitive difficulties of physical to virtual memory mapping***
  Because custom hardware works exclusively with physical addresses, even one manages to transport DOM tree into the main memory, he/she still has to face the impossible mission of mapping the data structure into virtual memory address of the heap. Aside from the improbable task for manipulating page tables, the mapping also requires that selected locations on the heap be unoccupied. Considering the dynamic nature of heap, this is simply too much to ask for.

DOM's memory inefficiency also has negative performance implications. Suppose a 1000x performance increase in building DOM is achievable, the cost of garbage collection, at around 10%~20% of CPU time of building a DOM tree, can put a cap on the overall performance gain. Therefore, whenever possible, one should attempt to optimize garbage collection as well.

SAX on a chip

As a lesser candidate for acceleration, SAX on a chip faces the following technical challenges:

- ***Lack of separation between parsing routine and application logic***
  Using SAX requires the application logic to communicate constantly with the parsing routine. When the SAX parser is replaced by custom hardware, the application logic becomes the weakest link that ties up the hardware and prevents it from accelerating other XML parsing tasks in a multi-threaded environment. The desirable behavior should be that custom hardware quickly processes one XML document, then move on to the next.

- ***Overhead of device driver calls***
  The callback nature of SAX parsing could force applications to make frequent and expensive device driver calls. As a result, any performance gain from the hardware acceleration would inevitably be offset by the overhead of the device calls.

- ***Intermediate format approach***
  Alternatively, one can choose to let custom hardware generate intermediate format, then build SAX on top. But the problem is that it diminishes SAX's ability

to process very large files, which is one of SAX's greatest strengths and a reason why it is invented in the first place. In other words, the "intermediate format" approach makes no sense unless it is DOM-like.

Aside from the above technical challenges, one has to beware of SAX's inherent usability issues. SAX parsing, with the overhead of object creation and tracking the state information, often isn't the most processing-intensive part of an application. In many cases, such as Apache AXIS, people claim to use SAX, but what actually happens underneath is that they are forced to construct in-memory data structure, which results in DOM-like performance and memory consumption. So accelerating SAX doesn't help much in boosting the overall performance.

All things considered, SAX on a chip doesn't solve the most critical problem.

XimpleWare's Solution

Problem Statement

Despite its promises, XML has many issues that need to be addressed before it can fulfill the potential as the foundation for next generation Web applications. As an example, poor performance of XML processing can force enterprises to either delay the rollout of XML applications or buy a lot more hardware to meet the scalability requirement. In addition, as the XML traffic in the network grows, the necessities to classify, route and secure XML data at wire-speed all point to the need for a high performance XML processing engine, which is very different from either network processors based on general-purpose architecture or layer-7 chips doing simple pattern-matching. Furthermore, performance and resource inefficiencies of current XML processing techniques have so far stymied industry-wide transitions to XML-enabled transaction-capable databases.

However, given their respective flaws and limitations, brute-force approaches to port existing XML processing models on chip are difficult to realize.

Technology Overview

All those lead to the significance of XimpleWare's patent-pending technology. *The best way to describe XimpleWare's invention is that it is two birds one stone: the only viable means to port XML on chip happens to be the most memory-efficient as well*. What's more, it possesses many interesting properties that overcome hidden shortcomings of existing processing models. We have sufficient reasons to believe that XimpleWare's technology will not only strongly influence how XML is used across many areas of applications, but also enable many XML applications previously thought impossible. In short, it will have a profound impact and far-reaching implications on the future of the XML revolution.

The starting point of XimpleWare's technology is a new XML processing model. By maintaining the entire document in memory, it gives a complete structural view of the document without incurring DOM's resource overhead. Unlike DOM or SAX, it requires that the original XML document be kept intact in memory. To traverse the data structure, an application navigates the intermediate format, which can be generated blinding fast by custom hardware. Moreover, the processing model allows

validation and XPATH to be implemented on chip as well. The properties of the processing model can be summarized as follows:

- **General purpose**
  The processing model and its ensuing advantages/properties apply to any XML document.

- **High performance**
  The processing model allows the intermediate format to be generated using custom hardware. Our chip running at 200MHz generates the intermediate format at 200MB/sec, outperforming DOM-building with 2GHz PIV by 100 times. What's more, the cost of garbage collection has been dramatically reduced as well. Its direct comparison with DOM again shows how poorly designed DOM is. In other words, no matter what the application logic does, DOM incurs the round-trip cost of tree construction and destruction, which XimpleWare's processing model bypasses altogether.

- **Low memory usage**
  If we have had a good enough discussion on DOM's memory inefficiency issue so far, it should become clear that picking apart the original XML document, which adheres to traditional text processing techniques, makes no sense! Going around it is one of the fundamental reasons why XimpleWare technology is able to reduce memory usage by 5x.

- **Language/Platform neutral and Standard Compliant**
  The processing model doesn't make any assumptions on the type of programming languages or architectures; therefore, any languages or platforms can implement APIs on top of the intermediate format as long as it conforms to the specification. XimpleWare plans to offer APIs compliant to XPATH/XQUERY specifications in addition to data-binding support and a DOM-like native API

- **Inherent Persistence**
  Alluding to the previous discussion on the limitation of custom hardware, one probably won't be surprised by the fact that the intermediate format is inherently persistent. This opens up two possibilities:(1) One can persist the intermediate format on disk along with the original XML document so that no parsing needs to be done when the application loads both the XML document and the intermediate format. (2) One can build the XML intermediary that generates the intermediate format and attaches it to the original XML data to dramatically speed up the performance of application servers at the receiving end. In other words, external XML offloading becomes a reality!

- **Fast serialization performance for remove, modify and insert**
  When applications only make minor changes to XML documents, both DOM and SAX incur the round-trip cost of taking apart the XML document then putting it back. XimpleWare's processing model avoids that cost as it allows the XML document to be manipulated like a piece of buffer. For example, it allows one to extract out any element node in its serialized form. Imagine a B-2-B application has to return, as a response, an XML invoice document that includes the large portion of the incoming XML-based purchase order, one can literally grab the intended elements from the PO and drop it into the invoice—a tremendous gain in serialization performance.

- **Validation and XPATH**
  Often considered critical in a real-time transactional environment, DTD or Schema based validation can significantly slow down XML parsing performance. When ported on chip, validation is done in parallel with the parsing, therefore incurs no extra latency. Our processing model also enables hardware implementation of XPATH.

Areas of Applications

Because XimpleWare's technology accelerates XML at the lowest layer, we expect it to enable a diverse range of XML applications. Below is an incomplete list of possible areas:

- **XML databases /Content Management**
  Our processing model coupled with the hardware acceleration is capable of dramatically boosting the XML database performance. Because of the persistent nature of the intermediate format and low overall memory footprint, when read into memory, the intermediate format allows the database server to query much bigger XML documents without the expensive parsing. Modifications to XML documents also become much more efficient.

- **XML transformations**
  It has been shown that the more optimization people do to the XSLT engine, the more visible that the parsing becomes a bottleneck. For that reason, we expect that our hardware XML engine is a must-have for any high performance XSLT transformation application or appliance.

- **XML middleware applications**
  People implementing performance-critical business applications have always faced the dilemma of choosing between DOM and SAX. The performance is the single biggest reason XML encounters much resistance in middleware applications. By making Web Services scalable and cost-effective at the same time, XimpleWare's technology fundamentally eliminates the toughest roadblocks standing in XML's way; in other words, with our technology, one can no longer make excuses for not using XML.

- **XML intermediaries/XML content switch/Traffic management**
  The primary functionality of XML intermediaries, such as XML firewalls, routers and proxies, is to store and forward XML messages. It usually examines the content of XML messages to determine (1) whether the message is valid and (2) where to forward the message. Depending on the situation, it can also make some changes to the message.

  There are many reasons that make XimpleWare's technology uniquely qualified for XML intermediaries. The most obvious one is performance: XimpleWare's hardware XML parsing engine can keep up with a Gig-bit network. Equally important is the fact that XimpleWare's processing model eliminates the round-trip penalties of de-serialization and serialization with other XML processing models. Not picking apart an XML document also fully preserves the original content and its authenticity—otherwise, security applications would be forced to digitally re-sign the document. Finally, on-chip implementation of validation and

XPATH evaluation further enhances the processing capability of the XML intermediary.

Summary

Performance and processing model are two tough and interwoven issues that prevent people from unleashing the power of XML. By pioneering a new, general-purpose, memory-efficient and hardware-accelerated XML processing model, XimpleWare hopes to help smooth enterprises' transition towards a more ubiquitous use of XML, and along the way, we would like to make this world more efficient, and ultimately a better place for everyone!