Tang J, Thanarungroj P, Liu C *et al.* Pinned OS/services: A case study of XML parsing on Intel SCC. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 28(1): 3–13 Jan. 2013. DOI 10.1007/s11390-013-1308-6

# Pinned OS/Services: A Case Study of XML Parsing on Intel SCC

Jie Tang[1] (唐 洁), *Student Member*, *IEEE*, Pollawat Thanarungroj[2], Chen Liu[2] (刘 晨)
Shao-Shan Liu[3] (刘少山), Zhi-Min Gu[1] (古志民), and Jean-Luc Gaudiot[4], *Fellow*, *IEEE*, *Member*, *ACM*

[1] *School of Computer, Beijing Institute of Technology, Beijing 100081, China*

[2] *Department of Electrical and Computer Engineering, Florida International University, Miami, Florida 33199, U.S.A.*

[3] *Microsoft, Redmond, Washington 98223, U.S.A.*

[4] *Department of Electrical Engineering and Computer Science, University of California, Irvine, California 92617, U.S.A.*

E-mail: tangjie.bit@gmail.com; pthan001@fiu.edu; cliu@fiu.edu; shaoliu@microsoft.com; zmgu@x263.net; gaudiot@uci.edu

Received December 31, 2011; revised May 10, 2012.

**Abstract** Nowadays, we are heading towards integrating hundreds to thousands of cores on a single chip. However, traditional system software and middleware are not well suited to manage and provide services at such large scale. To improve the scalability and adaptability of operating system and middleware services on future many-core platform, we propose the pinned OS/services. By porting each OS and runtime system (middleware) service to a separate core (special hardware acceleration), we expect to achieve maximal performance gain and energy efficiency in many-core environments. As a case study, we target on XML (Extensible Markup Language), the commonly used data transfer/store standard in the world. We have successfully implemented and evaluated the design of porting XML parsing service onto Intel 48-core Single-Chip Cloud Computer (SCC) platform. The results show that it can provide considerable energy saving. However, we also identified heavy performance penalties introduced from memory side, making the parsing service bloated. Hence, as a further step, we propose the memory-side hardware accelerator for XML parsing. With specified hardware design, we can further enhance the performance gain and energy efficiency, where the performance can be improved by 20% with 12.27% energy reduction.

**Keywords** XML parsing, homogeneous multi-core, Intel Single-Chip Cloud Computer

## 1 Introduction

As Moore's law[1] continues to take effect, general-purpose processor design enters the many-core era to break the limit of uni-processor. If the number of cores continues to double with each technology generation, within 20 years we would be looking at integrating more than 10 000 cores on a single chip[2]. However, how to generate enough parallelism at the software level to take advantage of the computing power these many cores provide remains a daunting task for system architect.

On the software side, traditional operating systems are designed for single-core and multi-core, but not for many-core. Usually, OS runs on one host processor and manages the system resources (CPU, memory, I/O) by the time-shared model. Besides, the host processor has to manage task creation and application mapping[3] to maximize the overall throughput of the system. In cloud computing data center environment, which scales up to tens of thousands of cores, the host processor will become the performance bottleneck and seriously hurt the availability and responsiveness of the server. In addition, it incurs extra energy consumption as well.

On the hardware side, Application-Specific IC (ASIC) and general-purpose microprocessor represent the trade-off between the performance and programmability on two ends of the spectrum. With the emergence of the multi-core processors, the two once "exclusive" designs are now seeing a chance for unification. With heterogeneous multi-core processor represented by CELL Broadband Engine[4], we could have a combination of general purpose processors and special processing elements, such that we can minimize the performance and energy overheads of system services in many-core environment.

In the future (and already happening), the majority

Regular Paper

This work is supported by the National Science Foundation of USA under Grant Nos. CCF-1065147, ECCS-1125762, the Scholarship Council of China, as well as the Beijing Institute of Technology Yu-Miao Ph.D. Scholarship of China. 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 neither of the National Science Foundation of USA nor of the Scholarship Council of China.

The preliminary version of the paper was published in the Proceedings of NPC 2011.

of applications would share the same middleware layer and OS services, such as scheduling, common language runtime, browser, security, web applications. Therefore, it is generic to make hardware acceleration for them in a many-core system. Given such background, we propose pinning each OS and runtime system (middleware) service onto a separate core, such that the server becomes always available and highly responsive. Here, we target at general OS services, all of which are ubiquitous enough and worth to accelerate by hardware. In future thousand-core scenario, we can turn off or wake up corresponding cores depending on the application load. If the service is not needed, its dedicated core will keep asleep or shut down, with low or no energy consumption. By doing so, we hope to provide superb FLOPS (floating-paint operations per second) per Watt ratio to system, to greatly reduce the non-recurring cost (hardware investment) and recurring cost (energy bill) of the deployment of servers in the cloud computing data center eco-system.

As a case study, we start our proposal by accelerating XML (Extensible Markup Language) parsing service. XML has been widely used as the standard in data exchange and representation①. It usually resides in the middleware layer in cloud computing environment. Although XML can provide benefits like language neutrality, application independency and flexibility, it also comes with heavy performance overhead[5-6] due to its verbosity and descriptive nature. Generally, in cloud computing environments, XML parsing is proven to be both memory and computation intensive[7-8]. A real-world example would be Morgan Stanley's financial services system, which spends 40% of its execution time on processing XML documents[9]. This situation is only going to get even worse as the XML dataset gets larger and more complicated.

To alleviate the pain in XML parsing, as we proposed, we port the XML parsing service to a dedicated core of Intel Single-Chip Cloud Computer (SCC)[10-13], which is a 48-core homogenous system. By doing so, we can study how it behaves on performance and power consumption. Our results show that when porting XML service onto a homogenous system, we can get considerable energy reduction but huge overhead from the memory side. To overcome this drawback, we further tailor the XML parsing service core into a specified memory-side hardware accelerator. The results turn out to be that the memory-side XML parsing accelerator can achieve both performance and energy efficiency; it is also feasible in terms of bandwidth and hardware cost.

The rest of the paper is organized as follows. We review background of our proposal in Section 2. Then, we introduce the Intel SCC system in Section 3, which is the platform of our proposal. In Section 4, we talk about our experiment methodology. In Section 5, we give the first step of the case study: porting XML parsing service to a dedicated core of SCC and analyzing its performance and energy behaviors. To overcome the overhead from memory-side in XML data parsing, we introduce the specified XML parsing accelerator in Section 6, showing its improvement in performance and energy consumption. In the last section, we make the conclusion and discuss our future work.

## 2 Backgrounds

We give the background information in this section, including our proposal, related work, and XML parsing basics.

### 2.1 Step-by-Step Pinned OS/Service

Considering the diversity of current system architectures, our grand plan is laid out in three steps.

As the first step, we choose to port OS/services onto homogeneous many-core design (such as Intel SCC platform) with one service per core, such that we can study its performance and power consumption. However, we expect that some cores are under-utilized and some cores are over-utilized in this situation since not all services are equally weighted or requested.

As the next step, in order to get the maximal performance gain and energy efficiency, we tailor specialized core (special hardware acceleration) for different service. For heavy-weighted and well-established services (e.g., browser, file manager), which are generic and static (no major changes for extended period of time), we can use ASIC cores for acceleration. For services that are less generic and prone to change (e.g., different cryptography algorithms, even future emerging applications), we can use FPGA accelerators which can be modified at runtime to adjust to applications' need.

As the final goal, we plan to construct a prototype Extremely Heterogeneous Architecture (EHA) by integrating above-mentioned pieces together. This EHA prototype will consist of multiple homogeneous light-weight cores, multiple ASIC (hard) accelerators, and multiple reconfigurable (soft) accelerators; each of these cores will host a service. It is supposed to have the best balance in performance, energy consumption and programmability.

In this paper, we focus on the first step by pinning XML parsing service onto one core in the homogeneous

①International HapMap project. http://hapmap.ncbi.nlm.nih.gov/, Dec. 2011.

many-core Intel SCC; we also delve into the second step by identifying hardware acceleration opportunities to improve the performance of XML parsing service.

### 2.2 Related Work

There have been some work discussing how to decompose OS/service in multi-core systems. FOS[14] is a factored operating system targeting multi-core, many-core, and cloud computing systems. In FOS, each operating system service is factored into a set of communicating servers, which in aggregate implement a system service via message passing. These servers provide traditional kernel services and replace traditional kernel data structures in a factored, spatially distributed manner. Corey[15] is another operating system for multi-core processors, which focuses on allowing applications to direct how shared memory data is shared between cores. However, we believe shared memory model will not scale well for future thousands-of-cores systems. GreenDroid[16] is a prototype mobile application processor designed to dramatically reduce energy consumption in smart phones. GreenDroid provides many specialized processors targeting key portions of Google's Android smart-phone platform. The resulting specialized circuits can deliver up to 18x increase in energy efficiency without sacrificing performance. It also focuses on reconfigurability that allows the system to adapt to small changes in the target application while still realizing efficiency gains. However, GreenDroid targets selected applications such as web browsers, email software and music players for embedded platform.

Different from previous studies, we are targeting the generic OS/middleware services of general systems and cloud platform. In addition to factoring out operating system services, we can also design special hardware (core) to accelerate each service. If available, we can extend more dedicated cores or specialized hardware for the purpose of acceleration. Therefore, it can scale up very well.

### 2.3 XML Parsing

In the experiments, we select XML service as our targeting application. XML has become the standard in data storage and exchange; however it produces high overhead to the system. It has been proven in cloud computing environments XML processing is both memory and computation intensive. It consumes about 30% of processing time in web service applications[7], and has come to be a major performance bottleneck in real-world database servers[8]. As the pre-requisite for any processing of an XML document, XML parsing scans through the input document, breaks it into small elements, and builds corresponding inner data representation or reports corresponding events according to the underneath parsing model② ③. The XML data can be accessed or modified only if it goes through the parsing stage at first. As a result, all those XML data based applications must include the overhead produced in the parsing stage when considering the entire system performance and energy consumption.

There are two kinds of commonly-used parsing models: tree-based model and event-driven model. DOM③ is the official W3C standard for tree-based parser. It reads the entire XML document and creates an inner tree structure to represent the meta-data information. Therefore, it has huge requirement for memory space to keep those structures. However, the constructed tree can be navigated and revised freely, providing flexibility for massive data updates. SAX② is the most popular implementation of event-driven parsing model. It does not store any information of XML document; it just transmits and parses infosets sequentially at runtime, reporting corresponding events. Compared with DOM, it does not stress storage but can only process partial data before parsing is completed.

## 3 Intel SCC

To start the case study on porting XML parsing service, we select Intel Single-Chip Cloud Computer (SCC)[10-13] as our platform, which is a homogenous many-core system with 48 identical cores on the same chip. Intel SCC is built to study many-core CPUs, including the performance and power characteristics, programmability and scalability of shared memory message-passing architecture, and benefits and costs of software-controlled dynamic voltage and frequency scaling. Through experimenting on it, we can discuss how pinning OS/service behaves on both performance and energy consumption.

### 3.1 Overview of SCC Architecture

As seen in Fig.1, Intel SCC contains an integration of 24 tiles and each tile contains two P54C-based full-IA (Intel Architecture) processing cores. The SCC chip is organized in a 6X4 2D mesh network with one embedded high-speed router (R in Fig.1) associated with each tile. The routers enable message passing capability

②SAX Parsing Model: http://sax.sourceforge.net, Dec. 2011.

③W3C. Document Object Model (DOM) level 2 core specification (Version 1.0). http://www.w3.org/TR/DOM-Level-2-Core, Dec. 2011

for SCC cores to communicate with each other. There also exist four on-die memory controllers (MCs in Fig.1) taking charge of off-die DDR3 memory access (DIMM in Fig.1 is the off-die memory module). For control of power, the Voltage Regulator Controller (VRC) is used to adjust the voltage as well as the frequency in both tiles and board area. The whole SCC board is connected to the PCIe (Peripheral Component Interconnect express) for the communication.

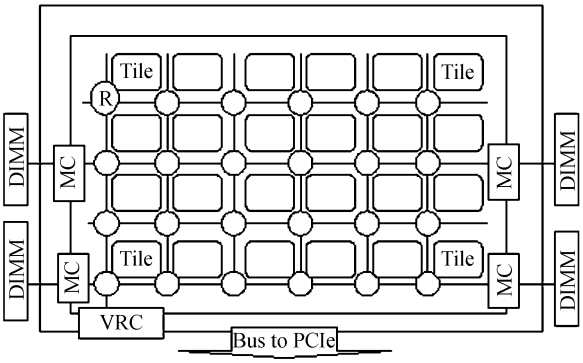



Fig.1. SCC architecture.

### 3.2 SCC Tile-Level Architecture

In Fig.2, we give the tile-level architecture of SCC. Each core on the chip has its own private $L1$ and $L2$ caches. The $L1$ cache has been divided into 16 KB instruction cache and 16 KB data cache. The $L2$ cache is a unified cache of 256 KB in size. A featured small memory unit, called Message Passing Buffer (MPB), is implemented to accelerate the message passing process of each core. Each MPB has a total size of 16 KB shared by two cores. Traffic generator (Gen.) is used to inject and check traffic patterns at runtime to test the performance capabilities of the mesh[13].

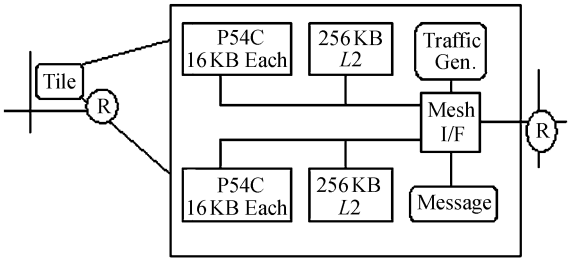



Fig.2. Tile architecture.

### 3.3 Power Management

The component named VRC is the voltage regulator controller. It allows full application control of the power state of the cores. As a result, SCC allows users to adjust voltage and frequency level of the SCC cores dynamically. SCC divides the 48 cores into different frequency and voltage domains as shown in Fig.3, where the square divided by dotted lines denotes the tile of SCC which has two cores inside. On one hand, the 48 cores can be separated into six voltage domains, ranging between 0.7 V and 1.3 V. Each voltage domain contains eight cores and can have its own voltage level. On the other hand, each tile can have its own operating frequency, ranging between 100 to 800 MHz. Thus, SCC has the ability to perform fine-grain dynamic power management[12-13].

Because of the dynamic power management capability of SCC platform, we can assign specific jobs with similar computation demand to one or more dedicated cores on SCC and manage the frequency and voltage levels for the domains those cores belong to. In our study, we dedicate one core to performing XML parsing function. Other cores are allowed to send the request of parsing an XML document to this core and keep working on their XML-independent jobs with high throughput. If needed, we can increase the number of cores dedicated for XML parsing, hence it can scale up well as required.

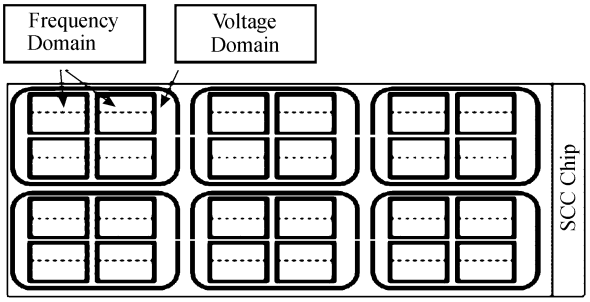



Fig.3. Frequency domains and voltage domains on SCC chip.

## 4 Methodology

In this section, we discuss the methodology to study the performance, power and energy consumption of our proposal, covering the experiment setup, the selected XML parsers we port on SCC and its frequency and voltage setup, the applied benchmarks and profiling tools.

### 4.1 Experimental Setup

Intel SCC platform consists of two hardware components. The first hardware component is called "Board Management Controller" (BMC), which houses the SCC chip. The second hardware component is called "Management Console PC" (MCPC), which is connected to BMC via both PCIe and Ethernet. MCPC allows us to configure and load programs to be executed on the SCC cores[12].

### 4.2 XML Parser on SCC

To study the pinned XML parsing on SCC, we intend to analyze the popular parsing workloads. As introduced in Section 2, there are two kinds of commonly-used parsing models: DOM and SAX. DOM model constructs a tree database to collect data from an XML document. It converts data elements into tree nodes and stores them in memory, which imposes high memory requirement and significant access delay. Especially, when the document size is large, the great impact from the memory access will hurt the system performance badly. That is because DOM needs a huge memory space to keep variables for the tags and data inside, and some of the tags and data are rewritten and recalled frequently. Considering the extreme case that future mass data processing will stress both the memory side and computation side, we focus on DOM parsing model in this study due to its more memory-intensive nature.

We port a lightweight and customizable DOM parser implementation named simplexml④ onto SCC chip. We further modify the parser for handling newer versions of XML and enhance it with the ability to accept XML version tags and short notation of tag ending.

### 4.3 Benchmarks

To make a complementary study, six XML benchmarks have been used in our experiment. These six benchmarks are different in size, number of elements and structure of XML data. Each of the benchmarks represents different types of workload to the XML parser. For some XML benchmarks that cannot be processed entirely due to the memory limitation of the ported XML parser program, we truncate them in size accordingly. For some small sized benchmarks, we run them for several iterations to get accumulated results. In Table 1, we summarize the benchmarks and their adaption.

### 4.4 Power and Energy Consumption Measurement on SCC

To describe the energy consumption characteristic, we monitor execution time and power consumption of XML parsing in different voltage and frequency setups on SCC. Power consumption is calculated from the product of voltage and current over the entire SCC chip. Together with power consumption, we also measure the time elapsed at the end of an iteration. The product between power consumption and execution time per iteration is the chip's energy consumption during the iteration. Thus, the sum of the energy consumption from all iterations is the total energy consumption. Hence, we run each benchmark on a single core with 21 pairs of frequency and voltage levels. To evaluate the performance of Intel SCC, we employ a performance monitoring tool, HPCToolkit⑤ to profile the XML parsing on SCC platform. It measures the hardware performance of a program execution via the performance counters available on the processor and merges all the profiling data into a statistical table.

## 5 XML Parsing on Intel SCC

In this section, we evaluate pinning XML parsing service in SCC from both performance and energy sides. We also give the analysis of how to get better execution efficiency on execution time as well as energy consumption.

### 5.1 Performance Evaluation

To evaluate computation and memory intensiveness in DOM parsing, using HPCToolkit[17] we compare the six real-world benchmarks with a synthetic computation-intensive benchmark named speedtest. It performs basic recursive arithmetic computation for 1 billion times, where the memory miss rate is extremely

**Table 1.** XML Parsing Benchmarks

| Benchmark | Size (KB) | Description | Iterations | Modification |
|---|---|---|---|---|
| long.xml (Long) | 65.7 | A few tags, long data length | 100 | None |
| mystic-library.xml (Mystic) | 1 384.0 | Massive repetition of the several tags, short data length | 1 | Truncated to 146.4 KB in size |
| personal-schema.xml (Personal) | 1.4 | A small case from employee database | 100 | None |
| physicsEngine.xml (Physics) | 1 171.0 | Massive repetition of the same tag, long data length | 1 | Truncated to 990.3 KB in size |
| resume_w_xsl.xml (Resume) | 51.8 | Various information under the same tag pattern | 10 | Truncated to 11.1 KB in size |
| test-opc.xml (Test-Opc) | 1.8 | A few repetitions of a few tags, short data length | 1 000 | None |

④Ecker B. Simple XML. http://www.omegadb.net/simplexml, Dec. 2011.

⑤Rice University. HPCToolkit, http://hpctoolkit.org, Dec. 2011.

low because of its high data locality. In Fig.4 we show the performance measurement in term of Instructions per cycle (IPC). Obviously, the computation-intensive benchmark speedtest achieves a good execution performance with 1.29 instructions per cycle. That is more than double of the best IPC of XML benchmarks. For the rest XML benchmarks, their IPCs are far less than that of speedtest. In the worst case, PhysicsEngine, the IPC is just about 0.06, even cannot reach 5% of speedtest's IPC. When we analyze the results, we can find that: speedtest is a computation-intensive benchmark; hence its performance is mostly determined by the computation rather than memory subsystem. Considering the huge gap between their IPCs, the results imply that the performance bottleneck of XML parsing may actually be memory access latency. Looking into each XML benchmark, different benchmarks perform differently according to their file size and structure. If there are massive tag repetitions and bulky data stored inside the tag, the XML benchmarks always have to pay considerable cost to memory accesses, leading to a very low IPC.

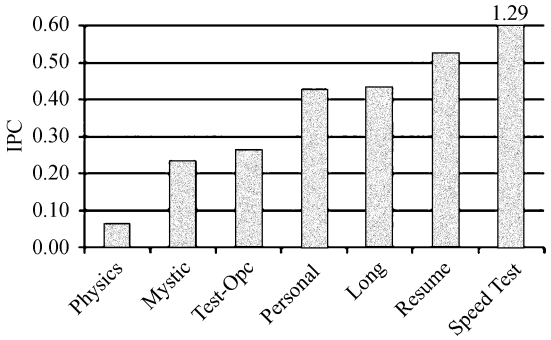



Fig.4. Comparison of the IPC among the benchmarks.

To further backup our observations in Fig.4, we collect the values of MPKI (misses per kilo instructions) on both $L1$ instruction cache and $L1$ data cache, and list them in Fig.5. Obviously, the speedtest benchmark has the lowest MPKI because it is really a computation-intensive benchmark with very few long-latency memory accesses. It matches our previous finding. All the XML benchmarks have comparative higher MPKIs. The worst case is benchmarks PhysicsEngine and Mystic-library. In $L1$ instruction cache, PhysicsEngine has about 9.25 MPKIs. In $L1$ data cache, it even has average 45.6 MPKIs. The data means when running parsing service, the parser keeps requesting long-latency memory accesses. The more cache misses incurred, the higher performance overhead the memory subsystem imposes. Hence, the IPC of PhysicsEngine is the lowest one as it possesses the highest MPKI.

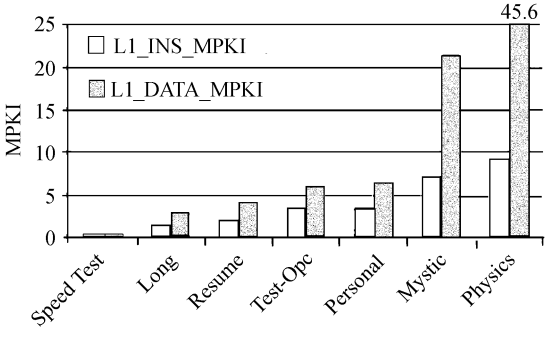



Fig.5. MPKI of both $L1$ instruction cache (L1_INS_MPKI) and $L1$ data cache (L1_DATA_MPKI).

However, $L1$ MPKIs cannot absolutely imply that the benchmark is memory-intensive. Therefore, we take advantage of another event, duration of pipeline stalled by data memory read, which provides us more information on the $L2$ cache and main memory accesses. This duration has been compared with the total execution time in a percentage format for each benchmark. Once a miss occurs in $L1$ cache, the targeted data will be requested from the lower memory hierarchy. No matter it results in a hit or miss, such memory behavior will directly influence the duration of pipeline stalled by data memory access. The longer the duration of pipeline stall and the lower level of memory subsystem the access happens, the more performance overhead it incurs to the system. Thus, we can use the duration of pipeline stalled by data memory access as a metric to determine how memory-intensive an application is.

In Fig.6 we compare the percentage of pipeline stall duration of each benchmark, normalized to the total execution time. Without any doubt, speedtest gets least stalled pipeline time percentage because it performs simple computations and most of its memory operation will result in cache hits. For XML benchmarks, averagely percentage of duration of pipeline stalled by data memory read is 13.8%, meaning 13.8% of total execution time is consumed by memory-side operations. For the most intensive one, Long, it even takes 35.29% of the overall execution time. This confirms the finding

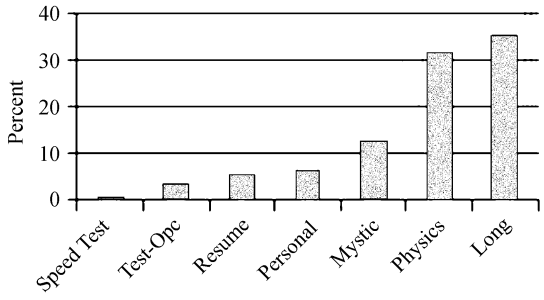



Fig.6. Percentage of duration of pipeline stalled.

that XML parsing service introduces high overhead to the system and most of the overhead is jammed in the memory side, leading to low performance.

### 5.2 Power and Energy Consumption

In this subsection, we list the results of power and energy consumption when porting the XML parsing service onto a dedicated core. We also show how power and energy consumption varies with different frequency and voltage combinations.

In Fig.7 we show results that relate the total execution time with the execution frequency of the SCC core. We do not plot them against the voltage level since the total execution time does not depend on the supply voltage; it only depends on the frequency of execution. Each benchmark finishes its execution earlier with higher frequency setup. The decaying execution time slope turns flat when we keep increasing the core's frequency. That is because higher execution frequency makes more instructions executed per unit time, shortening the total execution time. However, the total execution time is not only determined by the frequency. The overhead from long-latency memory accesses poses great limit for the execution optimization, leaving less room for higher frequency to get more performance gain. Actually, when we define the efficiency as the execution time over operating frequency, the efficiency degrads up to 14.24% from 100 MHz to 800 MHz.

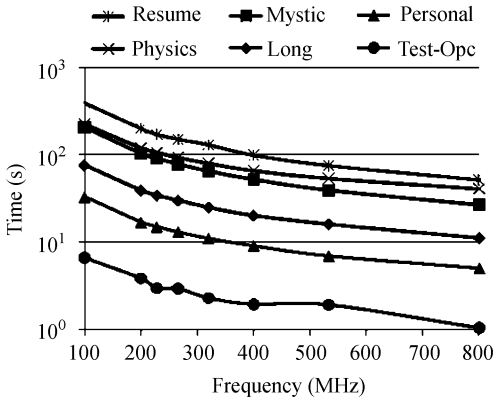



Fig.7. Total execution time spent at different frequency levels within 1.1 V.

Because we execute the XML parser on a single core, the time, power and energy characteristics of the benchmarks are following the same trend. Thus, we only show the rest of the results for benchmark Long.

In Fig.8, we show the average power consumption for benchmark Long at different frequency and voltage setups. We can find our results match perfectly with the power consumption mathematical model:

$$P \propto CV^2 f. \tag{1}$$

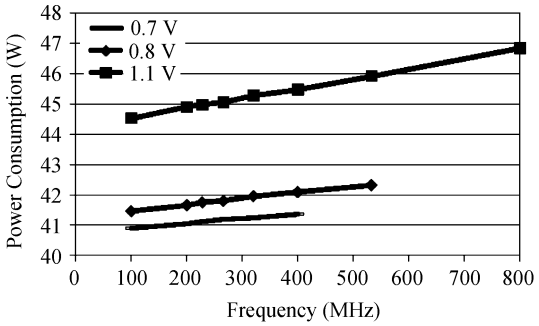



Fig.8. Average power consumption of the Long benchmark.

Power consumption is quadratic increasing with the supplying voltage: $P \propto V^2$. When configuring with the 1.1 V voltage, its power consumption jumps over those in other lower voltages. Meanwhile, power consumption is growing linearly with frequency: $P \propto f$, so that the power consumption gradually keeps growing with the increasing operating frequency on the same voltage setups.

Fig.9 shows the total energy consumption of parsing benchmark Long when applying three different supply voltage levels at different frequencies. The energy consumption ranges from 3 500 joules to 500 joules with different configurations. In 0.7 V and 100 MHz setting, the energy consumption is about 3 000 joules. Please note the energy consumption here refers to the entire SCC chip.

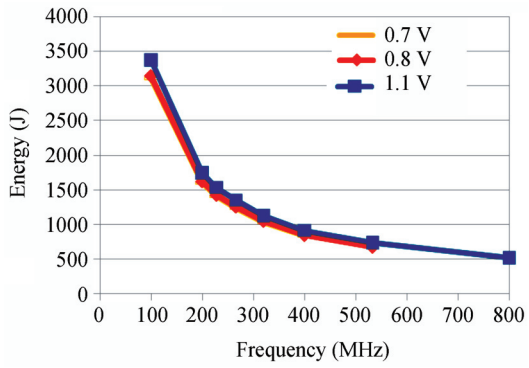



Fig.9. Total energy consumption of the Long benchmark.

To better understand these energy consumption numbers, we make a comparison between the energy taken to execute the benchmark on Intel SCC versus the energy taken to execute it on a normal desktop CPU: we run the XML parsing service on an Intel Core 2 Duo P8600 CPU and using the Vtune tool[⑥] to capture

[⑥] Intel® VTune™ Amplifier XE 2013. http://software.intel.com/en-us/intel-vtune-amplifier-xe, Dec. 2011.

its energy consumption. We find when porting parsing service onto a dedicated core of SCC, the energy consumption has been reduced by more than 40%, and note that with fine-grain power management techniques to reduce the static power consumption of the idle cores, this gap (energy saving) can be further increased. This piece of data indicates that by pinning services to each core of many-core machines, we could achieve great energy efficiency to the system, especially for cloud data center where innumerable XML data need be processed.

In Fig.9, we can also find that for a given frequency, SCC chip gets the highest energy consumption when we execute benchmark Long at 1.1 V. There is only subtle difference in the total energy consumption between 0.7 V and 0.8 V supply voltages. The results suggest that to achieve better energy efficiency for a configured frequency, it is better to lower voltage level in the execution. If we continue increasing the frequency, due to technology limitation, only voltage at 1.1 V can support such high frequencies (533 MHz to 800 MHz). Hence, when running at 800 MHz frequency within 1.1 V voltage, SCC chip just consumes 500 joules, 1/6 of that in 100 MHz and 0.7 V configuration. This is because in such configuration workload can be completed more quickly and the reduced execution time can compensate for the energy consumption.

However, as stated before, in SCC chip two cores in one tile are set with the same frequency, and eight cores in the same domain are set with the same voltage as well. Hence, our pinned XML parsing core cannot be configured separately. It has to share its frequency level with another SCC core in the same tile and share its voltage level with another seven cores in the same voltage domain. As a result, to enlarge the energy efficiency, there exist two scenarios: considering frequency wise and considering voltage wise. To consider voltage wise, given a fixed frequency level with a neighboring core, the XML parsing core should request the lowest supply voltage level available inside its voltage domain to save energy consumption. To consider frequency wise, given a fixed voltage level, the XML parsing core should request to run at the highest frequency level available inside its frequency domain to save execution time.

### 5.3 Summary

The evaluation of pinning XML parsing service on SCC from both performance and energy sides indicates that the performance of XML parsing suffers from intensive memory access pattern. The performance evaluation verifies the computation and memory intensiveness of the XML parsing by reporting up to 0.53 IPC while incurring up to 45.6 MPKI $L1$ misses and 35.29% pipeline stall duration due to memory access. These results recommend an improvement on memory side due to those penalties.

The chip voltage level, which is directly proportional to the power consumption, has a small influence on the overall energy consumption when compared with the change in operating frequency. Therefore, if the system supports, we would better maximize frequency and minimize voltage supply on SCC to get the optimal energy consumption. Our experimental results also indicate a potential on Intel SCC to elevate energy-efficient run-time execution on many-core environment.

## 6 Memory-Side Acceleration for XML Data Parsing

In previous section, we have found that: when pinning XML parsing service onto a dedicated core of Intel SCC, there are heavy penalties incurred from the memory side, which leaves less room for performance improvement. To alleviate this pain, we proposed and evaluated the memory-side XML parsing accelerator, which is a tailored hardware prefetching engine dedicated for memory-side acceleration. It is a generic acceleration scheme independent of the parsing model applied underneath. Even there are any updates for parsing algorithm, the memory-side accelerator keeps working. For more details of the acceleration work, please refer to [19].

In the following subsections, we first introduce the selected eight hardware prefetchers and applied profiling tools for evaluation. Second, we give the results to show how memory-side XML parsing accelerator can improve the performance of DOM modeled parsing.

### 6.1 Prefetchers

As we can see from previous sections, the performance bottleneck of XML parsing actually comes from memory access latency. Thus, as a natural step forward in our grand plan (please see Subsection 2.1 for more details), we start looking into hardware techniques to hide memory latency, or to accelerate from the memory-side. In our study, we evaluate how different prefetching techniques behave as the memory-side accelerator to impact the performance of XML. To make a complementary discuss, we select eight prefetchers named $n1$ through $n8$, all of which are different in size, algorithm and complexity. We summarize these prefetchers in Table 2:

*Cache hierarchy*: the coverage of the prefetching, means which hierarchy/hierarchies the prefetching is applied to;

*Prefetching degree*: suggests whether the prefetcher can adjust its aggressiveness statically or dynamically.

**Table 2.** Summary of Prefetchers

| | Cache Hierarchy | Prefetch Degree | Trigger $L1$ | Trigger $L2$ |
|---|---|---|---|---|
| $n1$ | $L1$ & $L2$ | Dynamic | Miss | Access |
| $n2$ | $L1$ | Static | Miss | N/A |
| $n3$ | $L1$ & $L2$ | Dynamic | Miss | Miss |
| $n4$ | $L1$ | Static | N/A | N/A |
| $n5$ | $L2$ | Static | N/A | Miss |
| $n6$ | $L1$ & $L2$ | Dynamic | Miss | Miss |
| $n7$ | $L2$ | Static | Miss | Access |
| $n8$ | $L2$ | Static | N/A | Access |

Usually, the dynamic prefetching degree can adapt itself to the phase change of the application so as to produce more efficient prefetching;

*Trigger* $L1$ and *trigger* $L2$: show the trigger set for covered cache hierarchy respectively. In this case, demand "*access*" stands for access requests from upper memory level regardless whether it is a miss or hit, and *N/A* means no prefetching is applied.

All selected prefetchers can filter out redundant access requests.

### 6.2 Performance and Memory Profiling Tools for Prefetchers

To study the performance of the memory-side acceleration, we utilize CMP$IM[20] 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 3.

**Table 3.** Simulation Parameters

| | |
|---|---|
| Frequency | 1 GHz |
| Issue width | 4 |
| Instruction window | 128 entries |
| $L1$ data cache | 32 KB, 8-way, 1 cycle |
| $L1$ instruction cache | 32 KB, 8-way, 1 cycle |
| $L2$ unified cache | 512 KB, 16-way, 20 cycles |
| Main memory | 256 MB, 200 cycles |

To understand the implementation feasibility of the memory-side accelerators, we also study the energy consumption of these designs. To model their energy consumption, we utilize CACTI[18], 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.

### 6.3 Performance Evaluation

We choose DOM parser implementations from Apache Xerces⑦ as the studied workload. Using CMP$IM, we can understand how much performance gain hardware accelerator can achieve. Table 4 summarizes the reduction of cache misses as a result of applying the prefetchers. Here, we only focus on the cache miss reduction of the lowest level cache that the prefetcher is applied to. For example, $n1$ is applied to both $L1$ and $L2$ caches, so 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 for 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 85% in DOM.

**Table 4.** Cache Miss Reduction

| | $n1$ | $n2$ | $n3$ | $n4$ | $n5$ | $n6$ | $n7$ | $n8$ |
|---|---|---|---|---|---|---|---|---|
| DOM (%) | 77 | 52 | 85 | 85 | 61 | 52 | 77 | 84 |
| Cache Level | $L2$ | $L1$ | $L2$ | $L1$ | $L2$ | $L2$ | $L2$ | $L2$ |

In Fig.10, we show how the cache miss reduction is translated into performance improvement on DOM parsing. The results indicate that prefetching techniques are able to improve DOM parsing performance by up to 20%. When averaging the results, memory-side acceleration produces 13.74% execution cycle reduction for mystic-library. It is obvious that $n3$ is the most effective prefetcher: even in the worst case, it can still reduce execution time by 6%. Since 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.

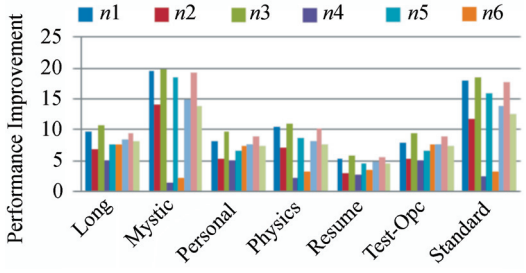



Fig.10. Performance improvement for DOM parsing.

⑦Apache Xerces™ project. http://xerces.apache.org/index.html, Dec. 2011.

## 7 Conclusions and Future Work

In this study, we proposed to port OS/service to a dedicated core to achieve both performance and energy efficiency. In our grand plan, it was implemented by three steps: first, to port OS/services onto homogeneous many-core design with one service per core; second, to tailor specialized core for different services to achieve performance and energy efficiency; third, to build a prototype extremely heterogeneous architecture (EHA) by integrating multiple homogeneous light-weight cores and multiple specified accelerators together.

In this paper, as a case study, we targeted XML parsing service, which is the bottleneck of all XML-based applications. We first ported XML parsing service onto Intel SCC, a 48-core homogeneous system. We found it shows great energy efficiency, which is very important for future energy-efficient applications. However, we also found that heavy performance penalty is introduced from memory side, making the parsing middleware bloated. Hence, as a further step, we proposed to make hardware acceleration for XML parsing. With specified hardware accelerator, we can get up to 20% performance gain. This case study focuses on the first stage of our grand plan and delves into the second stage as well. The results of this case study indicate that our proposal is feasible in achieving performance and energy efficiency, and encourage us to move forward with the grand plan.

To make further summary, we believe our design is suitable for the server side of cloud computing data center eco-system. At first, our design does not stress out one core but distributes the system services across cores instead. Thus this design can support extremely high throughput, allowing the server to be highly responsive. Meanwhile, distribution of workload brings fine-grain control of resources. In future thousand-core scenario, we will not be able to turn on all the cores at the same time due to the constrained overall power budget. In this case, most of the cores will be turned off and will be woken up depending on the application load. So do the system services. By running OS services on dedicated cores, we will have multiple specialized cores dedicated to the same system service. We can implement power management module to turn on and off the cores adaptively according to the operating system load as necessary, resulting in lower power consumption and energy overhead.

As future work, we aim to extend our technique to a broader scope: to pin and accelerate different heavy-weighted system services, for example the garbage collection[21-22]. By doing so, we can check how it can achieve middleware execution efficiency and maximum overall system throughput.

## References


[1] Moore G E. Cramming more components onto integrated circuits. *Electronics*, 1965, 38(8): 114-117.

[2] Gries M, Hoffmann U, Konow M, Riepen M. SCC: A flexible architecture for many-core platform research. *Computing in Science & Engineering*, 2011, 13(6): 79-83

[3] Liu L, Li X, Chen M, Ju R D C. A throughput-driven task creation and mapping for network processors. In *Proc. the 2nd Int. Conf. High Performance Embedded Architectures and Compilers*, January 2007, pp.227-241.

[4] Kahle J A, Day M N, Hofstee H P, Johns C R, Maeurer T R, Shippy D. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 2005, 49(4/5): 589-604.

[5] Chiu K, Govindaraju M, Bramley R. Investigating the limits of SOAP performance for scientific computing. In *Proc. the 11th Int. Symp. High Performance Distributed Computing*, July 2002, pp.246-254.

[6] Head M R, Govindaraju M, van Engelen R, Zhang W. Benchmarking XML processors for applications in grid web services. In *Proc. Conf. Supercomputing*, November 2006, Article No.121.

[7] Apparao P, Bhat M. A detailed look at the characteristics of XML parsing. In *Proc. the 1st Workshop on Building Block Engine Architectures for Computers and Networks*, October 2004.

[8] Nicola M, John J. XML parsing: A threat to database performance. In *Proc. the 12th Int. Conf. Information and Knowledge Management*, November 2003, pp.175-178.

[9] Apparao P, Iyer R, Morin R *et al.* Architectural characterization of an XML-centric commercial server workload. In *Proc. the 33rd Int. Conf. Parallel Processing*, August 2004, pp.292-300.

[10] Howard J, Dighe S, Hoskote Y *et al.* A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proc. IEEE Int. Solid-State Circuits Conference Digest of Technical Papers*, February 2010, pp.108-109.

[11] Mattson T G, Riepen M, Lehnig T *et al.* The 48-core SCC processor: The programmer's view. In *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis*, November 2010, pp.1-11.

[12] Intel labs. SCC platform overview. http://communities.intel.com/docs/DOC-5512.

[13] Jim H. Single-chip cloud computer. In *Proc. Intel Labs Single-Chip Cloud Computer Symposium*, February 2010.

[14] Wentzlaff D, Agarwal A. The case for a factored operating system (FOS). Technical Report, MIT-CSAIL-TR-2008-060, MIT CSAIL, October 2008.

[15] Boyd-Wickizer S, Chen H, Chen R *et al.* Corey: An operating system for many cores. In *Proc. the 8th USENIX Symp. Operating Systems Design and Implementation*, December 2008, pp.43-57.

[16] Goulding N, Sampson J, Venkatesh G *et al.* GreenDroid: A mobile application processor for a future of dark future. In *Proc. the 22nd Hot Chips*, Aug. 2010.

[17] Adhianto L, Banerjee S, Fagan M *et al.* HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 2010, 22(6): 685-701.

[18] Shivakumar P, Jouppi N P. CACTI3.0: An integrated cache timing, power, and area model. Technical Report, Compaq Western Research Laboratory, Feb. 2001.

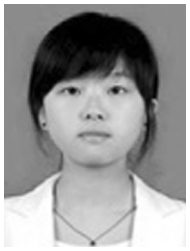
[19] Tang J, Liu S S, Gu Z M, Liu C, Gaudiot J. Memory-side acceleration for XML parsing. In *Proc. the 8th IFIP*

*Int. Conf. Network and Parallel Computing*, October 2011, pp.277-292.

[20] Jaleel A, Cohn R S, Luk C K, Jacob B. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *Proc. the 4th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2008.

[21] Tang J, Liu S S, Gu Z M *et al.* Hardware-assisted middleware: Acceleration of garbage collection operations. In *Proc. the 21st Int. Conf. Application-Specific Systems, Architectures and Processors*, July 2010, pp.281-284.

[22] Tang J, Liu S S, Gu Z M *et al.* Achieving middleware execution efficiency: Hardware-assisted garbage collection operations. *Journal of Supercomputing*, 2012, 59(3): 1101-1119.



**Jie Tang** is a Ph.D. candidate in Beijing Institute of Technology, China. She got a B.S. degree in computer science from National University of Defense Technology, Changsha. During her Ph.D. study, Jie also worked as a visiting researcher in the Center for Embedded Computer Systems, University of California, Irvine, USA.

**Pollawat Thanarungroj** is currently a Ph.D. student at Electrical and Computer Engineering Department of Florida International University, USA. He received his B.E. degree in computer and network engineering, in March 2009, from Assumption University, Bangkok, Thailand. His current work is to perform profiling for SPEC CPU 2000 benchmark programs using Pintool and Simpoint.

**Chen Liu** is an assistant professor in the Department of Electrical and Computer Engineering at Florida International University, Miami, USA. He received the B.E. degree in electronics and information engineering from University of Science and Technology of China, in 2000, the M.S. degree in electrical engineering from the University of California, Riverside, USA, in 2002, and the Ph.D. degree in electrical and computer engineering from the University of California, Irvine in 2008. 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 the IEEE and IEEE Computer Society. He also served as the chair of Computer Society Chapter, IEEE Miami Section from 2010 to 2011.

**Shao-Shan Liu** is currently with Microsoft. He received his Ph.D. degree in computer engineering, M.S. degree in biomedical engineering, M.S. degree in computer engineering, and B.S. degree in computer engineering, in 2010, 2007, 2006, and 2005 respectively, all from the University of California, USA. His research interests include parallel computer architectures, embedded systems, runtime systems, as well as biomedical engineering.

**Zhi-Min Gu** is a professor of computer science at Beijing Institute of Technology. Prior to that he was a visiting scholar of computer science at University of Birmingham in UK from 2003 to early period of 2004, and an associate professor and post-doctorate researcher of computer science at Northwest Polytechnic University from 1997 to 1999. He received the B.S. degree in computer science from Shanxi University in 1985, the M.S. degree in computer science from Harbin Institute of Technology in 1991, and the Ph.D. degree in computer science from Xi'an Jiaotong University in 1997, all in China.

**Jean-Luc Gaudiot** received the Diplôme d'Ingénieur from the Élole Supérieure d'Ingénieurs en Electrotechnique et Electronique, Paris, France in 1976 and the M.S. and Ph.D. degrees in computer science from the University of California, Los Angeles, USA 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 (UCI). 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 the director of the Computer Engineering Division for three years. His research interests include multithreaded architectures, fault-tolerant multiprocessors, and implementation of reconfigurable architectures. He has published over 170 journal and conference papers. In January 2006, he became the first Editor-in-Chief of IEEE Computer Architecture Letters. From 1999 to 2002, he was the Editor-in-Chief of the IEEE Transactions on Computers. In June 2001, he was elected as the 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, ACM SIGARCH, and IEEE. He has also served as program chair of several international conferences. In 1999, he became a Fellow of the IEEE. He was elevated to the rank of AAAS Fellow in 2007.