Importance of Coherence Protocols with Network Applications on Multi-Core Processors

Kyueun Yi, Won W. Ro, Member, IEEE, Jean-Luc Gaudiot, Fellow, IEEE

Abstract—As Internet and information technology have continued developing, the necessity for fast packet processing in computer networks has also grown in importance. All emerging network applications require deep packet classification as well as security-related processing and they should be run at line rates. Hence, network speed and the complexity of network applications will continue increasing and future network processors should simultaneously meet two requirements: high performance and high programmability. We will show that the performance of single processors will not be sufficient to support future demands. Instead, we will have to turn to multi-core processors which can exploit the parallelism in network workloads. In this paper, we focus on the cache coherence protocols which are central to the design of multi-core based network processors. We investigate the effects of two main categories of various cache coherence protocols with several network workloads on multi-core processors. Our simulation results show that token protocols have a significantly higher performance than directory protocols. With an 8-core configuration, token protocols improves the performance compared to directory protocols by a factor of nearly 4 on average.

Index Terms—Parallel processors, Cache memories, Multithreaded processors, Network communications, Data communications

1 INTRODUCTION

Network-based applications including Internet Protocol Television (IPTV), wireless communications, sensor network, ubiquitous computing, etc. have tremendously grown in importance in recent years. This trend translated in a need for ever more powerful network processors since applications will continue demanding even faster processing of incoming packets and requiring more powerful computing platforms dedicated to the processing of packets. Therefore, using high-performance microprocessors as a dedicated resource for network applications has been proposed by several authors, including Nemirovsky et al. [1], Crowley et al. [2] and many others [3].

Newer network applications such as QoS, URL matching, virus detection, intrusion detection, and load balancing require deep packet classification processing [4] and security-related processing which is more computation-intensive than any other network applications [5]. While all these network applications should be running at line rates, most programmable network processors on the market today aim at relatively low performance (from 100 Mbps to 10 Gbps) [6].

As modern processors have focused on exploiting Instruction-Level Parallelism (ILP), they have been quite successful at it and there is little room left for improvement [7]. This is why two alternate microarchitectures in wide-issue superscalar machines have appeared: Simultaneous MultiThreading (SMT) (Tullsen et al. [8]) and Chip MultiProcessor (CMP) (Olukotun et al. [9]). They both aim at exploiting multiple threads (Thread-Level Parallelism - TLP). However, the general trend in microprocessor design has been to embed an increasing number of cores in processor chips rather than the traditional approach of increasing the clock frequency to improve the performance [10]. This is due to the diminishing power-efficiency of high-frequency processors and the correspond-
ingly increasing power consumption problem of modern processors. Since further frequency scaling can no longer yield significant performance improvements without unreasonably increasing power consumption, the concept of single processor on a chip is likely to wane in the very near future [11]. Instead, multiprocessors or multithreaded architectures will likely be the baseline architectures of future network processors [10]. Indeed, it has been shown that single Chip Multiprocessors perform 50-100% better than wide-issue superscalar [9] with thread-level parallelism and multiprogramming workload. For all these reasons, in this paper, we select multi-core architecture as the baseline of network processors.

In a multi-core processor, the performance of the memory hierarchy is more important than ever, since multi-core processors share memory resources. When multiple processor cores are integrated on a single chip with a shared cache, two different processor cores can have different values for the same location of the shared cache. This cache coherence problem has been solved by various protocols (such as the snooping protocol or the directory protocol) [12], [13], [14]. The coherence protocol and the access latency for the shared memory inevitably affect overall system performance [15].

The architectural implications of cache coherence protocols have been investigated with commercial workloads, multiprogramming and OS workloads, and scientific/technical workloads [12]. In addition, Martin et al. [14] have presented the Token Protocol which uses tokens to control the read/write permission of shared cache blocks. They have compared the performance of the token protocol with other cache coherence protocols such as the snooping protocol and the directory protocol with commercial workloads [13]. More specifically, their approach outperforms directory protocols with only a relatively small bandwidth overhead.

In this paper, we investigate the performance implications of cache coherence protocols on multi-core based processors with network workloads. The performance numbers of each protocol (the directory protocol and the token protocol) are measured against each other, as the number of processor cores is made to vary. Our results show that the token protocol performs better with a variety of workloads than the directory protocol does. As the number of processor cores is increased, the number of instructions which are used to complete the application is also increased due to the multithreading mechanisms and the cache coherence protocol. The goal of this paper is to provide guidelines to the design of caches in the multi-core processors in the context of network workloads.

The rest of this paper is organized as follows: Section 2 describes past research on architectural implications of network workloads on single thread as well as cache coherence protocols on multi-core processors. Our simulation environment and the corresponding performance results are presented in Section 3 and Section 4, respectively. Finally, we summarize our observations in Section 5.

2 Background Research

To lay the groundwork for our study of the two different cache coherence protocols, we now first discuss some background on the use of network processors, followed by a brief review of cache coherence protocols. Finally, several related research projects on multi-core based network processors can be surveyed.

2.1 Use of Network Processors

As messages travel through the Internet, the intermediate routers process incoming packets for operations such as packet decoding, packet encoding, packet forwarding, etc. The processing speed of those operations becomes a crucial factor if we are to meet today’s high network speeds. In order to avoid making packet processing part of the bottleneck of computer communications, network processors have been developed: they are programmable processing units specifically intended to assist in the above mentioned operations [16].

Originally, network processing would take place in a fully software-oriented form of general-purpose processors. Using general-purpose processors provided great flexibility
and programmability: they were easy to program and modify [17]. However, software programs are inherently limited in terms of processing speed. In addition, using a single general-purpose processor leads to a scalability problem which is another inherent downside of any sequential software approach.

To remedy the shortfalls of general-purpose processors as network processors, ASIC (Application Specific Integrated Circuit)-style network processors have been used, typically for the purpose of managing traffic at very high speeds. However, this hardware-based approach is always costly and suffers from long development lags. Even worse, these systems do not exhibit the flexibility which would be needed as Internet traffic continues increasing [18] and the network protocols are becoming more dynamic and sophisticated [19].

To address these issues, new types of network processors have been developed. This means that today’s network processors comprise some form of ASIP (Application Specific Instruction Set Processor) which provides an instruction set specialized for network applications in order to provide sufficient flexibility [20]. These network processors are attached to each port of the routers to provide the required high processing speed. The main advantages of using an ASIP over an ASIC approach in the design of network processors include high flexibility and scalability which can be easily achieved by a simple software upgrade.

### 2.2 Cache Coherence Protocols

As we know, when multiple cores are integrated on a single chip, the individual cores often share a cache. If multiple processors share a cache, two different processors can have different values for the same location of the memory space, resulting in a cache coherence problem. A cache system is said to be coherent if any read of a memory location returns the most recently written value of that data element [21]. Cache coherence for multiple processors can be maintained with a variety of sophisticated cache coherence protocols which make all the processors have a consistent view of the shared cache and manage the reading and writing of data in the shared cache [13], [21].

<table>
<thead>
<tr>
<th>Name of State</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified state</td>
<td>No other processor has a copy of the data block. The copy of the data in main memory is incorrect.</td>
</tr>
<tr>
<td>Owned state</td>
<td>Only one processor can be in the owned state. All other processors can have a copy of the most recent in the shared state. The copy of the data in main memory can be incorrect.</td>
</tr>
<tr>
<td>Exclusive state</td>
<td>No other processor has a copy of the data. The copy of the data in main memory is also the most recent.</td>
</tr>
<tr>
<td>Shared state</td>
<td>Other processors can also have copies of the data in the shared state. The copy of data in main memory is the most recent.</td>
</tr>
<tr>
<td>Invalid state</td>
<td>Either main memory or another processor cache can have valid copies.</td>
</tr>
</tbody>
</table>

**TABLE 1: The MOESI coherence state model**

The primary duty of any cache coherence protocol is to track the state of the shared data blocks. In this work, we have assumed that the MOESI coherence state model [22] for our simulated architectures since it can be supported by the two cache coherence protocols used in this work. Each cache block can be in any of five states in the MOESI coherence state model shown in Table 1. The MOESI cache coherence state model can use different cache coherence protocols which track the sharing status of a copy of a block of a shared cache. The three possible kinds of protocols we consider are briefly described below:

- **Snooping protocols:** The cache of each processor has a copy of a block of shared cache and a copy of the sharing status. Snooping protocols do not have a centralized location to maintain the states of the various cache blocks. Instead, the cache controller of each processor continuously snoops on the bus to “hear” whether it has a copy of any block currently requested on the bus. The necessity for such information to be broadcast limits the scalability of bus-based snooping protocols.

- **Directory protocols:** The sharing status of a block is maintained in the directory of the home node. The directory keeps information as to which caches have
copies of the block, whether it is dirty, etc. Each access to a cache block of the shared cache requires to first access the directory to find the state of the cache block. Since directory protocols do not use the bus like snooping protocols do, directory protocols do not need for the processors to monitor the interconnection network.

- The Token Protocol with broadcast: The token is the base unit which is used to control the read/write permissions to the shared cache blocks in the token protocol. The token protocol [13] exchanges and counts tokens to control read/write permissions to the shared cache blocks. Each logical block of a shared cache has a fixed number of tokens. When each processor has at least one of the block’s tokens, it can read the cache block. When each processor has all of the block’s tokens, it can write the cache block.

2.3 Related Work

In an investigation of architectural implications with network workloads on CMP, Crowley et al. [23] have compared the performance of different architectures such as SuperScalar (SS), fine-grained multithreading (FGMT), single Chip MultiProcessing (CMP), and Simultaneous MultiThreading (SMT). By considering equivalent processor resources and dynamically exploiting both instruction-level parallelism and thread-level parallelism, this study shows that SMT yields better performance than CMP, FGMT, and SS by a factor of two.

Melvin et al. proposed massively multi-threaded packet processors for those “stateful” networking applications (those which are required to support a large amount of state with little locality [24], [25]). The processor with SMT capabilities supports 256 simultaneous threads in 8 processing engines. However, no evaluation of this processor is available.

Nahum et al. [26] have presented an experimental performance of packet-level parallelism on shared-memory multiprocessors. They found that limited packet-level parallelism exists within a single connection under TCP. However, packet-level parallelism is increased by implementing multiple connections.

The architectural implications of cache coherence protocols are investigated when the following parameters are changed: number of processors, cache size, and block size [21]. The investigation focused on evaluating the snooping protocol and the directory protocol under a variety of online transaction processing workloads (OLTP) and scientific/technical workloads. The investigation showed that i) when the cache size is increased, cache misses which would normally occur in a single processor are correspondingly reduced, while coherence misses which arise from inter-processor communication in multiprocessors are unchanged. In addition, ii) when the number of processors is increased, cache misses which would occur in a single processor are unchanged. However, the number of true sharing misses increases since an increase in the number of processors leads to an overall increase in the memory access cycles per instruction. Finally, iii) when block size is increased, true sharing misses are decreased by more than a factor of 2 and the number of false sharing misses is nearly doubled.

Martin et al. [13], [27] measured and compared the performance of cache coherence protocols such as the snooping protocol, the directory protocol, and the token protocol with commercial workloads. They found that the token protocol is 25-65% faster than the snooping protocol and 6-18% faster than the directory protocol.

Brooks et al. [28] implemented application-specific cache-coherence protocols in configurable hardware. The protocol they implemented showed an improvement in performance by a factor of 11 for a 32-node system. Kumar et al. [29] investigated the impact of cache coherence protocols in the network traffic processing on a real platform with processors based on an Intel® Core™ micro-architecture. Direct Cache Access which delivers network traffic into processor caches directly improved processing efficiency by 15.6% to 43.4% in receive-side. Ros et al. [30] presented DiCo-CMP which is cache coherence protocol for many-core Chip Multiprocessors. The DiCo-
CMP reduced the miss latency and network traffic compared to directory protocol and token protocol, respectively.

Qi et al. [31] have introduced three principles to design effective network algorithms on multi-core network processors. They re-designed two typical network algorithms such as a packet classification and a pattern matching with three principles. The algorithms which are redesigned with these three principles achieve better performance than many existing algorithms on multi-core network processors.

Cong et al. [32] have introduced three technologies to achieve high performance deep packet inspection on multi-core platform. These technologies are Connection-based Parallelism, Affinity-based Scheduling and Lock-free Data Structure [33] at the hardware-level, OS-level, and application-level, respectively.

2.4 Architecture of Actual Network Processors

The recent Intel IXP2800 integrates a parallel processing design on a single chip for the purpose of processing complex algorithms, deep packet inspection, traffic management, and forwarding at line rates [6]. The Intel IXP2800 consists of the Intel XScale core with sixteen 32-bit independent multithreaded microengines. The XScale core is an embedded 32-bit RISC core for higher layer processing such as high performance processing of complex algorithms, route table maintenance and system-level management functions. The microengine is a flexible multithreaded RISC processor that can be programmed to deliver intelligent transmit and receive processing, with robust software development environment for rapid product development. The microengines also have special instructions for packet processing, such as finding the first bit set, barrel shift, and extracting byte/word and providing the processing power to perform tasks that traditionally required high-speed ASICs. The IXP2800 delivers processing capability at OC-192/10 Gbps line rates.

3 Evaluation and Simulation Environment

We now present the simulator and benchmark programs which we used for our study.

3.1 Description of the Simulator

The Simics full-system functional execution-driven simulator [34] allows the modeling, early development, simulation, and quick test of multi-core processors. Simics provides virtual platforms to emulate any operating system to run with commercial workloads. However, Simics only shows the validity of program execution through functional simulation. Since Simics does not fully emulate the cycle delay caused by a functional operation, we need an additional simulation module to measure the machine cycle delay caused by cache coherence operations. Ruby, of GEMS [14], can be used as such a cache simulator: it can provide a count of the number of cycles spent due to cache operations as well as the number of cache misses.

The processor model used in our evaluation is the Ultra-SPARC III [35] which consists of a 4-way, 14-state non-stalling pipeline and a 16 K-entry branch prediction table. The simulated system runs an unmodified Solaris operating system version 9. One, two, four, and eight processor-cores can be simulated. To test the performance of the coherence protocols, two cache coherence protocols, MOESI-directory and MOESI-token, are used with network workloads. Figure 1 shows the various options for CMP implementation [10]. Figure 1 (a) shows a conventional microprocessor, (b) a simple chip multiprocessor, (c) a shared-cache chip multiprocessor, and (d) a multithreaded, shared-cache chip multiprocessor. In this paper, we used the CMP implementation option (c) which has a private L1 instruction cache, a private data cache and a shared L2 unified cache. The L2 cache is organized as a non-uniform cache architecture since non-uniform cache architectures provide the bandwidth demanded by the larger number of processors without incurring long access latency. The cache configuration is extremely sensitive to network workloads and impacts the overall performance. The
optimal instruction/data cache sizes are 16 KB and 32 KB for header-processing and payload-processing workloads [36], respectively. In this paper, 16 KB are used for instruction/data cache sizes since the network workloads used in the paper are header-processing workloads. The simulated CMP supports up to 8 cores and has 16 MB as shared L2 cache, 2 MB per each core. The configuration of the simulated CMP is shown in Table 2.

3.2 The NetBench Benchmark Suite

NetBench [37] is a set of nine benchmarks used for single thread execution, generally not for multithreaded parallel applications with a shared memory. It is commonly used for the evaluation of network processors. Multi-core architectures allow the exploitation of thread-level parallelism to fully utilize the capability, we had to modify NetBench so that it could exploit thread-level parallelism such as in SPLASH-2 [38], in which child processes share the same virtual address space as their parent process. To decompose a single thread into multiple threads, we exploit packet-level parallelism as shown in Figure 2. We convert single thread to multiple threads in which each thread processes its own packets and is independent of other threads. Figure 3 shows the implementation of the multiple threads which are proposed in Figure 2.

Among the nine programs of the NetBench benchmark suite, TL (Table Lookup), ROUTE, DRR (Deficit Round Robin), and NAT (Net-
network Address Translation) frequently refer to the routing table. When these four benchmark programs are modified to have multiple threads, they require synchronization mechanisms incurring additional spinlocks to share the routing table among multiple threads. Since this paper concentrates on the architectural implications of cache coherence protocols and excludes the effects of synchronization mechanisms in terms of network workloads, we do not use these four programs to investigate the architectural implications of cache coherence protocols with network workloads on CMP. The DH benchmark, a Diffie-Hellman public-key encryption-decryption mechanism, does not require a packet trace. We thus used CRC (Cyclic Redundancy Check), MD5 (Message Digest), and URL (Uniform Resource Locator) for our investigation of the architectural implications of cache coherence protocols. For the parallelization of each benchmark, the POSIX pthread library has been used and all three benchmarks have successfully been compiled and ported on the Simics simulator. These three benchmarks are compiled with gcc -O3 in SunOS 5.8. To “warm up” the cache, we ran these programs with 160 packets. Then we processed 5,000 packets. The simulation results are gathered between two MAGIC BREAKPOINT instructions [39] as shown in Figure 3.

The original NetBench uses the traces from Columbia University available in the public domain [40]. However, destination and source IP addresses of this trace are anonymized for privacy protection. Hence, for our purposes, we used other real packet traces [41].

4 RESULTS AND PERFORMANCE EVALUATION

We now present the evaluation results of two cache coherency protocols, the directory proto-
col and the token protocol, subjected to various network workloads. Those two protocols are major coherence protocols introduced in multicore systems. Snooping protocols are in many instances no longer considered effective coherence protocols due to their excessive overhead.

\[
\text{CPU time} = \frac{\text{Seconds}}{\text{Program Instructions}} = \left(\frac{\text{Program Instructions}}{\text{Program Clock cycles}}\right) \times \left(\frac{\text{Instruction Seconds}}{\text{Clock cycle}}\right) \tag{1}
\]

Equation (1) shows the CPU time needed to execute a program [21]. If two systems have the same clock frequency and run the same set of instructions, then the first term and the third term of Equation (1) are fixed and the CPU time depends only on the Clock cycles Per Instruction (CPI) parameter. Thus, to compare the performance of single processors which run single threaded and user-level programs, CPI (or IPC, the inverse of CPI) is commonly used. However, CPI is inaccurate for multithreaded workloads running on multiprocessors because it would be an incorrect implicit assumption that the number of instructions to be executed is constant during the execution of the programs. Instead, multithreaded workloads running on multiprocessors can have different instruction paths which are caused by spinlocks and other synchronization mechanisms. The different instruction paths change the number of instructions to perform the same amount of work [42]. For this reason, we use as a measure the total number of cycles required to complete the programs.

4.1 Performance comparison of protocols

As mentioned earlier in this paper, the token protocol shows better performance than the other two protocols (snooping protocol and directory protocol) with commercial workloads. Figure 4 shows the performance comparison between the directory protocol and the token protocol with three benchmark programs obtained from the NetBench benchmark suite. Regarding each benchmark program, the diagrams on the left show the total execution cycles measured in terms of RUBY cycles and the diagrams on the right show the total number of instructions executed.

With each benchmark, the token protocol exhibits better performance than the directory protocol. With the 8-core configuration, when compared to the directory protocol, the token protocol improves performance by a factor of 3.9 on average. With the 4-core configuration, the token protocol enhances the performance by a factor of 1.22. These results show that the token protocol yields an even better performance when more cores are included. This is a promising result, considering that the number of cores per die is likely to grow in the future.

As the number of processor cores increases, the additional number of cycles required to complete any application also increases. This is due to the fact that the number of instructions needed to deal with multiple threads in an operating system increases with the number of threads. This is not an auspicious observation since multi-core configurations do not necessarily yield higher performance. To explain this phenomenon, we will examine in the next subsection the instruction overhead due to multithreading.

4.2 Instruction overhead due to multithreading

In this part, the instruction overhead introduced by the code which enables multithreading has been measured. For example, there is some OS-related code to maintain memory consistency in order to guarantee the correct execution of the parallelized program. In fact, our parallelized model is programmed with the basic pthread functions. To measure the instruction overhead due to multithreading, the numbers of instructions used to complete the benchmark programs are compared under two scenarios of execution: normal program without any multithreading mechanism and multithreaded program with 1-thread. With the first configuration, only the sequential execution of
the program is measured. On the other hand, the threaded code with pthread operation is tested in the second case. This means that we can measure the thread-related instruction overhead by evaluating the results from the “normal” program and the multithreaded program with 1-thread.

Figure 5 shows the performance results with the two scenarios described above. When a multithreading mechanism is used (even with only one thread), the number of instructions for the programs execution under the directory protocol and the token protocol is increased by 9.7% and 10.3%, respectively. There is no noticeable difference between the two coherence mechanisms. The portion of code related to multithreading is believed to cause some overhead, particularly when more threads are executed in a multi-core environment.

4.3 Investigation of L2 cache misses

Under the multi-core execution model, misses from the shared L2 cache enormously degrade the overall performance. More specifically, they cause long memory access latencies and accordingly cause multiple cores to stall and wait until the misses are resolved. The overall performance of a shared cache is influenced by the number of cache misses that occur in a single processor and the number of coherence misses which arise from the inter-processor communi-
cation in multiprocessors. In this portion of our work, we have measured the number of cache misses in the shared L2 cache and the effect of those misses.

Figure 6 shows the number of cache misses in the shared L2 cache. As the number of processor cores increases, the number of L2 cache misses also increases. The reason for the increase in the number of L2 cache misses is the memory contention since the L2 cache is shared among all processors. More to the point, when the number of cores increases, the number of cache misses increase at a sharp rate. This is extremely detrimental to the overall performance as it also causes an increase in the number of instructions related to OS.

This causes excessive intervention of OS-related routines which severely degrades the performance of multithreaded execution. The token protocol shows comparatively fewer L2 cache misses than the directory protocol; this is one of the major reasons for the better performance of the token protocol. However, L2 cache misses in 8 processors of URL are fewer than those in 4 processors of URL. The reason is that 8 processors have fewer L1 data and instruction cache misses than 4 processors.

Let us examine in more detail the cache misses in order to identify the major cause behind them. Table 3 shows the ratio of the number of cache misses introduced by the execution in parallel over the number of cache misses in a sequential program. As shown in Table 3, the number of cache misses in the execution of user programs is negligible compared to the increased number of cache misses in the execution of OS-related routines. However, Branch-intensive network workloads [43], such as URL, have more cache misses in user mode than in OS mode. This explains the poor cache performance results in the multi-core environments.

We have observed a mix of causes for L2 cache misses. The results are shown in Figure 7. The light grey colored bars show the ratio of cache misses caused by the intervention of the operating system. The other bars show the ratio of cache misses caused by the user programs. These aggregated results show that most shared L2 cache misses are coherence misses of OS mode in the multi-core processor.

<table>
<thead>
<tr>
<th>Protocols</th>
<th>Category</th>
<th>CRC</th>
<th>MD5</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory</td>
<td>User mode</td>
<td>1.91</td>
<td>1.66</td>
<td>7.17</td>
</tr>
<tr>
<td></td>
<td>OS mode</td>
<td>4.31</td>
<td>3.60</td>
<td>2.09</td>
</tr>
<tr>
<td>Token</td>
<td>User mode</td>
<td>1.92</td>
<td>2.41</td>
<td>3.79</td>
</tr>
<tr>
<td></td>
<td>OS mode</td>
<td>3.82</td>
<td>4.01</td>
<td>1.84</td>
</tr>
</tbody>
</table>

TABLE 3: Increase of cache misses compared to the sequential program

5 Conclusions

In our work, we have found that token protocols provide much better performance than directory protocols. Most shared L2 cache misses are coherence misses in multi-core processors. The low performance observed in multi-core environments has two main causes; first, there exists a multithreading-related code which causes a significant amount of overhead. The second reason can be found in the frequent shared L2 cache misses in the operating system code which are actually unwittingly but inevitably inserted in order to maintain coherence. Thus, we need to reduce the instruction overhead due to multithreading and L2 shared cache misses for performance enhancement of future network processors which are based on multi-core processor.

As a result of our investigation, we claim that the multithreaded versions of network workloads suffer much from the existence of cache coherence mechanisms causing a low exploitation of instruction-level parallelism. Note that this is also the case for packet-level parallelism. Our target for future research will thus be to reduce any operating system-related overhead of multithreaded network workloads. We will also seek to improve the cache coherence protocol and tune it to provide better performance for network processors [28], [29]. As part of our future research, Lock-free Data Structure [33] will be exploited for the network workloads which require synchronization mechanism such as TI, ROUTE, DRR, and NAT. In addition, the architectural implications of cache coherence protocols need to be investigated with varying cache size and block size. We will also examine the architectural implications of simple multi-core multiprocessors.
with network workloads which exploit multiprogramming instead of multithreading.

**ACKNOWLEDGMENTS**

This manuscript is an extended version of a paper which appeared in the Proceedings of the IFIP International Conference on Network and Parallel Computing 2007. This work is partly supported by the National Science Foundation under grant number CCF-1065448. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

**REFERENCES**


References


**Kyueun Yi**

Kyueun Yi received the B.E. degree in Electronics and M.S. degree in Computer Science from Kyungpook National University, Korea in 1991 and 1993, respectively. He received the M.S. degree in Electrical Engineering from University of Southern California in 2002. He received Ph.D. in Electrical and Computer Engineering from University of California, Irvine in 2007. He has been a Principal Research Engineer at Mobile Handset R&D Center in LG Electronics Inc. since 2007. He has done image processing system design at POSTECH Information Research Labs, Korea (1993-1997). His research interests include multi-core and multithreaded architectures as well as application specific instruction processors. He is a member of the IEEE Computer Society.

**Won W. Ro**

Won W. Ro received the BS degree in electrical engineering from Yonsei University, Seoul, Korea, in 1996, and the M.S. and Ph.D. degrees in electrical engineering from the University of Southern California in 1999 and 2004, respectively. He worked as a research scientist in the Electrical Engineering and Computer Science Department, University of California, Irvine. He currently works as an assistant professor in the School of Electrical and Electronic Engineering at Yonsei University. Prior to joining Yonsei University, he had worked as an assistant professor in the Department of Electrical and Computer Engineering at California State University, Northridge. His industry experience also includes a college internship at Apple Computer, Inc., and a contract software engineer in ARM, Inc. His current research interests are high-performance microprocessor design, compiler optimization, and parallel processing. He is a member of the IEEE Computer Society.
Jean-Luc Gaudiot

Jean-Luc Gaudiot received the Diplôme d’Ingénieur from the École 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 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 joining 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 multi-processors, 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 Iguacu, 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.