On-Chip Multiprocessor with Simultaneous Multithreading

Kyoung Park, Sung-Hoon Choi, Yongwha Chung, Woo-Jong Hahn, and Suk-Han Yoon

As more transistors are integrated onto bigger die, an on-chip multiprocessor will become a promising alternative to the superscalar microprocessor that dominates today's microprocessor marketplace. This paper describes key parts of a new on-chip multiprocessor, called Raptor, which is composed of four 2-way superscalar processor cores and one graphic co-processor. To obtain performance characteristics of Raptor, a program-driven simulator and its programming environment were developed. The simulation results showed that Raptor can exploit thread level parallelism effectively and offer a promising architecture for future on-chip multiprocessor designs.

Manuscript received August 11, 1999 ; revised October 31, 2000.

Kyoung Park is with the Computer System Department, ETRI, Korea. (phone: +82 42 860 3809, e-mail: kyoung@etri.re.kr)

Sung-Hoon Choi is with the Venture Business Technology Department, ETRI, Korea. (phone: +82 2 525 8050, e-mail: shungchoi@etri.re.kr)

Yongwha Chung is with the Information Security Application Department, ETRI, Korea. (phone: +82 42 860 5381, e-mail: ywchung@etri.re.kr)

Woo-Jong Hahn is with the API Networks Inc., USA. (phone: 1 978 318 1149, e-mail: woo-jong.hahn@api-networks.com)

Suk-Han Yoon is with the SecureNetCom Inc., Korea. (phone: +82 42 866 6622, e-mail: shyoon@icu.ac.kr)

I. INTRODUCTION

The performance of microprocessors has been improving at a phenomenal rate for the last decade. This performance growth has been driven by the innovation in compiler, the improvements in architecture, and the tremendous improvement in VLSI technology. Currently, most of commercial microprocessors such as Intel Pentium, Compaq Alpha21264, IBM PowerPC620, Sun UltraSparc, HP PA8000 and MIPS R10000 use superscalar design technique [1]–[3]. Such superscalar microprocessor executes multiple instructions in a single cycle by exploiting Instruction-Level Parallelism (ILP) [4]. The latest superscalar microprocessors can execute four or six instructions concurrently with many non-trivial techniques including dynamic branch prediction, out-of-order execution, and speculative execution method. However, significant speed-up may not be achieved by using these techniques because of the limitation of the instruction window size and the ILP in a typical program [4], [8]. Moreover, considerable design efforts are required to develop such high performance microprocessors. Therefore, developing a complex wide-issue superscalar microprocessor as a next generation microprocessor may not be an efficient approach to satisfy the required performance [5]–[7]. Instead, researchers have studied some alternatives to superscalar architecture [5]–[10]. The On-chip Multiprocessor [7]–[9] is the one of the alternatives considered as next generation microprocessors.

This paper reports on the design of a next generation microprocessor, called *Raptor*, which has an On-chip Multiprocessor architecture. *Raptor* is composed of four 2-way superscalar processor cores and one graphic co-processor. The key idea of *Raptor* is a multiprocessor sharing an off-chip second level cache in a singe chip to exploit Thread-Level Parallelism (TLP) [9]–[16], in addition to ILP.

To illustrate the possibility of *Raptor* as a next generation microprocessor, we design it with Verilog Hardware Description Language (HDL) and conduct a performance simulation with a dedicated architectural simulator, called *RapSim* (*Raptor* Simulator). *RapSim* is a program-driven, cycle-level simulator consisting of a Pre-Processing Unit as an instruction simulator and a Post-Processing Unit as a performance simulator for each processor model. Also, a programming environment, called Multithreaded Mini-OS (*MMOS*), is developed to support a Simultaneous MultiThreading (SMT) environment for *RapSim*. Benchmarks programs are chosen from widely used scientific applications such as FFT and Gaussian Elimination.

Our simulation focused on the performance characteristics of *Raptor* including *Instructions Per Cycle (IPC)*, *execution cycle time*, *speed-up* and *thread overhead* as the number of processor cores increased. The results showed that the On-chip Multiprocessor could be a strong candidate having scalable *IPC* and *speed-up*. However, the overall performance of the On-chip Multiprocessor depended on the amount of available parallelism and *thread overhead* in SMT program.

The organization of the paper is as follows. A trend in the microprocessor architecture is given in Section II. Section III describes *Raptor* architecture. In Section IV, the architectural simulator of *Raptor* and its programming environment are explained. Simulation results are shown in Section V, and concluding remarks are made in Section VI.

II. TREND IN MICROPROCESSOR ARCHITECTURE

The major trend in commercial microprocessor architecture is the use of complex architecture to exploit the ILP. There are two approaches that are used to exploit the ILP: superscalar and Very Long Instruction Word (VLIW). Both approaches attempt to issue multiple instructions to independent functional units at every clock cycle. Superscalar [1]–[3] uses hardware to dynamically find data-independent instructions in an instruction window and issue them to independent functional units. On the other hand, VLIW [4], [17] relies on a compiler to find the ILP and schedule the execution of independent instructions statically.

Superscalar is more appealing in commercial microprocessors because it can improve the performance of existing application binaries. However, superscalar is complex to design and difficult to implement. Looking for parallelism in a large instruction window requires a significant amount of hardware and usually does not improve the performance as much as one might expect. Due to this complexity, it is difficult not only to make the architecture correct but also to optimize the pipeline and circuits to achieve high clock frequency [4]–[5].

On the other hand, VLIW relies on the compiler to find bunches of independent instructions. Since VLIW does not require the hardware for dynamic scheduling, it can be much simpler to design and implement. However, it requires significant compiler supports such as a trace scheduling to find out ILP in an application program. VLIW is preferred over superscalar when the issue width is so large that dynamic scheduling hardware in superscalar is too complex and expensive to implement. However, even in VLIW, such a wide-issue machine has a centralized register file that must have many ports to supply operands to independent functional units. The access time of the register file and complexity of the buses connecting the functional units may limit the clock frequency. Another disadvantage of VLIW is that it does not execute binary programs for an existing Instruction Set Architecture (ISA). Although this compatibility problem may be solved by the use of software for emulating existing ISA, this problem is serious in commercial market places. In addition, VLIW forces a bunch of instructions to execute together. If one instruction in the bunch stalls, then other instructions in the bunch must stall, too. This limits VLIW's ability to deal with unpredictable events such as data accesses causing cache misses [4]–[5].

Therefore, ILP architecture have some limitations on their performance improvements [6]–[9], even though ILP architectures are dominant in current commercial microprocessor marketplace. Researchers have proposed two alternative architectures to overcome the limitations of ILP architectures: Multithreading processor [6] and On-chip Multiprocessor [7]–[9] to exploit another type of parallelism, TLP besides ILP. Figure 1 shows the examples of two alternative architectures.

The Multithreading processor, shown in Fig.1(a), augments the wide-issue superscalar with the hardware allowing it to execute instructions from multiple threads of control concurrently and dynamically selecting and executing instructions from many active threads simultaneously. This improves utilization of the processor's execution resources and provides latency tolerance in case of a thread stall due to cache miss or data dependency. When multiple threads are not available, however, the Multithreading processor simply looks like a conventional wide-issue superscalar [6]. Moreover, to keep the processor's execution units busy, the Multithreading processor features advanced branch prediction, register renaming, out-of-order execution, and non-blocking caches as superscalar processor does. The design complexity of the dispatch unit increases, since it exploits ILP and TLP simultaneously [8].

The On-chip Multiprocessor, shown in Fig. 2(b), uses relatively simple single-thread processor cores to exploit only moderate amount of ILP within a thread, while executing multiple threads in parallel across multiple cores. If an application can not be effectively decomposed into threads, the On-chip Multiprocessor will be underutilized [7]–[9].

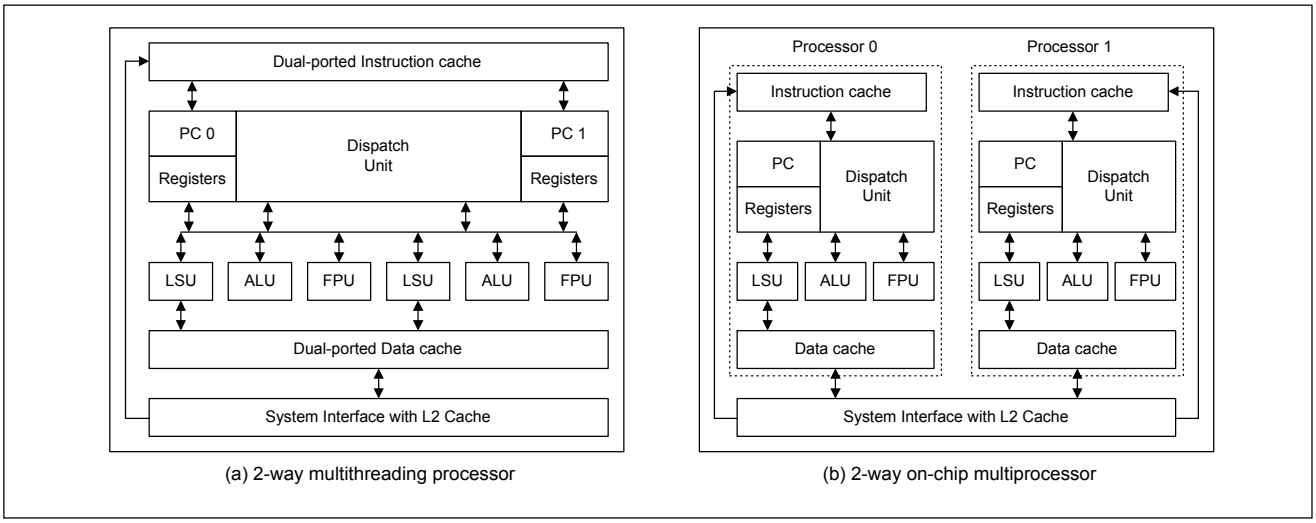

Fig. 1. The examples of two alternative architectures.

From a purely architectural point of view, Multithreading processor is superior. However, the inherent complexity of Multithreading processor results in hardware design problems that On-chip Multiprocessor solves by keeping the hardware simple. The important factor is the design complexity. As the complexity increases, more design efforts are needed to optimize critical timing path, increase clock frequency, and reduce interconnection delays. The effects of interconnection delays, which are becoming much slower than the transistor gate delays, will become more important in a billion-transistor CMOS implementation technology. The interconnection delay will force the architecture to be partitioned into small and localized blocks. For these reasons, the On-chip Multiprocessor is more promising because it is already partitioned into individual processor cores. The simple cores are amenable to speed optimization and can be designed relatively easily [8].

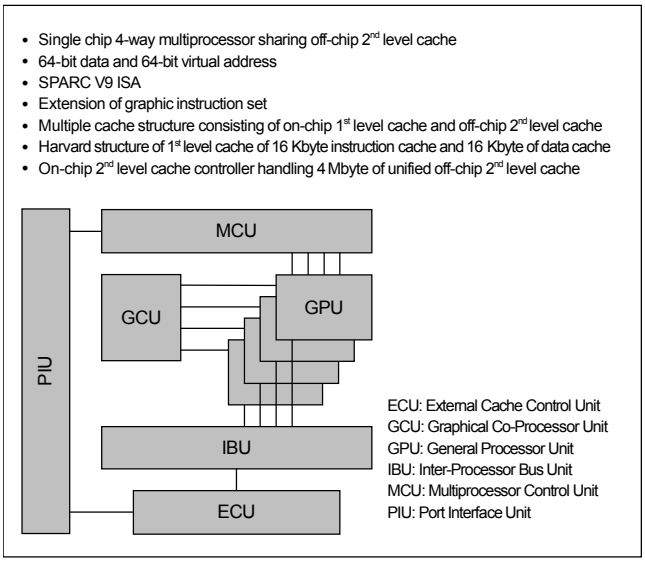

Fig. 2. Block diagram of Raptor microprocessor.

III. RAPTOR ARCHITECTURE

Raptor is an On-chip Multiprocessor microprocessor consisting of four independent processor cores, called General Processor Units (GPU), and one graphic co-processor, called Graphic Co-processor Unit (GCU). Inter-processor Bus Unit (IBU) is a shared bus connecting GPUs and External Cache control Unit (ECU). Multiprocessor Control Unit (MCU) distributes interrupts across GPUs and provides synchronization resources among GPUs. Port Interface Unit (PIU) is a multiprocessor-ready bus interface to communicate with the exterior of *Raptor*. Four GPUs execute all instructions except extended graphic instructions with their own register files and program counters, but share ECU through IBU. GCP is also shared by four GPUs and performs graphic instructions with Single Instruction stream Multiple Data Stream (SIMD) style pixel processing hardware. Figure 2 shows the block diagram of *Raptor* and main features of *Raptor.*

1. GPU

GPU is a simple 2-way superscalar RISC core having three functional units; Integer Execution Unit (IEU), Floating Point Unit (FPU) and Load Store Unit (LSU). It executes SPARC V9 [18] instruction set with branch folding and out of order execution capabilities. Fetch and PreDecoder (FPD) fetches instructions from Instruction Cache (I-Cache) and stored into Instruction Buffer (I-Buffer). In order to reduce the overhead caused by branch operation, branch folding technique [3] is

used during instruction prefetch stage. Two instructions in I-Buffer are decoded and issued into proper functional units in every cycle by Decoder. ReOrder Buffer (ROB) allocates entries for the issued instructions to support the out of order execution. Reservation Station (RS) of each functional unit resolves the dependency problems among instructions. After executing instructions, the functional units return their results to ROB through a Result Bus in order to update the ROB entries. ROB checks the status of each entry, updates register files, and deallocate the entries of the committed instructions.

The IEU has two Arithmetic Logic Units (ALU), a multi-cycle integer multiplier, and a multi-cycle integer divider to process integer data. The FPU executes floating point instructions using a floating point adder, a floating point multiplier and a floating point divider/SQuare RooT (SQRT). Most floating point instructions are fully pipelined, have a latency of three except for divide/SQRT instructions. The divide/SQRT instructions are not pipelined and take 12/22 cycles (single/double) to execute. The LSU is responsible for memory access using the virtual address of all loads and stores instructions.

The register file is organized as 8 windows, where each window has 32 entries and each register handles 64 bits of integer data. Floating point register handles single, double and quad precision floating point data. The floating point register file can store either 32 entries of single or double precision data, or 16 entries of quad precision data. The graphic register file supports the extended graphic instruction. 32 entries of graphic register file can store 32 bits of graphic data. The block diagram of a GPU is shown in Fig. 3.

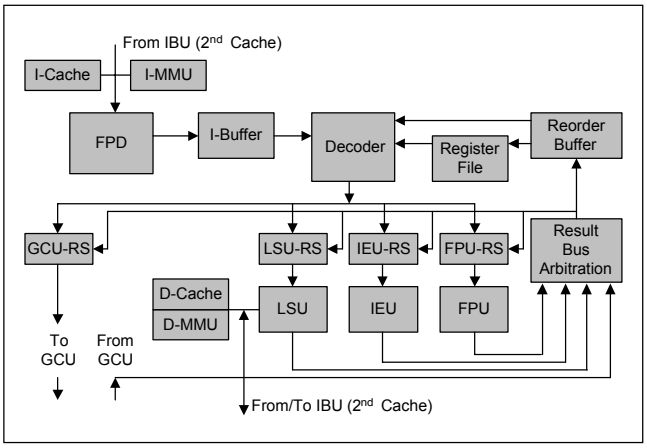

Fig. 3. Block diagram of GPU.

2. GCU

The instruction set supported by GCU is essential instructions widely used in multimedia and signal processing algorithms. The GCU architecture follows the design philosophy of a SIMD. The functional units of GCU can perform 8 bits, 16 bits, 32 bits of packed arithmetic operations, boolean algebras and bit-wise manipulations. Also, it can calculate the sum of absolute pixel distances so that MPEG algorithm can be handled more efficiently.

As we mentioned before, all GPUs share one GCU by issuing GCU instruction via corresponding buffers. The scheduler of GCU fetches the instructions and distributes them to proper functional units. Then, the results of GCU operations are returned to GPUs.

As shown in Fig. 4, GCU has the following four major functional units.

- `Graphic ALU(GALU)`: Performs packed arithmetic and boolean operations for graphics and multimedia data, synthesizes the results from GMUL and GSAD, and extracts max/min.
- `Graphic Multiplier(GMUL)`: Performs packed multiplication with four 16×16 multipliers and packing network.
- `Graphic Bit Manipulation Unit(GBMU)`: Performs bit-wise logic operations and shift operations within packed words, data copy and exchange operations among packed words, and packing and unpacking operations for type conversion.
- `Graphic Sum of Absolute Difference(GSAD)`: Performs SAD operations used for MPEG encoding with eight processing elements.

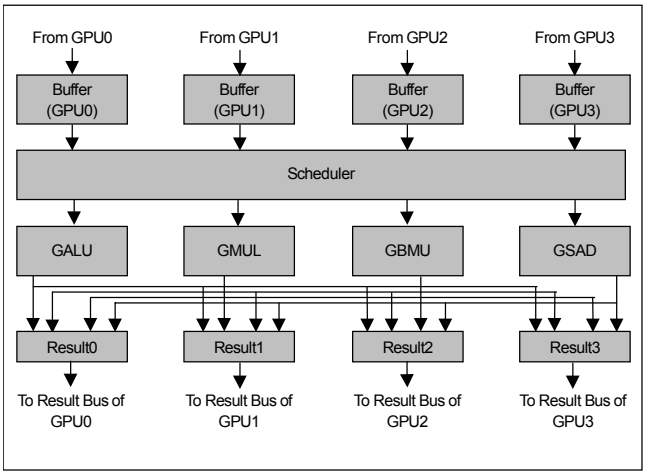

Fig. 4. Block diagram of GCU.

3. MCU

All GPUs should be the same in servicing external interrupts. Otherwise, software should identify which GPU on a chip is to run a particular service. MCU processes two classes of external interrupts. One is for a direct interrupt that should be serviced by

a specific GPU, and the other is for an arbitration interrupt that is serviced by any one of four GPUs. For arbitration interrupts, MCU gathers the external interrupts and distributes them equally among the four GPUs.

Furthermore, MCU provides GPUs with message passing resources for inter-GPU communications. When GPUs work together to perform a tightly coupled multithreaded task, an efficient inter-GPU communication mechanism is required. MCU has 32 entries of a message register file shared by all GPUs so that GPUs can use it as message buffers or synchronization resources among GPUs. In order to access MCU resources, each GPU uses special instructions. In addition, MCU initializes the whole chip on a reset and distributes a central clock.

4. IBU, ECU, and PIU

IBU is a shared bus connecting GPUs and ECU. Each GPU accesses ECU through IBU when it requires a memory access due to internal cache miss or a write-through.

ECU gets the request through IBU and returns a proper response according to modified MESI cache coherency protocol. It is responsible to keep the cache coherency among ECUs through PIU and maintain inclusion properties with the internal cache of GPU.

PIU acts as an interface between *Raptor* and outside world. It provides a multiprocessor-ready shared bus interface with a snooping cache coherency protocol. PIU also provides a SRAM module interface for accessing off-chip second level cache RAMs.

IV. SIMULATION ENVIRONMENT

To evaluate several tradeoffs in designing *Raptor* quantitatively, we developed a dedicated simulator, called *RapSim*. *RapSim,* shown in Fig. 5, is a program-driven microarchitecture simulator that models four GPUs and a memory hierarchy shared by four GPUs. Each GPU model consists of a Pre-Processing Unit as an instruction set simulator and a Post-Processing Unit as a performance simulator. Also, a programming environment, called Multithreaded Mini-OS (*MMOS*), was developed to support a SMT among GPUs. The overall configuration of *RapSim* and *MMOS* is shown in Fig. 6.

1. RapSim

Main features of *RapSim* are as follows:

- Execution of SPARC V9 instructions and graphic co-processor instructions
- Program-driven simulator having timing information

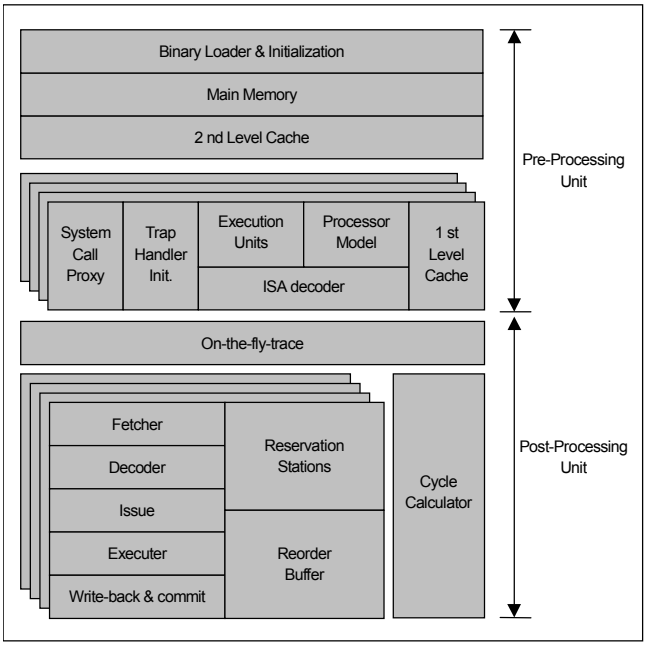

Fig. 5. Block diagram of *RapSim* simulator.

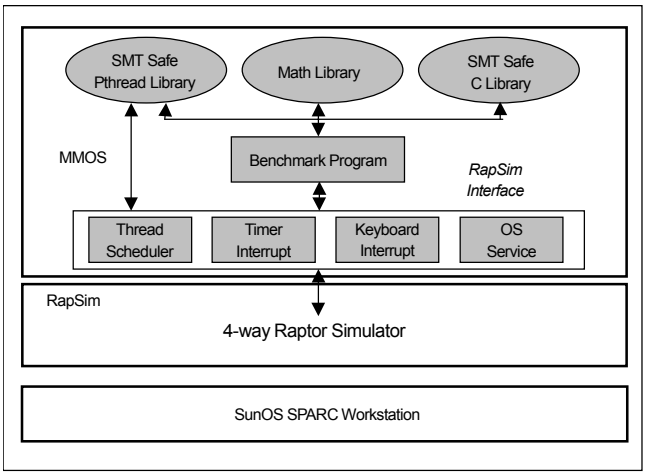

Fig. 6. Simulation environment.

- Multiprocessor model consisting of four processor cores and one graphic co-processor
- Support for out-of-order execution of a processor core
- Support SMT programming model
- Information gathering for performance evaluation

Pre-Processing Unit, shown in Fig. 7, is an instruction set simulator having a processor model for executing instructions, data structures for register files, proxy model for processing system calls, and first level cache model. Pre-Processing Unit fetches instructions and data from a shared memory hierarchy including second level cache model, executes the instructions, and generates an on-the-fly trace consumed by Post-Processing

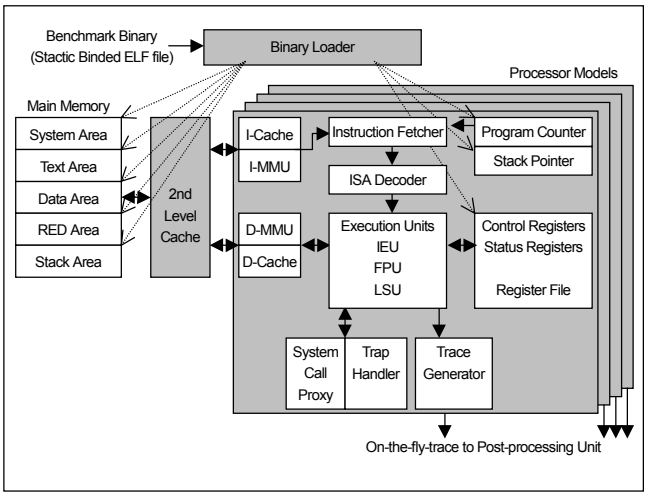

Fig. 7. Block diagram of pre-processing unit.

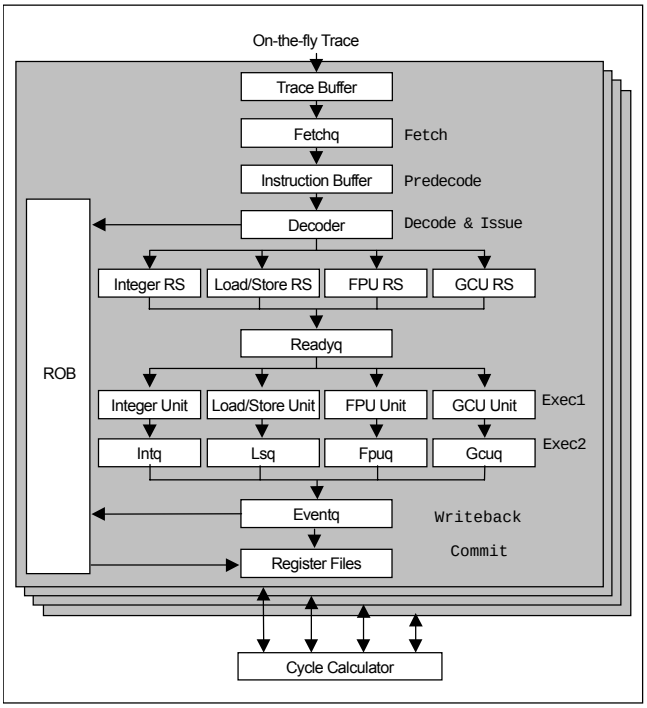

Fig. 8. Block diagram of post-processing unit.

Unit. Binary Loader of Pre-Processing Unit starts the simulation by loading a benchmark binary file, compiled and statically linked with *MMOS* library, into memory model. During the loading of the benchmark binary, a proper starting Program Counter (PC) and a Stack Pointer (SP) are set in the processor model, trap table and trap handlers are initialized in the memory model, and a stack is also constructed in the memory model. Then, Instruction Fetcher fetches instructions using PC and ISA Decoder issues the instructions to Execution Units after instruction decoding. Execution Units execute the instructions using the internal resources like IEU, FPU, LSU, register files, and first level cache model. As Pre-Processing Unit runs its instruction streams, Trace Generator generates an on-the-fly trace that is the sequence of executed instructions. Each entry of the on-the-fly trace contains enough information so that Post-Processing Unit can conduct the performance simulation using the on-the-fly trace as inputs.

As shown in Fig. 8, Post-Processing Unit is a RISC pipeline model conducting performance simulation by using the instruction traces generated from Pre-Processing Unit. It is modeled as a 2-way superscalar including RS and ROB to support the out-of-order executions. Two instructions in a Trace Buffer are fetched and pre-decoded in a cycle. The pre-decoded instructions in an Instruction Buffer are decoded and issued into proper RS, and ROB is updated simultaneously. Each execution unit runs safe instructions from a proper RS resolving dependency problems. The execution results of execution units reflect on ROB, and the terminated entries of ROB are updated into register files. The cycle calculator gathers the timing information from each pipeline model and calculates the performance information including execution cycle times and the number of total instructions.

The architectural parameters used in the simulation are listed in Table 1. In particular, our simulation focused on the effect of simultaneous multithreading parameter such as the number of

Table 1. Architectural configuration.

Feature	Default Value
Number of GPUs (P)	4 (1, 2, 4)
GPU issue width	2
1st level cache Size	16 Kbyte I-cache, 16 Kbyte D-cache / 32 bytes line
2nd level cache size	4 Mbyte / 32 bytes line
Write update policy	1st level cache to 2nd level cache: write through 2nd level cache to Main memory: write back
1st level cache access latency	1 cycle
2nd level cache access latency	4 cycle
Main memory access latency	10 cycle
Instruction Execution Latency [18]	Integer ALU = 1 cycle Integer Multiply = 4 ~ 34 cycle Integer Division = 36 (single), 68 (double) cycle Load / Store = 1 cycle Control Transfer = 1 cycle FP Addition/Subtraction = 1 cycle FP Multiply = 4 cycle FP Division / SQRT = 12 (single), 22 (double) cycle, not pipelined

GPUs in an On-chip Multiprocessor. Therefore, we fixed other architectural parameters as *Raptor*'s design specification used in HDL-design, except for the number of GPUs. The execution cycle and latency of each instruction are set to be equal to UltraSPARC-I microprocessor [19] executing SPARC-V9 ISA.

2. MMOS

In order to maximize the performance of the On-chip multiprocessor, TLP needs to be exploited. *MMOS*, modified Pthreads [16] for *RapSim,* help programmers to explicitly divide the code into threads to utilize four GPUs. *MMOS* has a SMT-safe Pthread library, SMT-safe C library, and *RapSim* interface.

The SMT-safe C library allows multiple thread to access the shared C library without synchronization problems, while the SMT-safe Pthread library provides a coarse-grain multithreading execution model with software support for creating, synchronizing, and scheduling threads. The *RapSim* interface assigns PCs, SPs, and register files in the processor models in order to allocate threads into the processor models in *RapSim*, which simulates 4-way On-chip Multiprocessor with quad threads simultaneously.

We chose some benchmarks from scientific applications [20], [21], and vision tasks from DARPA Image Understanding Benchmark [22]. This paper focuses on the results of the scientific applications only. The results of the vision tasks can be found in manuscript [22].

The codes of FFT, MP3D and LU have been obtained from SPLASH [20], [21] suite of benchmarks, whereas Matrix multiply (MMULT) and Gaussian Elimination (GAUSS) have written by the authors. MMULT and GAUSS were manually ported to be simultaneous multithreading program using SMT-safe Pthread library calls of *MMOS*. To port the SPLASH benchmarks to *RapSim*, the ANL macros were replaced with their SMT-safe Pthreads equivalents. Each benchmark is briefly described in the following:

- **MMULT** parses the matrix data into blocks and assigns them to threads. The data set for the threads is relatively disjoint, whereas the row by column operation produces considerable overlapping of data among threads. Moreover, there is no inter-thread communication or synchronization.
- **GAUSS** partitions $n \times n$ matrix into threads by using the row-wise block cyclic approach. Initially, one thread performs a division step with its pivot value, and then all other threads perform elimination step. These two steps are coordinated with barriers. GAUSS threads tend to have distinct data sets with a minimal data sharing besides the pivot value.
- **LU** factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense $n \times n$ array A is divided into $N \times N$ array of $B \times B$ blocks ($n = N \times B$). Every thread factors a $N/P \times N/P$ subarray. The data sets between threads are very localized.
- **FFT** implements a complex 1-D version of the $\sqrt{n}$ six step FFT algorithm. The data set consists of n complex data points and another n complex data points, called the roots of unity. Every thread is reponsible for transposing a contiguous submatrix $\sqrt{n}/P \times \sqrt{n}/P$ with every other thread and one submatrix by itself. The data sets between threads are very localized.
- **MP3D** is a simple simulator for rarefied gas flow over an object in a wind tunnel. The algorithm is primary occupied with a loop consisting of three phases. Each thread is given particles and proceeds to move them within a defined cell space. The thread continuously detects any possible collisions of its molecules with other molecules and updates the geometry of molecules each time step. MP3D contains data that is very localized and shares much of that data among threads. Also, each phase has to be completed by all the threads before continuing the next phase, requiring a larger amount of synchronization.

V. SIMULATION RESULTS AND ANALYSIS

In this paper, we focus on performance of the On-chip Multiprocessor, typically *IPC*, *execution cycles*, *speed-up*, and *thread overhead* are measured as our performance metrics.

Three sets of simulation runs were performed for each benchmark described in previous section. The first set was non-multithreaded benchmarks running on a single GPU to be served as a point of reference. The second set was 2-way SMT benchmarks running on two GPUs, and the final set was 4-way SMT benchmarks running on four GPUs. Three sets of simulation results were compared and analyzed to find out the architectural characteristics of *Raptor*.

1. Simultaneous Multithreading

In general, a workload (W) consists of a sequential part (W_{seq}) and a parallel part (W_{par}) as expressed in (1):

$$W = W_{seq} + W_{par}. \quad (1)$$

The parallel part can be partitioned into a number of parallel threads simultaneously running with multiple processors, while the sequential part cannot. *MMOS* converts the parallel part of a workload into multiple threads and assigns them into GPUs so that *Raptor* exploits TLP to utilize multiple processor cores. The SMT workload (W_{smt}) converted by *MMOS* can be express as (2):

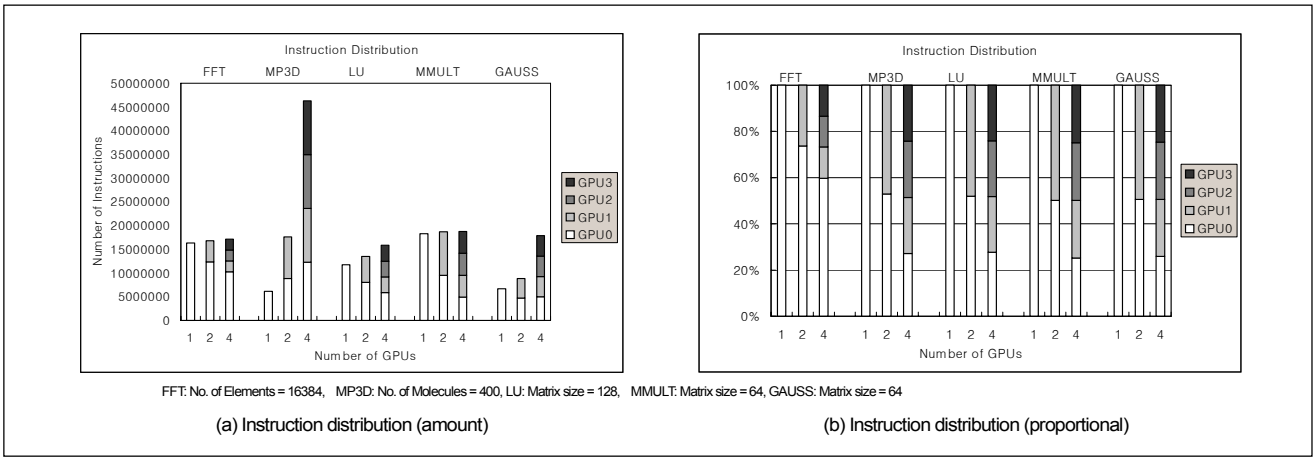

Fig. 9. Instructions distribution among GPUs.

$$W_{smt} = W_{seq} + (W_{par} / P) + W_{overhd} = W_{seq} + \sum_{n=1}^{p} W_{thread}[n] + W_{overhd}, \quad (2)$$

where P is the number of GPUs and W$_{thread}[n]$ is the n^{th} thread among P threads. The thread overhead(W_{overhd}) denotes the amount of additional works for the thread control and synchronizations.

Figure 9 shows the simultaneous multithreading effect on *Raptor*. Figure 9(a) and (b) show the instruction distributions of benchmarks in amount and proportional among GPUs respectively. As the number of GPUs increased, the instructions were distributed evenly over all of GPUs except FFT. FFT had about 45% of sequential part that computes 1D FFT, while others contained extremely low portion of the sequential part. Therefore, in all cases except for FFT, all GPUs were well utilized using simultaneous multithreading in most cases.

2. Execution Cycle vs. IPC

In general, the important performance metric of a microprocessor is *IPC*. The *IPC* is computed according to (3):

$$IPC = N_{inst} / T_{cycle}, \quad (3)$$

where N_{inst} and T_{cycle} refer to the number of executed instructions and the number of execution cycles, respectively.

The conventional processors exploit only ILP to increase the *IPC*. However, the processors face the limit on the *IPC*. *Raptor* uses another approach that exploits TLP using SMT to achieve the *IPC* scalability.

Figure 10 shows the scalability of *IPC*. As the number of GPUs increased, the *IPC* increased linearly except for FFT. In all cases except FFT, the *IPC* of *Raptor* having four GPUs ranges from 3.5 to 4.1, while the *IPC* of a single GPU ranges from 0.9 to 1.1. In FFT, the *IPC* of a GPU was about 0.9 and the *IPC* of 4-way *Raptor* was about 1.5. The major portion of FFT utilizes only single GPU, while others are idle. The amount of sequential part that can not be multithreaded is so large in FFT case.

The number of *execution cycles* is another interesting metric representing the performance. The number of *execution cycles* was reduced as the number of GPUs increased except for MP3D. Because MP3D had a significant amount of parallel part, it can be scalable in the *IPC*. However, as shown in Fig. 9(a), the number of instructions increased as the number of GPUs increased due to *thread overhead*. The issue regarding the *thread overhead* will be discussed in later section.

3. Speed-up vs. Thread Overhead

The amount of absolute performance gain of the simultaneous multithreading over a sequential programming can be expressed as (4):

$$\begin{aligned} &Absolute\ performance\ gain = W - W_{smt} \\ &= W_{par}(W_{par} / P) - W_{overhd} = W_{par} * (P-1) - W_{overhd}. \end{aligned} \quad (4)$$

As shown in (4), the absolute performance gain depends on two major factors: available parallelism (W_{par}) included in a workload and the thread overhead (W_{overhd}). If workload does not exhibit sufficient amount of parallelism for simultaneous multithreading, the GPU utilization will not increase. Though workload has sufficient amount of parallelism, the *thread overhead* of thread management, inter-thread communications, and synchronizations may limit the overall performance.

The *speed-up* is a general metric that presents the performance enhancement quantitatively. Let T and T_{smt} denote the execu-

tions times of W and W_{smt}, respectively. Then, the *speed-up,* t h e

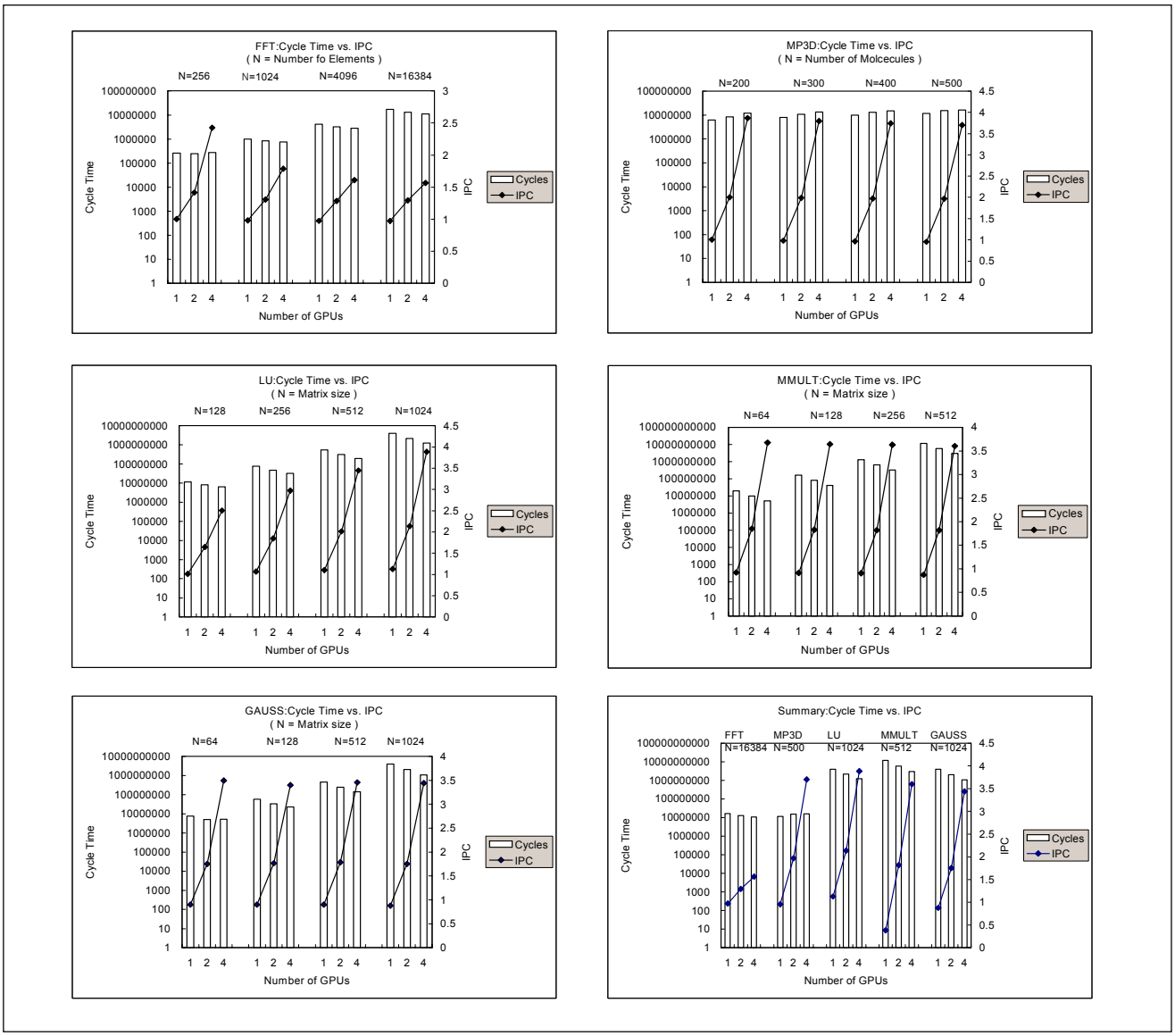

Fig. 10. Cycle time vs. IPC.

ratio of performance enhancement of the simultaneous multi-threading over a sequential programming, can be given as (5):

$$Speed\text{-}up = T / T_{smt}. \quad (5)$$

Figure 11 shows the *speed-up* as increasing the number of GPUs. As the number of GPUs increased, the *speed-up* increased except MP3D. MP3D had a sufficient amount of parallelism to be multithreaded, but heavy data sharing generated very frequent inter-thread communications and synchronizations. Such a heavy amount of thread overhead can limit the *speed-up*.

In FFT, the *speed-up* was small. As described before, FFT had a large amount of the sequential part and a little amount of available parallelism. Therefore, the *speed-up* was limited and the utilization of GPUs was not good.

VI. CONCLUDING REMARKS

The On-chip Multiprocessor is a promising candidate for a billion-transistor architecture, which can overcome the limit of ILP processors. By integrating simple processor cores, it can exploit TLP in addition to ILP.

In this paper, we have discussed the architecture of a new on-chip multiprocessor *Raptor* composed of four 2-way superscalar processor cores and one graphic co-processor. To evaluate performance of *Raptor*, we developed a program-driven, dedicated architecture simulator and its programming environment first. By conducting simulations, we analyzed the *IPC*, *execution cy-*

cle time, *speed-up*, and *thread overhead* as the performance metrics.

On the basis of our simulations, the *IPC* increased as the number

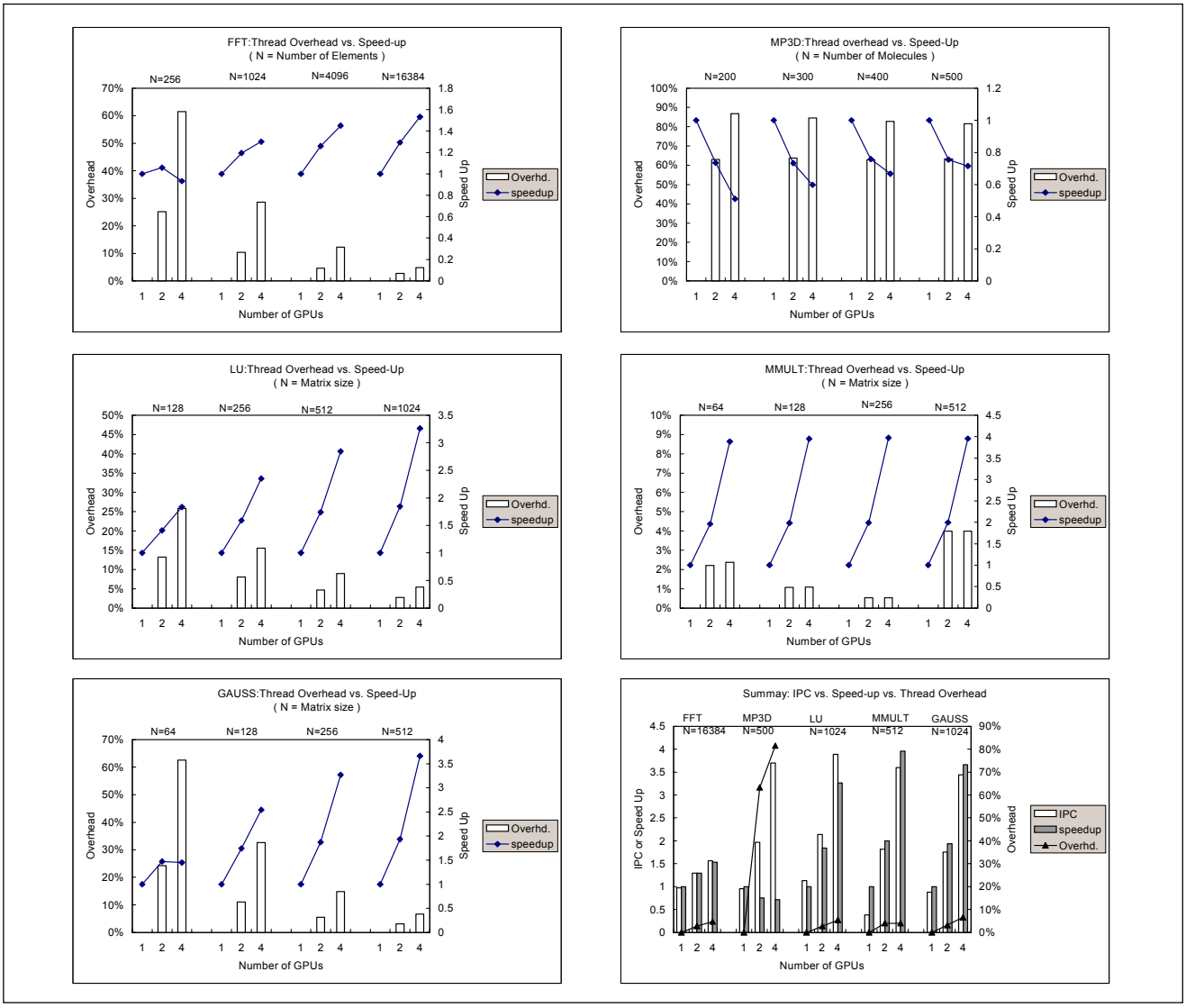

Fig. 11. Speed-up vs. thread overhead.

of GPUs increased. When four GPUs runs simultaneous multithreading workload the *IPC* ranges from 3.5 to 4.1, while the *IPC* of single GPU is about 0.9～1.1. In FFT, the *IPC* was not scalable since it contained 45% of a sequential part that can not be parallelized.

The *speed-up* obtained by increasing the number of GPUs over a single GPU heavily depended on the amount of available parallelism of workload and its thread overhead. Because MMULT, GAUSS and LU had a large amount of available parallelism and very small *thread overhead*, the *speed-up* was linearly increased. However, FFT had little improvement on the *speed-up* due to its small amount of available parallelism. MP3D was more severe. Even though MP3D had a large amount of available parallelism, the *speed-up* decreased as the number of GPUs increased. It is because MP3D had a large amount of the *thread overhead* caused by heavy inter-thread communications and synchronizations.

Based on our study, we plan to improve *RapSim* in a number of ways. First, we will provide multitasking environment where GPUs can run independents threads from individual multiple tasks. This improvement will increase the *IPC* by reducing the idle time of GPUs. Second, there is an increasing interest in applying multiprocessors to commercial applications such as *On-Line Transaction Processing* and *Decision Support System* [23], [24]. Therefore, it will be interesting to evaluate *Raptor* by using the commercial workload. Finally, *Raptor* can

be employed as a building block for large-scale multiprocessors [24]. We are currently expanding *RapSim* toward a Cache Coherent, Non-Uniform Memory Access (CC-NUMA) multiprocessor employing multiple *Raptor*s.

REFERENCES

[1] John L. Hennessy and David A. Patterson, *Computer Architecture A Quantitative Approach, second edition,* Morgan Kaufmann, Pub., 1996.

[2] Mike Johnson, *Superscalar Microprocessor Design*, Prentice Hall, 1991.

[3] M. Slater, "The Microprocessor Today," *IEEE Micro*, Vol. 16, No. 6, 1996, pp. 32–45.

[4] D.Wall, Limits of Instruction Level Parallelism, *WRL Research Report 93/6*, Digital Western Research Laboratory, Palo Alto, Calf., 1993.

[5] J. Wilson, "Challenges and Trends in Processor Design," *IEEE Computer*, Vol. 31, No. 1, 1998, pp. 39–50.

[6] S. Egger *et al*., "Simultaneous Multithreading: A Platform for Next-Generation Processor," *IEEE Micro*, Vol. 17, No. 5, 1997, pp. 12–19.

[7] B. Nayfe, L. Hammond, and K. Olukotun, "Evaluation of Design Alternatives for Multiprocessor Microprocessor," *Proc. of Int'l Symp. on Computer Architecture*, 1996, pp. 66–77.

[8] L. Hammond *et al*., "A Single-Chip Multiprocessor," *IEEE Computer*, Vol. 30, No. 9, 1997, pp. 79–85.

[9] S. Amarashinhe *et al*., "Multiprocessors From a Software Perspective," *IEEE Micro*, Vol. 16, No. 3, 1996, pp. 52–61.

[10] A. Agarwal *et al*., "Performance Tradeoff in Multithreading Processors," *IEEE Tr. on Parallel and Distributed Systems*, Vol. 3, No. 5, 1992, pp. 525–539.

[11] R. Alverson *et al*., "The Tera Computer System," *Proc. of Int'l Conf. on Supercomputing*, 1990, pp. 1–16.

[12] R. Saavedra, D. Culler, and T. Eicken, "Analysis of Multithreaded Architectures for Parallel Computing," *Proc. of Symp. On Parallel Algorithms and Architectures*, 1990, pp. 169–178.

[13] K. Kavi, B. Lee, and A. Hurson, "Multithreaded Systems," *Advances in Computers*, Vol. 46, 1998, pp. 287–328.

[14] A. Sohn, M. Sato, N. Yoo, and J. Gaudiot, "Data and Workload Distribution in a Multithreaded Architecture," *Journal of Parallel and Distributed Computing*, Vol. 40, 1997, pp. 256–264.

[15] B. Boothe and A. Ranade. "Improved multithreading techniques for hiding communication latency in multiprocessors," *Proc. of Int'l Symp. on Computer Architecture*, 1992, pp. 214–223.

[16] D. Botenhof, *Programming with POSIX Threads*, Addison Wesley, 1997.

[17] J. Fisher, "Very Long Instruction Word Architecture and the ELI-512," *Proc. of Int'l Symp. on Computer Architecture*, 1983, pp. 140–150.

[18] SPARC International, Inc., *The SPARC Architecture Manual version 9*, 1994.

[19] SPARC Technology Business, *UltraSPARC-I User's Manual Revision 1.0*, 1995.

[20] J. Singh, W. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared Memory," *Computer Architecture News*, Vol. 20, No. 1, 1992, pp. 5–44.

[21] C. Wang, and K. Hwang, "STAP Benchmark Evaluation of Three Massively Parallel Processors," *Proc. of Int'l Conf. on Parallel and Distributed Computing Systems*, 1997.

[22] Y. Chung *et al*., "Performance of On-Chip Multiprocessors for Vision Tasks," *IPDPS 2000 Workshop, LNCS 1800,* 2000, pp. 242–249.

[23] Woo-Jong Hahn, Suk-Han Yoon, Kwangwoo Lee, and Michel Dubois, "Evaluation of Cluster-Based System for OLTP Application," *ETRI Journal*, Vol. 20, No. 4, Dec. 1998, pp. 301–326.

[24] Yonghwa Chung, Jin-Won Park, and Suk-Han Yoon, "An Asynchronous Algorithm for Balancing Unpredictable Workload on Distributed-Memory Machine," *ETRI Journal*, Vol. 20, No. 4, Dec. 1998, pp. 346–360.

Kyoung Park received the B.E. and M.E. degrees in computer engineering from ChonBuk National University, Korea in 1991 and 1993, respectively. Since 1993 he is working on Electronics and Telecommunications Research Institute (ETRI) in Taejon, Korea and developed main memory board of a SMP system called TICOM-III, router switch of a high performance parallel system called SPAX, and On-chip multiprocessor called Raptor. He is working on developing a high performance multimedia server system as a Senior Member of Engineering Staff at ETRI. His main interests are computer architecture with a focus on multiprocessor, memory hierarchy and interconnection network in a large scale system, and next generation microprocessor architecture.

Sung-Hoon Choi received the M.S. degree in electronic engineering from Kyungpook National University, Taegu, Korea in 1988. He joined Electronics and Telecommunications Research Institute in 1988, where he is a Senior Member of Engineering Staff at ETRI. His main interests are computer architecture and microprocessor architecture. He is currently working to support ASIC design of small and medium companies in ASIC Design Center of ETRI.

Yonghwa Chung received his B.S. and M.S. degree from Hanyang University, Korea in 1984 and 1986, respectively. He received his Ph.D. degree from the University of Southern Californica, USA in 1997. He joined ETRI in 1986 and he is a Principal Member of Engineering Staff in Information Security Application Department. His research interests include computer architecture, parallel algorithm, distributed processing, and information security.

Woo-Jong Hahn received the B.S., M.S., and Ph.D. degrees from Korea University in 1981,

1984, and 1995 respectively. From 1984 to 2000, he was working on Electronics and Telecommunications Research Institute (ETRI) in Taejon, Korea as a Principal Member of Engineering Staff. From 1986 to 1988, he was working on 64-bit processor and workstation server development project, at AIT in Cupertino, CA., USA. From 1988 to 1991, he was working on developing SMP server, so called TICOM, and directed development and test of processor board. From 1991 to 1994, he coordinated hardware development and directed development of memory board of the next version of TICOM. From 1994, he was working on developing parallel processing architecture, so called SPAX, and also directing development of interconnection network. Currently, he is a Chief Architect in API Network, Inc. since October 2000. His research interests are computer architecture, memory hierarchy and interconnection network in a large scale system, microprocessor architecture.

Suk-Han Yoon received the B.E. degree in electronic engineering from Korea University, Seoul, Korea in 1977, the M.S. degree in computer science from KAIST, Taejon, Korea in 1986, and the Ph.D. degree in electronic engineering from Korea University, Seoul, Korea in 1995. He joined Electronics and Telecommunications Research Institute in 1977, where he was in charge of Mid-range SMP system Development Project, On-chip Multiprocessor Development Project, and High Performance Multimedia Server Development Project. He is currently the president of SecureNetCom Inc. since November 2000. His current research interests include high-performance computer architecture, parallel computing system and microprocessor architecture.