A Chip-Multiprocessor Architecture with Two Execution Modes

具雙重執行模式之單晶片多處理機架構

任朝欽, Chao-Chin Wu

國立彰化師範大學資訊工程學系

Department of Computer Science and Information Engineering

National Changhua University of Education, Changhua, Taiwan

E-mail: ccwu@cc.ncue.edu.tw

Abstract

Previous research show that, chip-multiprocessors have better speedup for floating-point-operation-intensive benchmark programs but worse for integer-operation-intensive application programs when compared with superscalar architectures. In this paper, we propose a novel microprocessor, combining the advantages of superscalar and chip-multiprocessor architectures, to provide the best performance regardless of workload types. Our architecture has two execution modes: one for multithreads and one for single thread. The new CPU can issue and execute sixteen instructions during each cycle regardless of the execution mode. In the first mode, the system behaves like a conventional chip-multiprocessor. On the other hand, we integrated separate four processing elements into a single logical superscalar processor in the second mode. When executing a program, the architecture keeps switching between two execution modes according to the feature of the subsequent codes to be run.

Keywords: Chip-Multiprocessor, Superscalar Processor, Multithreaded Architecture, Speculative Execution, Instruction-Level Parallelism.

摘要

單晶片多處理機雖然對大多數的測試程式都能提供相當不錯的執行效能，但是對於部分的整數型程式則無法提供如超純量架構一樣好的效能。本論文提出一種整合以上兩種架構之優點的新架構，不論是何種型態的測試程式均能有較好之效能。

在此新的架構中，每個處理單元均提供兩種執行模式：一種適合多引線平行處理，一種則適合單引線處理。在單引線模式下，我們整合四個處理單元為一個大的超純量架構。程式執行時會依其程式碼的特性，在兩種模式間切換。此架構不論在哪種執行模式下，最多均可於每個週期發行或執行 16 個指令。

關鍵詞：單晶片多處理機、超純量、多引線、冒險性執行、指令間平行度。

1. Introduction

Superscalar architectures, which are capable of issuing multiple instructions at the same cycle, have become the norm for today's high-performance microprocessors [1-4]. Although many researches still focus on boosting the speedup of superscalar architecture, various innovative microprocessor architectures have been proposed to exploit thread-level parallelism [6]. One interesting design is chip-multiprocessor (CMP) architecture that runs multiple threads on different processing elements at the same chip [5, 7-10].

Chip-multiprocessors have the following three advantages. (1) For the same chip size, chip-multiprocessors can provide higher issue rate compared with superscalar processors. According to previous research results, a chip-multiprocessor with eight 2-issue superscalar processing elements occupies the same die area as a 12-issue superscalar processor [11]. (2) Hardware design is simple because each processing element needs lower issue rate. Consequently, system clock can be faster and the duration of design validation phase can be shortened. (3) Communication in the processing elements' localized interconnect is faster because of decentralized network implementation.

There are two approaches to execute an application in parallel [12]. First, the programmer writes a parallel program with explicit instructions telling hardware when and how to execute the program in parallel. Second, the programmer

writes a sequential program and the compiler transforms it to a parallel one. The first approach is apparently difficult for most programmers while the second approach is simpler and it allows existing legacy codes to be able to be executed in the CMP architecture. Therefore, many chip-multiprocessors adopt the second approach for higher acceptability.

To run a sequential code in parallel in a CMP, loop iterations are spawned to multiple processing elements in a round-robin fashion. A processing element executes an iteration at a time. Because there are no explicit synchronization codes in the application code, speculative executions have been proposed to enforce dependences between iterations. The speculative execution approach assumes that data values are available when they are accessed by any iteration. During the course of execution, the hardware monitors whether any violation of data dependence occurs because a speculative iteration prematurely accesses a memory location. This will result in the squashing and then the restarting of the violating iteration along with its successors. Various schemes, including value predictions and data predictions, have been proposed to decrease the number of the data dependence violations [13-16].

Many research results have showed that the CMP architecture can provide superior system performance [5, 7-10]. However, it cannot compete against the superscalar processor with the same issue rate when running integer-operation-intensive applications [5]. This is because one single processing element issues a larger fraction of all the instructions in the integer application, with all other processing elements usually having no contribution on system performance during the course of execution.

A general-purpose microprocessor has to execute both integer applications and floating-point applications. It should provide excellent performances regardless of the workload type. Unfortunately, the CMP cannot meet the requirement. Therefore, we will propose an improved CMP to provide excellent performances for both types of workloads in this paper. The new processor supports two execution modes. One is the multithreaded mode that behaves like a conventional CMP. The other is the integrated superscalar mode that integrates all processing elements to behave like a conventional superscalar. When an application is executed, our CMP switches between two execution modes whenever reaching a loop entry point or a loop exit point. Because the improved CMP takes both the advantages of the conventional CMP and the conventional superscalar, it has the best system performance for both integer and floating-point applications.

The rest of the paper is organized as follows. Section 2 introduces the base CMP architecture and Section 3 describes how to implement the proposed microprocessor. Section 4 analyzes the performance gain and Section 5 concludes the paper.

2. CMP architecture supporting two execution modes

2.1 Conventional CMP microarchitecture

The chip-multiprocessor architecture we proposed is extended from the work of Krishnan and colleagues [5]. Their processor includes four processing elements and each of them is 4-issue superscalar architecture. Consequently, this processor is capable of issuing up to 16 instructions during every single cycle. They perform a compilation step on the sequential executable file without recompiling the program. As a result, legacy codes can be operated. In addition, they developed a binary annotator that identifies units of work for each thread and the register-level dependences between theses threads. The entry and exit points of each loop are marked. During the course of execution, when a loop entry point is reached, four threads are spawned to the four processing elements respectively to begin execution of successive iterations speculatively. Each processing element uses a special register to identify it is nonspeculative or speculative. At any time, only the processing element executing the first unfinished iteration is nonspeculative. When the first unfinished iteration is completed, the immediate successor processing element changes its status from speculative to nonspeculative. To enforce register dependences between threads and enable consumer threads to acquire correct values from the producer thread, a special hardware called synchronizing scoreboard has been designed. In addition, a modified cache-coherence protocol is proposed to enforce memory dependences. These two kinds of hardware also detect data dependence violations to squash and restart the execution of the corresponding thread. The fewer the dependence violations occur, the higher the system performance.

Although this processor outperforms superscalar processors for floating-point-operation-intensive benchmark programs, it cannot provide good system performance for all integer-operation-intensive programs.

2.2. Combining both advantages of CMPs and

superscalars

Since the CMP is suitable for floating-point applications and the superscalar is suitable for integer applications, our proposed CMP is designed to possess both the advantages of these two kinds of processors.

Our architecture provides two execution modes in a conventional CMP system: the speculative multithreading and the integrated superscalar modes. The proposed CMP acts like a conventional CMP in the speculative multithreading mode. On the other hand, it acts like a conventional superscalar in the integrated superscalar mode by integrating all processing elements (e.g., four processing elements) to be a conventional superscalar processor. In short, we will enhance the CMP to aggregate multiple processing elements (e.g. 4-issue each) into a wide superscalar (e.g., 16-issue aggregate).

The speculative multithreading mode is already available in the conventional CMP architecture, however, how to aggregate four separate processing elements into a wide-superscalar processor is a big challenge because each processing element in the CMP has its own program counter, fetch unit, decoding circuit, instruction window, functional units, reorder buffer, etc. There are two alternative ways to integrate a logical superscalar processor.

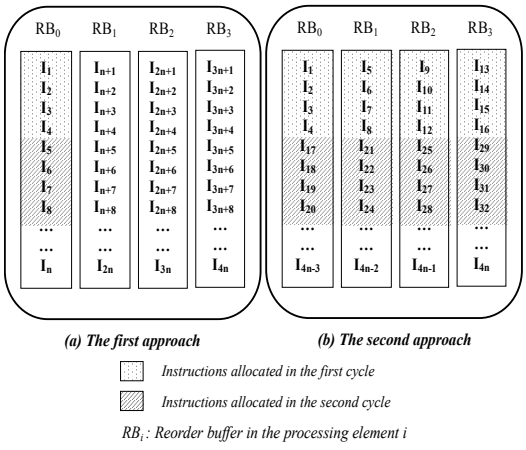


Figure 1. Two different approaches to integrate a logical superscalar processor.

The first approach is to let processing elements take turns to execute the codes sequentially as shown in Figure 1-(a). In other words, the first processing element will first execute the program until its instruction window or reorder buffer (RB) is full. Then, the second processing element will fetch the instructions immediately following the last instruction that the first processing element has fetched. To sum up, these four processing elements form a circular list. When an instruction window or reorder buffer is full, the following processing element continues to execute the subsequent codes.

The disadvantage of the first approach is that the maximum parallelism degree of the microprocessor is still four though these four processing elements may execute instructions concurrently. It is because the four processing elements are all 4-issue superscalar architecture, they take turns to fetch instructions to execute in different cycles. Consequently, four instructions are fetched at every single cycle in the whole system. The slight advantage of this approach is that the numbers of instruction window entries and functional units are logically increased.

The second approach is to let all the four processing elements fetch, issue and execute in every cycle as shown in Figure 1-(b). That is, four 4-issue superscalar processing elements will work together to make up a single logical superscalar processor with maximal parallelism degree of sixteen. To implement the approach, several requirements must be satisfied. First, we have to supply four instructions for each processing elements per cycle where these instructions are in the same dynamic trace. Second, a low-cost data communication approach is crucial for the processor. Third, data dependences between different processing elements must be enforced. Finally, precise interrupt across four processing elements have to be handled correctly. We will describe how to combine separate processing elements for each major stage of superscalar processing in the next section.

3. Implementing the integrated superscalar mode

3.1 Instruction fetch

We have to supply four instructions for each of the four processing elements at every cycle and the 16 instructions must be predicted to execute sequentially. The problem is that these instructions may be noncontiguous and they are across multiple basic blocks. Fortunately, the trace cache fetch mechanism provides a good solution [17]. It consists of a trace cache and a core fetch unit. The core fetch unit can fetch instructions up to the first predicted taken branch in each cycle by using the combination of an accurate multiple branch predictor, an interleaved branch target buffer, a return address stack, and a two-way interleaved instruction cache. The core fetch unit can only fetch contiguous sequences of instructions, i.e., it cannot fetch past a taken branch in the same cycle that the branch is fetched. The trace cache provides

this additional capability. It is a special instruction cache where each line stores a trace of the dynamic instruction stream. A trace is a sequence of multiple instructions and several basic blocks starting at any point in the dynamic instruction stream.

We apply the trace cache fetch mechanism in our architecture for the integrated superscalar execution mode. The fetch mechanism can fetch up to 16 instructions in a dynamic instruction stream per cycle. The fetched 16 instructions are divided into four ordered partitions where each partition is comprised of four contiguous instructions. The four instruction partitions will be dispatched to the four processing elements in order, respectively.

3.2 Data communications

Because sixteen contiguous instructions in a dynamic instruction stream are dispatched to four processing elements per cycle, data communications between different processing elements are frequent because of data dependences. In a conventional CMP, each processing element has its own local register file. To support the integrated superscalar mode, the four local register files must have the same data values at any time. In addition, we will use the bank-based register file proposed by NAN and colleagues [18] to replace the conventional register file in our chip-multiprocessor architecture.

The configuration of a bank-based register file is similar to that of a general register file, except that it is partitioned into three banks as shown in Figure 2. Every bank has the same number of register entries and the same register identifiers. That is, for each logical register identifier there is one corresponding register in each bank. To identify the bank of each register to be allocated for renaming, an in-order bank index table (IBIT) and a recently-updated bank index table (RBIT) are maintained. The IBIT is consistent with the sequential architectural state defined by program sequence, which is updated when an instruction completes in order. However, the superscalar processor allows out-of-order completion. To enforce in-order completion, a reorder buffer must be included but the field of register result is not required because register results are written directly to the bank-based register file. On the other hand, an entry of the RBIT is incremented by one whenever an instruction is issued. If an interrupt occurs, the RBIT contents will be replaced with the IBIT contents.

We use Figure 2 to illustrate functions of the bank-based register file. According to the IBIT, the sequential states of logical registers R_0, R_1 and R_{N-1} are stores in the Bank 0, respectively, while R_2 in the Bank 1. On the other hand, according to the RBIT, logical registers R_0, R_1, R_2 or R_{N-1} have been currently renamed to physical registers R_0 in the Bank 1, R_1 in the Bank 0, R_2 in the Bank 2 or R_{N-1} in the Bank 0. For instance, if a subsequent instruction requires to read logical register R_1, the content of the R_0 in the bank 1 will be supplied. In addition, when an instruction with destination register R_1 is issued, the entry corresponding R_1 in the RBIT is incremented by one. That is, the RBIT value for R_1 will be changed from 0 to 1. It means that the new result of R_1 will write to the next register bank for solving output dependence. On the other hand, the entry corresponding to R_1 in the IBIT will not incremented by one until the new result of R_1 is produced and the corresponding entry in the reorder buffer is at the head. In the following, we will describe how to use the bank-based register file and the simplified reorder buffer to enforce data dependences.

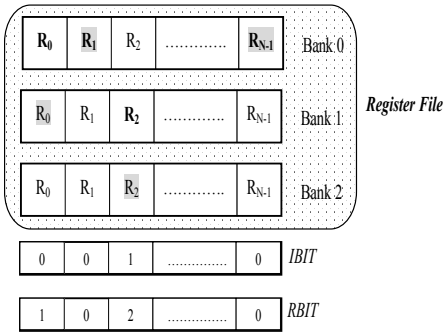


Figure 2. A bank-based register file

3.3 How to enforce data dependences

When an instruction is issued, the RBIT entry corresponding to the destination register is incremented by one. Moreover, a reorder buffer entry is allocated and the tag of that entry becomes the new name of the destination register until the result has produced. Meanwhile, the tag will be recorded in the register mapping table to indicate that the result is not available now. If the source operands of the subsequent instructions depend on the result and the result is not available when they are issued, the tag will be recorded in the instruction window. When the result comes out, it will directly write to the register file as well as the source operands with the same tag in the instruction window.

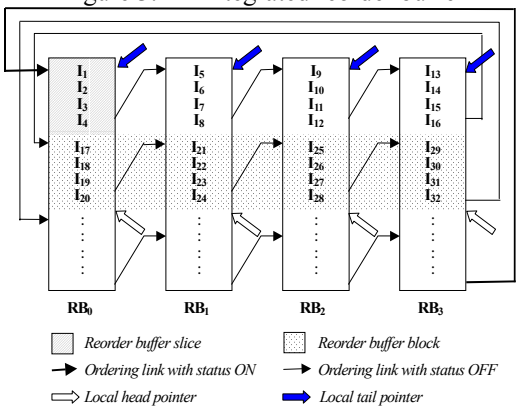
However, the above register renaming scheme cannot be directly applied to the integrated superscalar execution mode in our architecture. The first problem is how to maintain the program sequence by four independent reorder buffers. The second problem is how to perform register renaming. We explain how to handle the latter problem at first. Because we want to use the tag of reorder buffer to rename registers, the tag of each entry must be unique in the system. We add two bits in the most significant bits of the conventional reorder buffer tag. The additional bits are set to the identifier of the processing element the reorder buffer resides. Furthermore, a register mapping is maintained. In the following we will illustrate how to maintain program sequence.

3.4 Maintenance of program sequence

Four reorder buffers are linked and form a single logical reorder buffer as shown in Figure 3, where RB_i indicates the reorder buffer in the processing element $_i$. Because at most four instructions are allocated to each reorder buffer per cycle and we aim to simplify the hardware design, the number of the reorder buffer entries in a processing element is designed to be multiple of four. We here define a reorder buffer slice as every four contiguous reorder buffer entries, beginning from the first physical entry. For example, I_1, I_2, I_3 and I_4 form a reorder buffer slice. In addition, the four reorder buffer slices with the same physical order are together called a reorder buffer block. For instance, the sixteen instructions, from I_{17} to I_{32}, are in the same reorder buffer block. The reorder buffer slices belonging to the same reorder buffer block are ordered according to their own processing elements' identifiers. We define that the sixteen reorder buffer entries in the same reorder buffer block are ordered from the first entry in the first processing element to the fourth entry in the fourth processing element. In every cycle, sixteen fetched instructions are allocated sequentially to the reorder buffer entries in the same reorder buffer block. If less than sixteen instructions are fetched in a cycle, no-op instructions are automatically filled by hardware.

On the other hand, to enforce in-order completion, we add directed ordering links between every two consecutive reorder buffer slices as shown in Figure 3, resulting a circular list that keeps the feature of the conventional reorder buffer. Initially, all ordering links are OFF except that the one connecting to the first instruction in all four reorder buffers is ON. As Figure 3 shows, only the ordering link connecting to the instruction I_1 is ON, others are all OFF. An ordering link will be turned ON only when the head pointer points to the fourth entry in the corresponding reorder buffer slice and the completion field of the pointed entry has been set. The ordering link will be turned OFF again when the fourth entry in the reorder buffer slice is allocated for another new instruction. Ordering links together with the head pointer indicate whether the instruction in the first entry of a reorder buffer slice can commit or not. That is, if an instruction is not in the first entry of a reorder buffer slice, it cannot commit. Otherwise, the instruction can commit only when the corresponding ordering link is ON and the local head pointer points to it.

Figure 3. An integrated reorder buffer



3.5. Overview of our CMP architecture

Our CMP microprocessor as shown in Figure 4 is basically a conventional chip-multiprocessor where there are four 4-issue-superscalar processing elements. It has two execution modes. The multithreaded execution mode behaves like the CMP proposed by the Krishnan and colleagues, including all their architectural features: register-level communication with the synchronizing scoreboard, memory-level dependences handling with the memory disambiguation table, etc. However, bank-based register files are adopted instead. On the other hand, all the four processing elements are integrated to be a logical superscalar processor when in the integrated superscalar mode. The processor is initially in the integrated superscalar execution mode. When loop entry points are reached, the processor switches to the multithreaded mode. When exit points of loops are reached, the system switches back to the integrated superscalar mode again.

We have two kinds of instruction fetch and dispatch units and they are both responsible for fetching, decoding, and dispatching instructions. There is only one global instruction fetch and dispatch unit (GIFDU) in the processor. It adopts the trace cache fetch mechanism to fetch up to

16 instructions per cycle because it is for the integrated superscalar mode. On the other hand, there is one local instruction fetch and dispatch unit (LIFDU) for each processing element. LIFDUs are for the multithreaded mode and each of them can fetch up to four instructions per cycle. The GIFDU and LIFDUs are enabled exclusively according to the current execution mode.

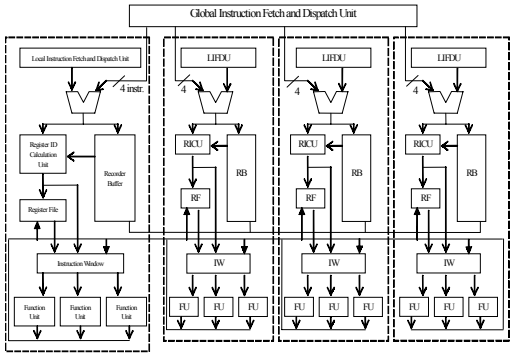


Figure 4. Our chip-multiprocessor with two execution modes

The GIFDU is also responsible for register renaming in the integrated superscalar mode. It contains a global head pointer, a global tail pointer, a register mapping table, an IBIT and an RBIT. The global head pointer and the global tail pointer together indicate if there are enough reorder buffer entries for further instruction fetch. Both of these two pointers maintain the status of the reorder buffer in the first processing element. It is because a reorder buffer block will be allocated in every cycle and the first fetched instructions must be allocated in the first processing element in our microprocessor. In addition, the global tail pointer is used to associate each fetched instruction with a unique reorder buffer tag. Finally, the IBIT is for precise interrupt handling and the RBIT is used to indicate which register bank will be accessed for each logical register when decoding an instruction. The register mapping table maintains the relationship between logical registers and physical registers.

In the integrated superscalar mode, the RBIT in each processing element is updated according to the contents of the RBIT from the GIFDU. Meanwhile, every consecutive four instructions are dispatched to different processing elements and local reorder buffer and instruction window entries are allocated. Instruction operands are read from the local register file if they are available. When the required operands are available, instructions are forward to functional units. After being produced by functional units, results together with its destination register tag are broadcast to all the instruction windows, reorder buffers, and register files. Pending instructions read the newly available operand values to release data dependences. Four bank-based register files are keep consistent during the integrated superscalar mode.

When switching to the multithreaded execution mode, the GIFDU is disabled and all LIFDUs are enabled. The LIFDU is responsible for instruction fetching, decoding, and dispatching. In addition, bank-based register files are operated independently and results from functional units do not broadcast to other processing element. All the specific CMP architectural features are enabled, including register-level data communication, detection of dependence violations. Consequently, four processing elements behave like a conventional CMP.

On the other hand, when switching back to the integrated superscalar mode, the GIFDU is enabled and all LIFDU are disabled. Four bank-based register files are made consistent with the contents of the register file in the non-speculative processing element. The supporting hardware for register renaming in the GIFDU is reset. Results are broadcasted across processing elements.

4. Performance Analysis

We construct the following evaluation model to analyze the performance of our microprocessor. Assume that the ratio of execution time in parallel mode is equal to p. In addition, assume that the instructions per cycle (IPC) in the sequential mode equals IPC_{seq} and the IPC in the parallel mode equals IPC_{par}. Consequently, the effective IPC of a system, IPC_{effect}, can be derived by the following equation:

$$\text{(I)} \quad IPC_{effect} = (1 - p) * IPC_{seq} + p * IPC_{par}$$

We will analyze our system performance based on the simulation data reported by Krishnan et al. since our microprocessor is extended from their architecture. The IPC is shown in Table 1, where the 4x4-issue indicates a chip-multiprocessor with four four-issue superscalar processing elements, the *n*-issue represents an *n*-issue superscalar microprocessor. Therefore, for a chip-multiprocessor system, the IPC_{effect} equals the value of the 4x4-issue and the IPC_{seq} equals the 4-issue value in Table 1. Next, we have to derive the p value indirectly from the figure of the *fraction of instructions issued by each of the four processors in the CMP* because it is not illustrated in [5]. In the following, we describe how to obtain the p value. We use F_i to denote the fraction of instruction issued by the

processing element *i*. According to the data reported in [5], the F_1 value is the largest and the values of F_2, F_3 and F_4 are almost the same. Moreover, to estimate the upper bound of performance improvements by our processor, we want the value of p to be larger as much as possible because our architecture can only improve the performance of the sequential part of executing application programs. Consequently, we assume that four processing elements are all busy in the parallel mode while only the processing element 1 is executing instructions when in the sequential mode. Let F_{three} be the average of F_2, F_3 and F_4. Then, we can have the value of p from the following equation.

$$p = \frac{F_{three}}{F_1} \qquad \text{(II)}$$

The p value derived from the above equation is listed in the first row of Table 2.

TABLE 1
The IPC values for SPEC95 applications under different processors.

	Compress	ijpeg	mpeg	eqntott
4 x 4-issue CMP	2.36	4.32	3.55	3.29
Four-issue superscalar	2.26	2.66	2.87	2.53
Sixteen-issue superscalar	3.07	3.26	3.63	3.17

TABLE 2
The p value and the IPC_{par} for the SPEC95 applications

	compress	ijpeg	mpeg	eqntott
***p* value**	0.06	0.5	0.57	0.65
IPC_{par}	3.96	5.98	4.07	3.70

Now that we have the data of the IPC_{effect} and the IPC_{seq} in Table 1 and the p value in Table 2, the IPC_{par} can be acquired according to the equation (I) and illustrate it in the second row of the Table 2.

Now we compare our processor under different overhead ratios with the chip-multiprocessor and the superscalar architectures as shown in Figure 5, where *Ours* stands for our microprocessor. We set the IPC_{seq} and the IPC_{par} to be the IPCs of the 16-issue superscalar architecture and the 4x4 CMP, respectively, as shown in Table 1 when calculating the IPC_{effect} of the *Ours* architecture. Our microprocessor provides the best performance for almost all the benchmark programs as shown in Figure 5. In particular, our processor can boost the performance of the benchmark program *compress* significantly: the IPC is from about 2.4 to about 3.1. In other words, our processor can outperform the conventional CMP up to 29.17%.

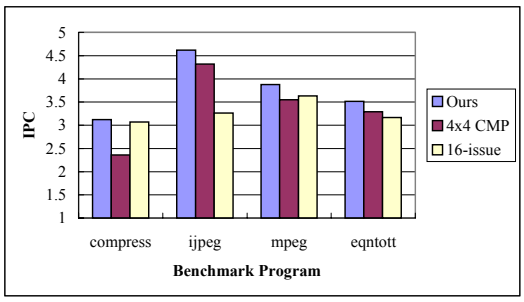


Figure 5. Performance comparison between our CMP, the conventional 4x4 CMP, and the 16-issue superscalar microprocessors.

5. Concluding Remarks

The chip-multiprocessor architecture provides a promising design methodology for the next-generation high-performance microprocessor by exploiting both instruction-level parallelism and thread-level parallelism. However, the conventional CMP cannot outperform the superscalar when executing integer-operation-intensive applications. In this paper we have introduced an improved CMP architecture providing superior performance for both integer and floating-point applications.

Our CMP supports the multithreaded and the integrated superscalar execution modes. Both execution modes provide the issue rate of sixteen, respectively. The multithreaded mode behaves like a conventional CMP while the integrated superscalar mode integrates all processing elements into a single logic superscalar. The proposed CMP is initially in the integrated superscalar mode. Whenever a loop entry point is reached, the processor switches to the multithreaded mode. The processor switches back to the integrated superscalar mode again whenever a loop exit point is reached.

The hardware supporting for the multithreaded mode is the same as that proposed by Krishnan and colleagues. On the other hand, the supporting for the integrated superscalar mode is not so apparent and it is the main concern in this paper. We have proposed that adding the GIFDU

to fetch, decode and dispatch instructions for the integrated superscalar mode. The GIFDU supplies sixteen instructions per cycle by using the trace cache fetch mechanism. In addition, we have adopted bank-based register files with modified reorder buffers for localized data communications, register renaming and precise interrupt handling. According to the performance analysis, our CMP outperforms the superscalar and the conventional CMP for all the benchmark programs regardless of the workload type. In particular, our processor can outperform the conventional CMP up to 29.17%.

Acknowledgements

This research was supported by the National Science Council of the Republic of China under the contract: NSC-90-2213-E-018-007.

References

[1] H. Sharangpani and H. Arora, "Itanium Processor Microarchitecture," IEEE Micro, vol. 20, no. 5, pp. 24-43, 2000.

[2] S. D. Naffziger, G. Colon-Bonet, T. Fischer, R. Riedlinger, T. J. Sullivan and T. Grutkowski, "The Implementation of the Itanium 2 Microprocessor," IEEE Journal of Solid-State Circuits, Vol. 37, no. 11, pp. 1148-1160, Nov. 2002.

[3] G. Lauterbach, D. Greenley, S. Ahmed, M. Boffey, J. Chamdani, Si-En Chang, D. Chen, Yu Fang, K. Holdbrook, M. Hsieh, B. Keish, R. Melanson, C. Narasimhaiah, J. Petolino, Tung Pham, Le Quach, Kit Tam, Duong Tong, Liuxi Yang and Kui Yau, "UltraSPARC-III: A 3rd Generation 64 B SPARC Microprocessor," 2000 IEEE International Solid-State Circuits Conference, pp. 410-411, 2000.

[4] A. Jain, W. Anderson, T. Benninghoff, D. Berucci, M. Braganza, J. Burnetie, T. Chang, J. Eble, R. Faber, O. Gowda, J. Grodstein, G. Hess, K J. owaleski, A. Kumar, B. Miller, R. Mueller, P. Paul, J. Pickholtz, S. Russell, M. Shen, T. Truex, A. Vardharajan, D. Xanthopoulos, T. Zou, "A 1.2 GHz Alpha microprocessor with 44.8 GB/s chip pin bandwidth," 2001 IEEE International Solid-State Circuits Conference, pp. 240-241, 2001.

[5] Venkata Krishnan and Josep Torrellas, "A Chip-Multiprocessor Architecture with Speculative Multithreading," IEEE Transactions on Computers, Vol. 48, No. 9, September, 1999, pp. 866-880.

[6] T. Ungerer, B. Robic and J. Silc., "Multithreaded Processors," The Computer Journal, Vol.45, No.3, pp. 320-348, 2002.

[7] L. Codrescu, D.S. Wills, and J. Meindl, "Architecture of the Atlas Chip-Multiprocessor: Dynamically Parallelizing Irregular Applications," IEEE Transactions on Computers, Vol. 50, No.1, pp. 67-82, January 2001.

[8] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun, "The Stanford Hydra CMP", IEEE Micro, Vol. 20, No. 2, pp. 71-84, March/April 2000.

[9] Jenn-Yuan Tsai, Jian Huang, Christoffer Amlo, David J. Lilja, and Pen-Chung Yew, "The Superthreaded Processor Architecture," IEEE Transactions on Computers, Vol. 48, No. 9, pp. 881-902, September, 1999.

[10] J. Smith and S. Vajapeyam, "Trace Processors: Moving to Fourth Generation Microarchitectures," Computer, vol. 30, no. 9, pp. 68-74, Sept. 1997.

[11] L. Hammond, B. Nayfeh and K. Olukotun, "A Single-Chip Multiprocessor," Computer, vol. 30, no. 9, pp. 79-85, Sept. 1997.

[12] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, Third Edition, Morgan Kaufmann Publishers, 2003.

[13] Sang-Jeong Lee, Pen-Chung Yew, "On table bandwidth and its update delay for value prediction on wide-issue ILP processors," IEEE Transactions on Computers, Vol.50 no.8, pp. 847 -852, Aug 2001.

[14] Burtscher, M., Zorn, B.G., "Hybrid load-value predictors," IEEE Transactions on Computers, Vol.51, no.7, pp. 759 -774, Jul 2001.

[15] L. Hammond, M. Willey, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," Proc. Eight Int'l Conf. Architecture Support for Programming Languages and Operating Systems (ASPLOS), Oct. 1998.

[16] K. Olukotun, L. Hammond and Mark Willey, "Improving the Performance of Speculatively Parallel Applications on the Hydra CMP," Proc. Int'l Conf. Supercomputing (ICS), 1999.

[17] E. Rotenberg, S. Bennett, J. E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," MICRO-29, pp.24-34, 1996.

[18] Nam, I.-C. Park, and C.-M. Kyung, "Fast Precise Interrupt Handling without Associative Searching in Multiple Out-Of-Order Issue Processors," IEICE Trans. Inf. & Syst., Vol. E82-D, No. 3 March 1999, pp. 645-6.