Dynamic Fast Issue (DFI) Mechanism for Dynamic Scheduled Processors

Abderazek BEN ABDALLAH†, Mudar SAREM††, Nonmembers, and Masahiro SOWA†, Regular Member

SUMMARY Superscalar processors can achieve increased performance by issuing instructions Out-of-Order (OoO) from the original instruction stream. Implementing an OoO instruction scheme requires a hardware mechanism to prevent incorrectly executed instructions from updating registers values. In addition, performance decreases if data dependencies, a branch or a trap among instructions appears. To this end, we propose a new mechanism named Dynamic Fast Issue (DFI) mechanism to issue instructions in an OoO fashion to multiple parallel functional units without considerable hardware complexity. The above system, which will be implemented in our Superscalar Functional Assignments Register Microprocessor (FARM) [1], solves data dependencies, supports precise interrupt and branch prediction, which are the main problems associated with the dynamic scheduling of instructions in superscalar machines. Results are written only once, Write-once, directly into the register file (RF). To ensure that results are written in order in their appropriate output registers, a record of instruction order and state is maintained by a status buffer (STB). A 64 entries integrated register file is implemented to hold both renamed and logical registers. To recover the processor state from an interrupt or a branch miss-prediction, a status buffer (STB) and a recovery list table (RLT) are implemented. Novel aspects of the above system architecture are instructions in an OoO fashion to multiple parallel functional units without considerable hardware complexity.

1. Introduction

Superscalar processing, the ability to initiate multiple instructions during the same clock cycle, is the latest in a long series of architectural innovations aimed at producing ever faster microprocessors. Because individual instructions are the entities being executed in parallel, superscalar processors exploit what is referred to as instruction level parallelism by issuing multiple instructions per cycle to each functional unit when dependencies allow [5], [21], [22]. In these processors instructions are not necessarily executed in order; an instruction is executed when processor resources are available. To execute instructions Out-of-Order (OoO), data dependencies among instructions must be disabled by the scheduling system during compilation, run time, or both. Dynamic scheduling detects dependencies in a set of dynamic instruction stream. The most general form of dynamic scheduling, the issue and execution of multiple OoO instructions, can significantly enhance system performance [2], [4], [13], [15], [17], [18]. Instructions with no dependencies are executed if they meet the constraints of the issuing algorithm (discussed in later section).

There are other issue difficulties: branch predictions, and fast precise interrupt particularly if a fast response time is desired [7]–[10]. Interrupts are precise in a processor with Out-of-Order execution, if after the execution of the interrupt routine, the processor state visible to the operating system and application can be reconstructed to the state a processor would have, had all instructions executed in sequence up to the point of an interrupt. Branch, represents about 15% to 30% of executed instructions for many applications [2], [19], [20], decreases the effectiveness of multiple issues to functional units if instructions following an undecided branch cannot be issued. Performance may be improved by enabling speculative executions as a predicted path information. If the gains on correct paths out-balance the losses from nullifying execution effects on incorrect paths, performance improves.

Simply stated, achieving higher performance means processing a given program in a smaller amount of time. Therefore, it is imperative that cycle time be considered when investigating new architecture. Since a processor performance depends on throughput and cycle time, if throughput is increased at the expense of cycle time, a net performance improvement may not occur. To this end, we propose in this work an issue system named Dynamic Fast Issue (DFI) mechanism to issue instructions in an OoO scheme to different functional units, and write the results directly to the register file.

The Out-of-Order completions information’s bits are stored simultaneously in the first-in-first-out status buffer (STB). New instructions are allocated to entries in the instruction buffer, while old instructions are discarded. Results are written once in their corresponding slots into the register file. This is different from conventional mechanism, where results must be written into...
a so-named reorder buffer then copied into the register file. The DFI system, shown in Fig.1, supports dependency resolution, fast precise interrupt handling, and branch mis-prediction resolutions. A Status Buffer (STB) [1], stores information in order about the resource components needed to execute and discard spurious instructions. A fast assignment table (FAT) stores the system state at a given time. A recovery list table (RLT) is used to store the system state after an interrupt or a branch occurs.

In section two, we describe the DFI architectural components and processor scheduling. Fast precise interrupt, and branch mis-prediction recovery are also given in this section. Section three presents our methodology and results. Related works are given in section four. In section five we present our conclusion and future work.

2. DFI’s Architecture Description and Processor Scheduling

2.1 Architecture Description

The DFI system, is implemented within a so named functional assignments register microprocessor FARM [1]. The DFI’s baseline block diagram is shown in Fig.1. It consists of the following components:

Register Assignment Unit (RAU) The RAU consists of a fast assignment table (FAT), and a recovery list table (RLT). Both tables are 32 words list and are implemented with eight read ports and four write ports. Read ports are used at decode stage, where it is necessary to obtain the physical address of three registers for each instruction. Write ports are used at dispatch stage to refresh the current FAT table with the new renamed register. The FAT stores the pointers to all machine registers within the registers file. The RLT stores the pointers, the working registers for the last completed instructions, to all machine registers within the register file.

Free Register Bank (FRB) It stores the physical addresses of the free registers in the register file, which holds both logical and renamed registers. It is organized as a first-in-first-out data structure and implemented with four read ports and four write ports.

Register File (RF) It contains 64 registers, 32 are the architectural registers and the other 32 registers are used in renaming. They are 33 bits length (32 bits for data and one bit to indicates the validity of the register).

Status Buffer (STB) [1] It is also arranged as a first-in-first-out buffer that holds information in order about the decoded instructions. The status buffer (STB) fields’ descriptions are shown in Table 1. These fields are explained in more details in Sect.2.2.

High Speed Switching Circuitry (HSSC) It is a fully connected crossbar network used to implement high-performance network switches. An intelligent centralized scheduler that can ensure fairness and high utilization determines this crossbar configuration, which is

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bit(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC</td>
<td>1</td>
<td>Instruction execution</td>
</tr>
<tr>
<td>LROA</td>
<td>5</td>
<td>Logical Register Output Address</td>
</tr>
<tr>
<td>SPC</td>
<td>2</td>
<td>Speculative execution</td>
</tr>
<tr>
<td>DPRA</td>
<td>6</td>
<td>Discarded physical register output address</td>
</tr>
<tr>
<td>PROA</td>
<td>6</td>
<td>Physical register output address</td>
</tr>
</tbody>
</table>

Table 1 Status buffer field description. The status buffer (STB) is implemented with 16 entries and works as a FIFO with 4 write ports and 4 read ports. Four write ports are used at the end of decode stage to add information in order from until four instructions per clock cycle. Read ports are used to retire in order until four completed instructions per clock cycle.
not of our interest in this paper.

2.2 System Scheduling

Our processor has a fetch bandwidth of four instructions per cycle and has five-pipeline stages.

During the **Fetch Stage**, four instructions are fetched from the instruction cache, and the address of the next instruction is computed. To help smooth out the instruction fetch irregularities caused by cache misses and branches, fetched instructions are buffered in the instruction buffer. Using this buffer, the fetch mechanism can build up a “stockpile” to carry the processor through periods when instruction fetching is stalled or restricted.

During the **Decode Stage**, the physical address of each source register is loaded from the Fast Assignment Table (FAT). The physical address of each output register is loaded from the First-in-first-out Free Register Bank (FRB), and the Status Buffer (STB) entry for each instruction is computed as follow:

- **PROA** (Physical Register Output Address): this address is read from the Free Register Bank (FRB).
- **DPRA** (Discarded/freed Physical Register Address): this address is read from the Fast Assignment Table (FAT).
- **LROA** (Logical Register Output Address): this address is obtained from the instruction opcode.
- **SPC** (Speculative execution): this is obtained from the branch logic.
- **IC** (Instruction execution): is reset when the instruction is executed.

At the **Dispatching Stage**, instructions are identified and sent to their respective reservation stations (not shown in the figure for visibility). PROA addresses are stored at their respective addresses (LROA) into the Fast Assignment Table (FAT). The data of the source registers is read from the register file and from the result bus.

At the **Execution Stage**, instructions with invalid source registers scan the result bus to get them. Each functional unit gets an instruction from its reservation station and executes it.

The last stage is the **Completion Stage**. The results of the executed instructions are stored directly into the register file. Their respective validity bit is set. The executed bit of the respective entries in the status buffer (STB) is also set. Until four instructions are completed in order, and for each completed instruction:

- **DPRA** is stored at the tail of the free register bank (FRB).
- Validity bit in the register file is reset. **PROA** is stored at the LROA in the recovery list table (RLT).

### 2.2.1 Completion Procedure

In our approach, the completion procedure is different from conventional mechanisms, where results are first written Out-of-Order into a reorder buffer (for ordering) then moved to the register file.

In DFI’s mechanism, the results of executed instructions are written directly to the register file, and their respective validity bit is set. The executed bit (IC) in the status buffer is also set. To ensure that results are written in order in the appropriate output register, a record of instruction order and state is maintained by the status buffer (STB).

At the decode stage, an entry in the Status Buffer is assigned to each instruction. So that when an instruction has been executed, information about its state is stored, and the instruction is ready to be completed in order. When an instruction is completed, previous renaming of the output register is freed (DPRA). Next instructions will take as source register the last renaming. Finally, information to free the register is stored in the Status Buffer (STB).

Figure 2 presents a description of the completion procedure, which is summarised in the following three phases:

1. Discarded/freed register is stored in the Free Register Bank
2. Validity bit in the register file is reset
3. Recovery List Table (RLT) is updated (to recover the processor state (Sect. 2.3))

#### 2.2.2 Concurrent Execution

Concurrent execution of instructions is limited by dependencies among instructions [2], [8], [9].

From the well known set of dependencies (described else where), Read After Write (RAW), Write After Write (WAW), and Write After Read (WAR), only RAW dependencies are actually inherent to the program. That is, the other two dependencies appear when Out-of-Order execution scheme is used.

Register renaming [9], [10], [12] technique, which dynamically allocates different physical address to a logical register, is used to overcome the above problem.

Most register renaming approaches use renaming buffer [6], [7], [22]. In these approaches, the renaming buffer is a small current addressable memory (CAM) that stores temporarily the output contents of dynamically renamed registers. When an instruction is completed its result is moved from the renaming buffer to its logical register in the register file. When an instruction needs a source register, it has to look for it in the rename buffer and in the register file getting the younger instance. This may cost multiple read cycles to both register file and CAM when an instruction needs a source register.

In our DFI mechanism, both renamed and logical registers are implemented in a single integrated register file and are addressed using a fast assignment table.
(FAT). Results are written only once in the register file and are not necessarily to rewrite them when the instruction completes. Hence, to read the contents of register \(R_j\) it is necessary to read the contents of the location \(j\) of the assignment table.

The register file has 64 entries, 32 are used to store logical registers and the remaining registers are used to solve dependencies. Any entry in the register file can store a logical register, a renamed register or no one of them. Initially the first 32 registers within the register file hold the logical registers. The address of a register is assigned at the dispatch time following this technique: If \(R_m\) (0 < \(m\) ≤ 31) is the output register of an instruction \(n\), then it is renamed and a free register address in the register file is reserved for it. This address is written in position \(m\) in the fast assignment table (FAT), so that future instructions that use this register as source will take it from the assigned address.

In order to see how the DFI function, the following simple instruction stream is used as an example.

\[
\begin{align*}
R4 &\leftarrow R1 - R3 : i_1; \\
R5 &\leftarrow R4 - R6 : i_2; \\
R4 &\leftarrow R5 + R3 : i_3; \\
R3 &\leftarrow R1 + R6 : i_4; \\
R2 &\leftarrow R2 + R3 : i_5; \\
R10 &\leftarrow R2 - R4 : i_6;
\end{align*}
\]

In the above instruction stream, the four output registers R4, R5, R4, and R3 are renamed by R9, R46, R43, and R49 respectively, which are obtained from the first-in-first-out register bank (Fig. 3), and logical registers are mapped to physical registers through the fast assignment table (FAT) also shown in Fig. 3. The logical register R1 of instruction i1 is mapped to the physical register address &R10. That is, register R10. The logical register R3 is mapped to the physical register address &R49. The same process is done for instruction i2 except that the logical register R4 is not mapped.
to the physical register address &R45, but it is mapped to the last renaming of the register R4 in i1, which is R9. This is to avoid registers conflict. Similar mapping and renaming process is done for i3, and i4. 

After dependencies resolution the new code is shown in the following instruction stream.

\[
R9 \leftarrow R10 - R49 : i_1; R46 \leftarrow R9 - R47 : i_2; R43 \leftarrow R46 + R49 : i_3; R49 \leftarrow R10 + R47 : i_4; R2 \leftarrow R2 + R3 : i_5; R10 \leftarrow R2 - R4 : i_6;
\]

Only RAW dependency, which is inherent to the program, appears in the final code, therefore the first three instructions must be executed sequentially but the fourth one can be concurrently executed with any other. Figure 4 shows the DFI components’ state after decoding stage. As shown in the above figure, the fast assignment table’s content has been updated, the head pointer in the FRB has been changed, and four new instructions have been added to the status buffer (STB). The information stored in the status buffer is used to recover the processor’s state when an interrupt occurs. When an interrupt is detected, the program counter changes to the above interrupt routine address, and the instruction processing continue (discussed in more details in the coming section).

2.3 DFI’s Precise Interrupt and Branch Mis-Prediction Handling

Providing support for interrupts is often a challenging part of the design of a pipelined processor because an instruction may be initiated before its predecessors have been completed [3], [11], [23].

In our system, a precise interrupt may be caused by instruction, \(q_j\), that is executed OoO, i.e., not all instructions preceding \(q_j\) have completed. To achieve a processor state that reflects that of a conventional machine that executes instructions up to \(q_j\), instructions that precede instruction \(q_j\) must complete execution. If an instruction \(q_i\), which precedes instruction \(q_j\), cause an interrupt while the conventional interrupt point for \(q_j\) is being achieved, the saved state is that of a conventional machine that executed instructions up to \(q_i\). The state of an interrupted process is saved by hardware through a recovery list table (RLT) and a status buffer (STB). The recovery list table is similar in size and format to the FAT. When an instruction is completed, its output physical register address (PROA) is saved into a RLT at the corresponding logical output register entry. Therefore, when the next instruction completes, the RLT has the same content as the FAT after the decoding stage. If the above instruction interrupts the instruction stream flow, the processor refers to the RLT, which holds the correct register mapping, to recover its original state. The DFI’s phases of precise interrupt handling is shown in Fig. 5. Invalidation signals to the reservations station and functional units are sent at the appropriate times. In the Status Buffer remain the entries related to invalidated instructions. These entries contain information about the registers used by the eliminated instructions that have to be discarded. These information is processed following a similar procedure to that already used for instructions completions: four entries will be processed in a clock cycle, PROA will be stored in the Free Register Bank, its validity bit in the register file is reset and the Recovery List Table is updated.

The above procedure is also applied to recover the state of the processor when a branch miss occurs.

The DFI scheme takes advantage of hardware to record and predict branch behaviour at run-time for reducing branch mis-prediction rates and recover the state of the processor in case of mis-predictions branch.
The SPC field, within the status buffer, is set when the instruction is speculative after an unresolved branch. If the branch has been mis-predicted, the processor changes the program counter to its right value, automatically invalidates all speculative instructions, and goes on processing unspeculative instructions. Instructions before the branch are executed and completed, when the branch gets the tail position in the STB, the Recovery List Table (RLT) and the Fast Assignment Table (FAT) are switched and speculative instructions are invalidated. The hardware function is similar to the case of interrupt discussed earlier. PROA are inserted in the free register back, their corresponding validity bits is set, fast assignment table is updated again, and resources occupied by speculative instructions are freed.

3. Methodology and Results

The simulators used in this study are derived from the SimpleScalar 2.0 tool set [23], a suite of functional and timing simulation tool for the AIX/MIPS ISA. The timing simulation executes only user-level instructions, performing a detailed timing simulation of a dynamically scheduled system with two levels of instructions and data cache memory. Simulation is execution driven, including execution down any speculative path until the detection of default, TLB miss, or branch mis-prediction.

To perform our evaluations, we collected results for four SPEC95 C benchmarks plus two other C test programs. Test-prnf is an output formatting code, and Test-math is a constraint solving system. The programs were compiled on a SPARC-Solaris 2.6 using full compiler optimization (-O4 -ifo).

We have selected the appropriate parameters to capture underlying trends in microarchitecture design. The processor has a large window of execution; it can fetch up to 8 instructions per cycle and issue up to 16 instructions per cycle. It has a 32 entries load/store buffer. Loads can only execute when all prior store addresses are known. The processor has also 4 integer ALU units, 2 load/store units, 1-FP adders, 1-integer MULT/DIV, and 1-FP/MULT/DIV. All functional units, except the divide units, are fully pipelined allowing a new instruction to initiate execution each cycle. All functional unit’s latencies assumptions of the SUPER-Base (baseline architecture with a reorder buffer) and the DFI system are shown in Table 2. We assume also a latency of 4 cycles for resynchronizing the register mapping tables, a minimum branch mis-prediction penalty of 7 cycles plus the number of cycles the branch instruction stalled in the processor pipeline.

Register file, where both renamed and logical registers are found, is implemented with RAM memories. Registers are addressed using the FAT table. Thereby register data are loaded faster and cycle time can be shortened. As mentioned earlier, to ensure that results are written in order in the appropriate output register, a record of instructions order and state are maintained by a status buffer. This is different from conventional mechanisms, where results are first written into a so-named reorder buffer and then copied into the register file. Branch prediction is provided by a branch target buffer (BTB) and pattern history table scheme. We use 2048 entries, direct mapped BTB and 2-bit saturating counters used as a finite-state machine to encode the past behaviour of the branch. The counters cycles through 4-states, ranging from strongly and weakly not taken to weakly and strongly taken.

### Table 2: Functional units latencies

<table>
<thead>
<tr>
<th>Function Unit</th>
<th>SUPER-Base’s Latency (cycles)</th>
<th>DFI’s Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Store</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Branch</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Integer</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Fl.Pt.</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Our workload primarily consists of Compress95, gcc, go, and Vortex from SPEC95 suite and two other C tests benchmark. We choose these benchmarks because they show varying levels of IPC and they have SPEC programs, high branch mis-predictions rates of 20%, 18%, 30%, and 6% respectively for the length of our run. Results are then reported for simulating each program for up to 100 million instructions.

We present the performance of the DFI system under a number of various scenarios and architectural alternatives. We examine (1) the effect of the issue width on execution time for different branch prediction accuracy, (2) the issue width for selecting which alternate path to spawn, (3) the effect of the issue width and branch prediction accuracy on throughputs.

Figures 6 and 7 show throughputs and percent execution time comparing the DFI system and the SUPER-Base architecture. To provide a fair comparison, both architectures were simulated with the same configurations. Overall, the DFI system provides better since it needs to write results only once to the registers file. These programs average a 12% over the SUPER-Base conventional architecture for programs with varying level of IPC and high branch mis-predictions rates of 6%, to 30%. Figures 6(a), (b), and (c) show the IPC performance for the six benchmarks programs: Vortex, Compress95, li, and gcc from SPEC95, Test-math, and Test-prnf when the system is running with Bimod, Perfect, and Combined branch prediction accuracy. In each of these measurements, the SUPER-Base is shown for comparison purpose with similar DFI's configuration. Bimod accuracy is bimodal predictor accuracy; it
uses a branch target buffer with 2048 entries and 2-bit counters. Combined accuracy is a bimodal and 2 level adaptive predictor with 1024 entries.

In Fig. 7, benchmark percent execution time of DFI system with register renaming, precise interrupt, and branch prediction mechanism with various branch prediction accuracy are given. The performance improvement in each benchmark when executed in Bimod, Perfect and Combined branch prediction accuracy machine of varying degree of issue width is presented. Three programs, Vortex, gcc and Compress95, among the benchmarks have shown the lowest percent instruction execution time in the tree different branch prediction measurements, shown in Figs. 7(a), (b), and (c). This is due to their low mis-prediction rates of 6%, 18%, and 20% respectively for the length of our run. It should be noted that, in both measurements little performance gain is seen when increasing the issue width beyond 8-way.

Considering both performance measurements, the ideal performance evaluation, when we run our benchmarks for different branch prediction accuracy over various issue width, is seen in programs which show low branch miss-prediction rates. Those programs are Vortex, gcc, and Test-math, which show also the highest throughput (Fig. 6) and the lowest percent instruction execution time (Fig. 7) when we run with 8-way issue width configuration, Bimod and Combined branch prediction accuracy. On the other hand, performance improvement of DFI over our baseline architecture is clearly seen when we run Vortex, Compress95, and Test-math programs with 4-way and 8-way issue width configuration.

4. Related Work

This research was motivated by a study done by I. Arise, et al. [22]. In their paper, they demonstrated the potential for issuing Out-of-Order and writing results only once to the registers file, but they did not simulate the architecture, nor did that paper presents a specific solution for branch mis-prediction recovery. Our research presents a full hardware architecture, which solves data dependencies, supports precise interrupt and branch prediction. Simulation results are also given. It presents then an architecture that realises much of the potential demonstrated by that work. The hardware is simpler than those architectures based in renaming buffer. This is different from conventional mechanisms implemented with a renaming buffer (CAM). In these systems, when an instruction is completed its result is moved from the renaming buffer to its corresponding logical register in the register file. When an instruction needs a source register, it has to looks for it in the rename buffer and in the register file getting the younger instance. This makes register data access and cycle time longer.

5. Conclusion and Future Work

We have presented an issuing mechanism architecture (DFI), which performs in an integrated fashion all tasks indispensable for a dynamical scheduled superscalar processor. The above mechanism uses a fast assignment table (FAT), a recovery list table (RLT) and a status buffer (STB). In addition to data dependencies handling, processor recovery after an interrupt or branch mis-prediction is also supported and no clock cycles are lost. The proposed architecture has the following main futures:

Write-Once: Results are written only once and directly to the register file.

Multipurpose Register File (MPRF): This was not discussed in this work for space limitation and will be our next topic.

The DFI is, therefore, faster since it writes results once, Write-once and is implemented with RAM memories that are faster and smaller than the CAM memories. Our performances evaluation results show that the DFI achieves a 12% average gain over the SUPERBase conventional architecture for programs with varying level of IPC and high branch mis-predictions rates of 6% to 30%. The simulations were done for a range of branch prediction accuracy and various issue widths. In addition, we notice that for our benchmarks, the gain of percent execution time (PET), and throughputs (IPC) is slightly improved when the issue width is beyond 8-way issue.

Finally, we conclude that our proposed architecture can be implemented without considerable hardware complexity. Therefore, the total power consumption and die area, which are still under investigation, are estimated to be satisfactory, comparing to other conventional architectures, especially when it is fabricated with the 0.25 µ CMOS process.

Our future work is to study the interaction between the DFI and the compiler. Further more, we expect to study its behaviour in other environments.

Acknowledgements

We would like to thank all anonymous reviewers for providing us useful comments. The first author gratefully acknowledges the generous support of Kevin Sharm from the Computer Science Department at the University of Virginia for the answers to numerous questions as we used the SimpleScalar simulation infrastructure. He also thanks Jin Zhu from the Department of Computer Science at Hong Kong University and Jim David from Microsoft for their constructive comments.

References

[22] H. Leister, “Quantitative analysis of precise interrupt mech-

Abderazek Ben Abdallah received his B.S degree in Electrical Engineering from the ISTIM University, Tunisia in 1990, the B.S degree in Electrical Engineering, and the M.S. degree in Computer Engineering from Huazhong University, China in 1994 and 1997 respectively. Then he worked as a research assistant at the National Storage Laboratory at the same University. Currently he is working for FARM processor project toward his Ph.D degree in Computer Engineering at the University of Electro-Communications in Japan. His research interests include Computer architecture, VLSI design and performance evaluations. He is a member of IEEE.

Mudar Sarem received his B.S degree in Electrical Engineering from Tishreen University in 1989. Then he worked at the general organization of Remote Sensing as a Processing Engineer. Then he joined Huazhong University of Science and Technology HUST in China in 1994, where he got his M.S and his Ph.D degree in computer Engineering. His research interests include Computer architecture, VLSI design, and Data base System. He is a member of IEEE.

Masahiro Sowa received his Ph.D degree in Electronics in 1974 from Nagoya University. In the same year he became an Assistant Professor at the Department of Computer Science of Gunma University. In 1976 he became an Associated Professor at Gunma University and in 1987 a Professor at Nagoya Institute of Technology. From 1993 he is a Professor at the Graduate School of Information Systems of the University of Electro-Communications. His research interests include parallel processing and computer architectures, especially new parallel and distributed computers. He is a member of IEEE, ACM and Information Processing Society of Japan.