The KIT COSMOS Processor: A Low-Complexity Superscalar Processor*

Toshinori Sato[a]
Kyushu Institute of Technology

Toshiyuki Yamamoto[b]
Alpha Systems, Inc.

Itsujiro Arita[a]
Kyushu Institute of Technology

Abstract
This paper introduces a new microarchitecture, which we call COSMOS, for realizing large-scale superscalar processors with high clock frequencies. In order to achieve the goal, several design techniques on instruction supply mechanism, instruction window, register files, and operand bypass logic are proposed. Based on simulation results of an 8-way dynamically scheduled superscalar processor, we estimate that the contribution of COSMOS microarchitecture on instruction level parallelism is only 1.3% degradation from the conventional superscalar processor with an equivalent scale. Due to its low complexity, a COSMOS superscalar processor works at higher clock frequencies than the conventional one. Therefore, COSMOS microarchitecture is one of the promising candidates for future superscalar processors.

Keywords:
Superscalar processors, Instruction supply mechanism, Instruction window design, Variable latency pipeline, Clustered architecture

1 Introduction

Future microprocessors will rely on higher clock frequencies, wider instruction issue width, deeper pipeline stages, and larger instruction windows in order to improve performance. As the width and size increase, the scaling of the clock frequencies becomes difficult to attain. Obstacles for the scaling are register renaming logic, large register files, and operand bypass logic as well as instruction window wakeup and select logic [11,14]. There are many proposals to alleviate these constraints on the renaming logic [1], the registers [7,24], and the instruction window [4,14]. This paper proposes a new microarchitecture, which realizes large-scale superscalar processors with high frequencies emerging in the near future. We introduce an implementation example, which we call KIT COSMOS processor [18], and describe its unique features. This paper focuses on its characteristics based on the superscalar paradigm, while it is a simultaneous multithreading processor [18,25].

The features of the KIT COSMOS processor, which improve clock frequencies, are summarized as follows.

1. Clustered architecture [18],
2. High bandwidth instruction cache [20],
3. Decoupled instruction window [19],
4. Simplified instruction issue logic [21], and
5. Variable latency pipeline [22]

It is important to consider impact of hardware complexity on processor cycle time for large-scale superscalar processors. In order to solve the problem, it is widely investigated to split the large processor into a number of small processing elements as the first feature. As instruction issue width is increase, requirement of instruction fetch bandwidth also increases. The conventional multiple-port cache combined with the instruction alignment circuit becomes the bottleneck deciding processor cycle time. The second feature of COSMOS attacks this problem. The next two deal with large scope instruction scheduling. The large instruction window is required for efficient instruction scheduling, while it is the most critical in the future [14]. The hierarchical and simplified design helps minimize performance degradation without increasing cycle time. The result bypass logic is another critical problem, which is considered by the last one.

The organization of the rest of this paper is as follows. Next section introduces the KIT COSMOS processor. Section 3 explains the high bandwidth instruction supply mechanism. The large instruction window and the operand bypass logic are dealt with in Sections 4 and 5 respectively. Section 6 surveys related works. Finally, our conclusions are presented in Section 7.

2 KIT COSMOS Processor

This section explains a COSMOS processor model and our evaluation methodology.

2.1 Processor model

Figure 1 shows the block diagram of the COSMOS processor, which consists of one instruction supply mechanism and two instruction execution mechanisms (clusters). The number of the clusters is not necessary two. The instruction supply mechanism will be explained in

*This work is supported in part by the Grant-in-Aid for Scientific Research from Japan Society for the Promotion of Science (No.12780273 and No.13558030). Toshinori Sato was supported in part by the grant from Fukuoka Industry, Science & Technology Foundation (No.H12-1).
[a] Department of Artificial Intelligence, 680-4, Kawazu, Iizuka, 820-8502 Japan.
[b] 2-17-5, Shibuya, Tokyo, 150-0002 Japan.

Section 3. The instruction buffer and the scheduling window will be described in Section 4. The detail of a functional unit will be found in Section 5. These components contribute to high clock frequencies. On the other hand, this section focuses on the clustered architecture, which is combined with the value predictor.

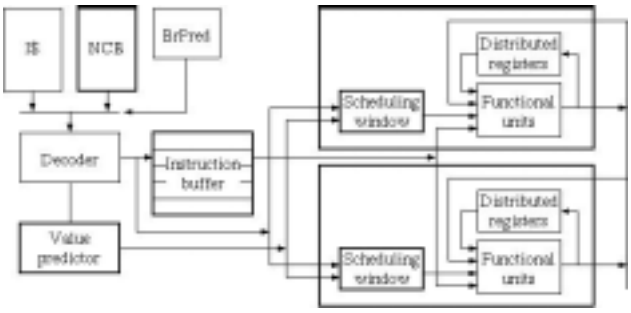

Figure 1: A dual-clustered COSMOS processor

The delay of both the large register file and the long bypass busses is severe and is degrades processor clock frequencies [14]. The clustered architecture attacks this problem by distributing the register files and the functional units across multiple clusters. Each register file becomes small and the bypass busses can be short. Therefore, the clustered processor can operate at higher frequencies than the superscalar processor with an equivalent scale.

In COSMOS processor microarchitecture, a register file is divided into multiple local register files and distributed across the clusters. There are not any copies of each register file nor any global register files. When an instruction is decoded, the steering logic decides in which cluster it is to be issued. If any source operand of the instruction is not held in the local register file, it is required to move the operand from the remote cluster. This incurs penalties at least one cycle. The proposed microarchitecture solves this problem by exploiting value prediction [9,13]. The instruction that requires any source operand located in the remote cluster uses its predicted value and executes speculatively. If the prediction is correct, the processor does not suffer the cycle penalty from moving the operand between clusters. Only when the prediction is incorrect, it suffers the penalty. One of the advantages of this value prediction strategy is that instructions held in the value prediction table are limited. Thus, the prediction table is efficiently utilized and its capacity can be reduced without performance loss.

In the remaining of this paper, we assume a single-clustered COSMOS processor as a representative for large-scale superscalar processors.

2.2 Evaluation methodology

An execution-driven simulator is used for this study. We implemented this simulator using the SimpleScalar tool set [2]. The SimpleScalar/PISA instruction set architecture (ISA) is based on the MIPS ISA.

The simulator models a realistic 8-way out-of-order execution superscalar processor. While register renaming is performed on register update unit (RUU) in the SimpleScalar processor, it can easily model the MIPS R10000's register mapping hardware [26]. The renaming registers, the active list, and the instruction queue share a single structure, which is the RUU. Each functional unit can execute any operations. The latency for execution is 1 cycle except in the case of multiplication (4 cycles) and division (12 cycles).

The default memory hierarchy is as follows. A 4-port, non-blocking, 128KB, 32B block, 2-way set-associative L1 data cache is used for data supply. It has a load latency of 1 cycle after the data address is calculated and a miss latency of 6 cycles. It has a backup of an 8MB, 64B block, direct-mapped L2 cache which has a miss latency of 18 cycles for the first word plus 2 cycles for each additional word. No memory operation can execute that follows a store whose data address is unknown. A 128KB, 32B block, 2-way set-associative L1 instruction cache is used for instruction supply and also has the backup of the L2 cache which is shared with the L1 data cache.

For control prediction, a 1K-entry 4-way set associative branch target buffer, a 4K-entry gshare-type branch predictor, and an 8-entry return address stack are used. The branch predictor is updated at instruction commit stage.

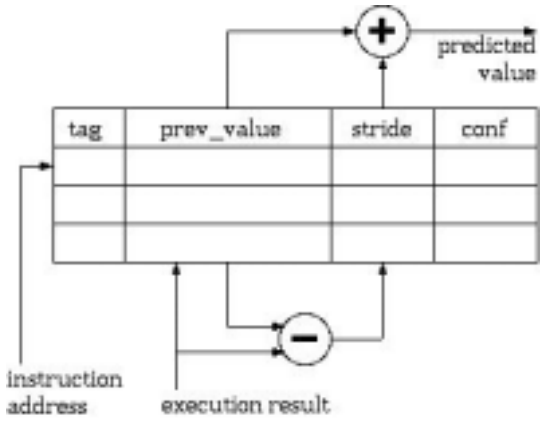

Figure 2: Stride value predictor

Value predictor used in this study is a 4096-entry direct-mapped stride predictor [9]. Figure 2 depicts the predictor. It is indexed by instruction addresses and each entry has a tag field (**tag**), a previous value field (**prev_value**), a stride field (**stride**), and a confidence field (**conf**). The tag field is used for distinguishing indi-

vidual instructions from each other. The previous value field holds the last value generated by the instruction. The stride field keeps a difference of the last two values generated by the instruction. The predicted value is produced as the sum of the previous value and the stride. The confidence filed is a 2-bit up-down saturated counter and decides if the speculation using the predicted value should be initiated. When a prediction is correct, it is incremented. Otherwise, it is decremented. When its most significant bit is 1, the speculation is initiated using the predicted value.

In order to recover the processor state when a misspeculation occurs, an instruction reissue mechanism is used. The instruction reissue re-executes only instructions dependent upon a misspeculated instruction. Those instructions are detected selectively and reissued in instruction window. We implemented an instruction reissue mechanism by extending the RUU. It is found that the reissue mechanism can be practically implemented without big hardware cost [23].

The SPECint92 and SPECint95 benchmark suites are used for this study. We focus on the performance of only integer programs because it tend to be difficult to obtain high levels of parallelism from these types of programs than from floating-point programs. The input files are modified in order that evaluation time is practical. For measuring performance, we use in this paper the committed instructions per cycle (IPC) as a metric. Only useful instructions are considered for counting the IPC. We do not count nop instructions.

3 Instruction Supply Mechanism

We propose a simpler trace cache [15,16], named non-consecutive basic block buffer (NCB)[20]. The followings describe the NCB fetch mechanism and explain the difference between two mechanisms.

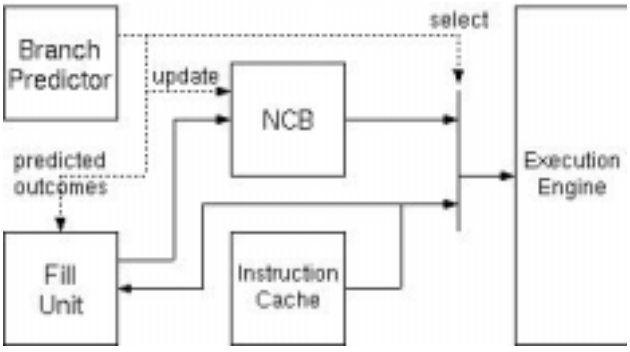

Figure 3: NCB fetch mechanism

The NCB is an extension of branch target buffer (BTB), and caches source basic blocks as well as target basic blocks, i.e. the NCB stores multiple non-consecutive basic blocks. Figure 3 indicates the NCB fetch mechanism. The instruction cache and the NCB are indexed by the program counter. The branch predictor is used for selecting an instruction sequence from the ones supplied by the instruction cache and the NCB. If the branch outcome is predicted taken, the instruction sequence (a trace) from the NCB is selected. Otherwise, the one from the instruction cache is selected. In the case that the trace is not cached in the NCB, instructions are provided from the instruction cache.

Trace constructing process executed in the fill unit is as follows. **Step 1:** The fill unit temporary buffers instructions from the instruction cache in a line buffer. **Step 2:** If there is a branch instruction (both conditional and unconditional) in the instructions and the branch outcome is predicted taken, the instruction sequence reaching the branch instruction is stored in the line buffer. **Step 3:** After next instructions from the instruction cache are fetched, the former instructions stored in the line buffer and the latter instructions are combined into a trace. The trace constructing process is finished when

1. The line buffer becomes full, or
2. The trace contains two branches predicted taken (including indirect jump).

And then, the constructed trace is stored in the NCB.

This process resembles the one performed by the collapsing buffer [6]. In the case of the collapsing buffer, **Step**s **1 - 3** have to be processed in one cycle. On the other hand, in the case of the proposed scheme, these steps can be pipelined because this process goes out from the instruction supplying process. Therefore, the complicated fill unit structure does not have any impact on the processor cycle time.

The differences between the trace cache and the NCB are summarized as follows.

1. The NCB stores only those traces containing taken branches, eliminating unnecessary replication of trace and basic block.
2. In the case of the NCB, a basic block can be split across two traces just like the trace packing [15]. And hence, starting addresses of traces do not always match with basic block boundary.
3. A single indirect jump, return or trap instruction does not terminate trace construction, reducing unused instruction slots.
4. The trace construction is based on predicted branch outcomes rather than actual ones. This decouples the NCB from the execution engine, reducing hardware complexity. In addition, this enables each trace to be constructed early.

According to these characteristics, the NCB fetch mechanism becomes simpler than the trace cache.

Figure 4 compares processor performance. We simulate three models: the baseline model (64K I-cache), the baseline model whose instruction cache is doubled in capacity (128K I-cache), and the evaluated model (64K I-

cache + 32K NCB). Results of the last two models are normalized by the first one. It can be easily found that the large cache model (denoted as **128**) cannot contribute to processor performance. On the other hand, it is observed that the IPC of the NCB model (denoted as **N**) is increased by 9.4% on average and a maximum of 13.7%. This confirms that the NCB can more efficiently supply instructions than the conventional instruction cache.

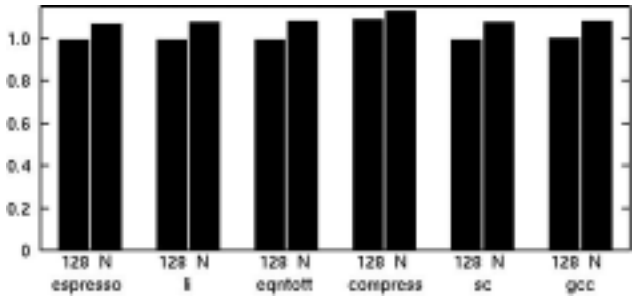

Figure 4: Performance contribution of NCB

4 Simple Instruction Window

This section considers how to realize large instruction window. First, we decouple the instruction window into relatively small instruction scheduling windows and a large instruction buffer, which is a backup for the first one. Second, a simplified instruction wakeup logic utilized in the small scheduling window is proposed. This removes associative lookup and thus the scalability is improved.

4.1 Decoupled instruction window

COSMOS processor architecture relies on data speculation technique as described in Section 2. The data speculation is a new technique removing serialization on a program execution caused by data dependences based on data value prediction [9,13]. When a predicted value is correct, it becomes possible to execute the predicted instruction and its dependent instructions simultaneously, thereby more instruction level parallelism (ILP) is extracted. Otherwise, it is necessary to revert processor state to a safe point where the speculation is initiated. Instruction reissue is such a technique recovering the processor state when a misspeculation occurs. It invalidates instructions dependent upon the misspeculated instruction selectively and then reissues them in the instruction window. It is possible to realize the instruction reissue with a simple hardware structure [23]. However, the instruction reissue technique has a problem. In order to realize the instruction reissue, each instruction remains in the instruction window until it is not speculative. In other words, it cannot release its entry in the instruction window until it is committed. Thus, effective capacity of the instruction window is reduced, diminishing the freedom of the instruction scheduling [5]. In order to maintain the freedom, it is necessary to increase the instruction window size.

In order to realize a large instruction window, we propose to decouple the recovery mechanism for data speculation from dynamic instruction scheduling structure [19]. As shown in Figure 1, the decoupled window consists of relatively small instruction windows for the scheduling and a large instruction buffer for the instruction reissue. The small scheduling windows are distributed across clusters. On the other hand, the large buffer is centralized. After an instruction is fetched and decoded, it enters both the scheduling window and the instruction buffer. When the instruction is dispatched to a functional unit, it leaves the scheduling window and releases its entry but remains in the instruction buffer. When it is committed, it leaves the instruction buffer and releases its entry. In the case that either the window or the buffer is full, instruction issuing stalls.

The small instruction window works for dynamic instruction scheduling. Thus, most of the times each instruction is dispatched from a small window. Since instructions are aggressively deallocated from the scheduling window when they are dispatched, the problem reducing its effective capacity is solved. In addition, its relatively small size does not have serious impact on processor cycle time. However, it is impossible to reissue misspeculated instructions inside the scheduling window. The large instruction buffer works as the backup for the small window and performs the instruction reissue. Each instruction remains in the buffer until it is committed. When a misspeculation occurs, reissued instructions are obtained from the instruction buffer. In order to aggressively speculate instructions, the instruction buffer should be very large. Since it is difficult to access such a large buffer in one cycle, the wakeup and selection logic of the buffer is pipelined to maintain high clock frequencies. It is expected that the pipelining does not degrade processor performance, since the instruction buffer is active only when misspeculations are detected.

It is straightforwardly decided which structure dispatches an instruction to a functional unit. When an instruction is misspeculated, its dependent instructions that have already dispatched and thus which should be reissued are obtained from the instruction buffer only. They have already left the scheduling window. The dependent instructions that have not been dispatched remain in both the scheduling window and the buffer. However, the instructions are obtained only from the scheduling window, because they are not candidates for reissue.

Figure 5 presents the performance contribution of the decoupled instruction window. We evaluate three models. The first (denoted as **B**) is a conventional processor model, which has a 512-entry instruction window. The second (denoted as **P**) is the one whose instruction window is pipelined by 2 cycles. And the last (denoted as **D**)

is the decoupled instruction window model. The instruction buffer and the scheduling window have 512 and 256 entries respectively. The buffer is pipelined by 2 cycles but the scheduling window is not pipelined. The 256-entry instruction window is still too large to achieve high clock frequencies. This is resolved in the next subsection. Each model utilizes 4096-entry stride value predictor [9], and its performance is normalized by the baseline model without the value predictor. In this evaluation we use 128K conventional instruction cache. It is observed that the pipelining of the instruction window is seriously degrades processor performance. However, it can be found that the performance degradation due to the pipelining is compensated by its decoupling. For all cases, processor performance is improved over the baseline model, 3.7% improvement on average. Compared with the conventional model with the value predictor, performance of the decoupled model is lower. However, it is expected that the clock frequencies of the decoupling model is faster than that of the conventional one. Therefore, the processor performance of the decoupling model will be comparable to that of the conventional one.

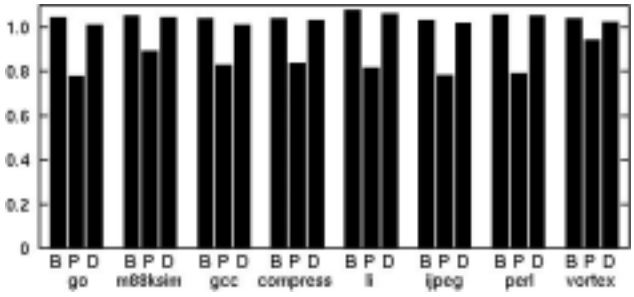

Figure 5: Performance contribution of decoupled instruction window

4.2 EDF Instruction Window

Several definitions are given here to simplify future references in this section. Modern microprocessors fetch multiple instructions per cycle. Following instruction fetch, the instructions are decoded and issued into instruction window. We use the term issue to indicate the process of placing the instructions into the instruction window. The instruction window consists of instruction queue and a buffer maintaining program order such as reorder buffer, and is operated as a FIFO buffer. The instructions remain in the instruction queue until their operands have been ready. Once their dependences have been resolved, instruction dispatch logic schedules the instructions and then dispatches them into functional units. The instruction queue entries containing the dispatched instructions are deallocated so that new instructions may be issued. We use the term dispatch to move the instructions from the instruction queue to the functional units, where they are executed. After completion of execution, the instructions still wait in the instruction window until their preceding instructions have been retired from the instruction window. When the instructions reach the head of the instruction window, they are retired from it. The instructions may be completed out-of-order but are retired in-order. We use the term instruction queue as the structure holding the instructions waiting for dispatch. On the other hand, the term instruction window is used as the structure holding the issued and completed instructions as well as the waiting ones.

In order to eliminate anti- and output-dependences, modern dynamically scheduled processors perform register renaming. There are two common ways to implement the register renaming. One is using separated renaming registers, which are usually constructed by reorder buffer. The other combines the renaming registers with architecture registers in a single register file. We focus on the latter case, especially based on MIPS R10000 [26].

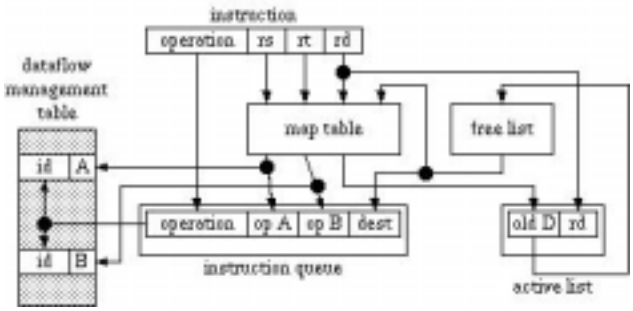

Figure 6: Dataflow management table

In order to improve the scalability of the instruction window by reducing the delay of the instruction wakeup logic, we propose the explicit data forwarding (EDF) instruction window [21]. The main purpose of the EDF instruction window is using RAM, which has more scalability than CAM. The EDF instruction window consists of the RAM instruction window and a table named dataflow management table (DMT). In order to replacing CAM by RAM, the dependences between instructions are definitely explained by any means. The DMT keeps the dependences. Figure 6 depicts the DMT with attached to the register mapping hardware. It is indexed by physical register number and each entry holds IDs indicating specified instruction window slots. In Figure 6, the number of IDs that are registered in each entry is one. However, it can be increased by analyzing the tradeoff between performance and its hardware cost. The dependences between instructions are registered when every instruction is issued, and the DMT is referred when instructions complete. The registration process is as follows. As shown in Figure 6, the DMT is indexed by the physical operand register numbers and the ID associated with the instruction that requires the operands are registered. Since the instruction in the figure has two

the instruction in the figure has two operands, the ID corresponding to the instruction is registered in 2 entries. The identifiers (denoted as **A** and **B**), that tell which operand for the instruction it is, are also held in the DMT.

The reference process and its following instruction wakeup process are explained in Figure 7. When an instruction completes, the DMT is indexed by the result tag of the instruction, which is physical destination register number. From the table, instruction ID that requires the execution result is obtained. Using the ID, the ready bit of an entry associated with the instruction is set. If all ready bits in the entry are set, the instruction is ready for execution (wakeup). As can be seen, there are no associative lookup in the instruction wakeup logic. And thus, the instruction window is implemented using RAM.

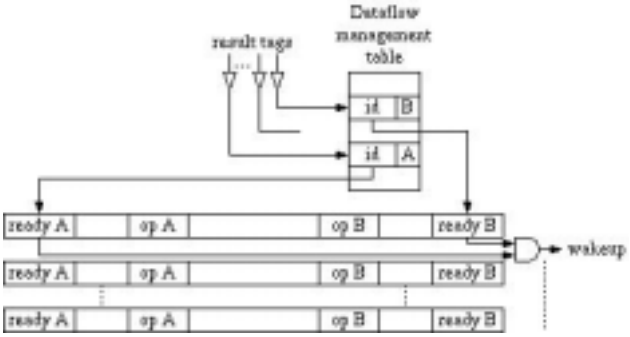

Figure 7: EDF instruction window

It is important to mention how to handle branch mispredictions. When a branch is mispredicted, dependences held in the DMT are incorrect. Therefore, it is necessary to revert the table to a safe point where the speculation is initiated. This is easily handled. Every time a branch is predicted, a checkpoint of the DMT is made, just like the map table[26].

Next, we consider the delay of the instruction windows. In the case of the conventional instruction window, it is realized using CAM in order to the process of the wakeup logic. Since each entry of the window requires *2 * IW* comparators, where *IW* is the instruction issue width, the CAM instruction window size is in the order of *O(IW * WS)*, where *WS* is the instruction window size. From the detailed analysis by Palacharla et al.[14], the delay of the CAM window is in the order of $O(IW^2 * WS^2)$. On the other hand, the delay of the proposed instruction window is estimated as follows. The sizes of the DMT implemented using RAM and the RAM instruction window are both in the order of *O(WS)*. Based on the analysis on the register renaming logic by Palacharla et al.[14], the delay of them are both in the order of $O(IW^2)$. Therefore, the EDF instruction window is more scalable than the conventional instruction window.

Figure 8 compares processor performance. Performance for the proposed RAM instruction window is normalized by that of the conventional CAM instruction window. Every instruction window has 256 entries. For each group of three bars in the figure, the left, middle, and right bars indicate the results in the cases where the number of the ID field in the DMT is 1, 2, and 3, respectively.

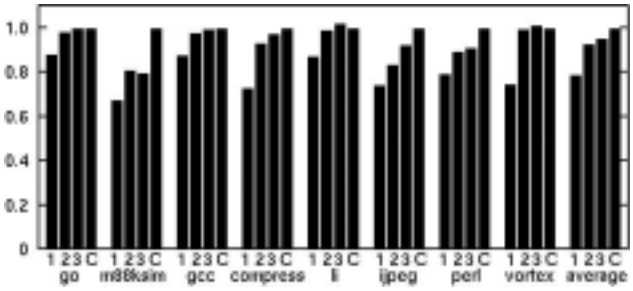

Figure 8: Performance contribution of EDF instruction window

It is found that the DMT with three ID slots is required for the EDF instruction window to achieve processor performance comparable to that of the conventional CAM instruction window, except for the case of 124.m88ksim. On average, it attains 95.4% of that of the conventional case. It is interesting that in the case of **130.li**, the processor performance of the RAM instruction window exceeds that of the CAM window. This is due to the indeterminate characteristics of out-of-order execution. For example, if the instruction issue stall reduces useless instructions that are executed on branch misprediction path, the performance is improved. When *ID* decreases to two, 92.7% of the conventional performance is achieved. If it still decreases to one, only 78.9% of the performance is achieved. Therefore, it is a good tradeoff point that the DMT has two ID slots.

5 Variable Latency Pipeline

The execution stage consists of execution latency and result drive time for the operand bypassing [11]. It is almost impossible to move the bypass logic from the execution stage to the issue stage, since the issue stage is already critical [26]. The delay of the bypass logic imposes the execution stage to be divided into several stages, increasing the execution latencies. Pipelining is one of the techniques realizing the high-speed circuits and can improve the throughput of a function. However, it also increases the latency of the function and thus processor performance sometimes cannot be improved by the technique. Therefore, careful considerations are required for applying the pipelining to the bypass logic. On the other hand, asynchronous and pseudo-asynchronous circuits are the other techniques realizing high-speed circuits. How-

ever, they have the following problems. The asynchronous and pseudo-asynchronous circuits need a completion detector of each operation. The detector becomes the critical path of the circuits and increases processor cycle time. Furthermore, the throughput can be diminished for specific operands. From these considerations, techniques to reduce execution latencies including bypassing are required.

In order to mitigate the constraints on the bypass logic, we exploit variable latency pipeline (VLP) structure [22]. Figure 9 shows the concept of the VLP [12]. A function can be implemented by several kinds of circuits whose design policy are different with each other. In Figure 8, two circuits are used for implementing the function. Circuit A is designed so that most of the longest path of each operation is shorter than a processor cycle time. Circuit B is designed so that the critical path of the circuit is shorter than the cycle time, and is pipelined. Combining these two kinds of circuits reduces the effective latency to execute the function and also maintains the throughput of the function even for the operations that are executed in two cycles.

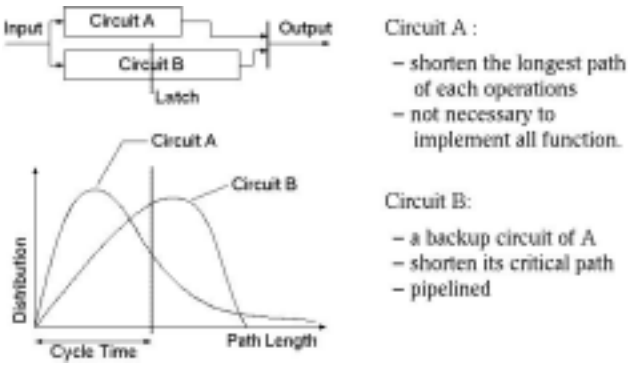

Figure 9: Variable latency pipeline structure

In order to select one from two results, a completion detector for circuit A is required. The detector can be pipelined since the pipelined circuit B works as the backup of the circuit A. Thus, the detector does not increase the critical path. From the above considerations, it can be seen that high frequencies and short latency function is implemented. It is true that the total transistor count increases in order to implement two circuits. However, it is possible to reduce the count if the circuits A and B share circuitry.

Figure 10 depicts the impact of the VLP on processor performance. We evaluate integer ALUs that utilize VLP. The latency of the ALU utilizing the VLP is 2 cycle when a carry propagation over 16 bits occurs. Otherwise it is 1 cycle. Candidate functions for the VLP include not only addition and subtract instructions but also load, store, and branch instructions. Multipliers and dividers do not utilize the VLP technique. For each group of four bars, the left bar is for the processor consisting of two latency pipelined ALUs and the right bar is for that consisting of the variable latency pipelined ALUs. Every simulation result is normalized by the processor model whose ALUs have 1 cycle latency. We can find that, utilizing the VLP, processor performance for most of the programs is comparable to performance of the baseline processor. It achieves 96.4% of the baseline.

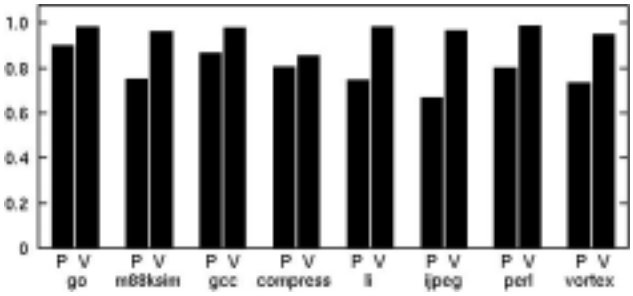

Figure 10: Performance contribution of VLP

6 Related Work

There are several studies on the clustered architecture [3,8,14], in which the register files, the instruction queue, and functional units are distributed across multiple clusters. Its motivation is to increase the clock frequencies by reducing the size and complexity of each cluster. The motivation of COSMOS clustered microarchitecture is the same, while there are some differences from the previous proposals. First, there is not any global register file nor copies of local register files in COSMOS microarchitecture. Second, COSMOS exploits the value prediction technique for the instruction assignment mechanism, which decides in which cluster an instruction is issued. Third, due to the introduction of the value prediction, the instruction window in a COSMOS processor has the hierarchical characteristics. And last, each cluster in COSMOS is a relatively large-scale superscalar processor. In this paper, we have evaluated 256-entry local instruction queue.

Trace processor architecture [17] is another related study. The NCB resembles the trace cache [15,16], while some differences have been explained in Section 3. The main difference from Trace processor is that Trace processor exploits both instruction and trace (thread) level parallelism, while a COSMOS processor explained in this paper only relies on ILP. And thus, the purpose of the value prediction in Trace processor is initiating thread level speculation, while that in COSMOS is supporting instruction assignment. In addition, Trace processor also has global registers and its instruction window is not hierarchical.

The EDF instruction window is strongly influenced by Dualflow architecture [10]. It owes the basic idea of ex-

plicitly explaining data communication for Dualflow, which hybridizes control- and data-driven architectures. Instruction sequence is control-driven, while result forwarding between instructions is data-driven. Destinations of a result are explicitly explained in each instruction, removing associative lookup in instruction wakeup logic while Dualflow performs out-of-order execution. One of the disadvantages of Dualflow is explosion of program code. It is reported [10] that the code size is increased by more than 100% and approximately 50% of dynamic instructions are useless. It also requires considerable changes in the instruction set architecture. On the other hand, the EDF instruction window maintains binary compatibility by translating implicit result forwarding into explicit ones dynamically.

A dependence-based instruction window [14] consists of a small number of FIFO buffers. Dependent instructions are steered to the same FIFO. Instructions are dispatched from the FIFO buffers in-order, and hence only heads of the FIFOs monitor the result tags. One of the hurdles for realizing the dependence-based window is its steering policy. While several heuristics are proposed [3,8,14], their complexities also become critical for processor cycle time.

Canal et al. [4] investigated first-use and distance schemes. The first-use scheme consists of Ready queue, where instructions that have available operands completely are issued, and First-use table, where the oldest instruction that reads each physical register. When every instruction finishes, the First-use table is referred and its corresponding instructions are issued to the Ready queue. This scheme eliminates the associative lookup, however performance degradation from the conventional instruction window is significant. In order to maintain the performance, they propose to attach a content addressable queue, named I-buffer. Therefore, the associative lookup is revived. In addition, pointer chasing is used in the First-use table thus there is the impact of serialized lookups of the Fist-use table entries on processor cycle time. The distance scheme consists of Register Available table, which holds available time for each register value, Wait queue, where instructions whose execution latencies are unknown, and Issue queue, where VLIW-style instruction is constructed using instructions from the decoder and the Wait queue. Instructions are dispatched to functional units from the Issue queue, where the associative lookup is removed. However, The Wait queue is a content addressable queue, and hence the distance scheme still relies on the associative lookup. On the other hand, the EDF instruction window eliminates the associative lookup completely. In addition, they only evaluate 64-entry instruction window and do not mention how effective their proposal is for larger windows.

7 Concluding Remarks

This paper introduced COSMOS microarchitecture for realizing large-scale superscalar processors with high clock frequencies. The key features are the partitioned execution engine (clustered architecture), the high bandwidth instruction supply mechanism (non-consecutive basic block buffer), the hierarchical (decoupled instruction window) and simplified (EDF instruction window) instruction scheduling mechanism, and the mitigated bypass logic (variable latency pipeline). The contributions on ILP of the components are 9.4% and 3.7% improvements of the NCB and the decoupled instruction window and 4.6% and 3.6% degradations of the EDF instruction window and the VLP. The total ILP is improved by 4.3%. Since a processor based on COSMOS microarchitecture can operate at higher frequencies than the superscalar processor with an equivalent scale, the total processor performance can further improved. This confirms that COSMOS is one of the promising candidates for future superscalar processors.

One of the primary future studies dealing with COSMOS microarchitecture is evaluating the cluster assignment mechanism based on the value prediction. This can affects the total processor performance. Currently, we expect that this is not a severe issue for COSMOS due to the following consideration. The degradation of ILP that is occurred by cluster partitioning is at most 6%[14]. The degradation in COSMOS cannot be worse, since we can remove the proposed cluster assignment strategy. Therefore, the total ILP including the clustering effect is only 1.3% degradation. This is easily compensated by COSMOS's high clock frequencies.

References

[1] B.Black and J.P.Shen, “Scalable register renaming via the quack register file,” Technical Report CMuART-2000-01, Department of Electrical Computer Engineering, Carnegie Mellon University, 2000.
[2] D.Burger and T.M.Austin, “The SimpleScalar tool set, version 2.0,” ACM SIGARCH Computer Architecture News, vol.25, no.3, 1997.
[3] R.Canal, J-M.Parcerisa, and A.Gonzalez, “Dynamic cluster assignment mechanisms,” 6th International Symposium on High Performance Computer Architecture, 2000.
[4] R.Canal and A.Gonzalez, “A low-complexity issue logic,” 14th International Conference on Supercomputing, 2000.
[5] G.Z.Chrysos and J.S.Emer, “Memory dependence prediction using store sets,” 25th International Symposium on Computer Architecture, 1998.
[6] T.M. Conte, K.N. Menezes, P.M. Mills, and B.A. Patel, “Optimization of instruction fetch mechanisms for high issue rates,” 22nd International Symposium on Computer Architecture, 1995.

[7] J-L.Cruz, A.Gonzlez, M.Valero, and N.Topham, "Multiple-banked register file architectures," 27th International Symposium on Computer Architecture, 2000.
[8] K.I.Farkas, P.Chow, N.P.Jouppi, and Z.Vranesic, "The Multicluster architecture: reducing cycle time through partitioning," 30th International Symposium on Microarchitecture, 1997.
[9] F.Gabbay, "Speculative execution based on value prediction," Technical Report #1080, Department of Electrical Engineering, Technion, 1996.
[10] M. Goshima, N. H. Ha, A. Agata, H. Mori, and S. Tomita, "Proposal of the Dualflow architecture," 12th Joint Symposium on Parallel Processing, 2000.
[11] T.Hara, H.Ando, C.Nakanishi, and M.Nakata, "Performance comparison of ILP machines with cycle time evaluation," 23rd International Symposium on Computer Architecture, 1996.
[12] Y.Kondo, N.Ikumi, K.Ueno, J.Mori, and M.Hirano, "An early-completion-detecting ALU for a 1GHz 64b datapath," International Solid State Circuit Conference, 1997.
[13] M.H.Lipasti, C.B.Wilkerson, and J.P.Shen, "Value locality and load value prediction," International Conference on Architectural Support for Programming Languages and Operation Systems VII, 1996.
[14] S.Palacharla, N.Jouppi, and J.Smith, "Complexity-effective superscalar processors," 24th International Symposium on Computer Architecture, 1997.
[15] S.J.Patel, D.H.Friendly, and Y.N.Patt, "Critical issue regarding the trace cache fetch mechanism," Technical Report CSE-TR-335-97, Department of Electronics Engineering and Computer Science, University of Michigan, 1997.
[16] E.Rotenberg, S.Bennet, and J.Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," 29th International Symposium on Microarchitecture, 1996.
[17] E.Rotenberg, Q.Jacobson, Y.Sazeidas, and J.Smith, "Trace processors," 30th International Symposium on Microarchitecture, 1997.
[18] T.Sato and I.Arita, "The KIT COSMOS processor: introducing CONDOR," International Conference on Parallel and Distributed Processing Techniques and Applications, 2000.
[19] T.Sato, "Quantitative evaluation of pipelining and decoupling a dynamic instruction scheduling mechanism," Journal of Systems Architecture, vol.46, no.13, 2000.
[20] T.Sato, "Evaluating trace cache on moderate scale processors," IEE Proceedings on Computers and Digital Techniques, vol.147, issue 6, 2000.
[21] T.Sato, Y.Nakamura, and I.Arita, "Revisiting direct tag search algorithm on superscalar processors," Workshop on Complexity-Effective Design held in conjunction with 28th International Symposium on Computer Architecture, 2001.
[22] T.Sato and I.Arita, "Execution latency reduction via variable latency pipeline and instruction reuse," 7th International Euro-Par Conference, 2001.
[23] T.Sato, "Evaluating the impact of reissued instructions on data speculative processor performance," Microprocessors and Microsystems, accepted.
[24] S.Wallace and N.Bagherzadeh, "A scalable register file architecture for dynamically scheduled processors," International Conference on Parallel Architecture and Compilation Techniques, 1996.
[25] T.Yamamoto, T.Sato, and I,Arita, "The KIT COSMOS processor: eliminating ineffectual branch instructions via concurrent dynamic optimization," COOL Chips IV, 2001.
[26] K.C.Yeager, "The MIPS R10000 superscalar microprocessor," IEEE Micro, vol.16, no.4, 1996.

Toshinori Sato is an associate professor in the Department of Artificial Intelligence at Kyushu Institute of Technology in Iizuka, Japan. He holds BE, ME, and PhD degrees in electronic engineering from Kyoto University. From 1991 to 1999 he was with Toshiba Corporation, where he worked on the design of several embedded processors such as EmotionEngine for PlayStation2. His research interests include high-performance, energy-efficient, and complexity-effective microarchitectures and design methodologies for VLSIs. He is a member of ACM, IEEE, IEICE, and IPSJ.

Toshiyuki Yamamoto received BE degree from Kyushu Institute of Technology in 2001. He worked on feedback oriented dynamic binary optimization technique for the KIT COSMOS processor. His research interests include processor architectures and parallel processing.

Itsujiro Arita received the PhD degree from Kyushu University in Fukuoka, Japan. He was with Kyushu University from 1965 to 1984, and currently is a professor in the Department of Artificial Intelligence at Kyushu Institute of Technology in Iizuka, Japan. His research interests include computer architecture, parallel processing, operating systems, and computer networks. He is a member of IEICE, IPSJ, and JSSST.