A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors

Masahiro Goshima Kengo Nishino Yasuhiko Nakashima Shin-ichiro Mori
Toshiaki Kitamura Shinji Tomita

Kyoto University
Yoshida Hon-machi, Sakyo-ku, Kyoto, Japan
{goshima, k-nisino, nakashim, moris, kitamura, tomita}@i.kyoto-u.ac.jp

Abstract

The wakeup logic is a part of the issuing window and is responsible to manage the ready flags of the operands for dynamic instruction scheduling. The conventional wakeup logic is based on association, and composed of a RAM and a CAM. Since the logic is not pipelinable and the delays of these memories are dominated by the wire delays, the logic will be more critical with deeper pipelines and smaller feature sizes. This paper describes a new scheduling scheme not based on the association but on matrices which represent the dependences between instructions. Since the update logic of the matrices detects the dependencies between instructions as the register renaming logic does, the wakeup operation is realized by just reading the matrices. This paper also describes a technique to reduce the effective size of the matrices for small IPC penalties. We designed the layouts of the logics guided by a $0.18\mu m$ *CMOS design rule provided by Fujitsu Limited, and calculated the delays. We also evaluated the penalties by cycle-level simulation. The results show that our scheme achieves 2.7GHz clock speed for the IPC degradation of about 1%.*

1. Introduction

One of the most straightforward ways to improve the IPC (instructions per cycle) of a superscalar processor is to increase the instruction issue width ($\boldsymbol{IW}$) and the instruction window size ($\boldsymbol{WS}$). In fact, the early superscalar processors considerably improved the IPC by increasing IW, WS as far as the number of the transistors permitted.

Recently, however, the clock speed becomes the main factor which restricts IW, WS instead of the number of the transistors. Since larger IW, WS does not necessarily result in greater IPC, thoughtless increas of IW, WS would probably degrade the total performance.

Among the components of a processor, the **wakeup** logic is estimated to be one of the main factors to restrict the clock speed. The wakeup logic is a part of the issuing window and is responsible to *wake up sleeping* instructions waiting for its source operands in the window. Over the past few years, gradually more studies have been made on reduction of the wakeup delay [6, 9, 2, 10]. Most of them, however, are base on the same principle as the most conventional scheme.

The conventional wakeup logic manages availability of source operands based on association of a **tag** allocated to the operand. When an instruction is issued, the tag allocated to its result is broadcasted to all the instructions in the window. Each instruction compares the broadcasted tag with the tags allocated to its source operands. If there is a match, the corresponding source operand is marked available.

The wakeup operation is a kind of dependence detection performed in the issuing window. The dependence, however, have been once detected by the register renaming logic in the frontend of a processor. The detection in the window is more complicated than in the frontend, because it can hardly utilize the information of the program order which is available in the frontend, and it is not pipelinable while it is in the frontend.

Although the detection itself is essential for the out-of-order issuing, the whole conventional wakeup operation does not have to be performed in the window. The key to reduce the complexity of the wakeup logic is to move *heavy* parts of the dependence detection to the frontend. This paper describes a way for it.

The rest of the paper is organized as follows: Section 2 describes the conventional dynamic instruction scheduling scheme, and explains why the conventional logic is estimated to be critical. Section 3 gives the detailed explanation of our scheme. Then Section 4 and 5 show the quantitative evaluation results. The related work is summarized in Section 6.

2. Conventional scheme

The delays of almost all the components of a superscalar processor except the execution units are given by increasing functions of IW and WS. Examples of these components are the instruction fetch logic, the register files, the caches and the TLBs, the operand bypasses, and a part of the dynamic instruction scheduling logic, which is the main theme of this paper. On the other hand, the delays of almost all the execution units except the cache part of the load/store units are independent of IW, WS.

Moreover, the delays of these components are more strongly influenced by wire delays than those of the most execution units. The wire delays will increasingly dominate the total delay in advanced technologies.

Therefore the delays of these components will become relatively long compared with those of the most execution units, as IW, WS are increased and the feature sizes are reduced.

But the relative increase of the delays of these components does not directly prevent the reduction of the cycle time, because the delays a cycle can be drastically reduced by pipelining and/or clustering [7, 9]. In other words, these decentralization techniques are indispensable for future processors of larger IW, WS and smaller feature sizes.

The pipelining, however, is not effective to a part of the dynamic instruction scheduling logic. This section describes the reason in detail. First, Section 2.1 summarizes the principle of the dynamic instruction scheduling. Next, Section 2.2 describes the conventional scheduling scheme. Section 2.3 and 2.4 show the logic circuits, then the *pipelinability* of the logics is discussed in Section 2.5.

2.1. Principle of dynamic instruction scheduling

An out-of-order superscalar processor has the buffer to temporally save the out-of-order results of the instructions, namely the reorder buffer, or the physical register file. We indistinctly refer to it as the **out-of-order buffer** or simply the **buffer**. The dynamic instruction scheduling can be considered as local data-driven computation utilizing the buffer. An instruction is allocated a free entry of the buffer, and write the execution result to it. On the other hand, an instruction can start execution when the buffer entries corresponding to all of its source operands have results written, independent of the program order.

Logically, an entry of the issuing window has flags rdyL and rdyR, which indicate whether the buffer entries corresponding to the left and right source operands of the instruction saved into the window entry have the results written[1]. The flags play important roles in the scheduling.

The flow of the dynamic instruction scheduling can be divided into the following five phases, namely the **rename**, **dispatch**, **wakeup**, **select**, and **issue** phases. The flow for an instruction $\boldsymbol{I_c}$ is explained as follows, where the left and right source operands of I_c is the result of a preceding instructions $\boldsymbol{I_p(L)}$ and $\boldsymbol{I_p(R)}$ respectively, and $I_p(L)$ has finished while $I_p(R)$ hasn't:

1. **Rename** I_c is allocated a free entry of the buffer to write the result, and finds the entries allocated to $I_p(L/R)$.
2. **Dispatch** I_c is stored into the window, and the rdyL/R of I_c are initialized. The buffer entry allocated to $I_p(L)$ has the result written, while that to $I_p(R)$ doesn't. Thus the rdyL/R of I_c are initialized to 1/0. I_c *sleeps* in the window waiting for the rdyR to be set.
3. **Wakeup** When the result of $I_p(R)$ is being produced, rdyR of I_c is set to 1 and I_c becomes ready.
4. **Select** I_c is selected from a maximum of WS ready instructions in the issuing window.
5. **Issue** I_c is read from the the issuing window, and sent to an execution unit.

2.2. Scheduling and tag

The conventional scheme introduces a **tag** to realize the wakeup operation described in the previous subsection. Since the tag is an ID to uniquely identify each data which simultaneously exist in the backend of a processor, any serial number functions as the tag. But usually the designator of the out-of-order buffer entry is used, because it has obvious one-to-one relationship with the result written into the designated entry. In this case, the designator is used both to designate an entry of the buffer and to wakeup the consumer instructions. It is self-evident but should be noticed because it is used in different pipeline stages.

The tags are assigned in the rename phase, and used in the wakeup phase. The explanation of these phases in the previous subsection is supplemented as follows:

1. **Rename** A free entry of the buffer is allocated to I_c to write the result. We refer to the designator of the entry as the tagD. In parallel with this, it find the tagDs allocated to $I_p(L)$ and $I_p(R)$. We distinctively refer to these tagDs as the tagL and the tagR.
3. **Wakeup** When an instruction is producing the result, the corresponding rdyL/Rs are set based on association of the tags. The instruction broadcasts its tagD to all instructions in the issuing window. Then each of them compares its tagL/R with the broadcasted tagD. If there is a match, the corresponding rdyL/R is set.

[1] We assume that the number of the source and destination operands are two and one for simplicity. The discussion can be easily extended to the cases of more operands.

In both of the rename and wakeup phases, dependences between instructions are detected. In the rename phase, I_c searches $I_p(L/R)$ to obtain the tagL/R. In the wakeup phase, I_p associatively searches I_c to update its rdyL/R. From the following subsections, we explain how the conventional scheme realizes these operations.

2.3. Rename logic

The rename operation is composed of allocation of tagDs and resolution of tagL/Rs. Since a tagD to be allocated can be decided in advance, for example in the previous cycle, the resolution of tagL/Rs is important for the performance.

The resolution of tagL/R is realized by the **register map table**. There are several ways to implement the table, we take the RAM scheme for its simplicity [8]. The scheme composes the table mainly of a RAM.

The RAM holds the *current* mapping from the logical register numbers to the tags, i.e., it holds the tags indexed by the register number. IW instructions being renamed write allocated tagDs to the entry designated by their destination register numbers. They simultaneously read the tagL/R from the entries designated by their left/right source register numbers. Thus the RAM has IW write and $2 \cdot IW$ read ports.

When there exist dependences between instructions being renamed in parallel, the RAM provides *old* mappings made in the previous cycle. A dependence detector, which is an array of comparators, compares the register numbers of the instructions. When it detects dependences, it replaces the *old* tagL/Rs by the *new* tagDs being written to the RAM.

Since the dependence detector is faster [8], the delay of the rename logic is given by the read delay of the RAM.

2.4. Wakeup logic

Figure 1 shows the block diagram of the conventional wakeup logic. The issuing window including the wakeup logic, is usually implemented not as a centralized logic shown in the figure, but as a set of decentralized logics. We will explain such decentralization in Section 2.6.

The select logic shown in the right of the figure arbitrates WS request signals, and asserts IW grant signals.

The wakeup logic is composed of a RAM-like and a CAM-like parts. Though they are not exactly a RAM and a CAM, we simply refer to them as the RAM and the CAM.

The RAM shown in the top saves tagDs. The read port of the RAM does not have a row decoder, and the grant signals from the select logic are connected to the read wordlines through a pipeline register. The tagDs read from the RAM are sent to the pipeline register shown in the top to be

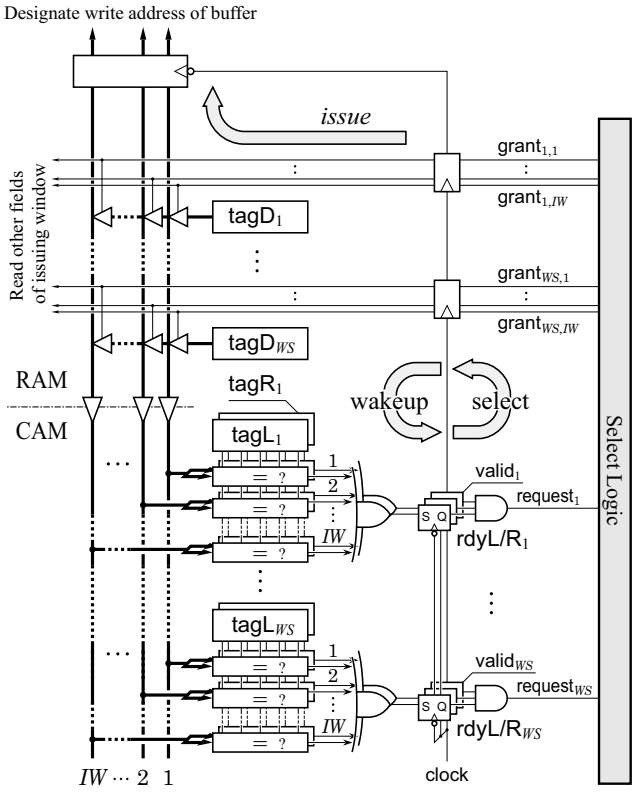

Figure 1. Conventional wakeup logic.

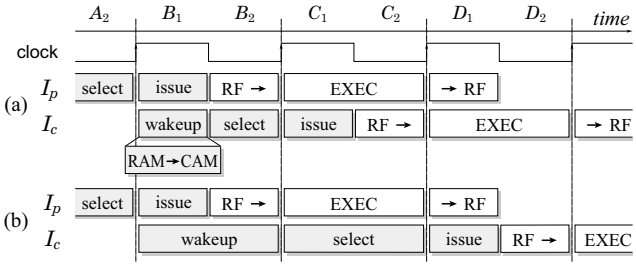

Figure 2. Phases and instruction pipeline.

used as the write addresses of the buffer, and to the comparison input ports of the CAM shown in the bottom. Thus the read operation of tagDs from the RAM is included in both of the issue and wakeup phases. The tagDs sent to the pipeline register shown in the top are delayed appropriate cycles before used as a write addresses of the buffer.

Then the CAM associatively searches the tagL/Rs which match the tagDs sent from the RAM, and sets the corresponding rdyL/Rs. The outputs of the rdyL/R registers are ANDed to generate the request signal to the select logic.

Figure 2 shows the position of the wakeup, select, and issue phases in the instruction pipeline. The figure follows the MIPS R10000 integer pipeline [12]. In this figure,

Exec indicates the execution stage, RF$\rightarrow$ and $\rightarrow$RF indicate read from and writeback to the physical register files respectively. Figure 2-a represents the most critical case for the wakeup and select phases, that is, one-cycle latency instruction I_p is followed by I_c in the consecutive cycle. The result of I_p is sent to I_c through the operand bypass. The behavior of the logics is summarized as follows:

$\boldsymbol{A_2}$ The select logic selects I_p, and asserts the grant signal.

$\boldsymbol{B_1}$ The issue operation for I_p and the wakeup for I_c proceed simultaneously. The tagD of I_p is read from the RAM and sent to the CAM. The CAM sets the rdyL of I_c, and I_c asserts the request signal to the select logic.

$\boldsymbol{B_2}$ This time the select logic selects I_c.

2.5. Pipelinability

Pipelinability of the instruction pipeline phases depends on feedback loops, and the possibility or the effectiveness of speculation. For example, all the five phases of the scheduling are included in a feedback loop from the execution of branch instructions back to the instruction fetch, but they are pipelinable owing to the effectiveness of the branch prediction. The cycles allocated to the phases only increase the misprediction penalty.

The phases of the scheduling has another important feedback loop, which is from the select back to the wakeup phases. The wakeup phase depends on the select phase, because the wakeup operation for I_c can start after the select logic decides the issue of I_p.

This feedback loop practically prevents the pipelining of the wakeup and select phases. Consider the case where one cycle is allocated to each of the wakeup and select phases as shown in Figure 2-b. As mentioned above, the wakeup operation for I_c can start after the issue logic selects I_p. Thus the wakeup and select phases for I_c proceed from B_1 to B_2 and from C_1 to C_2 respectively. Although the result of I_p has already been ready, I_c is not issued in the consecutive cycle of I_p,

It is equivalent to removing the operand bypasses from the one-cycle latency execution units, that is, the ALUs in the usual cases. As we will show in Section 4, it degrades the IPC about 15% at most, and it is unlikely to meet the gain of the clock speed. Thus we can conclude that the wakeup and select phases have to be executed in one cycle for one-cycle latency paths.

2.6. Decentralization

As mentioned before, the issuing window is usually decentralized to a set of subwindows. This technique is quite important, because it considerably reduces the delay of the logics at the cost of small IPC penalties. The decentralization reduces the effective size of each subwindow, and enables latency optimization.

Effective size reduction The structure of the logic circuit of a subwindow is basically the same as that of a centralized window shown in Figure 1, except that IW, WS are reduced to $\boldsymbol{IW'}$, $\boldsymbol{WS'}$, where IW' and WS' are the issue width and the size of the subwindow. When the window is divided into s subwindows, IW',WS' are about $1/q$ of IW, WS respectively.

Only the number of the comparison input ports of the CAM can not be reduced from IW to IW' to receive tagDs from the other subwindows. Although it is ideal that each CAM has IW ports, actual processors usually reduced it at the cost of IPC degradation.

Latency optimization As mentioned before, the wakeup and select phases have to be executed in one cycle for one-cycle latency paths. On the other hand, the phases for two or more cycle latency paths are pipelinable. For example, the issuing window of MIPS R10000 consists of three subwindows, namely the integer, load/store (**LS**), and floating-point (**FP**) subwindows. There are two one-cycle latency paths: from integer to integer, and from integer to LS. And the others are pipelinable. Thus the integer subwindow is the most critical among three subwindows, and the discussion about the critical path can be concentrated to it.

Summary

As mentioned earlier, recent superscalar processors employ increasingly deeper pipelines. Since the wakeup and select phases for one-cycle latency paths are not pipelinable, they will become more critical with deeper pipelines.

The wakeup logic will be more critical between these two phases. The delay of the select logic is composed mainly of the intrinsic gate delays, and it will be reduced steadily with the reduction of the feature sizes. On the other hand, the delay of the wakeup logic is composed mainly of the wire delays of the word, bit, and match lines of the RAM and the CAM. Therefore, the wakeup logic will become more critical with smaller feature sizes. The wakeup delay has already been considerably critical even for as small IW', WS' as recent processors, as we will show in Section 5.2.

For these reasons, the wakeup logic is considered as one of the main factors to restrict the clock speed of future processors. The next section describes the scheme we propose, which replaces the wakeup logic.

3. Proposed scheme

The out-of-order scheduling have to detect the dependences between instructions twice: in the rename and wakeup phases. A consumer instruction searches its producers to obtain its tagL/R in the rename phase, while a producer does its consumers to set their rdyL/Rs in the wakeup.

The logic of the wakeup phase, however, is considerably more complex than that of the rename as seen in the previous section. The reason can be summarized as follows:

1. **Associativity** In the rename phase, a consumer searches the only producer that produces each of its source operands. While in the wakeup phase, a producer has to search all the source operands of its consumers that depend on it. Therefore the search in the wakeup phase must be associative in principle.

2. **Position in instruction pipeline** The rename phase is in the frontend of the pipeline, while the wakeup isn't:

   a. **Pipelinability** The rename phase is pipelinable, while the wakeup isn't.

   b. **Availability of program order** Since instructions pass the rename phase in the program order, the target of search is restricted to the newest instruction that writes a logical register. Utilizing the program order in the wakeup phase is not impossible, but not easy either. The conventional scheme does not utilize it.

Although the associative search in the wakeup phase is essential for the out-of-order issuing, the whole conventional wakeup operation does not have to be performed in the phase. The key to reduce the complexity of the wakeup logic is to move *heavy* parts of the dependence detection required for the wakeup operation to the frontend of the processor. In fact, the scheme we propose removes the association from the wakeup phase in this way.

Our scheme uses matrices which hold the current dependences between instructions in the issuing window. The update logic of the matrices detects the dependences as the rename logic does in the frontend of the processor. Thus the wakeup operation is realized by just reading the matrices.

3.1. Dependence matrices

Figure 3 shows the concept of the dependence matrices. The dependence matrices are composed of the left and right $WS \times WS$ matrices corresponding to the dependences through the left and right source operands. The c-th row and p-th column element of the left/right matrix is 1 if and only if the left/right source operand of c-th instruction is the result produced by the p-th instruction. In short, c-th row of the left/right matrix is a bit-vector which indicates the producer of the left/right source operand of the c-th instruction.

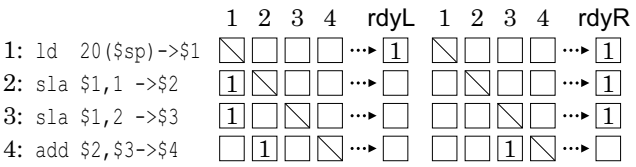

Figure 3. Dependence matrices.

In Figure 3, a series of four instructions are stored in the window from the first to fourth entries. Since the left source operand of the second instruction is produced by the first instruction, the first column element of the second row of the left matrix is set to 1. The other elements can be determined in this way.

In the wakeup phase, IW columns of issued instructions are bitwise ORed on each of left/right matrix, and the resulting column-vectors indicate rdyL/Rs to set. When the first instruction is issued, the second and third rdyLs are set by the first column of the left matrix, and the second and third instructions become ready. If these two instructions are issued simultaneously, the second and third columns are bitwise ORed to set rdyL/R, and the fourth instruction becomes ready.

3.2. Implementation

Figure 4 shows the block diagram of our wakeup logic. Compare it with the conventional one shown in Figure 1. The matrices receive the grant signals from the select logic, and produce the inputs of the rdyL/R registers, i.e., the RAM and the CAM of the conventional logic are replaced by the matrices.

Our scheme does not use a tag for the wakeup operation. If the tag is the designator of the out-of-order buffer, our scheme as well as the conventional scheme reads tagDs from the RAM to use them as the designators of the buffer[2]. The read operation of the tagDs, however, is included not in the wakeup operation but only in the issue in our scheme, as is directly shown in Figures 4. Note that the issue phase, unlike the wakeup, is pipelinable as mentioned in Section 2.5.

Although a bitwise-OR of multiple columns is required for the wakeup operation, no special logic is required for it. The usual RAMs can be used to implement the matrices with minor changes. We will explain the reason in Section 3.4. Figure 5 shows the logic circuit of the cells for a 2×2 minor matrix of the left matrix. A *4T cell* is put in the middle of each cell, and the write ports for update are

[2]Since it is used only as the designator of the buffer, the term "tag" is no longer appropriate for our scheme. But we use it for continuity.

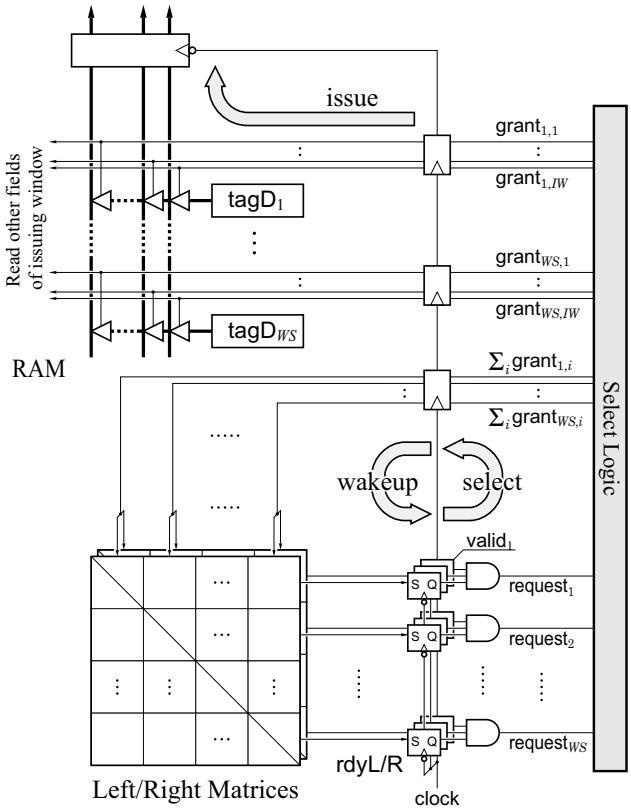

Figure 4. Proposed wakeup logic.

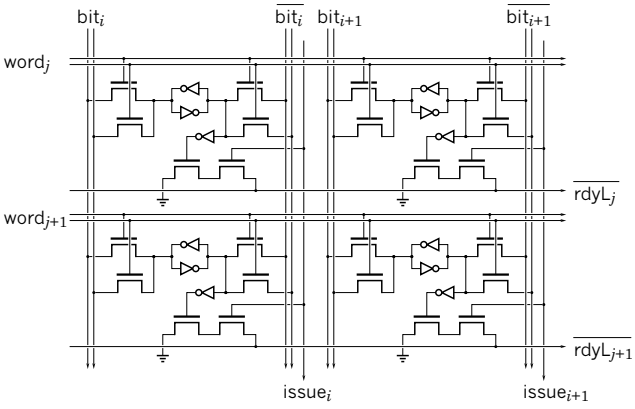

Figure 5. Logic of left matrix.

arranged on the both sides of the 4T cell. And A read port for the wakeup operation is in the bottom of each cell.

Following subsections describe three operations on the matrices: the update and wakeup operations, and status recovery from misprediction.

3.3. Update

Consider the case when I_c is being stored to the c-th entry of the window, and its producer instructions $I_p(L/R)$ has been stored to the $p(L/R)$-th entry. The update operation of the matrices can be divided into following two phases: 1. finding the producer designator $p(L/R)$, and 2. writing the decoded $p(L/R)$ to the c-th row of each matrix. The first and second phases proceed in parallel with the rename and dispatch phases respectively.

The producer designator $p(L/R)$ can be found in the same way as tagL/R are found in the rename phase. The explanation for the rename logic in Section 2.3 holds true for this logic if 'tagD/L/R' and 'register map table' are replaced by 'producer designator' and '**producer table**'. The producer table holds the mapping from the logical register numbers not to the tags but to the producer designators.

The same dependence detector as the rename logic is also required for this logic, and it can be shared between the rename and this logics. Therefore it is mainly the producer table that is required especially to update the matrices.

As mentioned above, the first and second phases proceed in parallel with the rename and dispatch phases respectively. Thus the delays of the phases can be compared with them as follows:

1. **Rename** The out-of-order buffer is usually from one to two times deeper than the instruction window. The tag which designates the buffer entry is thus by zero to one bit longer than the producer designator which designates the window entry. Therefore the register map table which holds the tags is by zero to one bit wider than the producer table which holds the producer designators. The other parameters of these tables are the same. Therefore the delay of the producer table is less than or equal to that of the register map table.

2. **Dispatch** The dispatch logic writes the instructions into the RAM of the issuing window. The decoders for the producer designator is functionally the same as the row decoder of the RAM of the window. Thus the delay of the second phase is almost the same as that of the dispatch logic.

From the above qualitative discussion, we can conclude that the delay to update the matrices is unlikely to lengthen the critical path of the rename and dispatch phases.

3.4. Wakeup

The wakeup operation is performed by the read ports of the circuit. As shown in Figure 5, the read port is basically the same as the conventional 1-read RAM with single-ended bitlines, except the following differences:

Structure The position of the word and bitlines of the read port is opposite from that of the conventional RAM. The wordlines issue run vertically, and the bitlines $\overline{\text{rdyL/R}}$ run horizontally.

This is realized by locally reconnecting the input and the output of the bitline driver of each cell. Thus it imposes nearly no cost.

Behavior The conventional RAM asserts only one wordline at a time, while each of the matrix asserts at most IW wordlines. The wordlines issue for instructions being issued are asserted in every cycle, and at most IW cells are connected to one bitline. If there exists at least one cell set among them, it pulls down the bitline. In short, asserting multiple wordlines of the conventional RAM with single-ended bitlines generates bitwise-OR of the multiple columns.

In order to generate bitwise-OR, the read port of the matrices can not be double-ended in principle, because asserting multiple wordlines can pull down both of the complementary bitlines.

3.5. Status recovery from misprediction

Speculative execution is indispensable for recent processors. For instruction scheduling schemes it is important that the status recovery from a misprediction is simple. In our scheme, the producer table requires checkpointing just like the register map table, while the matrices requires no special operation just like the conventional wakeup logic.

In general, invalid window entries including ones invalidated due to mispredictions partially keep obsolete information. Rows of the matrices keep obsolete elements set, as entries of the conventional logic keep obsolete tagL/R. They can produce *false* outputs.

It is enough to prevent the select logic from receiving *false* request signals. This is usually realized not by the tagL/R memories or the matrices but by glue logic around the rdyL/R registers, because it is less costly. For example, the circuits shown in Figure 1 and 4 use the valid flags of the window entry to do so.

Thus the rows of the matrices as well as the tagL/R memories can be left unchanged when invalidated due to mispredictions.

3.6. Decentralization

This and the next sections describe two techniques to speed up the accesses to the matrices, in particular the read access for the wakeup operation. The first is decentralization similar to that for the conventional scheme mentioned in Section 2.6, and the second is characteristic of our scheme. Though the techniques can be applied separately,

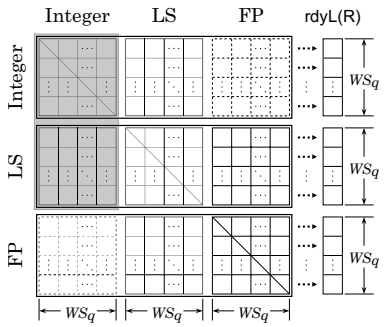

Figure 6. Decentralization of matrix.

they are more effective if applied together. Thus this section first describes the decentralization of the matrices, then the next section describes the way to apply the technique to the decentralized matrices.

When the instruction window is decentralized to s subwindows of which the issue widths and the sizes are IW' and WS', each matrix is segmented to s WS'-row WS-column partial matrix. For example, R10000 has subwindows for the integer, LS, and FP instructions, and WS' of each subwindow is 16. As shown in Figure 6, each matrix is segmented to three 16-row 48-column partial matrices.

Nine $WS' \times WS'$ minor matrices in Figure 6 correspond to the dependence between each two of the integer, LS, and FP subwindows. The right top and left bottom minor matrices are unnecessary for architectures which do not have move instructions between the integer and FP registers like SPARC. Even for architectures which have such instructions like MIPS, these minor matrices can also be omitted by stalling the frontend until the instructions finish. In this case, the rdyL/Rs of the consumer instructions are initialized to 1 in the dispatch phase (see Section 2.1). Though any of the minor matrices can also be omitted in this way, it is not recommendable because the impact on the IPC would possibly become too large.

Like the conventional scheme the decentralization of the matrices reduces the effective size of the matrices and enables latency optimization.

Effective size reduction The number of the write ports of each segment is reduced from IW to IW'. Thus the circuit area is reduced, and the delays are shortened.

Latency optimization As mentioned before, only the integer-to-integer and integer-to-LS paths have one-cycle latency in the case of R10000, and the wakeup and select operations for them have to be performed in one cycle. Thus the read operation for the shadowed segments in Figure 6 has to be performed in one cycle with the select, while the rest is pipelinable. We call the former segment **L-1 matrices**, while the latter **L-2 matrices**.

The read operation of L-2 matrices can consume two or more cycles with the select logic depending on the latency of the paths. If 0.5 cycles are assigned to each of the read operation of L-1 matrices and the select phase, the read of L-2 matrices can consume at least 1.5 cycles, which is three times longer than that of L-1 matrices. Therefore L-2 matrices is unlikely to be critical, and can be excluded from the discussion on the wakeup delay.

On the other hand, L-1 matrices is considerably smaller than the original matrices, and the delay becomes shorter. In addition, the technique described in the next subsection can be applied for L-1 matrices.

3.7. Narrowing

The distances between two dependent instructions are generally short. It is known that about 90% of them are less than 32 instructions [5]. The effective size of the L-1 matrices can be reduced utilizing this characteristics. Only the bits for the preceding $\boldsymbol{w}$ $(1 \leq w \leq WS' - 1)$ instructions are remained in the L-1 matrices, and the other $(WS' - w)$ bits can be moved to the L-2.

The rdyL/R are updated by the reduced L-1 matrices when the latency between two dependent instructions is one and the distance between them is less than or equal to w, otherwise the rdyL/R are updated by the L-2.

When the latency is one and the distance is greater than w, a penalty of one cycle is charged. The update of the rdyL/R are performed by L-2 matrices instead of L-1, and is delayed by one cycle. Consequently I_c can not be issued in the consecutive cycle of I_p. The impact, however, is enough small, as shown in the next section.

Figure 7 shows the way to reduce the minor matrix of $WS' = 8$ to $w = 4$. Though the figure shows only the integer-to-integer minor matrix, it can be applied to the integer-to-LS. The left of the figure represents the L-1 matrix before reducing, and the right represents after. The cells moved from L-1 matrices is drawn in gray lines in the left. There are several ways to collect remained cells into a rectangular area, and the figure shows one to give priority to reduce the length of the bitlines $\overline{\text{rdyL/R}}$ which are more critical. Now we can call w the **width** of L-1 matrices.

As shown in Figure 7, each of the wordlines issue and bitline $\overline{\text{rdyL/R}}$ of the narrowed L-1 matrix are connected to only w cells. Therefore the delay to read the L-1 matrices, that is, the wakeup delay is given by a function of w independent of WS or WS'.

Pace keeping operation The narrowed L-1 matrices can update rdyL/Rs when the distance between dependent instructions not in the instruction flow but in the window is less than or equal to w. Therefore the distance in the window has to be consistent with that of the instruction flow to a certain extent.

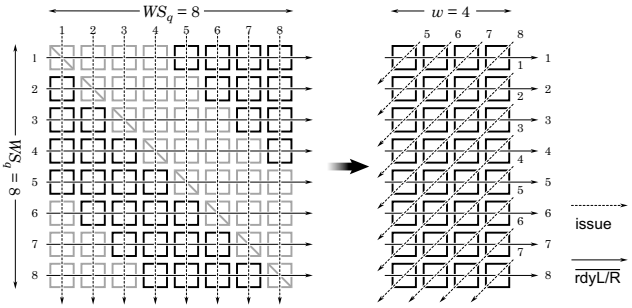

Figure 7. Narrowing L-1 matrix.

For instructions dispatched to the same subwindow, this condition is easily satisfied by reusing the subwindow entries cyclically. For instructions dispatched to different subwindows, however, extra control is required. The subwindows concerned with the narrowing, that is, the integer and LS subwindows in the case of R10000, have to *keep pace* with each other; otherwise the tails of the integer and LS subwindows are generally apart and dependent instructions can be stored into distant entries.

The simplest algorithm to keep pace is as follows: when two consecutive instructions are dispatched to the different subwindows, the succeeding instruction should be written into an entry *on the side of or after* the entry where the preceding instruction is written.

The operation itself is simple, and it would not impact the delay of the decode stage. Though the utilization of the entries of the subwindows is inevitably degraded, it is small as shown in the next section.

4. Evaluation of IPC

The rest of this paper gives quantitative evaluation results of our scheme compared with the conventional one. The evaluation items are the IPC penalties caused by the narrowing described in the previous section, and the areas and the delays of the circuits. The former is shown in this section, and the latter is in the next.

To evaluate the IPC, we modified the sim-outorder simulator in the SimpleScalar toolset [1].

We used MIPS R10000 as the base model with minor modifications. R10000 has three instruction subwindows for the integer, LS, FP instructions. The parameters are $(IW,' WS', TW, IW) = (2, 16, 6, 4)$, where IW' and WS' are the issue width and size of each subwindow, and TW is the width of tags (See Section 2.6). The size and the line size are 32KB and 32B for the first instruction/data cache, 1MB and 64B for the unified second cache. The latency of the first data cache is 1 cycle, the second is 6 cycles. On the second cache miss, the latency for the critical word is 18 cycles, and another 2 cycles a word is required for the oth-

Program	Input Set	No. of Insts
099.go	9 9	132M
124.m88ksim	dcrand.big	120M
126.gcc	genrecog.i	122M
129.compress	10000 q 2131	35M
130.li	train.lsp	183M
132.ijpeg	vigo.ppm -GO	26M
134.perl	primes.in	10M
147.vortex	persons.250	157M

Table 1. Programs in SPEC CINT95.

ers. We used a branch predictor based on two-bit saturating counters provided in the toolset (bimod).

We call this model **R10K$\times$1**. In addition, we evaluated two virtual models, namely **R10K$\times$2**, and **R10K$\times\infty$**. R10K$\times$2 has double computation resources except the caches. Thus the parameters are $(IW', WS', TW, IW) = (4, 32, 7, 8)$. R10K$\times\infty$ has infinite resources, and perfect caches and a perfect branch predictor. R10K$\times$1 and $\times$2 are also used for evaluation of the circuits in the next section.

We executed eight programs of SPEC CINT95 shown in Table 1. We don't show the result of CFP95. The critical paths of the programs in CFP95 is mainly composed of the

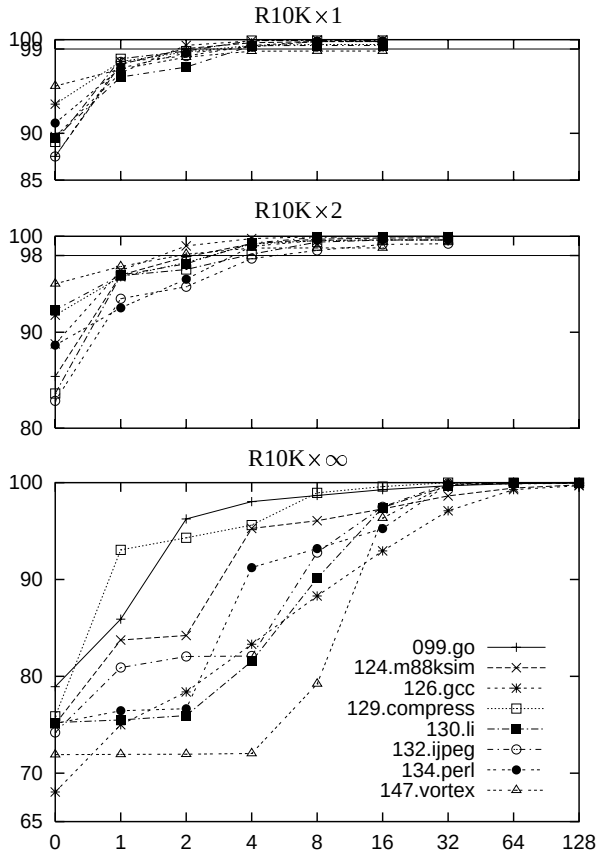

Figure 8. IPC ratios vs. L-1 matrices widths.

FP instructions of two or more cycle latencies. Thus the narrowing has little impact on them.

The graphs in Figure 8 correspond to R10K$\times$1, $\times$2, and $\times\infty$ respectively. The x-axes are the widths of L-1 matrices, and y-axes are the percentages of IPC to the conventional scheme.

Note that the values of $w = WS'$ are slightly less than 100% because of the pace keeping operation. Though the operation is unnecessary when $w = WS'$, we applied it to evaluate the net influence of the operation.

The graphs for R10K$\times$1 and $\times$2 show the quarter widths are enough to be comparable with conventional scheme. In these cases, the IPC degradations are caused mainly by the pace keeping operation, and are less than 1% for R10K$\times$1, 2% for $\times$2.

The cases of $w = 0$ are equivalent to that each of the wakeup and select phases are assigned one cycle described in Section 2.5. The IPC degradations reach about 15%.

The graph for R10K$\times\infty$ indicates that there exists the upper limit in the width. The L-1 matrices of widths 32 to 64 are enough even for an ideal processor with infinite resources.

5. Evaluation of circuits

We used the CS80A design rule provided by Fujitsu Limited. CS80A is a bulk CMOS process of 0.18μm gate width. The gate insulator and the inter-metal dielectric are SiO_2. The metal is six-layer aluminum, but we used only three.

Guided by the rule, we actually designed the layouts of the chief circuits of the integer subwindow of the conventional and our schemes, namely the RAM and the CAM of the conventional scheme, the matrices, as well as the select logic:

RAM The upper half of Figure 9 shows the logic circuit of the cell of the tagD RAM. The RAM is IW'-read IW'-write, TW b$\times WS'$ word. The figure is for $IW' = 2$, thus the cell has two read and two write ports.

CAM The lower half of Figure 9 shows the logic circuit of the CAM cell of the conventional scheme. The cell in the figure is for a bit of tagL. The cell is composed of an array of comparators shown in the top of the cell, and the RAM cell to save a bit of tagL shown in the bottom.

The RAM cell is logically the same as the tagD RAM cell described above.

We provide IW comparators for generality (see Section 2.6). The figure is for $IW = 4$, $IW' = 2$, thus two write ports and four comparators are provided.

We use a dynamic selector composed of a pair of nMOS pass gates for the one-bit comparator. One of a paired pass gates is ON depending on the state of the RAM cell. If the

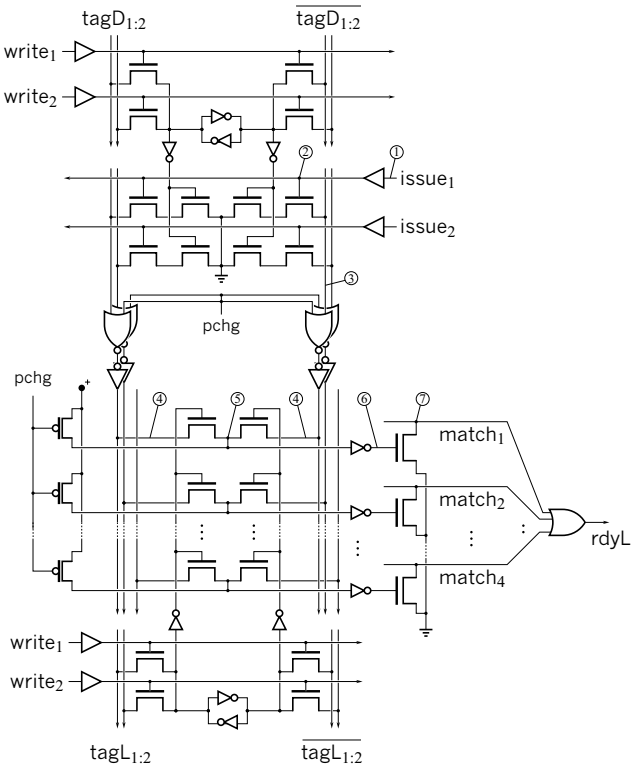

Figure 9. Conventional RAM and CAM cell.

input and the memorized value is different, one of the differential input ④ connected to the ON gate becomes low, and it discharges the node ⑤. If any one of TW one-bit comparators connected to a match line ⑦ detects the mismatch, it pulls down the precharged match line. An OR gate of IW match lines generates the final output for rdyL.

Matrices The logic circuit of the matrices has already shown in Figure 5. The matrix cell is basically the same as the tagD RAM cell described above with the following exceptions:

- The wordlines of narrowed L-1 matrices run diagonally (see Section 3.7). In the layout, they run in the *stair* shapes.
- The length of the word and bitlines are decided by the widths of L-1 matrices w, independent of WS' or TW.
- The number of the read port is always one, independent of IW'.

Select logic We tried the **prefix-sum** scheme for the select logic. A prefix sum is the sum of the requests of higher priorities. We used fixed priority for simplicity. A cyclic-priority scheme for cyclic reuse of the instruction window entries is described in [6].

component			R10K×1	R10K×2
RAM			22,053	78,819
CAM			35,529	214,560
Matrices	L-1	2	4,673	11,502
		4	9,354	23,003
		8	18,688	46,006
		16	37,376	92,013
		32	—	184,026
	L-1 + L-2		168,372	828,108

Table 2. Circuit areas (μm^2).

We measured the area of these circuits, extracted the RC data from the layouts based on the process parameters, and calculated the delays by the Hspice simulation.

5.1. Area

Table 2 shows the areas of circuits. As mentioned before, our scheme also requires the same tagD RAM as the conventional scheme to use tagDs as the write addresses of the out-of-order buffer. Thus the areas of the matrices could be compared with that of the CAMs.

The L-1 matrices are as large as the CAMs, and they become considerably smaller with the narrowing.

On the other hand, the L-2 matrices are about four times as large as the CAMs, because the areas of them are proportional to the square of WS. If larger WS causes problems, we can choose the conventional CAM instead of the L-2 matrices.

The overall areas, however, are less than 1mm^2 even for R10K×2, and it is negligible compared with the die area.

5.2. Delay

Figure 10 shows the results of the Hspice simulation. In this graph, the top six bars are for the delays of L-1 matrix, the next two for the conventional RAM + CAM delays, and the last two for the select logic. The marks ① to ⑦ in the graph correspond to those in Figure 9. The measurement condition is typical: the supply voltage is 1.8V, and the temperature is 85°C.

Without the narrowing, the widths of L-1 matrices for R10K×1 and ×2 are 16 and 32. Thus the bottom two bars for L-1 matrices can be compared with two bars for the conventional wakeup logic and two bars for the select logic respectively. When the widths of L-1 matrices are reduced to w, the bar for widths w can be compared.

Delay of L-1 matrices The delays of the L-1 matrices are comparable with those of the tagD RAMs. The tagD RAM is TW b × WS words, while the L-1 matrix without the narrowing is almost equivalent to WS b × WS words. Thus the delays of the L-1 matrices without the narrowing are slightly longer than those of the tagD RAMs because of

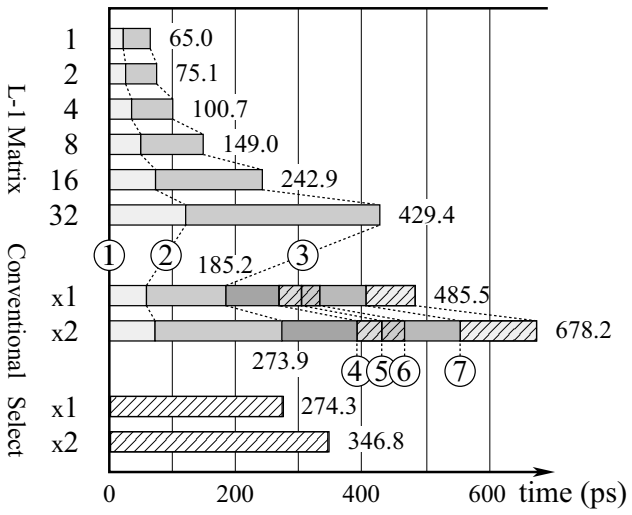

Figure 10. Delay of circuits.

the difference in bit widths, in other words, the loads of the wordlines.

The matrices are optimized for the width of 4 to 8, and there is relatively large room to optimize for the width of 16 to 32.

Issuing delay For R10K×1, our scheme reduces the wakeup+select delay to 68.0% of the conventional one, and the clock frequency limited by the delay is 1.93GHz. Thus the issuing delay would not limit the clock frequency, and the narrowing would be unnecessary.

For R10K×2, our scheme reduces the wakeup+select delay to 75.7% of the conventional one, and the clock frequency limit is 1.29GHz. When the widths of L-1 matrices are reduced to eight, the clock frequency limit reaches 2.02GHz, and the issuing delay would not be critical. The IPC degradation of the configuration is about 2% (see Section 4).

Wire delays The hatched segments of the bars are composed mainly of the intrinsic gate delays, while the others are subject to the wire delays.

The graph shows that the proportion of the latter is considerably high in the conventional wakeup logic. It indicates that the delay of it is hard to be scaled.

On the other hand, though the bars for the matrix are not hatched, the wire delays does not directly influence the delay of the matrices. As the width of the matrices are reduced, the word and bitlines are considerably shortened, and the influence of the wire delays are decreased. The influence is nearly zero for the width of one to two.

6. Related work

Dependence-based window Palacharla et al. proposed the dependence-based window [9]. This scheme decentralize the window into small FIFO queues each of which corresponds to a cluster of the execution units. Since dependent instructions are dispatched into the same queue, the target of wakeup for the one-cycle latency path can be restricted to the top of each queue. In addition, the scheme removes the select logic.

The potential problem of this method exists in the distribution of instructions among FIFO queues. The instructions are distributed based on heuristics, and the IPC is subject to its preciseness. Although several heuritstics are proposed [9, 4, 3], their IPC are relatively low. Complex heuristics could prolong the delay of the distribution stage and degrade the total performance.

The clustering of the execution units reduces the delay of the operand bypass logic, which is another main factor to limit the clock speed, is also indispnsable for future superscalar processors. The execution unit clustering, however, does not directly mean such clustring of the issuing window. The execution unit clustering could be combined with our scheme.

First-use and distance schemes Canal and González proposed the first-use and distance schemes [2]. And S. Weiss and J. E. Smith proposed a similar issue logic to the first-use scheme [11]. These schemes restrict the targets of the associative search required for the conventional wakeup operation at the cost of IPC penalties. The first-use scheme is based on the fact that most results are read once, while the distance scheme that the time when a result becomes ready can be determined from the latencies of the execution units and the starting time.

These schemes can also be combined with the clustering of the exececution units.

The IPC degradation of these schemes, however, are relatively high. The degradation is about -20% without any association, and an associative buffer of 16 entries is required for the IPC almost equal to the conventional scheme.

Though the results can not be directly compared, it could provide useful information that the matrices are faster than the conventional wakeup logic of 16 entries even without the narrowing, that is, for no IPC degradation.

CSP circuits Henry et al. widely applied a circuit to generate a cyclic segmented prefix (CSP) to the several components of the scheduling logic including the wakeup logic [6]. Their wakeup circuit is compose of CSP circuits for each logical register and a pairs of NR-input selectors for each window entry, where NR is the number of logical registers. The CSP circuit for a register generates whether the newest instruction that writes the register has finished in

$O(\log WS)$ time. Then the pair of selectors of a window entry selects outputs of the CSP circuits corresponding to the source registers of the instruction in $O(\log NR)$ time.

The CSP circuit is useful for the logis other than wakeup, especially for the select logic. But the application to the wakeup logic is not reasonable. For the CSP circuits to find the newest instruction, the program order have to be kept in the issuing window. Although this is another way to utilize the program order, it prohibits the decentralization of the issuing window.

EDF window Sato and Arita proposed a similar scheme to ours [10]. Their scheme also removes the association form the wakeup operation. Their scheme detects the dependence between instructions in the frontend of a processor like out scheme, while it saves the result of the detection not in the form of the matrices but of pointers to the consumer instructions. Their wakeup logic is composed of a RAM which holds the pointers and decoders for the pointers. Thus the main effect of the scheme is that the comparator array of the CAM of the conventional wakeup logic is replaced by the decoders.

However, the restriction over the number of the pointers causes severe tradeoff between IPC and the hardware complexity. In particular, the status recovery from mispredictions is complex.

The difference between their and our schemes comes from that the write to the matrices is performed to the row while the read is to the column. Owing to this feature, the row of the matrices as the entry of the issuing window can hold not a limited number of pointers to the consumers but the bit-vector to the only producer.

7. Conclusions

This paper describes a new dynamic instruction scheduling scheme for superscalar processors, not based on the association of the tags but on matrices which represent the dependence between instructions. Since the update logic of the matrices detects the dependences as the register renaming logic does in the frontend of a processor, the wakeup operation is realized by just reading the matrices.

We designed the layouts of the logics guided by the design rules of a CMOS process with 0.18μm gate width provided by Fujitsu Limited, and evaluated the delays by Hspice simulation.

Our scheme reduces the issuing delay of a processor of the MIPS R10000 configuration to 68.0% of that of the conventional one, and achives the clock frequency limit of 1.93GHz. It is unlikely that the scheduling logics form the critical path.

In addition, this paper also describes a technique called narrowing. The narrowing reduces the effective size of the matrices for small IPC penalties. The simulation results show that the IPC degradation is only 1 to 2% with the matrices of the quarter widths. The narrowing achieves the clock frequency limit of 2.0GHz for a virtual 12-issue processor.

Acknowledgment

I wold like to thank Fujitsu Limited for providing the design rules of the CMOS process.

This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research (B)(2) #12480072, #12558027, and #13480083.

References

[1] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar toolset. Technical Report CS-TR-1308, Univ. of Wisconsin-Madison, Jul 1996.

[2] R. Canal and A. González. A low-complexity issue logic. In *Proc. 14th Int'l Conf. on Supercomputing*, pages 327–335, 2000.

[3] R. Canal, J. M. Parcerisa, and A. González. Dynamic cluster assignment mechanisms. In *Proc. 6th Int'l Symp. on High-Performance Computer Architecture (HPCA6)*, 2000.

[4] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster architecture: reducing cycle time through partitioning. In *Proc. 30th Int'l Symp. on Microarchitecture*, 1997.

[5] M. Goshima, H. Nguyen, S. Mori, and S. Tomita. DualFlow: A hybrid processor architecture between control- and data-driven. *IPSJ SIG Notes 98–ARC–130 (SWoPP '98)*, pages 115–120, Aug 1998. (in Japanese).

[6] D. S. Henry, B. C. Kuszmaul, G. H. Loh, and R. Sami. Circuits for wide-window superscalar processors. In *Proc. 27th Int'l Symp. on Computer Architecture (ISCA27)*, 2000.

[7] J. Keller. The 21264: A superscalar alpha processor with out-of-order execution. In *Proc. 9th Annual Microprocessor Forum*, Oct 1996.

[8] S. Palacharla, N. P. Jouppi, and J. E. Smith. Quantifying the complexity of superscalar processors. Technical report, Univ. of Wisconsin-Madison, Nov 1996.

[9] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. 24th Int'l Symp. on Computer Architecture (ISCA24)*, Jun 1997.

[10] T. Sato and I. Arita. Simplifying wakeup logic in superrscalar processors. In *Joint Symp. on Parallel Processing 2001*, pages 23–30, Jun 2001. (in Japanese).

[11] S. Weiss and J. E. Smith. Instruction issue logic in pipelined supercomputers. *IEEE Trans. Comput.*, C-33(11):1013–1022, Nov 1984.

[12] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, (4):28–40, Apr 1996.