

## Indirect Tag Search Mechanism for Instruction Window Energy Reduction

Watanabe, Shingo  
Kyushu Institute of Technology

Chiyonobu, Akihiro  
Fujitsu Laboratories

Sato, Toshinori  
System LSI Research Center, Kyushu University

<https://hdl.handle.net/2324/8314>

---

出版情報 : 7th International Conference on Computer and Information Technology, pp.841-846, 2007-10-19

バージョン :

権利関係 :



# Indirect Tag Search Mechanism for Instruction Window Energy Reduction

Shingo Watanabe  
Kyushu Institute of Technology  
s-watanabe@klab.ai.kyutech.ac.jp

Akihiro Chiyonobu  
Fujitsu Laboratories  
chiyonobu@jp.fujitsu.com

Toshinori Sato  
Kyushu University  
toshinori.sato@computer.org

## Abstract

*Instruction window is a key component which extracts Instruction Level Parallelism (ILP) in modern out-of-order microprocessors. In order to exploit ILP for improving processor performance, instruction window size should be increased. However, it is difficult to increase the size, since instruction window is implemented by CAM whose power and delay are much large. This paper introduces a low power and scalable instruction window that replaces CAM with RAM. In this window, instructions are explicitly woken up. Evaluation results show that the proposed instruction window decreases performance by only 1.9% on average. Furthermore, dynamic energy is reduced by 67% on average and static power is reduced by 14%.*

## 1. Introduction

Modern microprocessors consume large power. The large power consumption implies that the system has to tolerate high current peaks and to deliver high and fast changing amounts of current. It also results in much heat generation, which requires efficient cooling. These factors lead to the increase in the system costs. On the other hand, smaller devices increases power density. The temperature of high activity circuit area is higher than other areas. The higher temperature might break the processor itself.

Until recently, microprocessors have obtained high performance by exploiting Instruction Level Parallelism (ILP). ILP is extracted in instruction window, which is a complex circuit. Since it has become a major power hungry component, it is difficult to increase the window size. Considering the above situations, the current trend is toward multicore processors, which attain high performance by utilizing Thread Level Parallelism (TLP). There is also a trend revisiting in-order execution instead of out-of-order execution. However, Amdahl's law tells us that the execution time of a parallelized program is dominated by a serial execution in the program, where a plenty of parallelism can not be found. In other words, much TLP can not be found there. Thus, both TLP and ILP are required to improve processor performance.

Instruction window is a key component which extracts ILP and is generally implemented using CAM logic. Unfortunately, CAM consumes large power. Therefore, large but still low power instruction window is required. This paper proposes such a scalable low power instruction window, where CAM is replaced by RAM. Evaluation results show that the proposed instruction window diminishes performance by only 1.9% on average, while its dynamic energy consumption is reduced by 67% on average and its static power is reduced by 14%.

The remainder of this paper is organized as follows: Section 2 describes the proposed instruction window. Section

3 explains the evaluation environment. Section 4 presents evaluation results. Section 5 summarizes related works. Lastly Section 6 concludes.

## 2. Indirect Tag Search instruction window

This section describes the proposed instruction window, which is coined Indirect Tag Search (ITS) instruction window. In the ITS instruction window, every issued instruction searches its dependent instructions through the wakeup list, which keeps data dependence information.

### 2.1. Wakeup list

The wakeup list is a linear list which connects every instruction with its dependent instructions. In the traditional CAM-based instruction window, when every instruction is finished, its destination tag is broadcasted against all instructions in the window. Each instruction in the instruction window entry compares the broadcasted tag with its source tags. If there is a match, the dependent instruction is ready to execute. The aim of the broadcast is to build data dependence information between instructions only with their register tags. In contrast, the proposed scheme builds the data dependence information in a different manner. All instructions which are dependent upon an instruction are connected by the wakeup list, as shown in Figure 1(a). When the producer instruction is finished, all instructions in the list are explicitly woken up one by one.

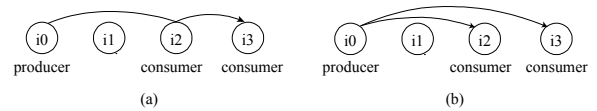
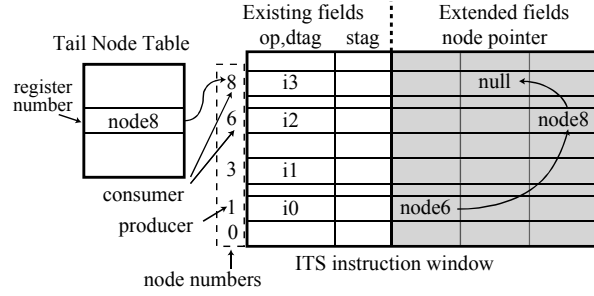


Figure 1: (a) Wakeup list, (b) Data flow graph

Although the data flow graph shown in Figure 1(b) also describes the same data dependence explained in Figure 1(a), keeping the graph in the instruction window requires high hardware cost, because each instruction might have many consumer instructions, each of which requires a pointer connecting it with the consumer instruction. Considering above, the data flow graph explained in Figure 1(a) is kept in the ITS instruction window instead of the one explained in Figure 1(b).

### 2.2. ITS instruction window

Figure 2 shows the ITS instruction window. Each entry in the ITS instruction window consists of the conventional fields that save op-code and tags, and the additional fields that save the wakeup lists explained in the previous section. The additional fields maintain node pointers that construct the list as shown in Figure 2. Every node and every node pointer present a node and an arc in the data flow graph, respectively.



**Figure 2: ITS instruction window**

For every instruction dispatched into the ITS instruction window, its associated node is appended to the tail of its corresponding wakeup list in the dispatched order. In order to identify which entry has the tail node, tail node pointers are saved in a dedicated table which is called Tail Node Table (TNT). The TNT saves the tail node pointers of all wakeup lists, each of which corresponds to a producer instruction. The tail node pointer is associated with the producer's logical destination register number. Therefore, every dispatched instruction obtains its corresponding tail node pointers by referring the TNT using its logical source register numbers.

The TNT can be implemented by extending the map table, which is implemented in modern microprocessors for register renaming. When every instruction refers the map table, it obtains the tail node pointers for its source operands and then it writes its destination register number into the TNT field in the map table.

The wakeup lists are constructed when every instruction is dispatched into the ITS instruction window. First, an entry is allocated in the instruction window for every dispatched instruction, and a node pointer associated with the instruction is determined. Second, the tail node pointer of the instruction's every source operand is obtained from the TNT using the logical source register numbers. In other words, pairs of producer and consumer instructions are found. The dispatched instruction is the consumer. Third, the consumer instruction's node pointer is written to the extended fields in the ITS instruction window entries, which keep the producer instructions. Now, the dispatched instruction is appended to its corresponding wakeup list. Last, since the tail node is changed by appending the dispatched instruction, the TNT is updated as described above. Note that third and fourth steps can be done in parallel.

Since most instructions have two operands, an instruction can belong to two wakeup lists. In addition, most instructions behave as a producer as well as a consumer, every ITS instruction window entry requires three fields to save the pointers.

### 2.3. Branch missprediction handling

When a branch missprediction occurs, a special missprediction handling is required since the wakeup list might be corrupted. In modern microprocessors, every branch missprediction flushes pipeline, and thus the wrong path instructions in the instruction window are squashed. In the case of the ITS instruction window, this squash breaks the wakeup lists. The TNT is also corrupted since some tail pointers kept in the TNT are removed from the ITS instruction

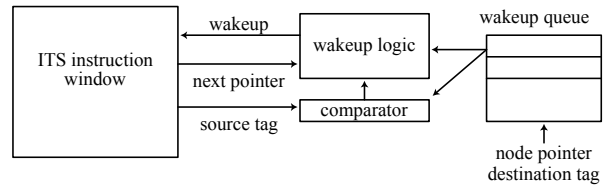
window. Using Figure 2, a troublesome example is shown. In this figure, instructions i0 and i3 are the oldest and the youngest, respectively, and instructions i2 and i3 are dependent upon instruction i0. Here, it is assumed that i1 is a branch instruction and misspredicted. Instructions i2 and i3 are squashed, and hence the node 8 is no longer a tail pointer. In addition, the node 1 might link instructions independent of instruction i0. Such a node that links independent instructions is called a violation node.

In order to restore the TNT, checkpointing is utilized. Since the TNT is a part of the map table as explained above, the checkpointing is easy to implement. The TNT is replicated whenever branch instructions appear. When a branch instruction is misspredicted, the TNT is restored using the checkpoint. This solves the corruption problem in the TNT.

However, unfortunately, checkpointing can not be used to restore the wakeup lists kept in the ITS instruction window. Since instruction windows are generally very costly in hardware budget, it is not easy to make checkpoints of the ITS instruction window. An alternative solution is to restore individual violation nodes in the window. This searches for violation nodes in all entries, and is very time consuming in the case of sequential search or requires large hardware cost in the case of parallel search. Hence, unfortunately, it can not be used. The adopted solution is to keep the wakeup lists corrupted. Instead of restoring the wakeup list, its validity is checked every wakeup. When every instruction is woken up, it is checked whether the instruction is really woken up or not. The source tag of a consumer instruction must be identical to the destination tag of its producer instruction. Hence, the tags are compared every wakeup, and only when they match, the consumer is woken up. This does not require tag broadcast.

### 2.4. Wakeup mechanism

Figure 3 shows the wakeup mechanism of the ITS instruction window. It consists of the following components; a wakeup queue, a comparator, and a wakeup logic. The wakeup queue saves the node pointers and their associated destination tags of issued instructions. The comparator is used to detect violation nodes as explained above. The wakeup logic controls the access to the instruction window.



**Figure 3: Wakeup mechanism**

When a producer instruction is issued, its node pointers and their corresponding destination tags are inserted into the wakeup queue. After that, wakeup is processed outside the pipeline. Wakeup is handled as follows. First, the wakeup logic and the comparator obtain a node pointer and its associated destination tag, respectively, from the wakeup queue. Second, the wakeup logic accesses the ITS window entry indexed by the node pointer. Third, the wakeup logic and the comparator obtain node pointers to consumer instructions and their associated source tags, respectively,

from the accessed entry. The source tags are compared with the destination tag. Forth, only for instructions whose source tag is identical to the destination tag, the source operand is marked as ready. When all operands are ready, the consumer instruction is woken up. This process continues until all or the predefined number of node pointers is processed.

Note that the wakeup mechanism shown in Figure 3 is necessary for every wakeup list. In order to process multiple wakeup lists simultaneously, multiple wakeup mechanisms are required.

### 3. Evaluation methodology

This section describes the evaluation methodology.

#### 3.1. Processor simulator

SimpleScalar tool set [2] is used to evaluate processor performance and to analyze how instruction windows are accessed. The ITS instruction window including its wakeup mechanisms and the TNT are implemented in the simulator. Table 1 shows the configurations of processor evaluated in this paper. The configurations of the ITS instruction window are as follows. There are four wakeup mechanisms, each of which wakes only one instruction up per cycle. Hence, at most four instructions are woken up per cycle. Note that the ITS instruction window is modeled in much detail, while the original SimpleScalar simulator utilizes the Register Update Unit for instruction scheduling, which combines the instruction window and the reorder buffer as one component.

**Table 1: Processor configurations**

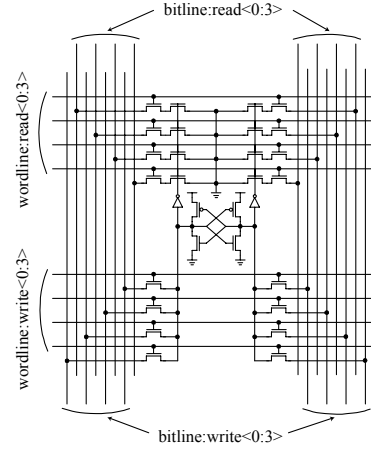
Instruction set	PISA
Issue width	4 instructions
Commit width	4 instructions
Branch prediction	Hybrid of bimodal and gshare
RUU size	128 entries
LSQ size	64 entries
L1 I cache	64 KB
L1 D cache	64 KB
L2 cache	2 MB

Spec2000 benchmark suit and MediaBench [9] are used for evaluation. For Spec2000 simulations, the ref input sets are used and 100 million committed instructions are executed after the forwarding the first 500 million instructions. For MediaBench simulations, the default input files provided from UCLA are used, and each program is executed until completion without any forwarding.

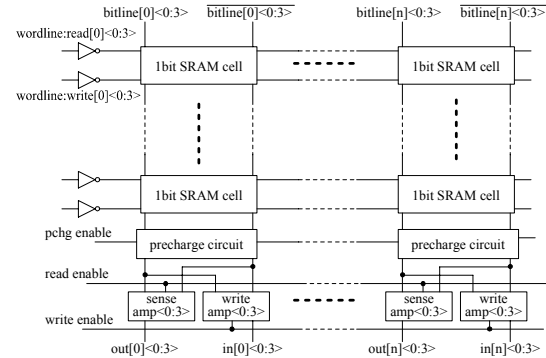
#### 3.2. Power estimation

HSPICE is used to estimate power consumed by the instruction windows. Transistor-level circuit designs on RAM and CAM are conducted based on the designs described in [5]. They are important elements in the instruction windows. Figure 4 shows a 1 bit SRAM cell designed in this study, and Figure 5 shows an SRAM array. The SRAM array and sense amplifiers are designed and evaluated in much detail, while the address decoder and the timing controller are out of considerations. The CAM logic is implemented by extending the RAM. It has four tag comparators in each line. The CAM requires the additional 4 ports for tag comparison. Figure 6 shows the CAM cell. The upper half is the cell and the lower

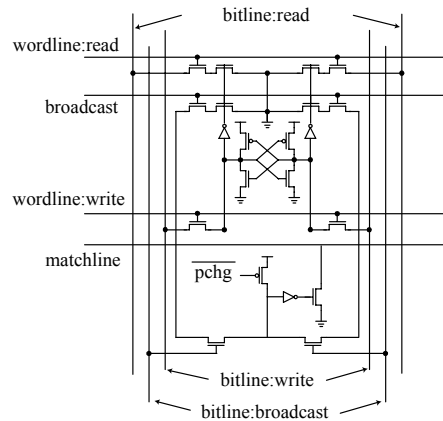
half is the tag comparator. While Figure 6 shows only one port per cell, the additional port connected to the next cell's comparators are required. Table 2 shows the configurations of the conventional CAM-based and the ITS instruction windows.



**Figure 4: SRAM cell**



**Figure 5: RAM array**



**Figure 6: CAM cell**

The power consumed by the CAM and the RAM is estimated by HSPICE for the 65nm technology. CMOS transistor and interconnect technology parameters used in the estimation are from Berkeley Predictive Technology Model [3,15]. Table 3 shows the detailed parameters used in this

study. Note that transistor sizes shown in the table are for those used in a cell.

Note that power consumed by the TNT is not considered in this estimation. The TNT has the same number of entries with the number of the logical registers, because it is attached to the map table. The size of each TNT entry is 9 bits; 7 bits for keeping the node pointer, 1 bit for indicating validity, and 1 bit for explaining which source operand is in the corresponding wakeup list. Moreover, multiple TNTs are necessary for checkpointing.

**Table 2: Instruction window configurations**

CAM-based instruction window	
op-code, destination tag	4 read and 4 write ports 23 bit width (op-code 16 bit, tag 7 bit)
source tag, (Implemented as CAM)	4 read and 4 write ports 4 broadcast ports 14 bit width (7 bit x2)
ITS instruction window	
op-code, destination tag	4 read and 4 write ports 23 bit width (op-code 16 bit, tag 7 bit)
source tag (Implemented as RAM)	8 read and 4 write ports 14 bit width (7 bit x2)
extended fields (node pointer)	4 read and 4 write ports 27 bit width (9 bit x3)

**Table 3: CMOS and interconnect parameters**

Process technology	65 nm
Supply voltage	1.1 V
Transistor size	nMOS L: 65 nm, W: 0.1 $\mu$ m pMOS L: 65 nm, W: 0.2 $\mu$ m
Interconnect width	0.1 $\mu$ m
Interconnect pitch	0.1 $\mu$ m

## 4. Results

This section discusses how performance is affected and how power consumption is reduced.

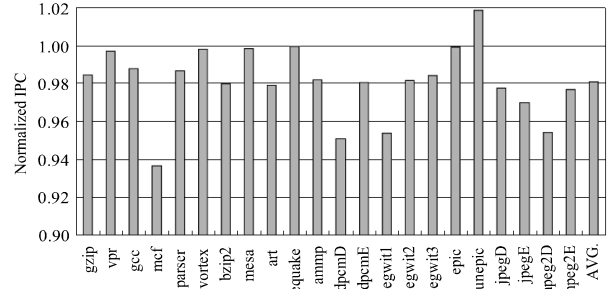
### 4.1. Processor performance

Instruction wakeup is generally delayed in the ITS instruction window in comparison with in the conventional CAM-based window. One of the reasons of the delay is that a wakeup mechanism in the ITS window corresponds with a wakeup list and thus wakes only one instruction up per cycle. Thus, for example, wakeup of instructions on the bottom of the wakeup list is late. The other reason is that the violation nodes waste wakeup bandwidth. Every violation node occupies one wakeup mechanism, which is not efficiently used since the wakeup is discarded after detecting a mismatch.

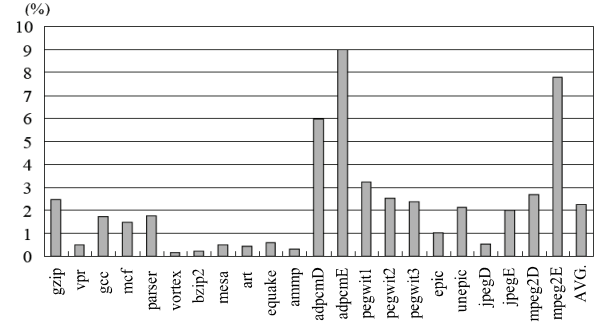
The wakeup delay degrades processor performance. Figure 7 presents performance of the processor utilizing the ITS window. Instructions per cycle (IPC) is used as a metric and the IPC result of each program is normalized by that of the processor utilizing the conventional CAM-based window. Y-axis shows the normalized performance and X-axis lists benchmark names. It is observed that the IPC loss is 1.9% on average. Even in the worst case of mcf, the IPC loss is only 6.3%. The results indicate that replacing the conventional

CAM-based instruction window with the ITS one does not have any serious impact on processor performance. It is well known that most producer instructions have only a few consumers [4,8]. Hence, the wakeup delay rarely occurs and thus does not have any severe impact on processor performance.

Figure 8 shows the percentage of wakeup lists that have violation nodes. It is only 2% on average. While adpcmD, adpcmE and mpegE have more violation nodes than other programs, their performance degradation is not considerably large as shown in Figure 7. The results indicate that the mechanism for handling branch misspredictions described above is a good choice in the point of the trade-off between hardware complexity and performance impact.



**Figure 7: Processor performance**



**Figure 8: Percentage of violating wakeup lists**

### 4.2. Power consumption

This section presents power consumption results. First, circuit-level power estimation is shown. Both static and dynamic power consumption is considered. Next, dynamic energy consumption considering program execution is shown.

#### 4.2.1. Circuit-level static power consumption

Figure 9 shows static power consumption of the instruction windows. In the figure, op-code, dtag, stag, and pointer mean power consumed by the operation code, destination tag, source tag, and node pointer fields in the instruction window entries, respectively. Note that the source tag fields in the conventional instruction window are implemented as CAM logic. The other fields both in the CAM-based window and in the ITS window are implemented as RAM.

An interesting result is that the source tags in the CAM-based window consume more static power than those in the ITS window. A reason is that the number of transistors in the

CAM is larger than that in the RAM since the CAM has four comparators in each entry.

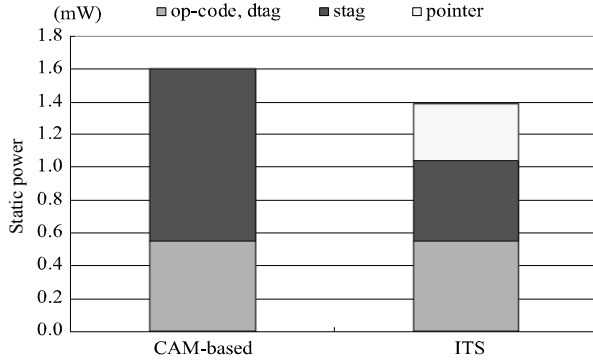


Figure 9: Static power of instruction window

While the node pointer fields in the ITS window consumes the additional power that is unnecessary in the CAM-based window, the total power consumption is 14% smaller in the ITS window than in the CAM-based window. This is due to the power consumed by the source tag field explained above.

#### 4.2.2. Circuit-level dynamic power consumption

The dynamic power consumption shown in Figure 10 is divided into four parts; dispatch, wakeup, issue, and append. Dynamic power is consumed when every instruction is dispatched, woken up, and issued. The additional power is consumed in the ITS window when the instruction is appended into a wakeup list constructed in the node pointer fields. Only ITS window consumes power for constructing the wakeup lists.

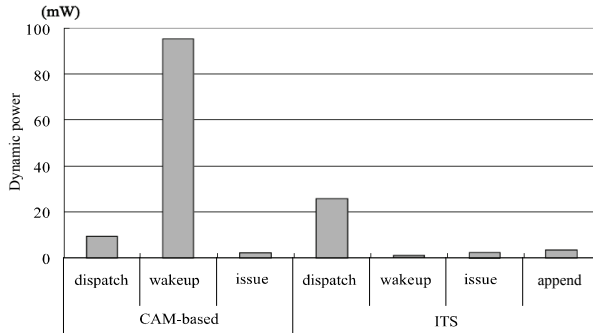


Figure 10: Dynamic power of access operations

First, it is observed that power consumption on dispatch in the ITS window is 2.7 times larger than in the CAM-based window. This is because a null pointer must be written to node pointer fields in the case of the ITS window. Second, in contrast, power consumption of wakeup is 99 times larger in the CAM-based window than in the ITS window. This is mainly due to the broadcast power. Third, there is not significant difference in the issue power between the CAM-based and the ITS windows. In total, every instruction consumes more power in the CAM-based window than in the ITS window through dispatch to issue.

#### 4.2.3. Energy consumption

Energy consumption is estimated by the instruction window activities observed during the SimpleScalar simulations. The numbers of operations, which are dispatch, wakeup, issue, and append, are counted. Energy consumption is calculated by multiplying the power shown in Figure 10 by the numbers. Figure 11 shows the energy breakdown of operations in the CAM-based window. The results show that the wakeup energy is major in the energy consumed by the CAM-based instruction window. The wakeup energy occupies 88% of the total energy on average.

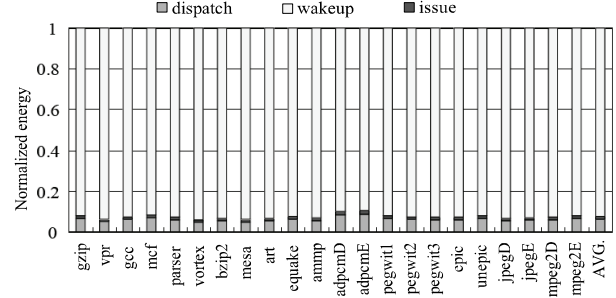


Figure 11: Energy of CAM-based window

Figure 12 shows the energy breakdown of operations in the ITS window. The total energy is normalized by that of the CAM-based window. In the results, it is observed that the wakeup energy is significantly decreased because the ITS window eliminates tag broadcasts and comparisons. Comparing Figure 12 with Figure 11, it can be seen that the dispatch energy is about two times increased. Only ITS window consumes the append energy, which occupies 10% of the total energy of the ITS instruction window on average. However, it is negligible when it is compared with the total energy of the CAM-based window. The total energy consumed by the ITS window is 67% smaller than that consumed by the CAM-based window on average. In the case of adpcmE, where energy reduction is smallest, 56% power reduction is achieved.

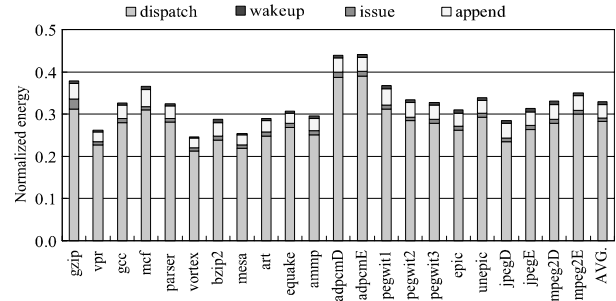


Figure 12: Energy of ITS window

## 5. Related works

There are a lot of studies that attempt to reduce power consumption of instruction windows. Abella et al. [1] provide a good survey of these techniques.

The works most related to the present paper are [8,10,12], which track dependencies among instructions and explicitly link producer and consumer instructions. In these mechanisms, a RAM structure replaces CAM logic with the use of a table to track dependencies. In other words, the associative broadcast is replaced with indexing to wake only one instruction up. They exploit the observation that most instructions have only one consumer [4,8], and a simplest implementation keeps only one consumer for every producer.

Onder et al. [10] propose source operand to source operand forwarding (SSF) to extend the links between a producer and its consumers. When every instruction is executed, it sends its source operands to their next use as well as its result. The differences between the SSF and the wakeup list proposed in the present paper are as follows. First, in the wakeup list, each consumer instruction explicitly makes the link on demand, while in the SSF, the relay instruction eagerly makes the links. Second, the ITS window completely eliminates CAM logic, while Onder's scheduler requires a small associative central structure called the Match Unit. Furthermore, Onder et al. do not consider power consumption.

Sato et al. [12] extend the table that track dependences, which they call the Dataflow Management Table (DMT), to keep up-to-three consumers for every producer. In order to deal with the case of more consumers, they introduce a scoreboarding mechanism, where instructions continuously monitor the register files for operand availability. The ITS window does not require such an expensive approach as scoreboarding. Sato et al. have not evaluated power efficiency, while they only mention that RAMs are lower in power dissipation than CAMs.

Huang et al. [8] introduce the Broadcast Bit in the instruction window to work together with a conventional CAM logic. Only when a producer instruction has more than one consumer, a tag broadcast is allowed to wake the consumers up. While it is limited in rare cases, associative search is still required. In contrast, the ITS window completely remove associative search. Huang et al. evaluate power efficiency based on the number of tag comparisons. In contrast, in the present paper, the detailed circuit design is conducted to estimate power consumption. Furthermore, leakage power is also considered in the present paper.

Researchers have been still interested in instruction windows design, especially in the low power domain [7,11,13, 14]. This is because embedded processors as well as general purpose processors utilize out-of-order execution, where the instruction window is the key structure. Hsiao et al. [7] propose two optimizations for power reduction. One is the selective match, which reduces the number of useless tag matches. The other is the wakeup range limitation, which splits tag broadcasting buses into several segments. Sasaki et al. [11] group multiple instructions as an atomic issue unit in order to reduce the required number of ports and the size of the instruction window. Sharkey et al. [13] propose Tag Memoization and Tagline Folding, both of which decrease the broadcasted tag bit width, resulting in power reduction in the wakeup tag broadcast. Vivekanandham et al. [14] propose the Scalable Low power Issue Queue (SLIQ), which combines the direct tag indexing [8,10,12] and the pipelined issue queue [6].

## 6. Conclusion

Modern processors have the conflicting requirements of large instruction window and its power reduction. This paper proposed a low power instruction window, which was named Indirect Tag Search (ITS) instruction window. Since CAM, which is required for broadcast and comparison operations, consumes large power, CAM is replaced by RAM that consumes lower power. The ITS instruction window constructs the wakeup lists, which is used to explicitly wake instructions up. Evaluation results show that the ITS instruction window decreases processor performance by only 1.9% on average. In contrast, dynamic energy consumption is reduced by 67% on average, and static power consumption is reduced by 14%.

## Acknowledgements

This work was partially supported by Grants-in-Aid for Scientific Research #16300019 and #176549 from Japan Society for the Promotion of Science, and is partially supported by the CREST program of Japan Science and Technology Agency.

## References

- [1] J. Abella et al., "Power- and complexity-aware issue queue designs," *IEEE Micro*, 23(5), 2003.
- [2] D. Burger et al., "The SimpleScalar tool set, version 2.0," *ACM Computer Architecture News*, 25(3), 1997.
- [3] Y. Cao et al., "New paradigm of predictive MOSFET and interconnect modeling for early circuit design," *Custom Integrated Circuits Conference*, 2000.
- [4] J.-L. Cruz et al., "Multiple-banked register file architectures," *Int. Symp. on Computer Architecture*, 2000.
- [5] M. Goshima, "Research on high-speed instruction scheduling logic for out-of-order ILP processors," Ph.D. Dissertation, Kyoto University, 2004 (in Japanese).
- [6] M. S. Hrishikesh et al., "The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays," *Int. Symp. on Computer Architecture*, 2002.
- [7] K.-S. Hsiao and C.-H. Chen, "Wake-up logic optimizations through selective match and wakeup range limitation," *IEEE Trans. VLSI Systems*, 14(10), 2006.
- [8] M. Huang et al., "Energy-efficient hybrid wakeup logic," *Int. Symp. on Low Power Electronics and Design*, 2002.
- [9] C. Lee et al., "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *Int. Symp. on Microarchitecture*, 1997.
- [10] S. Onder and R. Gupta, "Superscalar execution with dynamic data forwarding," *Int. Conf. on Parallel Architectures and Compilation Techniques*, 1998.
- [11] H. Sasaki et al., "Energy-efficient dynamic instruction scheduling logic through instruction grouping," *Int. Symp. on Low Power Electronics and Design*, 2006.
- [12] T. Sato, Y. Nakamura, and I. Arita, "Revisiting direct tag search algorithm on superscalar processors," *Workshop on Complexity Effective Design*, 2001.
- [13] J. J. Sharkey et al., "Power-efficient wakeup tag broadcast," *Int. Conf. on Computer Design*, 2005.
- [14] R. Vivekanandham, B. Amrutur, and R. Govindarajan, "A scalable low power issue queue for large instruction window processors," *Int. Conf. on Supercomputing*, 2006.
- [15] W. Zhao and Y. Cao, "New generation of Predictive Technology Model for sub-45nm design exploration," *Int. Symp. on Quality Electronic Design*, 2006.