LETTER

A Low-Cost Recovery Mechanism for Processors with Large Instruction Windows

In Pyo HONG[†a)], ***Student Member***, **Byung In MOON**[††], ***and*** **Yong Surk LEE**[†], ***Nonmembers***

SUMMARY The latest processors employ a large instruction window and longer pipelines to achieve higher performance. Although current branch predictors show high accuracy, the misprediction penalty is getting larger in proportion to the number of pipeline stages and pipeline width. This negative effect also happens in case of exceptions or interrupts. Therefore, it is important to recover processor state quickly and restart processing immediately. In this letter, we propose a low-cost recovery mechanism for processors with large instruction windows.

key words: *register renaming, recovery, checkpointing*

1. Introduction

In out-of-order superscalar processors, recovering processor state is done by restoring the rename map table contents to the in-order state of the fault-triggering instruction. To do this, conventional processors have separate in-order state mapping table at retire point. The processor waits for the faulty instruction to reach the retire point and then recovers register state by using the in-order state table. This technique wastes excessive cycles because the recovery process can be initiated only after the instruction reaches the retire point. Some checkpoint architectures that start recovery process immediately have been proposed in previous researches. One of them uses the reorder buffer (ROB) to store the register mapping of each instruction. If a branch misprediction or an exception is detected, the recovery process is initiated immediately and recovers register map table contents by walking each entry of the ROB in reverse order, canceling register mapping of wrong path instructions. Because this operation must be done in serial order, it takes several cycles to reconstruct register map table and renaming of new instructions is delayed [1]. Most enhanced techniques involve multicheckpointing, which uses multiple register map tables for recovery. When a checkpoint is generated, the processor saves current register map table as a checkpoint, and uses another free table for next cycle's renaming [2]. The most important factor in the multicheckpointing is the time when a checkpoint is taken. It is ideal to take checkpoints on every instruction as implemented in the Alpha 21264 [3], [4]. However, it induces excessive overhead and cannot be implemented on future processors with a large number of live instructions. Therefore, most recent work has focused on reducing the number of checkpoints by adjusting the conditions on which the processor creates checkpointing. The most frequent event that invokes register map table recovery is the branch misprediction. Instead of creating checkpoints to every instruction, MIPS R10K takes checkpoints only on branch instructions [5]. Although this branch-only checkpointing reduces overhead, larger instruction windows make this technique too big to be implemented. To mitigate the burden further, selective checkpointing mechanisms based on branch is proposed by Moshovos [6] and Akkary [7] respectively. In these techniques, a checkpoint is created only when a hard-to-predict branch instruction is decoded. In [8], Cristal employs a composite condition that invokes checkpointing. However, when a non-checkpointed branch instruction commit a fault, the recovery process must start from the latest checkpoint prior to the triggering instruction and reconstruct the latest register mapping by using information stored in the ROB in serial manner. That is, the selective checkpointing requires not only checkpoint recovery but also ROB-based recovery. Moreover, additional units, such as a branch confidence estimator, are needed in some cases. Because a recovery invoked by a non-checkpointed instruction takes several cycles, processors with selective checkpointing variations show lower performance than those with ideal checkpointing or branch-only checkpointing.

In this letter, we propose another register map recovery mechanism which is scalable for large instruction windows with low cost. In Sect. 2, the proposed architecture is presented. Section 3 presents estimated hardware cost and comparison to other mechanisms. The conclusion is given in Sect. 4.

2. Recovery Based on Instruction ID

Multicheckpointing saves register map tables for all registers although only a part of entries have valid renaming results that are modified during checkpoint interval. We paid our attention to it. Total number of in-flight and architectural registers is limited by the number of physical registers. If the processor records the order that physical registers are allocated, recovery logic can return the contents of the register map table to the state associated with any of fault-generating instructions. The ROB-based mechanism arranges renaming histories of individual instructions in program order, and recovers the state of an instruction by can-

Manuscript received October 6, 2005.

[†]The authors are with the Department of Electrical and Electronic Engineering, Yonsei University, Seoul, Korea.

[††]The author is with the School of Electrical Engineering & Computer Science, Kyungpook National University, Daegu, Korea.

a) E-mail: necross@dubiki.yonsei.ac.kr

DOI: 10.1093/ietisy/e89–d.6.1967

celing speculatively updated renaming results. This must be done in sequential order. This is the major reason why the ROB-based technique is replaced by multicheckpointing although it can be implemented with low cost. The principle of our proposal is the same with that of the ROB-based recovery. However, our proposal records the instruction order in another way and in another place, and can recover the map state instantly in a single cycle.

We assigned a unique number that represents instruction order to each fetched instruction. We call the number as *instruction ID*. Circular numbers generated by an increasing counter or index numbers of ROB entries that is assigned to the instructions can be used as an instruction ID. As you see in Fig. 1, the register map table has one entry per physical register and is implemented as a CAM. It has several information fields that represent the state of the physical register and the architectural register number associated with it. This map table architecture is similar to that of the Alpha 21264. In addition to that, we appended two fields that indicate the lifetime of the physical register by using the instruction ID. One is the creating instruction ID (CID) and the other is the removing instruction ID (RID). When an instruction with a valid destination register is renamed, an empty physical register is allocated to it and the CID field of corresponding map table entry is filled with its instruction ID. The RID is written when a succeeding instruction that updates the same architectural register is renamed. Figure 2 is an example which shows the renaming and recovery mechanism of proposed technique. All map table entries except those in in-order state and in recent state have valid CID and RID value. A physical register in the in-order state does not need the CID because it is clear that the physical register is the oldest mapped one for the corresponding architectural register. Similarly, the recent state means that the physical register is the last one allocated to the corresponding architectural register and has no RID value. In this example, the CID and RID fields are filled with the instruction IDs of creating and removing instructions. If the instruction 7 is mispredicted, the faulting instruction ID, 7, is transmitted to the register map table. Each entry compares the number to its CID and RID value. If both CID and RID are younger than the faulting instruction ID, as you see in P09, P25 and P26, it means that the physical register was allocated later than the faulting instruction and must be freed. Therefore, the empty bit is set. If both CID and RID are older than 7 as in P07 and P23, the physical register was allocated before the instruction 7 and the entry remains unchanged. Finally, if the CID is older and the RID is younger than the mispredicted instruction or the physical register is set to the recent state, as in P08 and P24, it means that the physical register was most recently mapped one to its corresponding architectural register when the instruction 7 was renamed. Therefore, the physical register is not freed and its recent bit is set.

As described above, comparing the lifetimes of each physical register with the instruction ID of a fault-generating instruction enables the recovery logic to determine whether the physical register is returned to empty state or not. At the same time, the recent bit can also re-established by the com-

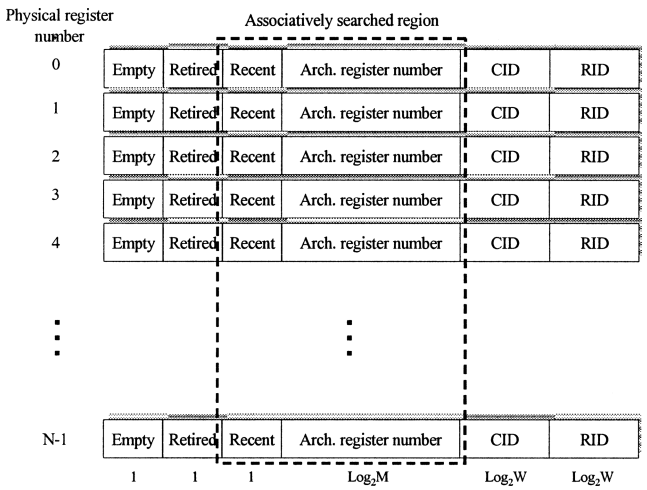

Fig. 1 Proposed rename map table (N is the number of physical registers, M is the number of architectural registers, and W is instruction window size).

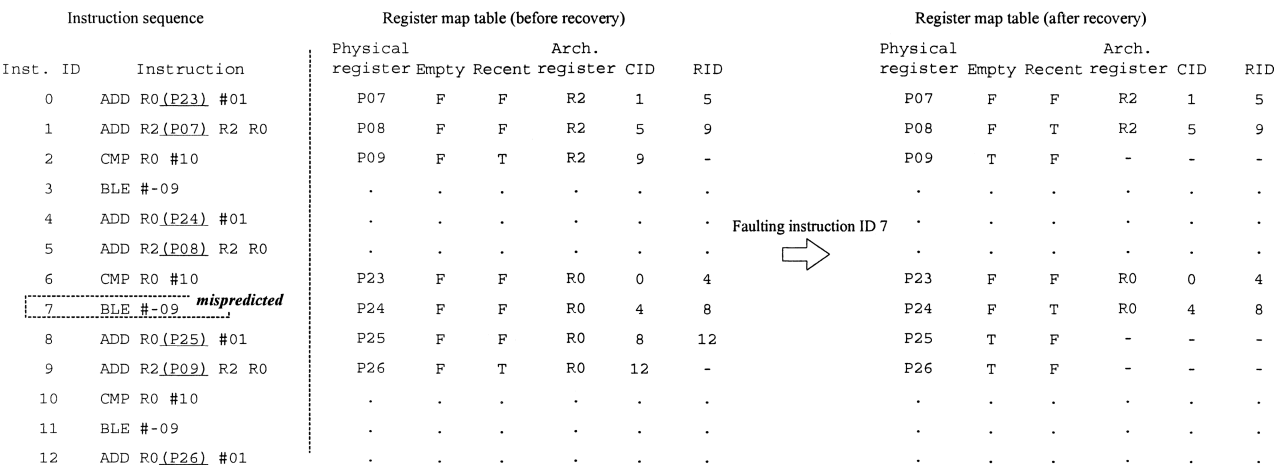

Fig. 2 Recovery example showing how the register map table is restored by using instruction ID.

parison result. If an instruction accesses the register map table for source operands, the CAM searches the map table and provides the physical register number with the recent bit set and with the architectural register number matched. There is no need to walk through the ROB canceling speculative renaming results in sequential order. Recovery can be performed inside each map table entry without centralized control in a single cycle.

3. Cost Estimation

Evaluating the performance by simulations is not an urgent work to prove the merits of the proposed mechanism. Because the instruction ID is granted to every live instruction, the recovery logic can restore the contents of the map table whatever instruction invokes the recovery process. Therefore, the performance is the same with a multicheckpointing that takes a checkpoint on every instruction. It does not induce any negative effect on architectural performance. Although previous selective checkpointing induces performance loss due to the non-checkpointed fault-triggering instructions, the proposed mechanism always shows the ideal architectural performance. Therefore, more important factor to be evaluated is the cost rather than the performance.

To estimate the cost, we calculated and compared the equivalent storage bits on the basis of a single SRAM cell. First of all, our proposal employs additional information fields, the CID and the RID. The width of the fields is determined by the number of live instructions. One bit must be added when the number of live instructions is doubled. Another overhead is computational logics. Our recovery mechanism uses some decision logics rather than employs huge storage blocks. The largest component is comparators that determine whether the physical register is older than the faulting instruction or not. Because the instruction IDs are circular numbers, some compensation is needed. It can be done easily by appending a single bit to the instruction IDs and the comparators to prevent errors due to roll-backed instruction IDs. After comparison, recovery logic must determine the next value of map table entry fields according to the results. The hardware needed to make a final decision can be implemented by using several transistors per a map table entry and is negligible. On the other side, the hardware devoted to the comparators is very large. A single bit adder used by the comparators occupy about 4-times larger area than a 6-transistor SRAM cell [9]. Therefore, we regard a single bit of the adder as four storage bits. The other overhead that must be considered is the ports of CAM memory block. When an architectural destination register is renamed, the map table must find the most recent entry that is associated with it and fill the RID field of the entry. It requires additional CAM read ports. Because the maximum number of destination registers is a half of the sum of source registers that access the CAM, the CAM in our proposal occupies 1.5-times larger area than the CAM of the multicheckpointing. The area of general CAM's is 3-times larger than that of the SRAM. Therefore, we counted

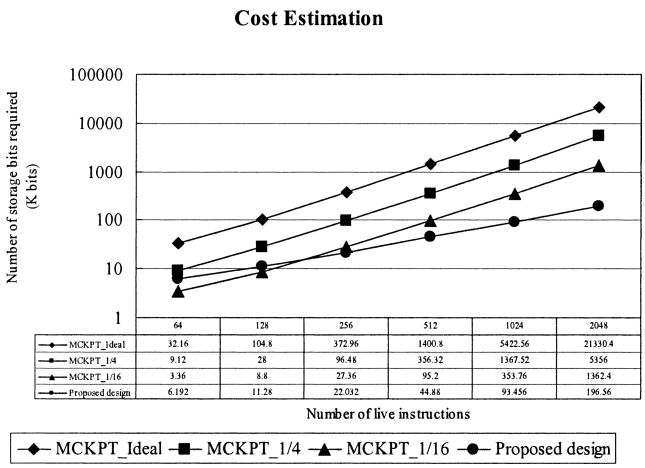

Fig. 3 Estimated cost of proposed checkpointing mechanism.

4.5 bits for a single bit of our proposed CAM [10].

It seems that our proposal induces a huge hardware overhead. However, the proposed recovery mechanism needs only a single copy of the register map table regardless of the number of checkpoints. In general, the number of physical registers is proportional to that of the live instructions. Therefore, the number of map table entries is increased as more live instructions is permitted. In Fig. 3, we calculated and compared the costs of the multicheckpointing and the proposed design. We assumed that the number of physical registers is the num of the number of live instructions and the number of architectural registers. MCKPT_Ideal creates a checkpoint on every instruction. MCKPT_1/4 and MCKPT_1/16 take a checkpoint periodically on every 4 instructions and on every 16 instructions respectively. Because the average length of basic blocks is about 4~6 instructions [11], MCKPT_1/4 can represent a cost of branch-only multicheckpointing. MCKPT_1/16 is presented to compare the cost of the proposed mechanism with the Moshovos' selective checkpointing that requires only 16 checkpoints when maximum 256 instructions can be alive in the processor. The cost only counts the number of storage bits for register map tables and excludes other extra-units such as branch confidence estimators and hardware for sequential recovery that support the selective checkpointing. Therefore, the merit of our proposal is under-estimated in the figure. The initial cost of the proposed recovery mechanism is not the smallest. However, the rate of increase is lower than that of the multicheckpointing variations. As a result, the cost of the proposed one is the minimum with a large difference when the number of live instructions is 128 or above.

4. Conclusion

In this letter, we proposed an enhanced register map table recovery mechanism. By storing the lifetimes of individual physical registers in register map tables, the proposed architecture can perform recovery without costly multiple

checkpoints. According to the cost estimation described above, it shows better results than previous multicheckpointing mechanisms, while maintaining the performance at an ideal level. Moreover, it can be used for the ROB-free architectures with huge instruction windows if another unit, such as an increasing counter, can provide instruction IDs. Therefore, our low cost fast recovery mechanism can be an alternative choice for future processors with hundreds or thousands of live instructions.

References

[1] M. Johnson, Superscalar Microprocessor Design, Prentice Hall, 1991.
[2] W.W. Hwu and Y.N. Patt, "Checkpoint repair for out-of-order execution machines," Proc. 14th Annual International Symposium on Computer Architectures, pp.18–26, 1987.
[3] D. Leibholz and R. Razdan, "The alpha 21264: A 500 MHz out-of-order execution microprocessor," Proc. 42nd IEEE International Computer Conference, pp.28–36, 1997.
[4] "Alpha 21264 microprocessor hardware reference manual," Compaq Computer Corporation, 1999.
[5] K.C. Yeager, "The MIPS R10000 superscalar microprocessor," IEEE Micro, vol.16, no.2, pp.28–41, April 1996.
[6] A. Moshovos, "Checkpointing alternatives for high performance, power-aware processors," Proc. International Symposium on Low Power Electronics and Design, pp.318–321, 2003.
[7] H. Akkary, R. Rajwar, and S.T. Srinivasan, "Checkpoint processing and recovery: An efficient, scalable alternatives to reorder buffers," IEEE Micro, vol.23, no.6, pp.11–19, Nov. 2003.
[8] A. Cristal, O.J. Santana, and M. Valero, "Toward kilo-instruction processors," ACM Transactions on Architecture and Code Optimization, vol.1, no.4, pp.389–417, Dec. 2004.
[9] C.J. Fang, C.H. Huang, J.S. Wang, and C.W. Yeh, "Fast and compact dynamic ripple carry adder design," Proc. 3rd IEEE Asia-Pacific Conference on ASIC, pp.25–28, 2002.
[10] M. Motomura, J. Toyoura, K. Hirata, H. Ooka, H. Yamada, and T. Enomoto, "A 1.2-million transistor, 33-MHz, 20-b dictionary search processor (DISP) ULSI with a 160-kb CAM," IEEE J. Solid-State Circuits, vol.25, no.5, pp.1158–1165, Oct. 1990.
[11] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-block ahead branch predictors," Proc. seventh international conference on Architectural support for programming languages and operating systems, pp.116–127, 1996.