복귀주소 스택을 활용한 얕은 파이프라인 EISC 아키텍처의 명령어 큐 효율성 향상연구

김한이[†] · 이승은[††] · 김관영[†††] · 서태원[††††]

요 약

EISC 프로세서에서 LERI 폴딩과 루프 버퍼링을 지원하는 명령어 큐는 하드웨어적으로 20%를 차지하며, 그 효율성은 성능에 직결된다. 본 연구에서는 EISC 프로세서의 명령어 큐 아키텍처 효율성 향상을 복귀주소 스택(RAS)을 통해 실현하였다. 구현한 아키텍처는 EISC의 얕은 파이프라인 구조의 이점을 활용하여 잘못된 명령어 수행으로 인한 RAS Corruption 문제를 제거하였다. 실험에서, 4개 엔트리의 RAS는 명령어 큐의 플러시를 기존보다 최대 58.90% 줄였고, 8개 엔트리의 RAS는 이를 최대 61.28% 줄였다. 또한 실험 결과 8개 엔트리의 RAS는 3.47%의 성능향상을 보여주었고, 4개 엔트리의 RAS는 3.15%의 성능향상을 보여주었다.

주제어 : EISC, 명령어 큐, 복귀주소 스택, 분기 예측기, 마이크로아키텍처

Enhancing Instruction Queue Efficiency with Return Address Stack in Shallow-Pipelined EISC Architecture

Han-Yee Kim[†] · SeungEun Lee[††] · Kwan-Young Kim[†††] · Taeweon Suh[††††]

ABSTRACT

In the EISC processor, the Instruction Queue (IQ) supporting LERI folding and loop buffering occupies roughly 20% of real estate, and its efficient utilization is a key for performance. This paper presents an architectural enhancement for the IQ utilization with return address stack (RAS) in the EISC processor. The proposed architecture eliminates the RAS corruption from the wrong-path, taking advantage of shallow pipeline. In experiments, a 4-entry RAS reduces the number of IQ flushes by up to 58.90% over baseline, and an 8-entry RAS by up to 61.28%. The experiments show up to 3.47% performance improvement with 8-entry RAS and up to 3.15% performance improvement with 4-entry RAS.

Keywords : EISC, Instruction Queue, Return Address Stack, Branch Predictor, Microarchitecture

---

† 정 회 원: 고려대학교 컴퓨터교육학과 석박통합과정
†† 정 회 원: 서울과학기술대학교 전자공학과 교수
††† 정 회 원: 에이디칩스 상무이사
†††† 종신회원: 고려대학교 컴퓨터교육과 교수(교신저자)
논문접수: 2014년 12월 10일, 심사완료: 2015년 1월 24일, 게재확정: 2015년 3월 9일
* 본 논문은 2015년 정부(교육과학기수부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행되었음 (NRF-2014R1A1A2059652)

1. Introduction

In modern high-performance computers, the instruction queue (IQ) is typically employed to bridge the latency difference between CPU core and cache, and feeds the CPU with proper instructions. For example, Nehalem [1] is able to store up to 18 instructions in IQ to mitigate the impact of the 3-cycle latency of L1 instruction cache. Thus, for the CPU performance, it is crucial to place proper instructions in IQ, and it is closely related to the hit rate of branch predictors. The higher the hit rate is, the CPU is better serviced with the instruction stream.

The branch predictors [2-4] are commonly incorporated in commercial CPUs ranging from embedded processors to server processors [1, 5-7]. For example, ARM1136 [6] has a 3-entry return address stack (RAS) and a 128-entry BTB. Power3 [5] has a 8-entry RAS and a 256-entry BTB. The branch predictor especially plays a key role for performance in the deep-pipelined processors, where the control hazard incurs a considerable number of cycle loss. In addition, in superscalar processors such as x86, multiple instructions are executed simultaneously. Considering that roughly 20% of the instructions in programs are branches [8], the importance of branch predictor cannot be over-emphasized. For the low-cost embedded processors with shallow pipeline stages, the role of branch predictor has been overlooked since the control hazard incurs only a few cycle penalty, and the hardware overhead of the branch predictor could be an overkill. Thus, there are few studies quantifying its impact on performance.

In this paper, we have used a commercial embedded processor, a 5-stage pipelined Extendable Instruction Set Computer (EISC) processor from Adchips [9] and enhanced the IQ efficiency and performance with RAS. Unlike the previous work, the proposed scheme completely removes the RAS corruption from the wrong-path prediction, taking advantage of the shallow pipeline. According to experiment results, the proposed RAS has a significant impact on IQ efficiency and performance. The experiments show that the number of IQ flushes can be reduced to as low as 38.72%, compared to the baseline, boosting the application performance by as high as 3.47%. Unlike the previous studies, we have used the actual RTL code for the experiments.

The rest of the paper is organized as follows. Section 2 summarizes the related work. Section 3 explains the IQ organization and its role in the EISC processor. Section 4 discloses the detailed design of the proposed RAS and its operation with IQ. Section 5 presents the experiment results and its analysis. Finally, Section 6 concludes our paper.

2. Related work

There are few studies or reports quantifying the IQ efficiency with branch prediction. Most of studies simply measured performance and/or power consumption of branch predictors assuming superscalar processors with wide issue widths. In addition, unlike our study, most of researches conducted with software simulators, without actual implementation of RTL code.

Parith [10] utilized SimpleScalar and Wattch with an out-of-order CPU configuration to investigate the power consumption problem with branch predictor. The study concludes that it is worth spending more power in branch predictor if it results in more accurate prediction that improves execution time. Das [11] investigated the impact of faulty branch predictor on power consumption. The study reveals that the design inaccuracy could lead to a huge power loss (up to 95% additional power). Thus, fault-tolerant predictor design is essential for chip multiprocessors.

Works on Return Address Stack (RAS) were first performed by Webb [12] and Kaeli [13]. Jordan [14] proposed a checkpointed RAS that provides a recovery mechanism in case RAS is corrupted from the wrong-path instructions. The proposed RAS scheme provides a link to the next element such as the linked list in data structure. Skadron [15] proposed an cost-effective RAS with a recovery mechanism, which stores the RAS pointer and data (i.e., return address) whenever the processor speculates past a branch. The RAS pointer and data are stored along with the register-rename map, which is already implemented in superscalar processors. More recently, Wang [16] proposed another RAS structure referred to as self-aligning RAS (SARAS). SARAS has an aligning queue where the popped entry from RAS is recorded along with its index. The popped entries are recovered from the aligning queue to the RAS upon the detection of wrong-path execution. Vandierendonck [17] proposed a low-cost corruption detector for RAS (CT-RAS), which adds a corruption bit in every entry of RAS and includes a separate table called CheckTOS. If the corruption bit is set in RAS, it indicates that the corresponding entry is overwritten via a wrong-path call. The experiments show that CT-RAS provides a superior performance to the checkpointed RAS when the number of entries in RAS is less than 32.

Table 1 summarizes the hardware details of commercial processors. ARM1136 [6] and ARM1156 [7] have a 3-entry RAS with 3-cycle miss penalty with 8 and 9 pipelines, respectively. Power3 [5] has an 8-entry RAS with 3-cycle miss penalty with variable pipelines depending on integer or floating-point processing. Commercial embedded processors [6] [7] rarely disclose the microarchitectural details such as IQ due in part to confidentiality. To the best of our knowledge, the performance impact of RAS in terms of IQ on [6] [7] has not been published.

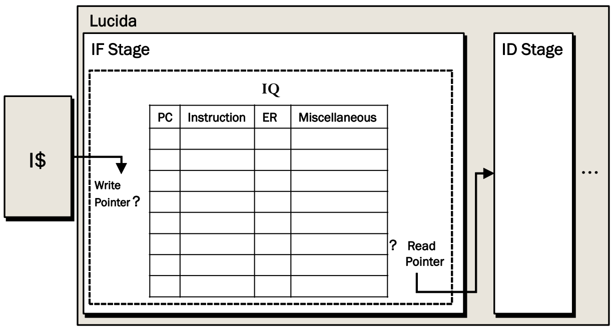
<Table 1> RAS and Miss Penalty in Commercial Processors

Processors	ARM1136 [6]	ARM1156 [7]	Power3 [5]
Pipeline stages	8	9	7 ~ 10
# RAS entries	3	3	8
Miss penalty	5 cycle	7 cycle	3 cycle

3. Instruction Queue in EISC Processor

EISC is a 32-bit architecture with 16-bit instruction set for embedded systems. With the 16-bit instructions, it improves the code density compared to 32-bit instruction set architecture. One of the key features in the EISC microarchitecture (referred to as Lucida hereafter) is that it provides a capability of creating 32-bit immediate with 16-bit instructions without accessing memory. It is achieved through the consecutive instructions called LERI that store parts of 32-bit immediate. In Lucida, hardware formulates the 32-bit immediate with the LERI folding [18] logic in the fetch stage. Then, the immediate is passed through to subsequent pipeline stages.

Lucida has a generic 5-stage pipeline composed of Fetch, Decode, Execution, Memory Access and Writeback. It incorporates the 8-entry IQ in the Fetch stage. Fig. 1 shows the schematic diagram of IQ. The IQ stores instructions from instruction cache where 2 instructions are brought to the CPU core at a time. Before buffering instructions to IQ, the LERI folding logic detects and combines the consecutive LERI instructions to create 32-bit immediate. Each entry in IQ stores not only an instruction, but also 32-bit immediate, flags, and program counter for precise interrupt support. Thus, the LERI instruction itself does not occupy an entry in IQ.



<Fig. 1> Instruction Queue in Lucida

Another important role of IQ is for providing the loop buffering mechanism. It is similar to the Loop Stream Detection (LSD) in x86. In case of Nehalem [1], LSD is activated if ?ops in a loop is within 28 slots. In Lucida, the loop buffering is activated if the number of instructions in a loop is equal to or less than 8 instructions without counting the LERI instructions in the loop. The loop buffering provides a significant performance and power advantage since instructions are directly supplied from the IQ while executing the loop without accessing instruction cache.

To determine the loop, Lucida incorporates a simple branch predictor which has only 4-entry branch target buffer (BTB) in a fully associative configuration with the pseudo LRU replacement. If a branch instruction hits one of BTB in the Fetch stage and the branch destination is within 8 instructions away, the loop buffering is activated. In Lucida, the branch is resolved in the Decode stage. In case of mis-prediction, the corresponding entry in BTB is invalidated and the correct destination is recorded. In addition, IQ is flushed since it was buffering wrong-path instructions from the branch destination.

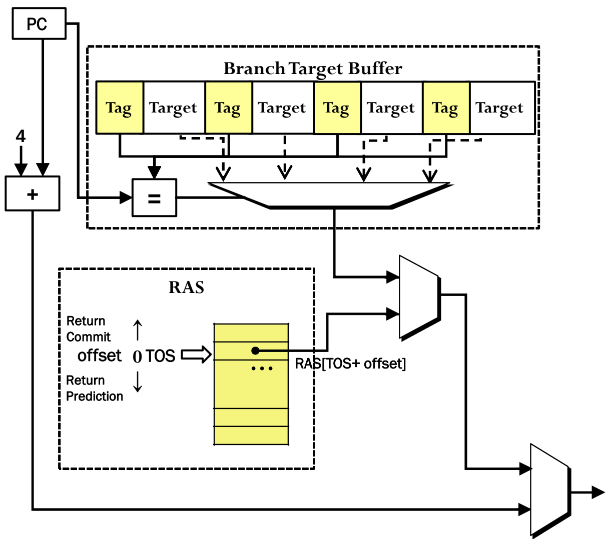
<Table 2> Branch Instruction in EISC and Branch Predictor

Branch Instructions in EISC	Branch Predictor
JMP (Jump) **JNV** (Jump on overflow clear) **JV** (Jump on overflow set) **JP** (Jump on sign clear) **JN** (Jump on sign set) **JNZ** (Jump on non-zero) **JZ** (Jump on zero) **JNC** (Jump carry clear) **JC** (Jump carry set) **JGT** (Jump signed greater) **JLT** (Jump signed less) **JGE** (Jump signed greater or equal) **JLE** (Jump signed less or equal) **JHI** (Jump unsigned higher) **JLS** (Jump unsigned lower or equal)	BTB
JAL (Jump and link): LR ← PC+2	BTB, RAS Push
JPLR (Jump link register): PC ← LR **POP PC**: PC ← popped entry from stack	RAS Pop
JR (Jump register indirect): PC ← %RS **JALR** (Jump register indirect and link): LR ← PC+2, PC ← %RS	None

Table 2 summarizes the branch instructions in EISC. Among them, the branch predictor in the original Lucida handles direct branch instructions, whereas the destination of the return instructions (JPLR and POP PC) is not predicted in BTB, resulting in control hazard. The control hazard in Lucida pipeline incurs at least a 1-cycle loss in the pipeline; The long cache access latency or memory access in case of a cache miss incurs additional penalty. As mentioned, mis-prediction and consequent control hazard incurs the IQ flush. The IQ flush adversely affects the performance and power consumption since the IQ should newly be filled from cache and the additional cache accesses consume power. Thus, it is essential to reduce the number of IQ flushes as long as the new hardware investment does not overkill its benefits.

When ported to a Virtex-6 FPGA, the IQ in Lucida requires 20.29% of registers and 14.79% of LUTs out of the total hardware consumption. Considering that Lucida is used in embedded systems, the hardware cost of IQ is significant and its efficient utilization is a key issue for performance.

4. IQ and RAS in Lucida

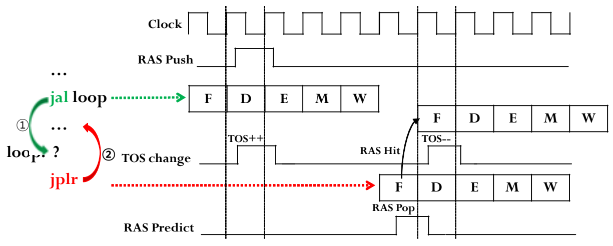


<Fig. 2> BTB and RAS in Lucida

Fig. 2 shows the schematic diagram of the branch predictors in Lucida. As mentioned, the BTB is incorporated in the original Lucida. The RAS was newly designed for the study to support the prediction for the return instructions. The number of evaluated entries in RAS is 4 or 8. It was chosen not to invest too much hardware because Lucida is a processor used in embedded systems.

4.1 Basic Operation of RAS in Lucida

The RAS in Lucida is essentially a circular buffer, which is indexed by top-of-stack (TOS). Fig. 3 demonstrates the timing diagram of the RAS push and pop operations. The previous work [15] [16] assumes that TOS is updated in the Fetch stage and the branch is resolved in writeback stage. Thus, it introduces the risk of RAS corruption from the wrong-path instructions upon branch misprediction. In Lucida, TOS is updated at the Decode stage. It completely eliminates the RAS corruption if not overflowed, because the branch is resolved in the Decode stage. The prediction for the return instructions occurs in the Fetch stage when filling up IQ, as depicted in Fig. 3. To handle the TOS skew between prediction and RAS update, an additional pointer called offset is used to adjust TOS so that the right position in RAS is accessed for prediction. The detailed operation of offset is explained in the following section with an example.



<Fig. 3> Pipeline Timing Diagram of push and pop in Lucida RAS

TOS and offset are initialized to zero at reset. TOS is incremented by 1 after pushing the return address to an entry in RAS when a subroutine call instruction is detected. TOS is decremented by 1 after the return instruction is detected in the Decode stage. TOS wraps around to the beginning when it reaches to the top of RAS.

4.2 IO and RAS Detailed Operation

The impact of RAS on the IQ efficiency and performance can be well explained with an example. In Lucida, all the instructions in a program go through IQ to the subsequent stages for execution. Fig. 4 shows two C-code examples where RAS can be well utilized. Fig. 4 (a) is a recursive call to compute factorials where the return instructions are executed almost consecutively. Fig. 4 (b) is a nested function call where the return instructions are used to return to the callers.

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```

(a) Recursive Procedure Call

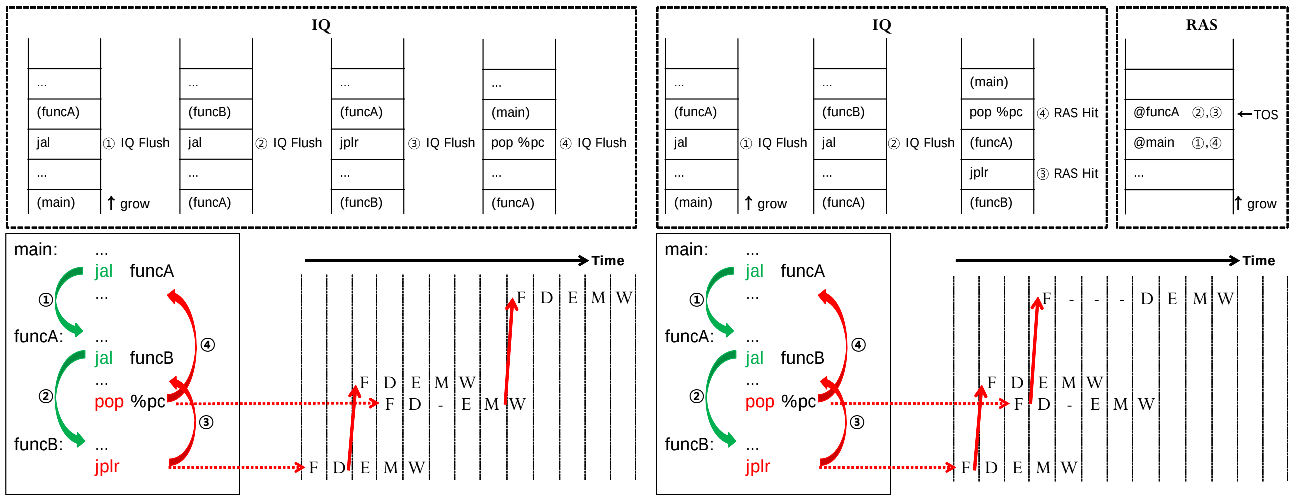
```
main() {                 funcA() {
   ...                      ...
   funcA                    funcB();
   ...                      ...
   return 0;                return 0;
}                        }
                         funcB() {
                            ...
                            return 0;
                         }
```

(b) Nested Procedure Call

<Fig. 4>

Fig. 5 shows the IQ and RAS status without and with RAS for comparison, as Lucida runs the instructions compiled for the code of Fig. 4 (b). Fig. 5 (a) illustrates the detailed operations of the Lucida without RAS, and Fig. 5 (b) demonstrates the enhanced version with RAS. The jal instructions (① and ②) are used for calling functions (funcA and funcB). The jplr instruction (③) is used to return to funcA, which triggers the control hazard. The pop pc instruction (④) is used to return to main. The pop pc loads from memory to the PC register, and it occurs in the Memory Access stage. In Lucida, the pop pc instruction stalls the pipeline because the subsequent instructions would be nullified anyway. In case of Fig. 5 (a), the first jal (①) incurs an IQ flush, which is inevitable since it changes the instruction flow with a BTB miss (first call). The second jal (②) incurs another IQ flush as well for the same reason. The destination of the first return instruction (jplr (③)) is fetched after the Decode stage due to the control hazard. The destination of the second return instruction (pop pc (④)) is fetched after the Memory Access stage due to the control hazard. The return instructions (jplr (③) and pop pc (④)) also flushes IQ due to the changes in instruction flow and the lack of the prediction logic.

On the other hand, in case of Fig. 5 (b), two RAS hits prevent the last two IQ Flushes in Fig. 5 (a). The first jal (①) incurs a push of the return address in main to RAS in the Decode stage and flushes IQ. The second jal (②) also incurs a push of the return address in funcA to RAS with another IQ flush. At this point, the RAS status is shown in Fig. 5 (b) where TOS points to the return address to funcA. When the first return instruction (jplr) is filled up in IQ, RAS makes a prediction (③:



(a) Nested Procedure Call without RAS

(b) Nested Procedure Call with RAS

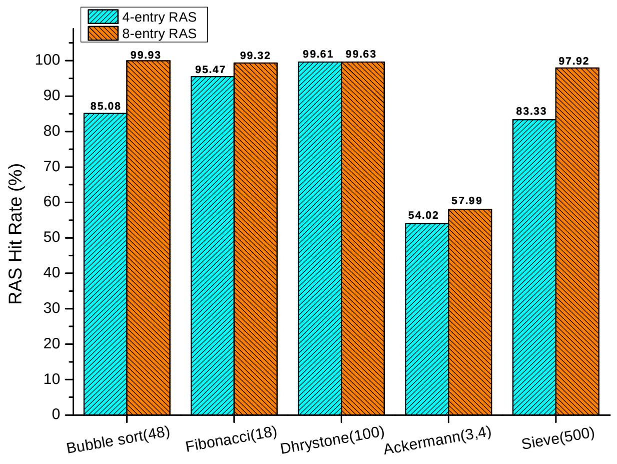
<Fig. 5>

RAS Hit). However, TOS is not decremented since the jplr is not in the Decode stage yet (jplr is still in IQ). Note that TOS is updated in the Decode stage. To account for the skew, the offset is decremented by 1. Thus, offset becomes -1 at this point in time. Following the RAS prediction, IQ is continuously filled with the instructions after the funcB call in funcA. Assume that the next return instruction (pop pc) is just a few instructions away from the first one (jplr). When pop pc (④) is inserted in IQ, the RAS makes another prediction with the adjusted TOS (TOS + offset), which points to the return address in main (@main in RAS). The offset is incremented by 1 when the return instruction is in the Decode stage. With the proposed microarchitecture, the RAS prevents the corruption from wrong-path instructions. It also makes the right predictions while filling up the IQ when the return instructions are a few instructions apart, which occurs frequently with recursive or nested calls.

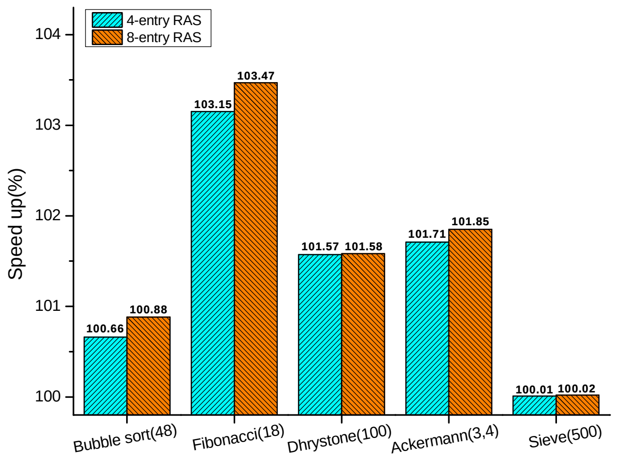
5. Evaluation and Analysis

We have implemented the proposed architecture with Verilog-HDL and evaluated the performance using the RTL simulation with Xilinx ISIM [20]. The experiments were performed with two RAS configurations: 4-entry and 8-entry, and five benchmark programs are executed: Dhrystone [21], Ackermann, Sieve, Fibonacci, and Bubble Sort. Dhrystone is a general benchmark for embedded processors. Other benchmarks are synthetic benchmarks. Each synthetic benchmarks has various characteristics. Ackermann and Fibonacci are recursive-based function with conditional statement, whereas Bubble Sort and Sieve are loop-based function. So, the program flow is changed by the parameter value.

The baseline is the original Lucida, which does not have RAS. Lucida has two scratchpad memories (SPM): Instruction SPM (ISPM) and Data SPM (DSPM). The benchmark program was loaded into ISPM before simulations. The Lucida requires a 1-cycle to access SPMs.



<Fig. 6> RAS Hit Rates



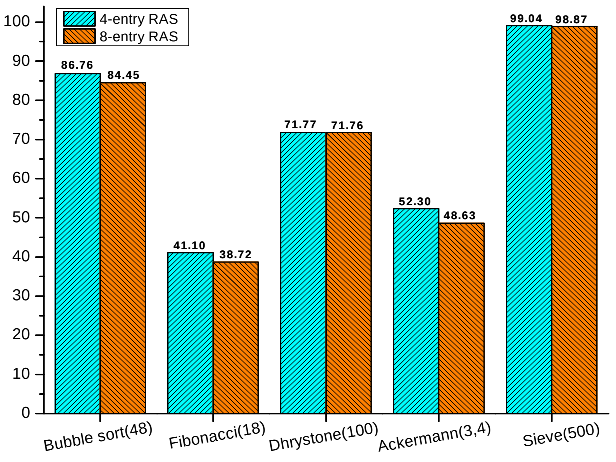
<Fig. 7> Speedups over the baseline

Fig. 6 shows the hit rate of RAS for each benchmark. As expected, the 8-entry RAS always provides a superior performance to the 4-entry RAS. In Fibonacci and Dhrystone, the hit rate is higher than 95% with both 4-entry and 8-entry. In Bubble Sort and Sieve, there is roughly 14% difference in hit rate between 4-entry and 8-entry. It comes from the overflow and underflow of RAS. In case of the 8-entry RAS, over eight consecutive calls without return make the RAS overflow, as opposed to over 4 consecutive calls in the 4-entry RAS. Ackermann reports the lowest hit rate under 58% even with the 8-entry RAS. The intensive calls and returns in Ackermann are blamed for the lowest hit rate. Depending on the input parameter, Ackermann has three different instruction flows, which are intensively called in a recursive manner.

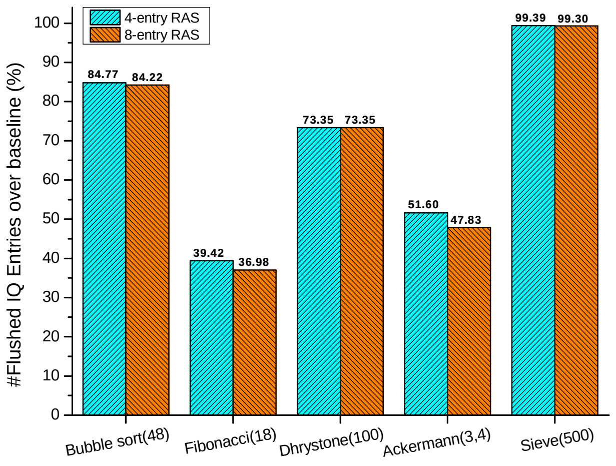
Fig. 7 shows the speedup over the baseline for benchmark programs. Compared to the baseline, Fibonacci reports the highest speedup (3.47%) with the 8-entry RAS. On the other hand, Sieve shows mere 0.01 ~ 0.02% speedup even though the RAS hit rate in Fig. 6 is relatively high. It is because, in Sieve, a tiny fraction of the compiled code is return instructions, influencing little on performance. In contrast, Ackermann contains a much higher portion of return instructions. Thus, RAS is able to boost performance by 1.71% with the 4-entry RAS and by 1.85% with the 8-entry RAS even though the hit rate is the lowest. Considering that Lucida has a 5-stage shallow pipeline and the control hazard incurs a 1-cycle loss, it is a significant gain to achieve up to a 3.47% speedup. The performance benefit will be increased much further with the long cache access latency and even longer memory access latency upon cache misses.

The impact of RAS on IQ utilization was measured by two metrics: the number of IQ flushes in Fig. 8 and the number of flushed IQ entries in Fig. 9. As shown in Fig. 8, Fibonacci reports the highest decrease in IQ flushes over the baseline (58.90% decrease with 4-entry and 61.28% decrease with 8-entry). Ackermann shows the second highest with 47.70% decrease with 4-entry and 51.37% decrease with 8-entry.

The number of flushed entries in IQ, as shown in Fig. 9, shows a similar trend to the number of IQ flushes. Table 3 shows the average number of flushed entries in IQ, which shows a marginal difference between baseline and architectures with the proposed RAS. When flushed, more than 80% of IQ entries were occupied in case of Fibonacci and Ackermann.



<Fig. 8> Ratio of the number of flushes over baseline



<Fig. 9> Ratio of the number of flushed IQ entries over baseline

<Table 3> Average Number of Flushed IQ Entries

Benchmark	Bubble sort	Fibonacci	Dhrystone	Ackermann	Sieve
Baseline	4.38	6.80	4.85	6.80	6.06
4-entry	4.27	6.53	4.95	6.71	6.08
8-entry	4.35	6.50	4.95	6.69	6.08

<Table 4> Performance Average and Hardware Cost over Baseline

Section	Subsection	4-entry	8-entry
Performance	Avg. speedup	101.42%	101.56%
	Avg. hit rate of RAS	83.50%	90.96%
	# IQ flushes on avg.	70.19%	68.42%
	# flushed IQ entries on avg.	69.71%	68.27%
Hardware Cost	# Slice Register	103.84%	107.12%
	# of Slice LUTs	101.49%	104.11%

Even the lowest (Bubble Sort) reports that more than half of entries in IQ were occupied when flushed. The number of IQ flushes is closely related to the speedup; the less number of flushes incurs the higher performance because the right stream of instructions is prepared for execution in IQ. It is clearly shown by comparing Fig. 8 and Fig. 9 with Fig. 7. The highest reduction in flushes returns the highest performance (speedup), as observed in the Fibonacci case. The small reduction in flushes comes with the minimal benefit in speedup, as in Sieve. The reduction in the number of IQ flushes has positive implications for power consumption. Considering that the real estate occupied by IQ is roughly 20% in Lucida, the reduction in unnecessary IQ buffering and cache accesses would positively influence the overall power consumption.

Table 4 summarizes the performance average across all benchmark programs. It also reports the hardware cost associated with the implemented RAS. In general, the more investment in hardware (8-entry) always delivers the superior performance in all metrics. The overall performance was increased by 0.14% with 8-entry, compared to 4-entry. The RAS hit rate is increased by roughly 7% with 8-entry, compared to 4-entry. The number of flushes was also decreased with 8-entry roughly by 1.5% when compared to 4-entry. The hardware cost was measured by porting the RTL design to a Virtex6 FPGA. The 8-entry RAS requires 3.28% more registers and 2.62% more LUTs compared to the 4-entry RAS. As concern with the cost effectiveness, 4-entry RAS seems more efficient because additional hardware cost in 8-entry is quite remarkable. As mentioned, Lucida is a processor for embedded systems. It is offered as a synthesizable format. It means that the hardware configuration can be fine-tuned depending on the performance target of applications.

6. Conclusion

This paper presented a microarchitectural enhancement for the IQ efficiency with RAS in a commercial EISC processor. Lucida is equipped with the 8-entry IQ to store the instruction stream with the LERI folding. The IQ occupies a significant amount of real estate in Lucida. The efficient utilization of IQ is a key for performance and power. Taking advantage of a shallow 5-stage pipeline in Lucida, the proposed RAS architecture completely eliminates the corruption by wrong-path instructions. With the 4-entry and 8-entry RAS, the number of IQ flushes was decreased by up to 58.90% and 61.28%, respectively. The performance of benchmark programs is closely related to the decrease in IQ flushes. The more reduction in IQ flushes returns the higher performance (speedup). The experiments show up to 3.47% performance improvement with 8-entry RAS.

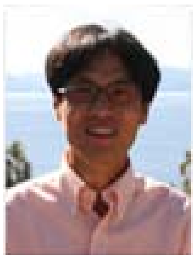
참 고 문 헌

[1] Thomadakis, M. E. (2011). The architecture of the Nehalem processor and Nehalem-EP smp platforms. Resource, 3, 2.
[2] Seznec, A., & Michaud, P. (1999). De-aliased hybrid branch predictors.
[3] Yeh, T. Y., & Patt, Y. N. (1993). A comparison of dynamic branch predictors that use two levels of branch history. ACM SIGARCH Computer Architecture News, 21(2), 257-266.
[4] McFarling, S. (1993). Combining branch predictors (Vol. 49). Technical Report TN-36, Digital Western Research Laboratory.
[5] Papermaster, M., Dinkjian, R., Jayfiield, M., Lenk, P., Ciarfella, B., O'Conell, F., & DuPont, R. (1998). POWER3: Next generation 64-bit PowerPC processor design. IBM White Paper, October.
[6] McFarling, S. (1993). Combining branch predictors (Vol. 49). Technical Report TN-36, Digital Western Research Laboratory.
[7] ARM. ARM1156T2-STM Revision: r0p4 Technical Reference Manual.
[8] Hennessy JL, Patterson DA. (2002) Computer architecture: a quantitative approach: Morgan Kaufmann
[9] Lee, H., Beckett, P., & Appelbe, B. (2001, January). High-performance extendable instruction set computing. In Australian Computer Science Communications (Vol. 23, No. 4, pp. 89-94). IEEE Computer Society.
[10] Parikh, D., Skadron, K., Zhang, Y., & Stan, M. (2004). Power-aware branch prediction: Characterization and design. Computers, IEEE Transactions on, 53(2), 168-186.
[11] Das, B., Bhattacharya, G., Maity, I., & Sikdar, B. K. (2011, December). Impact of Inaccurate Design of Branch Predictors on Processors' Power Consumption. In Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on (pp. 335-342). IEEE.
[12] Webb, C. F. (1988). Subroutine call/return stack. IBM Technical Disclosure Bulletin, 30(11), 221-225.
[13] Kaeli, D. R., & Emma, P. G. (1991, April). Branch history table prediction of moving target branches due to subroutine returns. In ACM SIGARCH Computer Architecture News (Vol. 19, No. 3, pp. 34-42). ACM.
[14] Jourdan, S., Stark, J., Hsing, T. H., & Patt, Y. N. (1997). Recovery requirements of branch prediction storage structures in the presence of mispredicted-path execution. International Journal of Parallel Programming, 25(5), 363-383.
[15] Skadron, K., Ahuja, P. S., Martonosi, M., & Clark, D. W. (1998, November). Improving prediction for procedure returns with return-address-stack repair mechanisms. In Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture (pp. 259-271). IEEE Computer Society Press.
[16] Wang, G., Hu, X., Zhu, Y., & Zhang, Y. (2012, June). Self-Aligning Return Address Stack. In Networking, Architecture and Storage (NAS), 2012 IEEE 7th International Conference on (pp. 278-282). IEEE.
[17] Vandierendonck, H., & Seznec, A. (2008). Speculative return address stack management revisited. ACM Transactions on Architecture and Code Optimization (TACO), 5(3), 15.
[18] Kim, H. G., Jung, D. Y., Jung, H. S., Choi, Y. M., Han, J. S., Min, B. G., & Oh, H. C. (2003). AE32000B: a fully synthesizable 32-bit embedded microprocessor core. ETRI journal, 25(5), 337-344.
[19] Intel 64 and IA-32 Architectures

Optimization Reference Manual. http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf.

[20] Xilinx. ISE Simulator (ISim). http://www.xilinx.com/tools/isim.htm.

[21] Weicker, R. P. (1984). Dhrystone: a synthetic systems programming benchmark. Communications of the ACM, 27(10), 1013-1030.



이 승 은

1998 한국과학기술원 전기공학과 (학사)
2000 한국과학기술원 전기공학과 (석사)
2008 UC Irvine Electrical and Computer Engineering
2008~2010 Intel Corporation Platform Architect
2010~현재 서울과학기술대학교 교수
관심분야: 컴퓨터구조, 밀티프로세서 시스템온칩, 네트워크온칩
E-Mail: seung.lee@seoultech.ac.kr

김 관 영

1990 성균관대학교 전자공학과 (학사)
1989~1999 삼성전자 연구원
2000~현재 Adchips 상무이사
관심분야: 임베디드 CPU, 컴퓨터구조, 멀티미디어 시스템온칩
E-Mail: kevinkky@adc.co.kr

서 태 원

1993 고려대학교 전기공학과 (학사)
1995 서울대학교 전자공학과 (석사)
1995~1998 LG종합기술원 주임연구원
1998~2001 하이닉스반도체 선임연구원
2006 Georgia Institute of Technology Computer Engineering (공학박사)
2007~2008 Intel Corporation System Engineer
2008~현재 고려대학교 컴퓨터교육과 교수
관심분야: 컴퓨터구조, 임베디드 시스템, 컴퓨터교육, 멀티프로세서
E-Mail: suhtwa@korea.ac.kr

김 한 이

2012 고려대학교 컴퓨터교육과 (학사)
2012~현재 고려대학교 컴퓨터교육과 석박사통합과정
관심분야: 컴퓨터구조, 임베디드 시스템
E-Mail: hanyeemy@korea.ac.kr