J Supercomput (2013) 65:484–495
DOI 10.1007/s11227-011-0734-x

Revisiting reorder buffer architecture for next generation high performance computing

Min Choi · Jong Hyuk Park · Young-Sik Jeong

Published online: 1 February 2012

Abstract Modern microprocessors achieve high application performance at an acceptable level of power dissipation. Reorder buffer is used for out-of-order instructions to be committed in-order. The reorder buffer plays a key role in modern microprocessors because performance improvement techniques highly rely on aggressive speculation to feed wider issue, out-of-order, and deep pipelines. In terms of power to performance trade-off, reorder buffer is particularly important. This is because enlarging the reorder buffer size achieves high performance but naive scaling of the conventional reorder buffer architecture can severely increase the complexity and power consumption. In this paper, we propose low-power reorder buffer techniques for contemporary microprocessors. First, the separated reorder buffer reduces power dissipation by deferred allocation and early release. The deferred allocation delays the SROB allocation of instructions until all their data dependencies are resolved. Then, the instructions are executed in program order and they are released faster from the SROB. The result of the instruction is written into rename buffers immediately after the execution completes. Then, the result values in the rename buffer are written into the architectural register file at the commit state. The proposed approaches in this paper provide higher resource utilization and low power consumption.

M. Choi
Department of Information and Communication Engineering, Chungbuk National University, Cheongju, Republic of Korea
e-mail: mchoi@cbnu.ac.kr

J.H. Park
Seoul National University of Science and Technology, Seoul, Republic of Korea
e-mail: parkjonghyuk1@hotmail.com

Y.-S. Jeong (✉)
Wonkwang University, Iksan, Republic of Korea
e-mail: ysjeong@wku.ac.kr

Keywords Reorder buffer · High performance computing · Separated reorder buffer · Energy efficiency

1 Introduction

Enlarging the size of reorder buffer can lead to performance improvement. However, naive scaling of the conventional reorder buffer severely affects the complexity and power consumption. In fact, Folegnani and Gonzalez [1] showed that the reorder buffer is the most complex and power-dense parts in dynamically scheduled processors. Thus, much research has been conducted to increase the size of the reorder buffer without negatively impacting power consumption.

In this context, we propose a novel technique for reducing power dissipation and improving performance of the reorder buffer (ROB). Our proposed method, called separated reorder buffer (SROB), is distinct from other approaches in that we achieve early release without depending on any checkpointing. This feature gives us good performance with relatively low power dissipation. In this paper, we introduce a separated architecture for reorder buffer. First, we focus on the fact that many instructions waste ROB resources without doing any useful work during data dependency resolution. To reduce such wasteful resource usage, we introduce the novel concept of reorder buffer, named a separated reorder buffer (SROB). The SROB structure consists of two parts which are in charge of dependent instructions and independent instructions, respectively. For the dependent instructions, our SROB architecture executes the instructions in program order and releases the instructions faster. This results in higher resource utilization and low power consumption. The power reduction stems from deferred allocation and early release. The deferred allocation technique inserts instructions into the SROB only after fulfilling the data dependency. The SROB releases instructions earlier immediately after the execution completes, because precise exception is trivial under in-order execution. Second, in order to deal with the power problem on issue queue, we focus on a well known fact that the vast majority of instruction dependency exists within a basic block. In practice, a basic block is comprised of about six instructions on average [2–6].

The rest of this paper is organized as follows. Section 2 presents a brief review of the existing approaches. Section 3 describes our modified reorder buffer architecture, the separated reorder buffer, and the concept of deferred allocation and early release. We evaluate its performance and power consumption in Sect. 4. Finally, we conclude by summarizing our results in Sect. 5.

2 Related work

A wealth of work has been undertaken to design a low power instruction window architecture. We begin with categorizing and summarizing a number of related works for power reduction of the reorder buffer and the issue queue.

The K-instruction processor [7] proposed the concept of early releasing reorder buffer resources via checkpointing. For the case of a long-latency operation, the processor takes a checkpoint and releases the reorder buffer resource early. This achieves high resource utilization, but it closely depends on multicheckpointing.

The early load retirement [8] mechanism combines register checkpointing and early release of the load instruction. This allows instructions dependent on the long-latency load to execute sooner. However, this scheme is still based on checkpointing and it requires the checkpointing overhead. The cherry [9] is developed to recycle physical registers and load/store queue entries aggressively using combination of reorder buffer and periodic checkpointing.

The concept for runahead execution was proposed by Dundas [10] to improve the data cache performance on in-order execution. More recently Mutlu [11] has extended the concept for out-of-order execution processors. Mutlu [11] proposes a runahead execution to realize large instruction windows by reducing the penalties from long-latency memory instructions. When the instruction window is blocked by the long-latency instruction, the architectural register state is checkpointed and retires early. In this way, the instructions following the blocking operation are executed quickly. This work is close to our early release scheme in separated reorder buffer, but it is different in that we eliminate the use of the checkpointing.

Kucuk [12] proposed approaches for reducing the complexity and power dissipation in processors that use separate register file to maintain committed register values. The first approach relies on a distributed implementation of the reorder buffer that spreads the centralized reorder buffer structure across the functional units. The second approach combines the use of the previously proposed retention latches and a distributed reorder buffer implementation that uses minimally ported distributed components. The implementation strategy for the centralized reorder buffer is functionality focused with little emphasis on power, whereas the distributed reorder buffer theory was proposed. In this work, it is shown that distributed reorder buffer design is a more power efficient design because it handles dynamic power in a more robust and less complex manner than the centralized reorder buffer. It does so through design optimizations which ultimately lower switching activity throughout the internal logic to accomplish the output to the architected register file.

Smith [13] proposed the concept of future file that is similar to the history buffer method; however, it uses two separate register files. One register file reflects the state of the architectural machine. This file will be referred to as the architectural file. A second register file is updated as soon as instructions finish and therefore runs ahead of the architectural file. This future file is the working file used for computation by the functional units.

Brown [14] and Mehrara [15] focus on performance degradation during thread migration between cores due to the loss of cache state. A significant boost in post-migration performance is possible if the cache working set can be moved, proactively, with the thread. These works accelerate thread startup performance after migration by predicting and prefetching the working set of the application into the new cache.

3 The separated reorder buffer

In general, the function of the reorder buffer (ROB) is to put the instructions back into the original program order after the instructions have finished execution possibly out of order. The ROB maintains an ordered list of the instructions and takes into

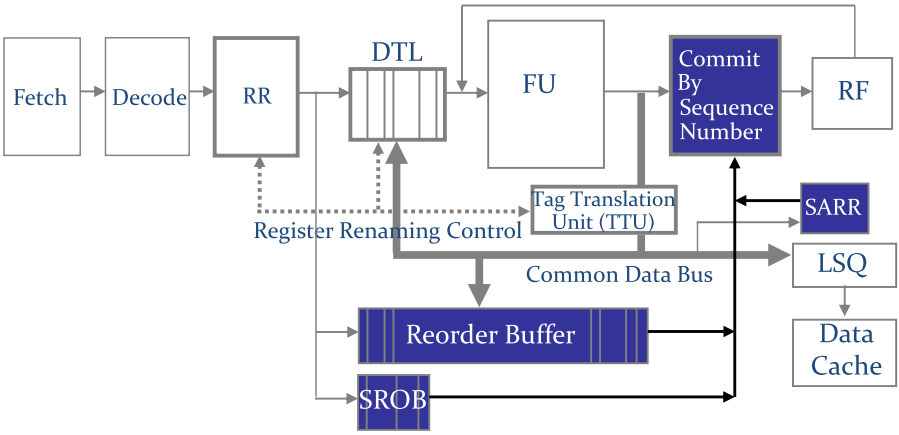

Fig. 1 Pipeline architecture model

account recovery and precise exception. Conventionally, the instructions are inserted into both the issue queue and the ROB. The instructions stay in the ROB until the instruction commits. As soon as the dependency of an instruction is fulfilled, the processor executes the instruction sequentially in program order. In addition to the ordinary ROB, we propose the concept of the separated ROB (SROB), as depicted in Fig. 1.

Figure 1 shows the overall pipeline architecture, in which the colored components represent the modified (or newly added) parts in this work. The processor decodes fetched instructions (FETCH) and assigns physical registers to hold their results. This register renaming process (RR) maps the architected registers into a larger set of physical registers. Decoded instructions (DECODE) are inserted in both instruction window and reorder buffer (ROB) at dispatch time in program order. For load and store instructions, they are assigned to entries in load-store queues (LSQ). Instructions leave the instruction queue when they are issued, and free their reorder buffer entries when they commit. As shown in Fig. 2, the reorder buffer holds the result of an instruction between the time the operation associated with the instruction completes and the time the instruction commits.

The functional units (FU) can execute an operation of a certain type. The system retrieves the operands from register file (RF), and stores the operands into the register file. The stand-alone rename registers (SARR) are split register files to implement the rename buffers.

Usually in conventional reorder buffer, the rename registers are integrated into reorder buffer. Each entry in the SROB has the same structure as the ordinary ROB, but in the SROB architecture the rename registers are stand-alone, so we named that SARR. The structural difference of separated ROB is as shown in Fig. 3. Each separated part of the SROB manages dependent and independent instructions, respectively. One part of SROB processes control instructions, independent instructions, and load/store instructions. Another part of SROB is in charge of dependent instructions. The execution of dependent instructions is serialized inherently by true de-

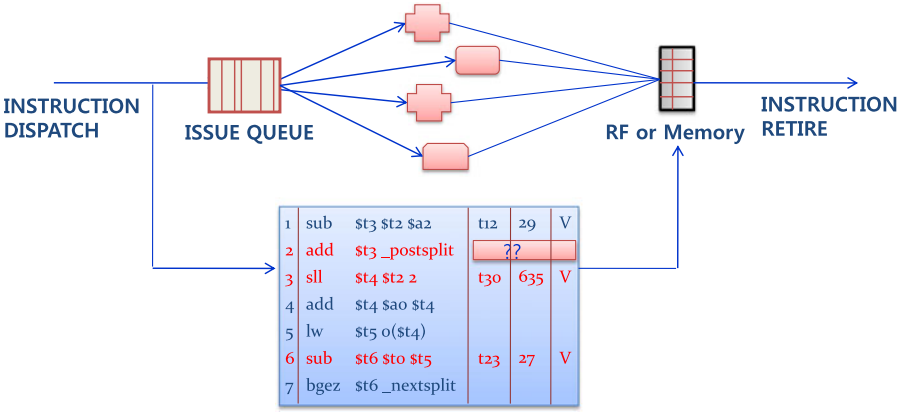

Fig. 2 Instruction Window constisting of Issue queue and Reorder buffer

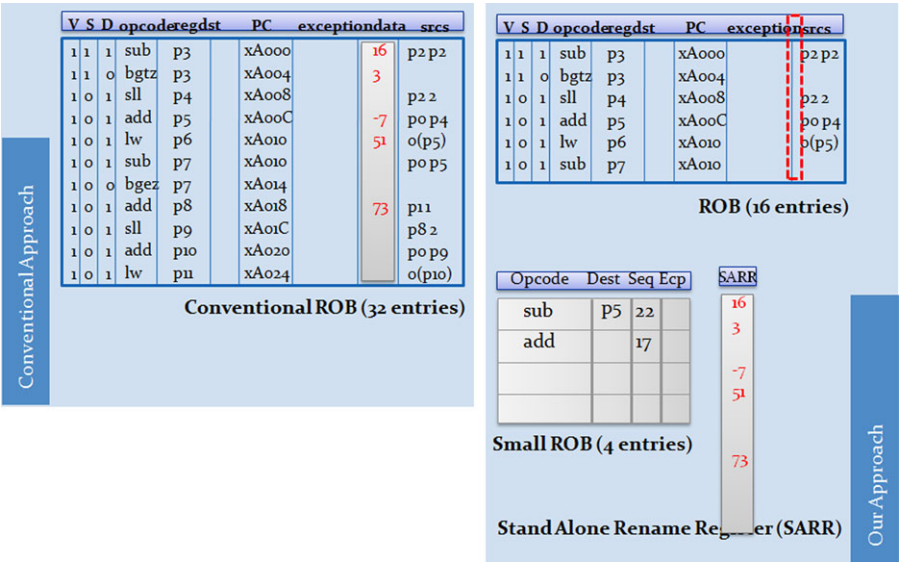

Fig. 3 Structural difference between ordinary ROB and SROB

pendency. The dependent instructions will wait for a long time to resolve their data dependencies, even if we put the dependent instructions into the general ROB.

Figure 4 shows the example of instruction allocation in the SROB architecture. The instructions waiting in the ROB do not any useful work and severely affect the power consumption and the instruction level parallelism (ILP). This is because the ROB is a complex multi-ported structure and represents a significant source of power dissipation. Moreover, if the dependent instructions are in a long dependency chain, the power and performance problem gets worse.

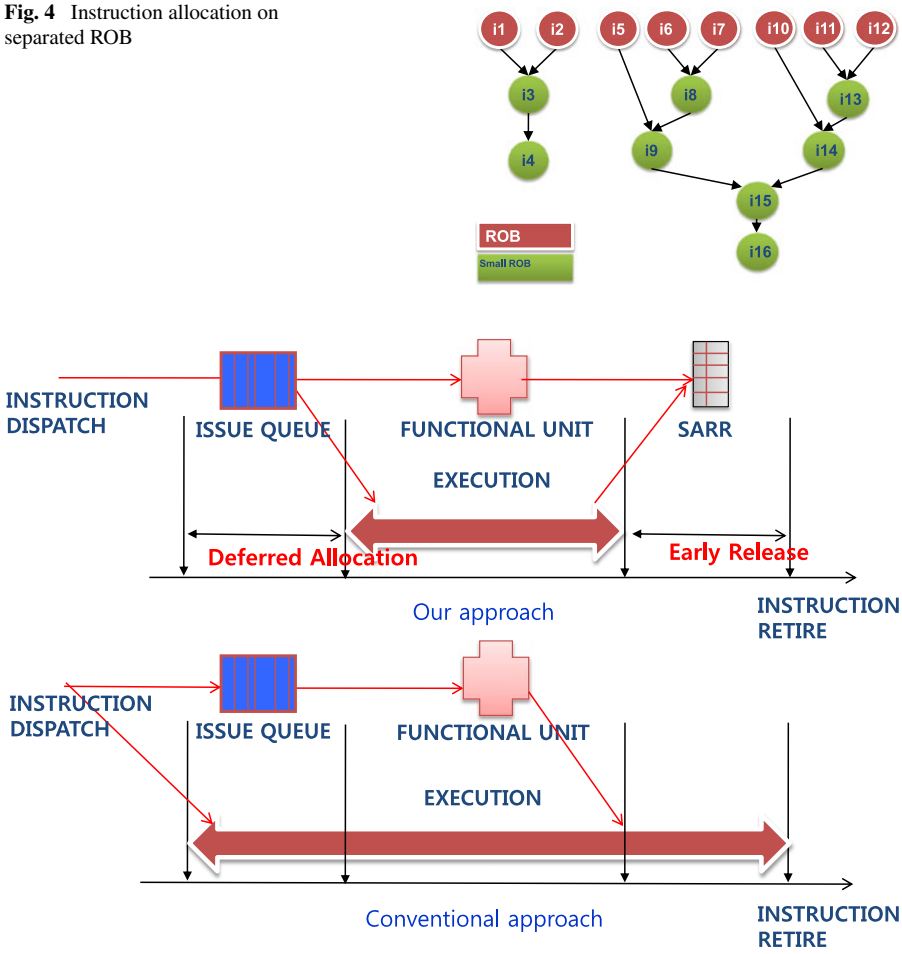

Fig. 4 Instruction allocation on separated ROB

Fig. 5 Resource usage between SROB and conventional ROB

3.1 Deferred allocation and early release

In order to resolve the power and performance problems, we prevent dependent instructions from moving through the ROB at dispatch time. The instructions wait for issue on the instruction queue, not on the ROB. After the instruction dependency is fulfilled, the instructions can go to the SROB. As a result, one instruction of a dependency chain executes in the SROB at a time naturally as shown in Fig. 5.

We call this the deferred allocation feature of the SROB. Moreover, the instructions in the SROB are released earlier and the result of the instruction is written into rename buffers immediately after the execution completes. Then, the result values in the rename buffer are written into the architectural register file at the commit state. Since the instructions in the SROB are executed in program order, we need not maintain the order of instructions and thus we have to take the results only. For implementation of the deferred allocation, we need to check whether an instruction is

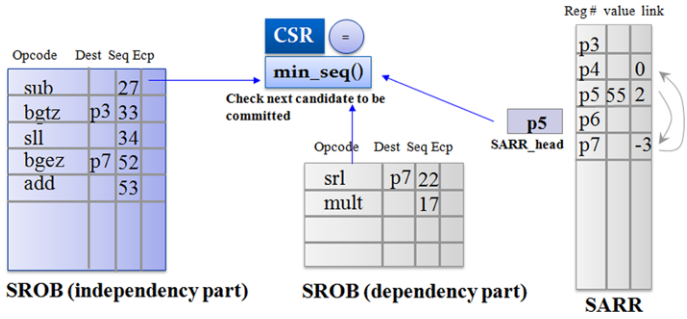

Fig. 6 Sequence number-based commit mechanism

in a certain dependency chain or not. However, facilitating such a hardware checker causes complexity at the front-end. So, we take a straightforward approach to realize a simple instruction classification at the decoding stage. Our classifier checks only the operand availability of each instruction. If operands are available, the instruction is independent. Otherwise, the instruction is dependent and it thus goes to the SROB. This classification mechanism is very simple, yet able to capture in a uniform way all the dependency chains through a given microarchitectural execution of a program.

3.2 Mechanism for in-order commit

In conventional reorder buffer architecture, a processor only looks at the bottom of the centralized reorder buffer and it commits the instruction reaching the bottom of the reorder buffer to the architectural register. As depicted in Fig. 6, we have a slightly different commit mechanism based on sequence numbers which is similar to a timestamp scheme. The commit mechanism must take into account the following three different buffers: (1) stand-alone rename registers (SARR); (2) one part of SROB for dependent instructions; and (3) another part of SROB for independent instructions. The SARR is a split register file to implement the rename buffer. Although we use one part of SROB in our architecture, it only keeps track of a subset of instructions such as control instructions or independent instructions.

Therefore, we need to separate the rename buffers from the ROB and facilitate the SARR to cover all types of instruction. Nevertheless, this is not a severe limitation because recent processors except the Intel Pentium typically incorporate the stand-alone rename register scheme [16–18]. In our commit mechanism, the fetch unit tags each instruction with a sequence number that is unique for a processor. The number is increased by one for each instruction. The sequence number ensures that the instructions retire in order. This is necessary because the instructions are waiting for retirement from three different distributed storages in our architecture. Looking for an eligible instruction to commit is achieved by the min_seq scheduling logic. The min_seq scheduler determines the candidate instruction by comparing the sequence numbers among the three buffers. As a result, an instruction retires at commit stage only if it meets the condition that all lower numbered instructions are already committed. This strategy serializes the commit sequence among all in-flight instructions.

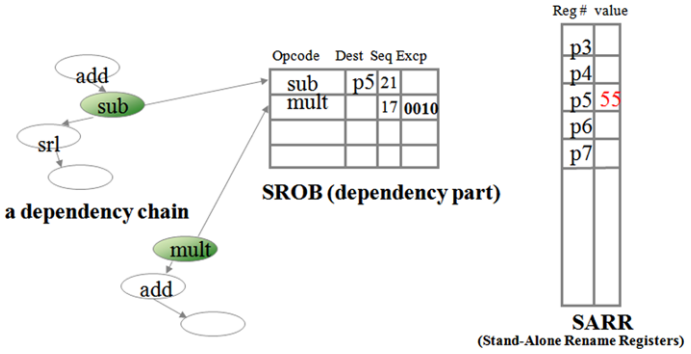

Fig. 7 Enforsing precise exception

In addition, the comparator that determines the minimum sequence number has compensation logic. This is because the sequence number register becomes zero when it overflows; we provide a compensation mechanism to handle the relative order of in-flight instructions correctly.

3.3 Enforcing precise exceptions

The SROB also provides precise exception. The precise exception occurs when an exception is raised and the processor state looks exactly as if the instructions were executed sequentially in strict program order. Since the ROB in our configuration manages only subsets of all instructions, providing a precise exception only in the ROB is not enough. In order to deal with precise exception for all instructions, we use a straightforward strategy to, as shown in Fig. 7, enforce precise exception in the SROB. If an exception is detected, the SROB just holds the instruction and the status flags until all uncompleted instructions prior to the instruction are executed and retired. Then, the exception is processed and the operations are re-executed where necessary to process the exception. Due to the true data dependency, the instructions in the SROB are executed in order. The precise exception during in-order execution is trivial.

However, this straightforward approach is only proper for sequentially executed instructions such as in the SROB. Providing precise exception in conventional ROB is still necessary because the SROB in our configuration manages only subsets of all instructions. When an exception occurs in the ROB, retiring all instructions prior to the excepting instruction is the state of the art solution. After handling from the exception, the processor resumes executing instructions in the correct state.

4 Experimental results

All tests and evaluations were performed with programs from the SPEC2000 CPU benchmark suite on Sim-Panalyzer [11]. The Sim-Panalyzer is a cycle accurate and architecture level power simulator which is built on the SimpleScalar simulator. The

	Baseline	Separated ROB method
fetch:speed	4/16	4/16
decode:width	4/16	4/16
issue:width	4/16	4/16
commit:width	4/16	4/16
rn:size	32	32
rob:size	8/16/32/64	8/16/32/64
srob:size	–	4/16
ialu, imult, fpalu, fpmult	8, 2, 2, 2/8, 8, 8, 8	8, 2, 2, 2/8, 8, 8, 8
Branch Prediction		
bpredictor	2lev.Bimodal, 2K	2lev.Bimodal, 2K
btb	128, 4-way	128, 4-way
return address stack	8entry	16entry

Fig. 8 Simulation parameters

total power dissipation in a CMOS circuit can be expressed as the sum of three main components.

$$\text{Total power dissipation} = \text{VI}_{\text{leak}} + \text{CV}^2\text{f} + \text{ACV}^2\text{f}.$$

In order to calculate power using the above equation, the Sim-Panalyzer makes use of both analytical and experimental models. We need to count how much switching has happened for each module during simulation. Then, we multiply the unit power for a microarchitectural block by the number of accesses.

The Sim-Panalyzer lumps the issue queue, the reorder buffer, and the physical register file into a register update unit (RUU). In order to better model the power consumption of contemporary microprocessor architecture, we split the RUU into the reorder buffer and the issue queues.

To evaluate the performance of the SROB architecture, we use the Alpha 21264 architecture as the baseline platform. The Alpha is an out-of-order-issue microprocessor that can fetch and execute up to four instructions per cycle. It also features dynamic scheduling and speculative execution to maximize performance. The Alpha pipeline contains four integer execution units. The two of the integer execution units can perform memory address calculations for load and store operations. The 21264 pipeline also contains two floating-point execution units to perform add, divide, square root, and multiply functions. The 21264 pipeline has seven stages which consist of instruction fetch, branch prediction, register renaming, instruction issue, register access, execution and writeback. The architecture parameters used in our Sim-Panalyzer simulations are listed in Fig. 8.

The rn:size is for adjusting the range of register renaming. It indicates how many physical registers are mapped to logical register names. Without register renaming, running a binary executable compiled for 32 registers on 64 register machine will repetitively make use of first 32 registers only. This is because the renamed register tag is used as an index to lookup the IW in DLT architecture. This technique avoids recompilation overhead when a binary executes on different architecture in terms

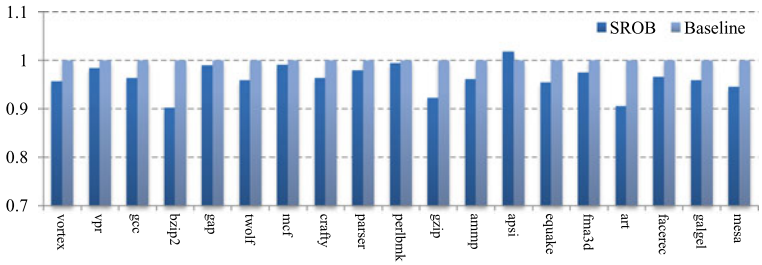

Fig. 9 IPC (ROB8SROB4, BaseROB12)

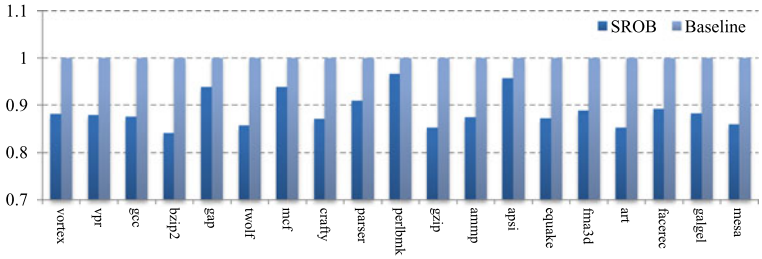

Fig. 10 Power dissipation (ROB8SROB4, BaseROB12)

of physical register size. The srob:size configures the size of the SROB buffer. The reason we set this parameter as 4 is to make the SROB size equal to the issue/commit bandwidth. If the size is more or less than the bandwidth, it may result in performance bottleneck or resource waste.

Figure 9 shows an average of IPC attained by SpecFP and SpecInt applications in simulations. The results are normalized to the baseline values. The performance degradation is due to the SROB contention. The exception is that apsi delivers even better performance while maintaining an effective power consumption level (4.9% less than the baseline power).

Figure 10 represents the evaluated power dissipation. The SROB method achieved power reduction to 11.2% of baseline power. The power reduction stems from deferred allocation and early release in the SROB. The power savings come with a performance penalty of only 3.7% on average. We note that power saving of the 11.2% is not total system savings, but a portion of the total system savings. The savings only applies to the power saving in the ROB unit. However, the overall power savings in the perspective of total system are not negligible. This is because the ROB consumes the most significant amount of energy among all structures. In fact, it takes 27.1% of total system power dissipation. At the same time, we achieved power reduction of the ROB unit to 11.2%. Therefore, the overall power savings in the perspective of total system are 3.04%.

Figure 11 describes the evaluation result of exploiting our proposed approaches on larger reorder buffer, especially 32 to 128 entries of reorder buffer configuration. This configuration reflects the real environment. B8W4 represents the issue bandwidth is eight on a baseline model. B16W4 is four wide issue of 16 reorder buffer on a baseline

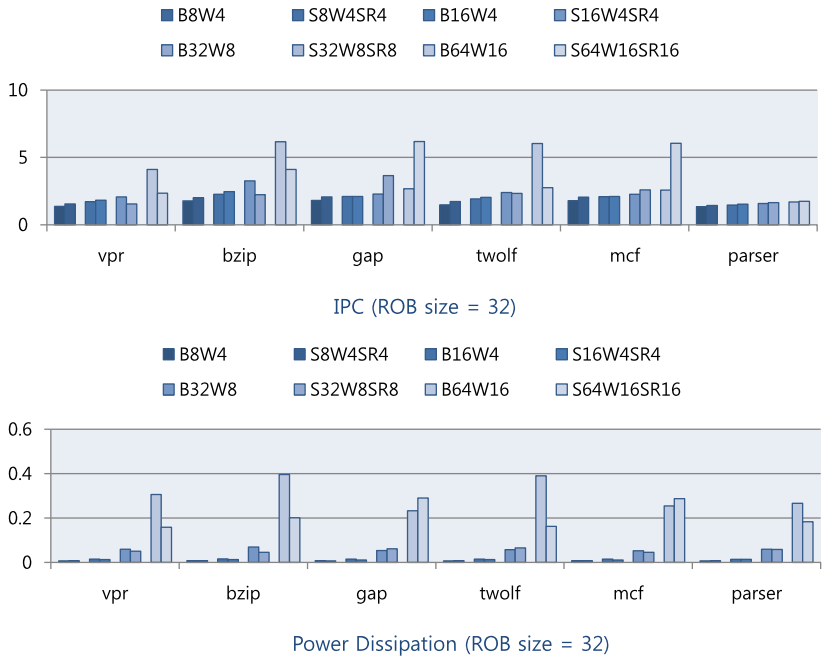

Fig. 11 Evaluation on 32, 40, 64, 80 entry reorder buffer configuration

platform. S32W8SR8 means our proposed approach that has 32 reorder buffer with eight wide issue bandwidth and eight entries of separated reorder buffer. Likewise, S64W16SR16 has in total 80 entries that consist of 64 reorder buffers and 16 separated reorder buffers. So, "SR" is the acronym of separated reorder buffer, the issue bandwidth referred to briefly as 'W', and B is the baseline approach. From Fig. 11, we see that mcf and gap show notable performance improvements with a little increase of power dissipation. In case of vpr, bzip, and twolf, there are no performance improvements, but significant reduction of power dissipation.

5 Concluding remarks

The separated reorder buffer (SROB) improves performance and reduces power dissipation by deferred allocation and early release. These two techniques result in higher resource utilization and low power consumption. Therefore, up to 3.04% of power saving comes with an average of only 3.7% performance penalty. In the current version of implementation, we limited the role of the SROB to process only dependent instructions. The power saving will be much increased if the SROB approach is extended to all types of instruction as future work. Even though there is a little performance penalty, our SROB technique for reducing the power dissipation is still meaningful, especially on the embedded computing. In the embedded environment, the energy saving is most critical due to the limited battery capacity.

Acknowledgements This research was jointly supported by the research grant of Chungbuk National University in 2011 and the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0027161 and 2011-0025975). This research was also supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2011-C1090-1131-0004).

References

1. Folegnani D, Gonzalez A (2001) Energy-effective issue logic. In: The proceedings of the IEEE international symposium on computer architecture (ISCA)
2. Nan H, Kim KK, Wang W, Choi K (2011) Dynamic voltage and frequency scaling for power-constrained design using process voltage and temperature sensor circuits. J Inf Process Syst 7(1)
3. Åsberg M, Nolte T, Pettersson P Prototyping and code synthesis of hierarchically scheduled systems using TIMES. J Converg 1(1):75–84
4. Sathappan OL, Chitra P, Venkatesh P, Prabhu M Modified genetic algorithm for multiobjective task scheduling on heterogeneous computing system. Int J Inf Technol, Commun Converg 1(2):146–158
5. Ye Y, Li X, Wu B, Li Y A comparative study of feature weighting methods for document co-clustering. Int J Inf Technol, Commun Converg 1(2):206–220
6. Fisher JD (2009) Design and implementation of low power reorder buffer. Dissertation of University of Texas at San Antonio, 77 p
7. Cristal A, Santana O, Cazorla F, Galluzzi M, Ramirez T, Pericas M, Valero M (2005) Kilo-instruction processors: overcoming the memory wall. IEEE micro
8. Kirman N, Kirman M, Chaudhuri M, Martinez J (2005) Checkpointed early load retirement. In: Proceedings of the international symposium on high-performance computer architecture (HPCA)
9. Martinez J, Renau J, Huang M, Prvulovic M, Torrellas J (2002) Cherry: Checkpointed early resource recycling in our-of-order microprocessors. In: Proceedings of the IEEE international symposium on microarchitecture (MICRO)
10. Dundas J, Mudge T (1997) Improving data cache performance by pre-executing instructions under a cache miss. In: Proceedings of the ACM international conference on supercomputing (ICS), July 1997
11. Mutlu O, Stark J, Wilkerson C, Patt YN (2003) Runahead execution: An alternative to very large instruction windows for out-of-order processors. In: Proceedings of the IEEE international symposium on high performance computer architecture (HPCA), February 2003, pp 129–140
12. Kucuk G, Ergin O, Ponomarev D, Ghose K (2003) Distributed reorder buffer schemes for low power. In: International conference on computer design (ICCD)
13. Smith JE (1985) Implementation of precise interrupts in pipelined processors. The anatomy of a microprocessor: A system perspective. IEEE CS Press, Los Alamitos
14. Brown JA, Porter L, Tullsen DM (2011) Fast thread migration via cache working set prediction. In: International symposium on high performance computer architecture (HPCA)
15. Mehrara M, Hsu PC, Samadi M, Mahlke S (2011) Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism. In: International symposium on high performance computer architecture (HPCA)
16. Sima D (2000) The design space of register renaming techniques. IEEE micro
17. Obaidat MS, Dhurandher SK, Gupta D, Gupta N, Asthana A (2010) DEESR, dynamic energy efficient and secure routing protocol for wireless sensor networks in urban environments. J Inf Process 6(3)
18. Jerbi K, Wipliez M, Raulet M, Babel M, Déforges O, Abid M Automatic method for efficient hardware implementation from RVC-CAL dataflow: A LAR coder baseline case study. J Converg 1(1):85–92