큐잉 모델을 이용한 분산된 리오더 버퍼 수퍼스칼라 프로세서의 성능분석

*백석균, 정진하, 신광식, 최상방
인하대학교 전자공학과
e-mail : s960800@gmail.com

The Performance Analysis of Distributed Reorder Buffer Superscalar Processor using Queuing Model

*Seock-Kyun Baek, Jin-Ha Jung,
Kwang-Sik Shin, Sang-Bang Choi
Dept. of Electronic Engineering
Inha University

Abstract

In all contemporary superscalar processors, the result repositories are implemented as the Reorder Buffer(ROB) slots. In such designs, the ROB is a large multi-ported structure. There are several approaches for reducing the ROB complexity in processors. The one technique relies on a distributed implementation that spreads the centralized ROB structure across the function units(FUs). Each distributed component sized to match the FU workload and with one write port and one read port on each component. We are using M/M/1 Queuing theory to determine the number of entries in each ROB component that the performance of processor depends on. Our schemes are evaluated using the simulation of CPU2000 benchmarks.

I. 서론

요즘 대부분의 프로세서(processor)는 성능을 높이기 위해 명령어를 병렬로 수행한다. 연속되는 명령어를 한 클럭에 dispatch해서 동시에 수행 하는 것을 수퍼스칼라 프로세서라고 한다. 수퍼스칼라 프로세서는 ILP를 높이기 위해 명령어를 비순차 수행을 한다(ort-of-order execution). 이때 명령어들 사이에 거짓 데이터 의존성(false data dependency)이 생길 경우 이를 해결하기 위해 Register Renaming을 사용한다. 이때 ROB(Reorder Buffer)를 이용해 Register Renaming을 한다. ROB는 비순차적으로(out-of-order) 수행된 명령어들이 정해진 순서대로 레지스터(Architecture Register Files)에 쓰여 지게 함으로서 Exception이나 인터럽트 또는 분기 예측 실패(Branch miss prediction) 발생시점에서 다시 명령어가 수행 되도록 해 준다. 즉, ROB는 비순차적으로(out-of-order) 수행된 명령어를 정해진 순차적으로(in-order) 레지스터에 쓰도록(commit) 해 준다[1].

W-way 슈퍼스칼라인 경우 ROB에는 적어도 3W개의 읽기 포트와 W개의 쓰기 포트가 필요하다. ROB에 많은 입출력 포트가 존재하기 때문에 ROB구조가 복잡해진다. 따라서 칩에서 차지하는 면적도 많아지고 전력 소모도 많아진다. 그리고 이를 이용하는 로직에 지연(critical path - delay)이 생겨 결국 프로세서의 클럭 레이트가 떨어지게 된다. 이를 해결하기 위해 ROB를 나누어 각각의 FU(Functional Unit)에 할당한다. 즉 하나의 ROB를 FU의 개수만큼 나눈 ROBC(ROB component)를 각각의 FU에 할당하는 것이다. 이렇게 했을 때 ROBC는 FCFS(First Come First Served) Queue로서 동작하므로 하드웨어를 단순하게 할 수 있고 각 FU의 부하(work load)에 따라 ROBC entry수를 조절 할 수 있다. 즉 자주 사용하는 FU에는 ROBC크기를 크게 하고 반대의 경우에는 작게 할당할 수 있다. 또 평균 6%미만의 명령어가 명령어 디스패치시에 ROB로부터 operand를 읽어 오기 때문에 이 포트도 제거함으로써 ROB를 더 간단하게 할 수 있다. 결과적으로 미미한 성능 저하를 초래하지만 하드웨어 자원을 적게 사용하므로 전력소모를 많이 줄일 수 있다[2,3].

ROB를 ROBC로 나누는 목적은 성능 저하를 최소화 하면서 ROB의 복잡도를 낮추는 것이다. 하지만 ROBC의 크기에 따라 성능 변화가 매우 크기 때문에 ROBC의 적절한 크기를 결정하는 것은 매우 어렵고 중요하다. 그러나 지금까지 논문들은 ROB를 ROBC로 나눌 때 실험 결과를 참고해 직관적으로 각 ROBC의 크기를 유추하였다. 본 논문에서는 완벽한 branch predictor와 인터럽트, Exception이 발생 하지 않는다고 가정하고 ROBC를 "큐"로 간주하여 "Queuing Theory"에 기반한 해석적 모델을 제시한다[4]. ROBC 크기와 성능과의 관계를 본 논문에서 제시한 수학적 모델을 통해 분석하여 최적화된 ROBC 크기를 결정 한다

II. 분산된 ROB의 해석적 모델

2.1. 분산된 ROB(Reorder Buffer)의 배경

명령어가 디코드 될 때 소스 오퍼랜드(Source Operand)를 어디서 가져 오는지 조사해 보면, 대부분 레지스터 파일(ARF)이나 바이패스 네트워크(Bypass Network)에서 가져오고 단지 작은 부분만 ROB에서 가져 온다. 절반 이상의 소스 오퍼랜드가 바이패스 네트워크로부터 오고 평균 6% 미만의 소스 오퍼랜드가 ROB로부터 읽혀온다[3]. 이것은 ROB에서 소스 오퍼랜드를 읽기 위한 포트를 완전히 제거해도 전체 성능의 저하가 그리 크지 않다는 것을 말한다.

ROB에서 소스 오퍼랜드를 읽기 위한 포트를 완전히 제거한다는 것은 읽기 포트를 줄이는 것과는 근본적으로 다르다. 포트를 없애는 것은, 첫째 ROB에서 Critical Path를 제거 하는 것이 되고 두 번째로 ROB 설계상에서 복잡도가 많이 줄어드는 것을 의미한다. 비록 ROB에서 소스 오퍼랜드를 읽기 위한 포트가 없어졌다 해도 W-way 수퍼스칼라 프로세서에서 FU에서 연산된 결과값을 저장하는 W개의 쓰기 포트와 결과값을 ARF에 적을 때 필요한 W개의 읽기 포트는 제거 할 수 없다. ROB를 각 FU의 개수만큼 나누어 분산된 ROBC(ROB component)를 만들면 ROB는 다음과 같이 더 간단해 질 수 있다.

- 각 ROBC는 FU의 부하에 따라 적절한 크기로 만들 수 있다.
- 각 FU는 한 싸이클에 한 개의 결과만 저장을 하므로 ROBC에는 한 개의 쓰기 포트만으로 충분하다. 즉, 하나의 ROB를 사용할 때처럼 여러 개의 결과가 동시에 쓰지 않기 때문에 이를 중재하는 로직이나 버퍼등의 추가 회로가 필요하지 않다.
- ROBC에 있는 결과를 레지스터(ARF)에 쓸 때 한 개의 읽기 포트만 필요하다.

CPU2000 benchmrks에서 4-way 수퍼스칼라 프로세서 경우 평균 1.7%의 성능저하가 생긴다[3].

2.2. 분산된 ROB(Reorder Buffer)

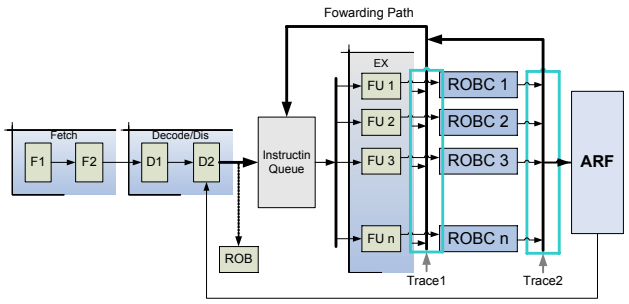

그림 1. 분산된 ROB 수퍼스칼라 프로세서의 구조

그림1은 각 FU에 하나의 ROBC를 할당한 분산된 구조를 보여주는 블록 다이어그램이다. 그림에서 ROB는 분산된 ROBC의 포인터를 갖고 있어서 ROBC에 있는 연산 결과들이 프로그램 순서대로 commit(or retire)될 수 있도록 해 주고 Exception이나 인터럽트의 발생 시 정확한 상태에서 프로그램이 다시 실행될 수 있게 해 준다[1]. 분산된 구조에서 ROB는 W-way 수퍼스칼라 프로세서에서 Dispatch시의 프로그램 순서대로 ROBC의 포인터를 만드는데 필요한 W개의 쓰기 포트와 ROBC의 연산결과를 ARF에 적을 때 포인터를 가져올 W개의 읽기 포트만으로 충분하다. 또는 1개의 읽기, 쓰기 포트로 구성하기 위해서 ROB는 한번에 W개의 포인터를 저장할 entry를 가지면 된다. ROB에서 각 포인터는 FU의 id와 ROBC의 주소를 저장하게 된다.

그림1에서 각 ROBC는 FU의 연산결과를 적기 위한 하나의 쓰기 포트와 ROBC의 결과값을 ARF에 적을 때 필요한 읽기 포트로 구성되어 있다. 그리고 각 ROBC의 크기는 FU의 작업부하에 따라 적당한 크기를 정할 수 있다. 본 논문에서 최적화된 ROBC의 크기를 결정할 수 있는 M/M/1 Queuing 모델을 제시한다.

2.3. ROBC의 해석적 모델

M/M/1 큐잉 모델은 많은 분야에서 시스템의 성능을 예측하는데 자주 사용되는 모델링 방법이다. 하나의 서버가 대기 큐(Waiting Queue)에 있는 Customer를 처리하는 M/M/1 큐잉 모델을 가정하자. 여기서 Customer interarrival 과 서비스 타임은 지수 분포(exponentially distributed)를 가져야 한다[4].

다음은 Poisson arrival process를 갖는 시스템의 특

성을 보여준다[4].

- 일정 시간 사이에 발생한 사건의 수는 다른 시간대의 발생한 사건의 수와 독립이다. 즉, Poisson process는 무기억(no Memory)이다.
- 매우 짧은 시간에 하나의 사건이 발생할 확률은 시간에 비례한다.
- 짧은 시간동안 한개 그 이상의 사건이 발생할 확률은 무시할 수 있다.

하나의 ROBC를 놓고 볼 때 ROBC는 하나의 쓰기 포트와 하나의 읽기 포트를 갖고 있다. 여기서 Exception과 인터럽트가 발생하지 않고 완벽한 Branch predictor를 갖고 있다고 가정하면 ROBC는 FCFS Queue로서 간주될 수 있다. ROBC에는 하나의 읽기 포트와 하나의 쓰기 포트가 있고 매우 짧은 시간을 클럭의 주기라고 한다면 다음과 같은 결론을 내릴 수 있다.

- 매 클럭당 ROBC에 값을 쓰거나 읽는 것은 이전의 읽기 쓰기와는 독립이다.
- 매 클럭당 한 번의 읽기나 쓰기를 할 확률은 클럭수에 비례한다.
- ROBC에서 한 클럭에 최대 한 번의 읽기와 쓰기 밖에 할 수 없다.

따라서 ROBC에 연산 결과값을 읽고 쓰는 것은 Poisson arrival process이고 M/M/1 Queueing 시스템으로 모델링 할 수 있다.

단위 클럭당 FU에서 연산을 마치고 ROBC에 결과값을 쓰거나 ROBC에서 commit되어 ARF에 값을 쓰는 것은 M/M/1 Queueing 시스템에서 Arrival process와 Service로 볼 수 있다. 그리고 ROBC에서 한 클럭에 최대 한 개의 명령어 결과를 ARF에 적기 때문에 하나의 서버를 갖는다고 볼 수 있다.

단위 시간(클럭)당 ROBC에 값이 쓰여질(average arrival rate) 확률 λ와 ARF에 commit(average service rate)될 확률 μ를 안다면 ROBC에서 service rate(ρ)와 평균 ROBC에 있는 값들의 개수(N_s)를 알 수 있다[5].

$$\rho = \frac{\lambda}{\mu} \tag{1}$$

$$N_s = \frac{\rho}{1-\rho} \tag{2}$$

식(2)에서 N_s는 ROBC에서 commit되기를 기다리는 값들의 평균 개수이다. 즉, ROBC의 크기는 최소한 N_s보다 커야 ROBC가 가득차지 않을 수 있고 성능저하가 최소화 될 수 있다. ROBC의 크기를 크게 할수록 성능 저하는 줄어들게 된다. 하지만 하드웨어 비용은 늘어나게 된다. 성능 저하는 거의 없으면서 ROBC의 크기는 최소가 되도록 정해야 한다. ROBC에 써진 값들의 개수(N_s)가 x보다 크거나 같을 확률은 수식(3)과 같다.

$$P\{N_s \geq x\} = \rho^{x} \tag{3}$$

ROBC가 가득찰 확률을 10% 이내로 하기 위해서는 ROBC의 크기가 다음과 같은 값을 가져야 한다.

$$\rho^{x} \leq 0.1$$

$$x > \frac{\log(10^{-1})}{\log(\rho)} = 1/\log(\frac{1}{\rho}) \tag{4}$$

여러 가지 응용 프로그램을 수행하고 이것을 추적하여 ROBC에서 Arrival rate(λ)와 Service rate(μ)를 구하고 (4)번 수식을 이용하여 ROBC의 적절한 크기를 구하고 여기서 구한 ROBC의 크기로 시뮬레이션 하여 기존의 하드웨어와 성능을 비교하고 전체 ROBC의 크기에 따른 성능 변화를 확인해 본다.

표1. ROBC에 있는 결과값 개수의 평균과 최댓값 그리고 Arrival rate(λ)과 평균 Service time(μ)과 x

	Int ADD/SUB				Int MUL/DIV				FP ADD/SUB				FP MUL/DIV				LOAD			
	ρ	N_s	max	ave	ρ	N_s	max	ave	ρ	N_s	max	ave	ρ	N_s	max	ave	ρ	N_s	max	ave
bzip2	0.88	7.15	18.00	10.30	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.89	8.14	27.00	9.93
gcc	0.67	2.02	20.00	2.78	0.42	0.72	5.00	0.05	0.44	0.80	1.00	0.00	0.00	0.00	2.00	0.00	0.83	5.02	30.00	5.40
parser	0.77	3.27	13.00	4.00	0.62	1.64	3.00	0.25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.90	9.16	29.00	10.2
perl	0.69	2.23	17.00	2.75	0.56	1.26	6.00	0.24	0.66	1.90	3.00	0.10	0.53	1.12	10.00	0.60	0.87	6.84	28.00	7.95
vpr	0.84	5.23	15.00	6.32	0.40	0.67	2.00	0.02	0.71	2.49	4.00	0.26	0.65	1.84	11.00	0.35	0.93	12.6	28.00	13.1
art	0.78	3.50	11.00	4.51	0.00	0.00	0.00	0.00	0.74	2.83	8.00	0.49	0.67	2.01	16.00	1.38	0.88	7.31	26.00	8.31
equake	0.81	4.38	12.00	5.91	0.00	0.00	4.00	0.06	0.56	1.26	0.00	0.57	0.00	0.00	0.00	0.00	0.85	5.84	24.00	6.88
mesa	0.67	2.06	11.00	2.94	0.65	1.83	9.00	0.49	0.75	2.97	6.00	0.75	0.67	2.06	11.00	1.10	0.88	7.10	27.00	7.60
average	0.76	3.73	14.63	4.94	0.33	0.77	3.63	0.14	0.48	1.53	2.75	0.27	0.31	0.88	6.25	0.43	0.88	7.94	27.38	8.67
$x = 1/\log\frac{1}{\rho}$	8.53				2.08				3.16				1.99				17.91			

III. 시뮬레이션 및 성능분석

3.1 시뮬레이션 배경

프로세서의 성능을 테스트하기 위해 널리 사용되는 Simplescalar simulator[6]를 이용하였다. 그리고 CPU2000에서 사용하는 테스트 코드를 수행하여 성능을 측정 하였다. Simplescalar로 구현한 프로세서는 4-way issue, commit을 하고 각각 4개의 정수/실수 덧셈뺄셈기와 각각 1개의 정수/실수 곱셈나눗셈기와 2개의 메모리 읽기쓰기 FU를 갖는다. 실험에서 8-2-4-2-18로 표기된 것은 정수 덧셈뺄셈기, 정수 곱셈나눗셈기, 실수 덧셈뺄셈기, 실수 곱셈나눗셈기, 메모리 읽기쓰기 FU에 할당된 ROBC를 나타낸다.
4개의 정수연산 코드(bzip2, gcc, parser, per)와 4개의 실수연산 코드(vpr, art, equake, mesa)를 이용하여 실제 ROBC에 있는 값들의 평균 개수와 ROBC의 크기를 결정하기 위한 λ, μ ρ의 값을 구하였다. 여기서 구한 λ, μ ρ의 값을 식(4)에 대입해 적절한 ROBC의 크기를 결정한다. 그리고 그렇게 결정한 값이 ROBC에서 평균 존재하는 값들의 개수와 비교해 본다. 마지막으로 ROBC의 크기에 따른 성능의 변화를 확인한다.

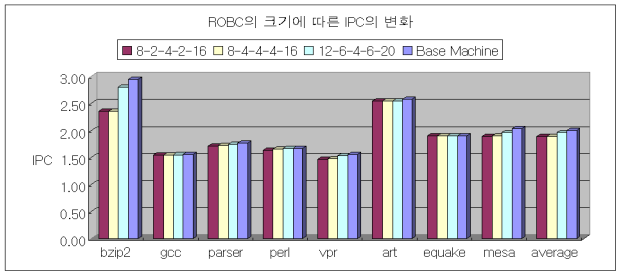

그림2. ROBC의 크기에 따른 IPC의 변화

3.2. 실험 결과

각 ROBC가 매우 크다고 가정하고 실수 정수 각 4개의 테스트 코드를 돌려서 ROBC에 있는 연산 결과 값들 개수의 평균과 최댓값을 기록하고 그림1의 Trace1에서 Arrival rate(λ)과 Trace2에서 평균 Service time(μ)을 구한다. Arrival rate(λ)과 평균 Service time(μ)을 이용 얻은 값(ρ)을 식(4)에 대입하면 각 ROBC의 크기 x값을 구할 수 있다. 구한 결과 값은 표1과 같다. x값을 이용해 2의 배수에 가장 가까운 값으로 실험 하였다. FU에 할당되는 각 ROBC의 크기가 그림2에 제시한 값일 때 각 시스템의 IPC (Instruction Per Cycle)는 그림2와 같다. ROBC의 값이 커질수록 성능이 좋아 지지만 어느 정도 이상이 되면 성능 향상에 큰 도움을 주지 못한다. 따라서 하드웨어를 최대한 절약하면서 최대한의 성능을 낼 수 있는 ROBC의 크기를 결정해야 한다. 실험에서 보듯이 각 ROBC크기는 8-2-4-2-18일 때 성능저하와 하드웨어 자원을 최소로 할 수 있다.

V. 결론 및 향후 연구 방향

ROBC의 크기에 따라 성능 변화가 매우 크기 때문에 ROBC의 적절한 크기를 결정하는 것은 매우 중요하다. 그러나 지금까지 논문들은 ROB를 ROBC로 나눌 때 실험 결과를 참고해 직관적으로 각 ROBC의 크기를 유추하였다. 본 논문에서는 분산된 ROB를 갖는 수퍼스칼라 프로세서의 성능을 분석하고 적당한 ROBC의 값을 찾기 위하여 M/M/1 Queuing 모델을 이용하였다. 이러한 해석적 모델을 사용해 얻을 수 있는 장점은 프로그램의 실행에 필요한 ROBC의 크기를 예측할 수 있으므로 자원의 낭비를 줄일 수 있는 것이다. 본 논문에서는 인터럽트와 Exception이 발생하지 않는다는 가정에 해석적 모델을 만들었지만 이를 조금 더 발전시켜 인터럽트나 Exception이 발생할 때의 상황을 좀 더 다각적으로 해석하기 위해 연구 중이다..

참고문헌

[1] Gurindar S. and Sriram Vajapeyam "Instruction Issue Logic for High-performance, Interruptable Multiple Functional unit, Pipelined Computer" IEEE Transaction on Computers,39(3):349-359, March 1990.

[2] G. Kucuk, D. Ponomarev, and K. Ghose, “Low-Complexity Reorder Buffer Architecture,” Proc. Int’l Conf. Supercomputing(ICS ’02), pp. 57-66, 2002.

[3] Gurhan Kucuk, Dmitry V. Ponomarev, Oguz Ergin, and Kanad Ghose "Complexity-Effective Reorder Buffer Designs for Superscalar Processors" IEEE TRANSACTIONS ON COMPUTERS, VOL. 53, NO. 6, JUNE 2004

[4] Bertsekas&Gallager, Data Networks, Prentice Hall, 2002.

[5] Hassan & Jain, High Performance TCP/IP Networking, Prentice Hall, 2000.

[6] Burger, D. and Austin, T. M., “The SimpleScalar tool set : Version 2.0”, Tech. Report, Dept. of CS, Univ. of Wisconsin-Madison, June 1997