비순차이슈 수퍼스칼라 프로세서에서 리오더버퍼의 성능개선

(Performance Improvement of Reorder Buffer in Out-of-order Issue Superscalar Processors)

장 문 석 † 이 정 우 †† 최 상 방 †††
(Moon Suk Jang) (Jung Woo Lee) (Sang Bang Choi)

요 약 리오더버퍼는 명령어를 비순차로 이슈하는 수퍼스칼라 파이프라인에서의 명령어 실행을 순차적으로 완료하는데 사용된다. 본 논문에서는 리오더버퍼에 의하여 발생할 수 있는 명령어의 스테그네이션(stagnation)을 효율적으로 제거시킬 뿐만 아니라 리오더버퍼의 크기를 감소시킬 수 있는 쉘터버퍼를 사용한 리오더버퍼 구조를 제안하였다. 시뮬레이션을 수행한 결과 리오더버퍼의 엔트리 개수가 8개에서 32개 사이일 때 쉘터버퍼는 단지 1개 또는 2개만 사용하여도 뚜렷한 성능 향상을 얻을 수 있음을 보여준다. 쉘터버퍼를 4개 사용했을 때는 2개만 사용했을 경우와 비교하여 주목할만한 성능 향상은 없었다. 이는 쉘터버퍼를 2개만 사용하여도 대부분의 스테그네이션을 제거하는데 충분함을 보여준다. 실행율의 손실이 없는 상태에서 2개의 쉘터버퍼를 사용하면 Whetstone 벤치마크 프로그램에서는 44%, FFT 벤치마크 프로그램에서는 50%, FM 벤치마크 프로그램에서는 60%, Linpack 벤치마크 프로그램에서는 75%의 리오더버퍼의 크기를 줄일 수 있었다. 쉘터버퍼를 사용했을 때 수행 시간 역시 Whetstone에서는 19.78%, FFT에서는 19.67%, FM에서는 23.93%, Linpack에서는 8.65%의 성능 향상을 얻을 수 있었다.

Abstract The reorder buffer is usually employed to maintain the instruction execution in the correct order for a superscalar pipeline with out-of-order issue. In this paper, we propose a reorder buffer structure with shelter buffer for out-of-order issue superscalar processors not only to control stagnation efficiently, but also to reduce the buffer size. Simulation results show that we can get remarkable performance improvement with only one or two shelter buffers when the size of reorder buffer is between 8 and 32. For the shelter buffer of size 4, there is not a noticeable performance improvement compared to that of size 2, which means that the shelter buffer of size 2 is large enough to handle most of the stagnation. If the shelter buffer of size 2 is employed, we can reduce the reorder buffer by 44% in Whetstone, 50% in FFT, 60% in FM, and 75% in Linpack benchmark program without loss of any throughput. Execution time is also improved by 19.78% in Whetstone, 19.67% in FFT, 23.93% in FM, and 8.65% in Linpack benchmark when the shelter buffer is used.

1. 서 론

프로그램의 수행 속도를 향상시키기 위해 여러 명령어를 동시에 처리할 수 있는 다양한 프로세서 구조들이 개발되었다. 이러한 프로세서 구조들 중 수퍼스칼라 프로세서(superscalar processor)는 기존의 구조와 기계어 수준의 호환성(binary-level compatibility)을 유지하면서 여러 명령어를 동시에 수행할 수 있기 때문에 널리 사용되어진다. 수퍼스칼라라는 용어는 과학적 컴퓨팅에서 일반적인 벡터나 배열의 병렬 처리와 구별해서 스칼라 데이타에 대해 동시에 여러 연산을 수행함을 의미한다[1].

수퍼스칼라 프로세서에서는 여러 명령어들이 파이프

† 비 회 원 : 인하대학교 전자공학과
sirius93@hanmail.net
†† 비 회 원 : 인하대학교 전자공학과
nemesisq@chollian.net
††† 종신회원 : 인하대학교 전자공학과 교수
sangbang@inha.ac.kr
논문접수 : 2000년 7월 12일
심사완료 : 2000년 12월 14일

라인의 디코드단(decode stage)에서 실행단(execution stage)으로 전달된 후 여러 개의 실행유닛에 의해 동시에 수행된다. 실행유닛에 의해 생성된 결과들은 마치 하나의 파이프라인 프로세서에 의해 실행된 것처럼 메모리나 레지스터에 갱신된다. 디코드단에서 실행단으로의 명령어 전달을 명령어 이슈(instruction issue)라고 하며, 명령어의 이슈 방법은 순차이슈(in-order issue)와 비순차이슈(out-of-order issue)로 분류할 수 있다. 순차이슈에서는 명령어들이 캐쉬로부터 인출(fetch)되고 디코드된 순서로 이슈된다. 순차이슈는 프로그램 자체의 명령어 수준 병렬성(instruction-level parallelism)을 충분히 이용할 수 없기 때문에 만족할 만한 성능향상을 얻을 수 없다. 그 이유는 명령어들 사이의 자원 충돌(resource conflict)이 연속적으로 명령어들이 이슈되는 것을 막으며, 또한 명령어 수행의 흐름은 제어와 데이타 의존(control and data dependency)에 의해 파이프라인 정지(stall)가 빈번히 발생하기 때문이다. 따라서 수퍼스칼라 프로세서는 성능 향상을 위해 이러한 충돌이나 의존이 발생한 명령어의 한계를 넘어 실행될 수 있는 다음 명령어를 찾아야 한다. 따라서 비순차이슈 수퍼스칼라 프로세서에서는 독립적인 명령어들을 명령어 윈도우(instruction window)라고 하는 명령어 풀(pool)로부터 선택하여 명령어가 인출되어 디코딩되는 순서와 관계없이 이슈하여 프로그램을 수행한다.

응용 프로그램이 어느 정도의 명령어 수준 병렬성을 포함한다면 비순차이슈 방법은 현저한 성능향상을 얻을 수 있다. 그러나 순차이슈와 비교하여 이 방법은 분기예측(branch prediction)의 정확도에 많은 영향을 받고 독립적인 명령어들을 검색하기 위한 추가적인 하드웨어가 필요하다. 즉, 비순차이슈로 수행되기 위한 독립적인 명령어의 선택은 명령어 윈도우를 필요로 하며, 이 명령어 윈도우 크기는 프로세서의 성능에 상당한 영향을 미친다. Jouppi는 주어진 프로그램에 대해 명령어 수준 병렬성을 측정했고 일반적인 스칼라 프로세서에 비해 수퍼스칼라와 수퍼파이프라인 프로세서의 상대적인 성능 향상을 비교했다[2]. 또한 Smith는 복잡한 수학 연산이 없는 프로그램은 사이클 당 2개의 명령어를 수행할 만큼 충분한 병렬성이 없다고 결론지었다[3].

비순차이슈에서 명령어들은 프로그램의 실행 순서대로 이슈되지 않고 각 연산은 해당 실행유닛에서 서로 다른 지연시간(latency)을 가지고 있기 때문에 명령어의 종료 역시 비순차적으로 이루어진다. 그러나 비순차 종료는 명령어의 실행 결과를 내부 버스를 통해 레지스터 파일(register file)에 쓸 때 그 순서를 조정할 필요가 있으며, 또한 인터럽트 처리를 매우 어렵게 한다. 따라서 여러 실행유닛에 의해 동시에 생성된 결과들은 순차적으로 프로그램이 수행된 것처럼 메모리나 레지스터에 갱신되는 것이 바람직하다. 리오더버퍼는 디코드된 명령어의 실행 상태를 기록하여 종료된 명령어의 결과들이 올바른 순서로 시스템에 쓰여지도록 한다. 즉, 실행이 끝난 명령어는 먼저 등록된 순서로 리오더버퍼를 빠져나오므로 명령어가 인출되어 디코드된 순서대로 명령어의 실행 결과가 시스템에 갱신된다.

그러나 긴 지연시간을 갖는 부동소수점 연산 명령어에 의해서 다음 명령어들이 실행유닛에서 동작이 종료되었음에도 불구하고 이 명령어가 수행이 완료될 때까지 기다려야 하는 명령어 정체현상이 발생한다. 본 논문에서 이를 스테그네이션(stagnation)이라 한다. 리오더버퍼는 명령어 윈도우에 저장된 명령어, 실행 유닛에서 수행중인 명령어, 또한 연산을 마쳤으나 레지스터 파일에 결과를 갱신하지 못한 명령어의 상태를 기록하고 있어야 한다. 따라서 리오더버퍼에서 스테그네이션을 일으킨 명령어가 시스템 성능 저하를 유발하지 못하도록 리오더버퍼는 매우 커야한다.

기존의 연구에서 Inoue와 Takeda는 최대 명령어 이슈능력, 정수와 부동소수점 ALU들의 특성, 분기유닛 같은 다양한 파라미터들을 사용하여 이슈율을 측정하였으며, 최적의 수퍼스칼라 프로세서의 특성을 제시하였다[4]. Butler는 그의 시뮬레이션에서 레지스터 변경(register renaming), 모험적 수행(speculative execution), 비순차이슈를 사용하면 2.0에서 5.8사이의 명령어 이슈율을 얻을 수 있다고 주장하였다[5]. Wallace는 고속의 리오더버퍼를 구현했지만 스테그네이션에 의하여 발생하는 리오더버퍼의 구조적 문제는 해결할 수 없었다[6][7][8].

본 논문에서는 명령어의 스테그네이션을 해결하고 리오더버퍼의 사용을 효과적으로 줄이기 위한 새로운 리오더버퍼 구조를 제안한다. 제안된 리오더버퍼 구조에서 긴 지연시간을 갖는 명령어는 다른 명령어들의 진행을 막지 않기 위해 쉘터버퍼로 옮겨진다. 이 쉘터버퍼는 비순차로 이슈하는 수퍼스칼라 프로세서 내의 파이프라인에서 명령어들의 균일하게 흐르게 한다.

4개의 벤치마크 프로그램에 대한 명령어 트레이스를 이용하여 쉘터버퍼를 사용한 리오더버퍼를 시뮬레이션 하였다. 대부분의 경우 리오더버퍼의 엔트리 개수가 8개에서 32개 사이일 때 쉘터버퍼는 단지 1개 또는 2개만 사용하여도 현저한 성능 향상을 얻을 수 있었다. 쉘터버퍼를 4개 사용했을 때는 2개만 사용했을 경우와 비교하

여 주목할만한 성능 향상은 없었다. 실행율의 손실이 없는 상태에서 2개의 쉘터버퍼를 사용하면 Whetstone 벤치마크 프로그램에서는 44%, FFT 벤치마크 프로그램에서는 50%, FM 벤치마크 프로그램에서는 60%, Linpack 벤치마크 프로그램에서는 75%의 리오더버퍼의 크기를 줄일 수 있었다. 쉘터버퍼를 사용했을 때 수행시간 역시 Whetstone에서는 19.78%, FFT에서는 19.67%, FM에서는 23.93%, Linpack에서는 8.65%의 성능 향상을 얻을 수 있었다.

본 논문은 다음과 같이 구성된다. 2장에서는 비순차이슈 수퍼스칼라에서 사용되는 명령어 윈도우와 리오더버퍼의 동작을 설명한다. 리오더버퍼에서 발생할 수 있는 스테그네이션과 이를 제거하기 위한 쉘터버퍼의 기능은 3장에서 논의한다. 4장에서는 수퍼스칼라의 시뮬레이션 모델과 시뮬레이션을 수행한 결과 및 분석을 설명하고, 5장에서는 본 논문의 결론을 맺는다.

2. 리오더버퍼

2.1 명령어윈도우

응용프로그램에서 동시에 이슈될 수 있는 평균 명령어 개수를 명령어 수준의 병렬성이라 한다. 주어진 프로그램에서 데이타 의존도와 분기 명령의 빈도는 명령어 병렬성을 결정한다. 파이프라인의 구조에 따라 각 명령어의 지연시간은 다르며, 데이타 의존도 역시 수퍼스칼라 프로세서의 구조와 밀접한 관계가 있다. 컴퓨터의 병렬성(machine parallelism)은 수퍼스칼라 프로세서가 명령어 병렬성을 얼마나 이용할 수 있는지 나타내는 척도이다. 컴퓨터의 병렬성은 동시에 인출되어 수행될 수 있는 명령어의 개수와 프로세서가 서로 독립적인 명령어들을 찾기 위하여 사용하는 메커니즘에 의해 결정된다. 수퍼스칼라 프로세서가 최대한의 성능을 얻기 위해서는 명령어 수준의 병렬성과 컴퓨터의 병렬성을 효율적으로 이용하여야 한다.

수퍼스칼라 프로세서에서 명령어를 이슈하는 방법은 크게 두 가지로 나눌 수 있다. 첫 번째 방법은 프로그램의 실행 순서대로 명령어를 이슈하는 방법으로 순차이슈라 한다. 이러한 순차이슈를 사용하는 수퍼스칼라는 데이타 의존이나 자원 충돌로 인하여 수행이 정지된 명령어의 다음 명령어들은 실행될 수 없다. 이러한 방법은 간단한 논리회로로 구현 가능하나 일반적으로 이슈율이 매우 낮다. 두 번째 방법은 프로그램 내에서의 실행 순서와 관계없이 독립적인 명령어를 찾아 이슈하는 방법으로 비순차이슈라 한다. 최근 설계되는 대부분의 고성능 마이크로 프로세서는 비순차이슈를 사용한다. 순차이슈와 달리 비순차이슈는 명령어가 프로그램 내에서 수행되는 순서와 관계없이 어떤 조합의 독립적인 명령어도 동시에 이슈할 수 있다. 따라서 비순차이슈는 순차이슈와 비교하여 이슈율이나 전체적인 성능이 월등히 우수하다.

그러나 수퍼스칼라 프로세서가 명령어를 비순차로 이슈하기 위해서는 디코딩된 여러 명령어를 검색하여 서로 독립된 명령어를 찾아야 한다. 일반적으로 비순차이슈 프로세서들은 디코딩단과 실행단을 분리하여 명령어 윈도우라고 불리는 버퍼를 둔다[9]. 명령어 윈도우의 크기와 검색방법은 실행유닛과 함께 비순차이슈 수퍼스칼라의 명령어 이슈율을 결정하는 주된 요소이다. 명령어 윈도우의 크기가 커진다면 윈도우에서 더 많은 독립된 명령어를 찾을 수 있다. 명령어 윈도우의 크기가 커질수록 프로세서는 더 많은 명령어에 대하여 독립성을 검색할 수 있으므로 이슈율이 증가하나, 의존도를 검색하여야 하는 명령어의 수가 증가하여 (명령어 윈도우의 크기가 n 인 경우 $O(n^2)$의 비교를 수행하여야 함) 회로의 복잡도는 급격히 증가하고 검색에 소요되는 시간이 길어지게 되어 프로세서의 성능을 제한하는 원인이 될 수 있다. 따라서 수퍼스칼라 프로세서의 성능은 명령어 윈도우의 크기에 비례하여 향상되는 것이 아니기 때문에 검색 방법과 실행유닛의 개수를 고려하여 윈도우의 크기를 결정하여야 한다. 또한 명령어 윈도우 내에서 한 분기명령의 분기예측이 틀리는 경우, 분기명령어 이후의 명령어들은 비록 이슈된다 하더라도 그 수행을 완료할 수 없으므로 결과적으로 명령어의 크기를 줄이는 효과를 가져온다[10].

2.2 리오더버퍼

본 연구에서는 명령어가 수행을 마친 상태를 명령어의 수행이 종료(completion)되었다고 하고, 종료된 명령어의 결과가 메모리나 레지스터에 갱신(update)되는 것을 명령어가 완료(commit)되었다고 구분하여 정의한다. 프로세서가 서로 독립적인 명령어를 검색하여 비순차적으로 명령어를 이슈하는 프로세서는 명령어 종료 역시 비순차적으로 이루어져 정확한 예외상황(exception) 처리를 어렵게 한다. 즉, 비순차로 명령어들이 이슈되거나 명령어들이 실행되는 시간이 다른 경우 인터럽트 처리가 매우 복잡하여 진다. 정확한 인터럽트 처리(precise interrupt handling)를 위해서는 명령어의 상태를 저장하는 프로세서 내의 하드웨어를 사용하여 명령어의 수행이 이미 종료되었더라도 먼저 이슈된 명령어의 연산이 끝날 때까지 연산결과를 임시로 저장하여 마치 순차적으로 명령어가 완료되는 것처럼 보이게 해야한다. 이

러한 목적으로 사용되는 하드웨어가 리오더버퍼(reorder buffer)이다. 이 방법을 사용하면 인터럽트가 발생하는 경우 프로세서를 인터럽트가 발생하기 이전의 상태로 되돌리는 것이 가능하다. 그러나 리오더버퍼를 사용할 경우 명령어 지연시간의 차이가 클수록 버퍼의 크기가 커져야 하므로 하드웨어 비용이 증가하며, 명령어가 수행을 종료한 후 그 완료를 기다리는 중에도 다른 명령어의 이슈가 계속될 수 있도록 버퍼에 저장된 결과가 우회경로(bypass)를 통하여 사용될 수 있도록 하는 회로가 필요하다.

그림 1은 본 논문의 기본 모델로 사용하는 비순차이슈 수퍼스칼라 프로세서의 파이프라인 구조를 보여주고 있다. 앞에서 설명하였듯이 명령어 윈도우는 디코드단과 실행단 사이에 위치하며, 명령어들은 명령어 윈도우로부터 실행유닛으로 비순차적으로 이슈되고 또한 비순차적으로 실행유닛에서 수행된다. 그러나, 실행유닛에 의하여 생성된 결과들은 리오더버퍼의 도움으로 명령어가 프로그램 내에서 수행되는 순서대로 메모리나 레지스터에 갱신된다. 이러한 리오더버퍼는 FIFO 큐(first-in first-out queue)로 구현될 수 있다.

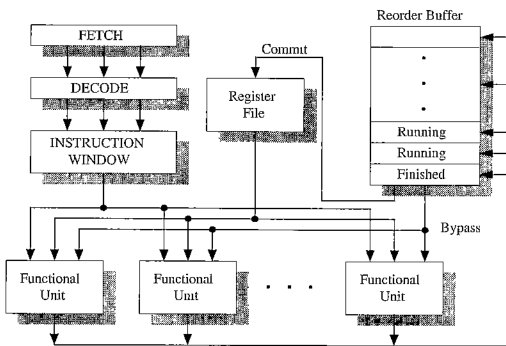


그림 1 비순차이슈 수퍼스칼라 프로세서의 기본 모델

리오더버퍼가 명령어 이슈와 관련된 타이밍 측면에서 결정적인 부분은 아닐지라도 구현시 많은 하드웨어가 필요하고 부동소수점 연산 같은 긴 지연시간을 갖는 명령어가 리오더버퍼의 출구를 막는다면 스테그네이션이 발생하기 때문에 프로세서 성능에 있어서는 매우 중요하다. 이러한 스테그네이션은 리오더버퍼를 크게 설계하거나 단순히 비순차완료를 허용하여 해결할 수 있으나, 리오더버퍼의 크기를 크게 하면 칩(chip)의 다이(die) 면적이 넓어지고 제어 로직은 더욱 복잡하여 진다. 더구나 주로 과학 연산을 수행하는 프로그램들만이 긴 지연시간을 갖은 부동소수점 명령어를 포함하므로 리오더버퍼를 단순히 크게 하는 것은 매우 비효율적인 해결방법이다. 또한 명령어가 비순차로 완료하는 방법은 명령어의 결과가 프로그램에서의 실행 순서와 관계없이 시스템에 갱신되므로 페이지 폴트(page faults)와 같은 예외상황 처리를 어렵게 한다.

명령어 윈도우와 리오더버퍼의 크기를 적절히 결정하는 것은 수퍼스칼라 프로세서 설계에서 가장 중요하게 고려되어야 할 요소 중의 하나이다. 명령어 윈도우의 크기를 증가시키면 수퍼스칼라 프로세서는 독립된 명령어를 위해 더 많은 디코드된 명령어들을 검색할 수 있으며, 리오더버퍼를 크게 하면 긴 실행 지연시간을 가지는 명령어에 의한 스테그네이션을 줄일 수 있다. 한편, 리오더버퍼의 크기가 작고 리오더버퍼의 출구 부분에 지연시간이 긴 명령어가 위치한다면 잇따르는 다음 명령어들은 완료되지 못하며, 리오더버퍼가 포화되었을 때는 더 이상의 명령어들이 이슈되지 못한다. 곱셈 명령어나 나눗셈 명령어와 같은 부동소수점 연산 명령어들은 할당된 실행유닛에서의 지연시간이 길다. 그리고 일부 과학 연산을 수행하는 프로그램들은 많은 양의 부동소수점 연산 명령어를 가지고 있다. 본 논문에서는 리오더버퍼의 지나친 하드웨어 비용을 줄이고 명령어의 흐름을 일정하게 유지하여 프로세서의 성능을 향상시킬 수 있는 쉘터버퍼를 사용한 리오더버퍼를 제안하고 그 성능을 분석한다.

3. 리오더버퍼의 스테그네이션

3.1 쉘터버퍼를 사용한 리오더버퍼

그림 2와 같이 긴 지연시간을 갖는 명령어가 리오더버퍼의 출구에 위치하여 실행중이고 오퍼랜드의 검사결과 인터럽트가 발생하지 않을 것이 확실한 부동소수점 연산 명령어라고 가정하자. 이 경우에 정체현상을 유발한 0번 엔트리(entry)의 명령어를 예비버퍼(쉘터버퍼)로 옮겨서(instruction sheltering) 실행하여 수행이 종료된 명령어들(1, 2, 3)을 완료(commit)시킨다면 명령어의 정체현상을 해결할 수 있을 것이다. 여기서 쉘터버퍼로 옮길 수 있는 명령어를 인터럽트가 발생할 가능성이 없는 명령어로 제한하지 않더라도 리오더버퍼의 크기가 아주 작지 않은 한 모든 명령어는 리오더버퍼에 있는 동안 인터럽트 발생여부를 알 수 있어, 쉘터링되는 명령어는 자연히 실행 지연시간이 긴 정수나 실수 연산으로 제한된다. 따라서 어떠한 예외상황이 발생하는 경우에도 원래의 프로그램 흐름을 깨지 않고 정확한 인터럽트 서비스가 가능할 것이다.

일반적으로 RISC(reduced instruction set computer)

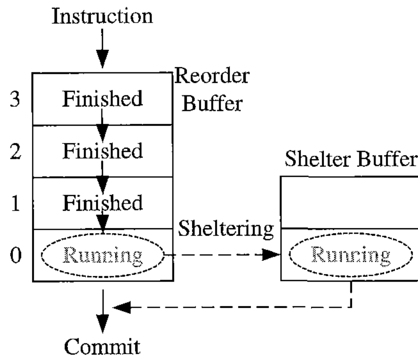


그림 2 쉘터버퍼를 사용한 리오더버퍼의 구조

프로세서에서 수행되는 실수연산들은 주로 레지스터 레벨에서 이루어지고, 언더플로우(underflow)나 오버플로우(overflow) 등의 연산 과정에서 발생하는 인터럽트(interrupt)는 연산 수행 초기에 해당 오퍼랜드(operand)를 검사해봄으로써 그 발생 여부를 알아낼 수 있다. 인터럽트를 발생시키는 명령어는 FPRS(floating-point registers state)와 같은 간단한 하드웨어를 사용하여 명령어가 실행유닛에서 수행되는 한 두 사이클 내에 발견할 수 있다[11]. MIPS나 Intel Pentium에서도 이러한 방법을 사용하고 있다. 정체현상을 일으키는 명령어들은 대개 실수 곱셈이나 나눗셈 등의 실행 지연시간(execution latency)이 큰 실수연산이 대부분을 차지한다. 따라서 명령어가 리오더버퍼의 출구부분에 도달하기 전에 인터럽트 발생 여부를 알 수 있다.

쉘터버퍼를 사용하여 얻을 수 있는 장점들은 다음과 같다. 먼저 리오더버퍼의 크기를 감소시킴으로서 칩의 다이 크기를 줄일 수 있다. 고정된 다이 크기 경우에는 감소된 공간에 명령어 캐쉬나 데이타 캐쉬 같은 다른 구성요소들을 크기를 늘려 프로세서의 성능을 향상시킬 수 있다. 또한 긴 지연시간을 갖는 명령어를 쉘터버퍼로 보냄으로써 제거된 스테그네이션은 파이프라인의 각 스테이지를 따라 이동하는 명령어 흐름을 순조롭게 하여 명령어의 실행율을 높일 수 있다.

3.2 쉘터버퍼의 분석

쉘터버퍼로 얻어지는 성능향상을 분석하고 적절한 버퍼 크기를 결정하기 위하여 리오더버퍼의 간단한 해석적 모델을 이용한다. 본 논문의 해석적 모델은 다음과 같은 가정을 사용한다. 첫째로 리오더버퍼는 ($n+1$)개의 엔트리를 가지고 있고 FIFO 큐에 의해 구현된다. 큐의 출구 부분 엔트리는 0번 엔트리라고 하고 나머지 엔트리들은 오름차순으로 번호를 매긴다. 따라서 n은 입구 부분의 엔트리 번호가 된다. 둘째로 리오더버퍼에 이용 가능한 공간이 있다면 하나의 명령어만이 리오더버퍼에 주입되며, 명령어가 연산을 마치고 리오더버퍼의 출구에 도달할 때 하나의 명령어만이 리오더버퍼에서 빠져나간다. 이러한 가정은 동시에 k개의 명령어들을 이슈할 수 있는 수퍼스칼라의 리오더버퍼에 대하여도 쉽게 일반화시킬 수 있다. 셋째로 명령어들 사이에는 데이타 의존과 제어 의존이 없다. 또한 모든 명령어들이 실행유닛에서 수행하는 동안은 어떠한 예외상황도 발생되지 않는다. 그러므로 리오더버퍼가 어떠한 스테그네이션도 유발시키지 않는 일반적인 명령어들로 채워져 있다면 매 사이클마다 한 개의 명령어가 완료될 수 있다.

해석적 모델에서 사용되는 명령어들은 짧은 지연시간을 갖는 명령어와 긴 지연시간을 갖는 명령어의 두 그룹으로 나뉘어 질 수 있다. 일반적으로 명령어는 짧은 지연시간을 가지며 L_{norm}으로 표시하고 ($n+1$)개의 엔트리를 갖는 리오더버퍼에서 어떠한 스테그네이션도 유발시키지 않는다($L_{norm} \leq n+1$). 긴 지연시간을 가지는 명령어는 L_{stag}로 표시하며 같은 크기의 리오더버퍼에서 스테그네이션을 유발시킨다($L_{stag} > n+1$).

본 논문에서는 리오더버퍼의 명령어 상태를 보여주기 위해 다음과 같은 표현을 사용한다. 각 명령어는 I_j ($0 \leq j \geq n$)로 표시하고 " : " 표기 오른쪽은 명령어가 연산을 마치기 위해 남아있는 지연시간을 나타낸다. 즉, 일반적인 명령어는 $I_j : L_{norm}-x$로 표시하고, 긴 지연시간을 갖는 명령어는 $I_j : L_{stag}-y$로 나타낸다. 여기서 x와 y는 각 명령어가 리오더버퍼에 입력된 이후의 경과된 시간을 나타낸다. 또한 T_i ($i \geq 0$) 는 클락 사이클 시간을 가리킨다.

그림 3에서 리오더버퍼는 클락 사이클 시간 T_n에 L_{norm} 지연시간을 갖는 일반적인 명령어들 I_0, I_1, …, I_n으로 채워져 있다고 가정한다. 다음 사이클 (T_n+1)에 리오더버퍼의 출구 부분에 있는 명령어 I_0는 레지스터 파일에 그 결과를 저장하기 위하여 버퍼를 빠져나가고, 새로운 명령어 I_{n+1}은 입력된다. 새로이 입력된 명령어는 $L_{stag} > n$의 긴 지연시간을 갖는다고 가정을 한다. 다시 n 사이클 후에 명령어 I_{n+1}은 리오더버퍼의 출구부분에 위치하고 $L_{stag}-n$ 사이클 동안 스테그네이션을 일으켜 새로운 명령어가 실행되는 것을 막는다. 다음 조건 $n \geq L_{stag}$가 만족되도록 리오더버퍼가 충

분히 크다면 비록 명령어는 완료될 수 없지만 새로운 명령어가 $L_{stag}-n$ 사이클 동안 리오더버퍼에 주입되어 명령어 수행은 계속될 수 있다. 수퍼스칼라에서 k개의 명령어가 동시에 이슈되고 완료된다면 리오더버퍼는 일정한 명령어 이슈율을 유지하기 위해 $k \cdot L_{stag}$보다 커야만 한다. 즉, 지연시간이 $L_{stag} > \lfloor n/k \rfloor$인 명령어가 리오더버퍼에 주입되면 그 명령어는 언젠가 다른 명령어의 이슈를 막아 파이프라인 정지를 유발시킨다. 그림 3에서 $L > x$이면 $L \sim x = L - x$이고 $L \le x$이면 $L \sim x = 0$이다.

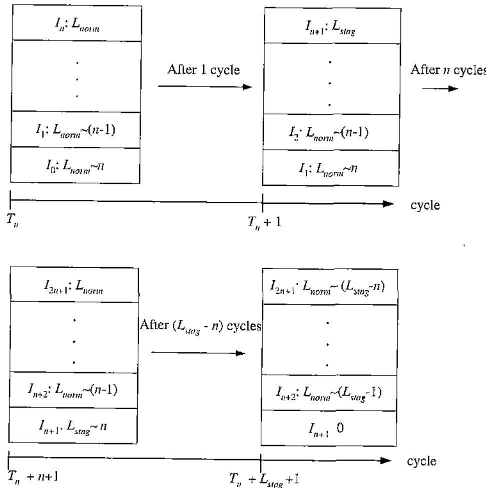


그림 3 리오더버퍼에서의 명령어 흐름

그림 4에서 보는 봐와 같이 쉘터버퍼를 사용하면 (T_n+n+1) 클락 사이클에 0번 엔트리의 명령어 I_{n+1}은 리오더버퍼에서 쉘터버퍼로 이동한다. 이는 리오더버퍼의 입장에서 보면 I_{n+1}은 가상적으로 수행을 마치고 리오더버퍼를 빠져나가는 것과 동일하게 동작한다. 명령어 I_{n+1}은 남은 ($L_{stag}-n$) 사이클 동안 쉘터버퍼에 머무르면서 수행을 계속한 후, 그 결과를 레지스터에 저장하여 동작을 완료하게 된다. 만약 쉘터버퍼가 사용되지 않는다면 어느 명령어도 ($L_{stag}-n$) 사이클 동안 완료될 수 없으며, 리오더버퍼가 충분히 크지 않아 이용 가능한 공간이 없다면 새로운 명령어조차도 이슈될 수 없다. 수퍼스칼라에서 k개의 명령어가 동시에 이슈되고 완료될 수 있다면 쉘터버퍼를 사용한 리오더버퍼는 ($L_{stag}-n$) 사이클 동안 최대 $k \cdot (L_{stag}-n)$ 명령어들을 더 완료할 수 있다.

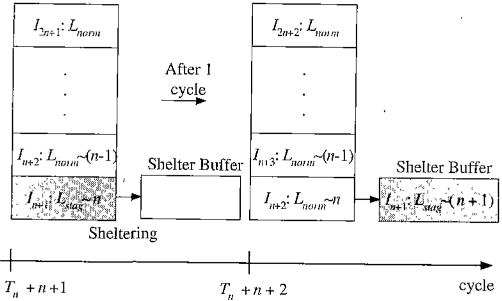


그림 4 쉘터버퍼를 사용한 리오더버퍼에서의 명령어 흐름

프로세서가 일부 명령어에 대해 긴 지연시간을 가지고 있다면 쉘터버퍼는 리오더버퍼의 스테그네이션에 의해 발생되는 성능 저하를 해결할 수 있는 매우 효율적이고 경제적인 방법이다. 본 논문에서 수행한 시뮬레이션에서 긴 지연시간을 갖는 명령어가 리오더버퍼에 연속적으로 위치하는 것은 자주 볼 수 없었다. 이것은 아주 작은 크기의 쉘터버퍼만으로도 대부분의 스테그네이션을 처리하는데 충분함을 의미한다.

데이타 해저드 혹은 제어 해저드가 발생하지 않고 프로세서가 매 사이클에 대해 병렬로 종료할 수 있는 명령어 개수는 k라고 가정하면, 리오더버퍼에서 발생하는 스테그네이션의 영향만을 고려한 IPC(instructions per cycle)는 다음 식으로부터 얻어진다.

$$IPC = \frac{k}{1 + \sum_j (L_j \sim \lfloor \frac{n}{k} \rfloor) \Pr(I_j)}$$

위 식에서 I_j는 명령어 j, L_j는 명령어 I_j의 지연시간, $\Pr(I_j)$는 명령어 I_j가 수행되는 확률을 나타낸다. 짧은 지연시간을 갖는 명령어만을 포함하는 프로그램에서는 $L_j \le \lfloor n/k \rfloor$가 되어, $IPC = k$가 된다. 위 식으로부터 실행유닛의 개수(k) 혹은 지연시간(L_j)이 커짐에 따라 리오더버퍼의 크기는 이에 비례하여 반드시 커져야한다. 이러한 조건이 만족되지 않는다면 리오더버퍼의 스테그네이션에 의한 심각한 성능 저하가 발생한다.

리오더버퍼의 동작은 평균 명령어 이슈율, 명령어의 데이타 및 제어 의존, 그리고 각 명령어의 지연시간에 의해 영향을 받는다. 또한 긴 지연시간을 가지는 명령어의 분포는 프로그램 수행시 사용되는 리오더버퍼의 엔트리 개수와 직접 연관된다. 연속적인 명령어들이 긴 지연시간을 가지고 있고 각 명령어가 앞서의 명령어에 대해 데이타 의존도를 가지고 있다면 최악의 연쇄적인 스테그네이션이 발생한다. 따라서 명령어의 데이타 의존과 분포의 영향을 고려한다면 리오더버퍼의 크기는 명령어

이슈가 중지되는 것을 피하기 위해서 매우 커야 한다. 그러나 작은 크기의 쉘터버퍼는 대부분의 이러한 영향들을 효율적으로 해결할 수 있다. 해석적 모델에서는 이러한 모든 파라미터들의 영향을 자세하게 다룰 수 없으므로 본 논문에서는 명령어 트레이스를 사용한 시뮬레이션을 통해 쉘터버퍼의 성능을 분석한다. 다음 장에서는 수퍼스칼라의 시뮬레이션 모델과 시뮬레이션을 수행한 결과를 분석할 것이다.

분기 명령어들은 프로세서에서 명령어 수행의 순서를 바꿀 수 있기 때문에 비순차이슈 수퍼스칼라의 성능에 매우 큰 영향을 미친다. 분기 예측과 모험적 수행은 이러한 분기명령의 영향을 감소시키기 위해 사용된다. 분기 예측이 실패한다면 분기명령어 다음에 인출되어 수행되는 모든 명령어들은 버려져야하고 새로운 목적지로부터 명령어들이 인출되어야 한다. 쉘터버퍼를 사용한 리오더버퍼에서 사용되는 엔트리들의 개수는 리오더버퍼만을 사용하는 경우보다 훨씬 적기 때문에 버려질 수 있는 엔트리 개수 또한 적어진다. 제안된 쉘터버퍼를 사용함으로서 리오더버퍼의 크기를 실질적으로 줄일 수 있으며 부동소수점 연산을 포함하는 응용 프로그램에 대하여는 매우 높은 성능 향상을 기대할 수 있다.

4. 시뮬레이션 및 분석

4.1 시뮬레이션 모델

본 연구에서 사용하는 시뮬레이터는 그림 1과 같은 RISC 수퍼스칼라 프로세서를 기본 모델로 하였으며 명령어 윈도우를 사용하여 비순차로 명령어를 이슈하여 실행하며 리오더버퍼를 사용하여 비순차로 종료된 명령어를 순차 완료시킨다. 시뮬레이션은 정확한 프로세서의 동작을 모델링할 수 있어야 하므로 C++언어를 사용해서 작성하였으며 클록 사이클 단위로 동작한다. 시뮬레이터에서 실행유닛의 수는 사용자에 의해서 설정될 수 있으나 본 논문에서 정수 연산 유닛의 수는 4개로 고정한다. 그 이상의 정수 유닛을 사용하여도 시뮬레이션 결과는 거의 변화가 없다. 정수 연산 유닛은 덧셈, 뺄셈, 메모리(LD/ST) 동작을 수행할 수 있다. 정수 유닛 외에 분기명령을 수행하는 별도의 분기 유닛 하나를 사용한다. 따라서 동일 사이클에는 단지 한 개의 분기명령어만이 실행될 수 있다. 또한 시뮬레이터는 부동소수점 덧셈 유닛과 정수 및 실수 곱셈 그리고 나눗셈을 수행할 수 있는 부동소수점 곱셈 유닛을 포함한다. 분기 예측의 정확도는 90%로 가정하고 리오더버퍼는 쉘터버퍼 기능을 선택적으로 사용할 수 있다. 표 1은 리오더버퍼 시뮬레이션에서 사용된 수퍼스칼라 프로세서 시뮬레이터의 구성을 나타내고 있다. 또한 시뮬레이터에서 사용 가능한 실행유닛의 수를 충분히 하여 실질적으로 무한대로 설정하는 것과 같은 경우의 시뮬레이션 수행 결과도 4.2절에서 분석한다.

시뮬레이션을 수행하기 위해서는 선택한 벤치마크 프로그램을 실제로 수행시켜 얻은 명령어 트레이스가 필요하다[12]. 시뮬레이션을 수행하기 위해 필요한 트레이스 파일은 SPA[13] 프로그램을 사용하여 Sun SPARC -20 워크스테이션 기종에서 GNU C/C++의 -O0 옵션으로 컴파일된 파일에 대하여 생성하였다. 리오더버퍼의 동작은 응용 프로그램의 명령어의 분포에 의해 결정된다. 본 논문에서 -O0, -O1, -O2, -O3의 최적화 레벨로 많은 시뮬레이션을 수행했으나 리오더버퍼와 쉘터버퍼의 동작은 최적화 레벨과 특별히 주목할 만한 관계를 찾지 못하였다. 물론 프로그램의 수행시간은 최적화 레벨이 높을수록 향상된다. 본 시뮬레이션에서 중요한 것은 부동소수점 명령어의 분포이다. 따라서 트레이스 파일들은 최적화를 수행하지 않는 기본 레벨에서 컴파일된 실행 파일을 사용하여 생성된다.

표 1 리오더버퍼 시뮬레이션에서 사용된 수퍼스칼라 프로세서의 시뮬레이터 구성

구 성		설 명
명령어 이슈		최대 이슈 수는 사용자가 설정할 수 있다. 본 시뮬레이션에서는 최대 이슈 수를 4로 고정한다.
명령어 윈도우		명령어 윈도우의 엔트리 개수는 사용자가 설정할 수 있다. 본 시뮬레이션에서는 명령어 윈도우의 엔트리 개수를 16으로 고정한다.
리오더 버퍼		리오더버퍼의 엔트리 개수는 사용자가 설정할 수 있다. 쉘터버퍼를 사용하는 경우와 그렇지 않은 경우로 나누어 시뮬레이션 한다.
ALU	정수 연산 유닛	정수 덧셈과 뺄셈 연산을 수행한다. 또한 LD(load)나 ST(store) 명령어와 같은 메모리 동작도 수행한다. 본 시뮬레이션에서는 정수 연산 유닛의 수를 4개로 고정하고, 별도의 분기 유닛 하나를 사용한다.
ALU	부동소수점 덧셈 유닛	부동소수점 덧셈과 뺄셈 연산을 수행한다. 본 시뮬레이션에서는 완전히 파이프라인 화된 부동소수점 덧셈 유닛을 1개로 고정한다.
ALU	부동소수점 곱셈 유닛	부동소수점 곱셈과 나눗셈 연산을 수행하며 정수 곱셈 및 나눗셈 연산도 이 유닛에서 수행한다. 본 시뮬레이션에서는 완전히 파이프라인 화된 부동소수점 곱셈 유닛을 1개로 고정한다.
레지스터 파일		레지스터 파일에는 동시에 여러 명령어의 연산 결과가 저장될 수 있다. 본 시뮬레이션에서는 최대 4개까지 연산 결과가 동시에 저장된다.
명령어 집합		스팍(SPARC) V7 구조의 명령어 집합을 사용한다.

표 2 명령어 타입별 파이프라인 지연시간 (단위: 사이클)

명령어 타입	지연시간	명령어 타입	지연시간
Memory LD/ST	1	Float ADD/SUB	3
Integer ADD/SUB	1	Float MUL	6
Integer MUL	6	Float DIV/SQRT	14

표 2는 시뮬레이션에서 사용되는 지연시간을 명령어 타입별로 설명한 것이다. 실행유닛으로 명령어를 이슈할 수 있는 최소 사이클 수인 이슈 지연시간은 1로 가정한다. 부동소수점 덧셈 및 곱셈 유닛은 완전히 파이프라인 화되어 있어 지연시간이 1보다 큰 명령어들도 해당 실행 유닛으로 매 사이클 당 이슈될 수 있다. 스토어 버퍼는 충분히 커 메모리로 데이타를 저장할 때 추가적인 지연은 발생하지 않는다. 분기 지연시간은 1로 설정하였으며, 캐쉬의 히트율은 100%로 가정하였다.

표 3은 시뮬레이션에서 사용된 Whetstone, FFT Fast Fourier Transform), FM(Float Matrix), 그리고 Linpack 벤치마크 프로그램의 명령어 분포를 나타낸 것이다. 이 표에서 6번째 열인 스테그네이션은 정수 곱셈, 부동소수점 덧셈/뺄셈, 부동소수점 곱셈/나눗셈/루트(SQRT) 연산처럼 스테그네이션을 유발시킬 수 있는 명령어들의 백분율을 나타낸 것이다. Linpack의 경우 부동소수점 연산 명령어는 3.19% 밖에 되지 않으나 스테그네이션을 일으키는 명령어의 분포는 14.74%이다. 이것은 Linpack 벤치마크 프로그램의 명령어의 분포에서 정수 곱셈 명령과 나눗셈 명령이 많기 때문에 스테그네이션을 일으키는 명령어의 분포가 높게 나타난다.

표 4는 각 실행유닛에 의해 처리되는 명령어의 비율

표 3 명령어 트레이스의 생성에 사용된 벤치마크 프로그램 및 명령어 타입별 분포 (단위 : %)

벤치마크 프로그램	분기 명령어	메모리 참조	정수 연산	부동소수점 연산	스테그네이션
Whetstone	9.16	23.02	59.14	8.68	11.45
FFT	3.42	32.66	52.69	11.23	11.23
FM	10.38	42.64	38.70	8.28	9.01
Linpack	10.09	26.31	60.41	3.19	14.74

표 4 각 실행유닛에 의해 처리되는 명령어의 비율 (단위 : %)

벤치마크 프로그램	정수연산 유닛	부동소수점 덧셈 유닛	부동소수점 곱셈 유닛	분기유닛
Whetstone	79.40	5.13	6.31	9.16
FFT	85.35	6.90	4.33	3.42
FM	80.61	5.35	3.66	10.38
Linpack	75.17	2.15	12.59	10.09

을 벤치마크 프로그램별로 나타낸 것이다. 부동소수점 덧셈과 곱셈 연산 유닛에 의하여 수행되는 명령어의 비율은 Linpack을 제외하면 10%보다 작은 것을 알 수 있다. 따라서 모든 실행유닛들이 완전히 파이프라인 화되었기 때문에 각각 한 개의 부동소수점 덧셈 유닛과 곱셈 유닛만으로도 본 시뮬레이션에서는 충분하다.

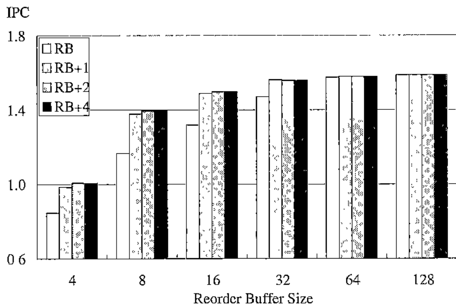
4.2 시뮬레이션 분석

그림 5는 리오더버퍼의 크기가 변할 때 시뮬레이션으로부터 얻어진 각 벤치마크 프로그램의 실행율(IPC)을 나타낸다. 각 그래프의 (RB + *num*)에서 RB는 리오더버퍼의 크기를 의미하고 *num*은 쉘터버퍼의 수를 나타낸다. 즉, RB + 2는 주어진 RB 크기의 리오더버퍼에 2개의 쉘터버퍼를 함께 사용한 것을 나타낸다. 벤치마크 프로그램의 수행시간은 다음과 같이 명령어 실행율인 IPC로부터 계산될 수 있다.

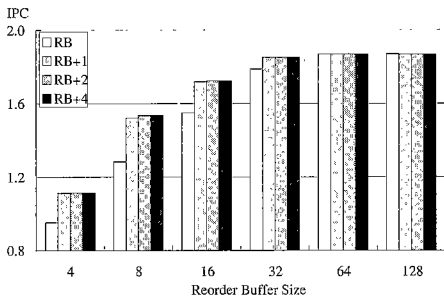
$$(\text{프로그램 수행시간}) = (\text{클록 사이클 시간}) \times \frac{(\text{실행되어진 전체 명령어의 수})}{(IPC)}$$

시뮬레이터에서 최대 명령어 이슈 수는 사용자에 의하여 설정될 수 있다. 이슈 수를 8로 하여 시뮬레이션을 수행하는 경우 4인 경우에 비하여 실행율은 어느 정도 향상되나 쉘터버퍼와 리오더버퍼 사이의 관계에서는 관심을 끌만한 차이가 없어, 본 논문에서는 이슈 수가 4인 경우에 대하여 설명한다. 리오더버퍼의 크기가 4개 이하면 이슈되는 명령어의 수보다 더 작기 때문에 리오더버퍼는 대부분 포화된 상태에서 동작하게 되므로 성능 평가는 의미를 갖지 못한다. 따라서 본 논문에서는 리오더버퍼의 크기를 4개 이상으로 설정하여 시뮬레이션 하였다. 리오더버퍼의 크기가 64개보다 큰 경우에는 Linpack 벤치마크 프로그램을 제외하면 쉘터버퍼를 사용하여도 실행율의 추가적인 향상은 없다. 이것은 리오더버퍼의 크기가 충분히 크기 때문에 대부분의 경우 스테그네이션이 발생하여도 리오더버퍼는 이슈되는 명령어를 계속 받아들일 수 있어 파이프라인 정지가 발생하지 않기 때문이다. 그러나 Linpack 벤치마크 프로그램은 그림 5의 (d)에서와 같이 리오더버퍼를 128개를 사용하여도 그 크기가 충분하지 않기 때문에 쉘터버퍼를 사용하면 더 좋은 실행율을 얻을 수 있다.

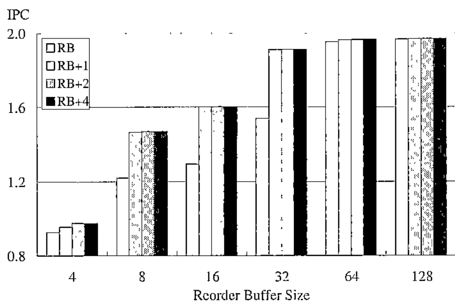
리오더버퍼의 크기가 8에서 32개 사이일 때 쉘터버퍼를 사용하면 실행율과 프로그램 수행시간에 있어서 두드러진 향상을 보인다. 리오더버퍼의 크기가 작다면 두 버퍼의 합이 같다는 조건 하에서 2개의 쉘터버퍼를 사용한 것보다 1개를 사용하는 경우 더 좋은 성능 향상을 얻을 수도 있다. 4개의 쉘터버퍼를 사용하는 것은 2개의



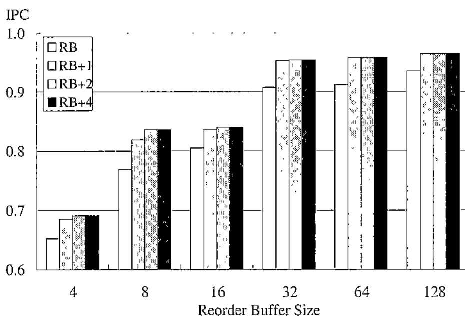
(a) Whetstone Benchmark



(b) FFT Benchmark



(c) FM Benchmark



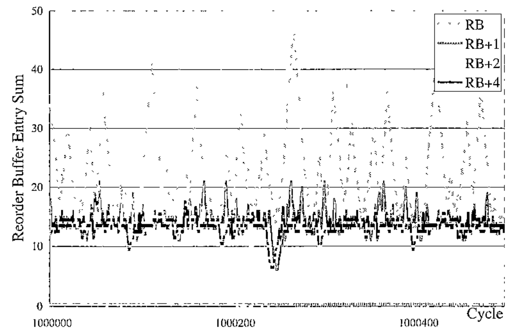
(d) Linpack Benchmark

그림 5 각 벤치마크 프로그램별 IPC

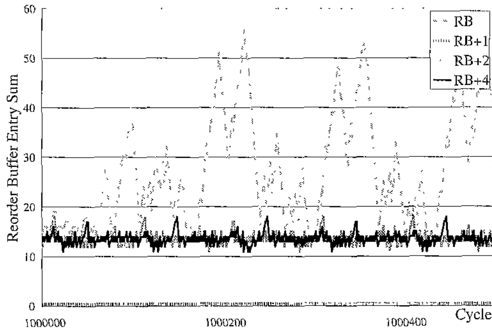
쉘터버퍼를 사용하는 경우와 비교할 때 실행율과 프로그램 수행시간에 있어서 그 차이가 거의 없다. 표 3에서 알 수 있듯이 스테그네이션을 일으키는 명령어의 분포는 시뮬레이션에서 사용된 벤치마크 프로그램에서 약 10%를 차지한다. 따라서 긴 지연시간을 갖는 명령어가 세 개 또는 그 이상 연속하여 수행되는 경우는 매우 드물어 쉘터버퍼의 세 번째나 네 번째 엔트리는 거의 사용되지 않기 때문이다.

그림 6은 리오더버퍼의 크기를 128개로 고정하였을 때 리오더버퍼에서 명령어들이 점유하고 있는 엔트리의 개수를 보여준다. 대부분의 벤치마크 프로그램에서 이 정도 크기는 리오더버퍼가 어떠한 파이프라인 정지도 없이 이슈되는 명령어를 받아들일 수 있을 정도로 충분하다. 각 벤치마크 프로그램에 대해 1,000,000번째 클록 사이클에서 1,000,500번째 사이클까지 리오더버퍼를 점유하고 있는 명령어의 수를 측정한 후 시간에 따른 변화를 그래프로 표현한 것이다. 부동소수점 벤치마크 프로그램은 대개 많은 루프를 포함하고 있다. 따라서 리오더버퍼는 각 벤치마크 프로그램이 실행되는 동안 특정한 반복된 동작을 보여준다. 만약 다른 주기를 선택하면 리오더버퍼는 비록 다른 모습이지만 비슷한 반복된 동작을 보일 것이다.

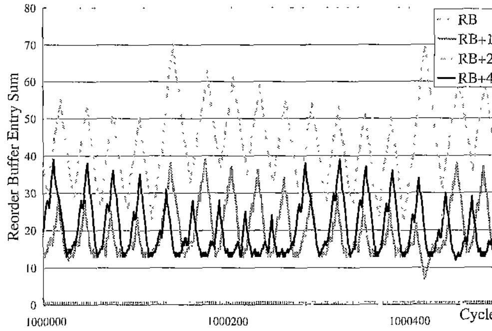
표 3에서 보는 바와 같이 FFT 벤치마크 프로그램은 스테그네이션을 유발시키는 명령어를 11.23%밖에 포함하지 않으나 쉘터버퍼를 사용하지 않는 경우 자주 50개 이상의 리오더버퍼 엔트리를 점유한다. 이에 반하여 쉘터버퍼를 2개 이상 사용하였을 때에는 스테그네이션을 유발하는 명령어가 쉘터버퍼로 이동되기 때문에 리오더버퍼 엔트리 개수는 단지 10에서 20개 정도만 사용된다. Linpack 벤치마크 프로그램에서는 스테그네이션을 유발하는 명령어가 14.74%나 된다. 그러므로, 쉘터버퍼 없이 리오더버퍼만을 사용하는 경우에는 128개 엔트리가 모



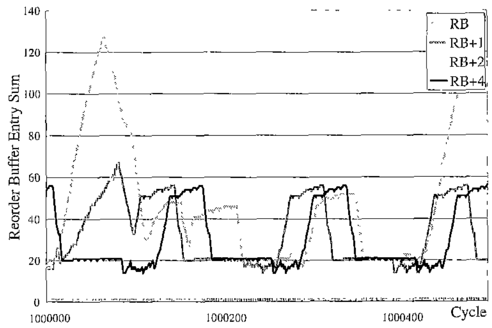
(a) Whetstone Benchmark



(b) FFT Benchmark



(c) FM Benchmark

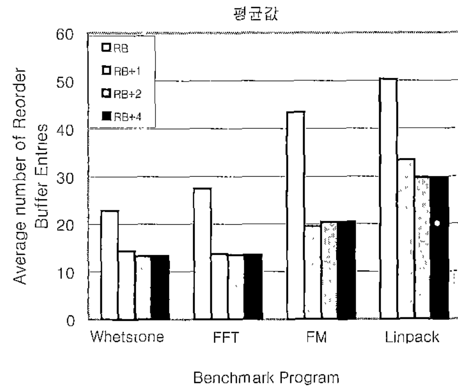


(d) Linpack Benchmark

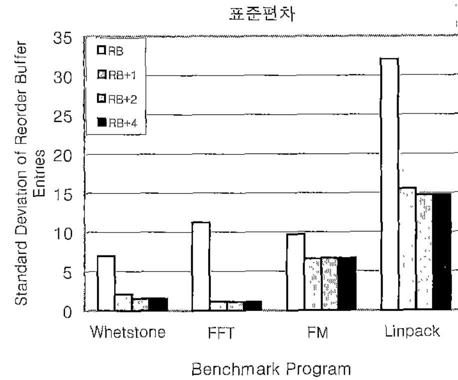
그림 6 사용되어진 리오더버퍼 엔트리 개수의 시간에 따른 변화

두 사용되는 때도 발생하나 쉘터버퍼를 사용하면 20에서 55개 정도의 엔트리만 사용된다. 그림 7은 각 벤치마크 프로그램의 수행중에 사용된 리오더버퍼 엔트리 개수의 평균값과 표준편차를 그래프로 나타낸 것이다.

표 5는 최대 실행율을 유지하기 위해 필요한 리오더버퍼의 크기를 비교한 것이다. 이 표로부터 Whetstone에서는 44%, FFT에서는 50%, FM에서는 60%, Linpack에서는 75%의 리오더버퍼 크기를 감소시킬 수 있음을 알 수 있다. 표 6은 2개의 쉘터버퍼를 사용하였을 때 프로그램 수행시간의 향상율을 나타낸 것이다. 수



(a) 평균값



(b) 표준편차

그림 7 사용되어진 리오더버퍼 엔트리 개수의 평균값과 표준편차

표 5 최대 실행율을 얻기 위하여 필요한 리오더버퍼의 크기

벤치마크 프로그램	최대 실행율	리오더버퍼만 사용한 경우(개)	쉘터버퍼를 사용한 리오더 버퍼(RB + SB)	사용된 리오더 버퍼의 감소율(%)
Whetstone	1.59	64	34 + 2	44%
FFT	1.87	64	30 + 2	50%
FM	1.97	128	50 + 2	60%
Linpack	0.94	128	30 + 2	75%

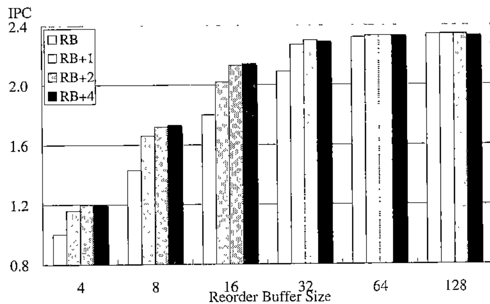
표 6 프로그램 수행시간의 향상

벤치마크 프로그램	사용되어진 리오더 버퍼의 크기(개)	실행 시간 (사이클)		실행 시간의 향상율 (%)
		리오더 버퍼만 사용한 경우	쉘터버퍼를 사용한 리오더 버퍼의 경우	
Whetstone	8	1951218	1629026	19.78
FFT	8	23246971	19425272	19.67
FM	16	3604697	2908715	23.93
Linpack	8	13905977	12798307	8.65

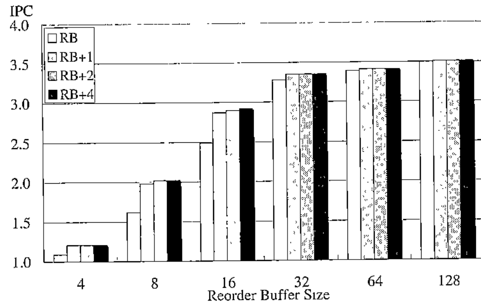
행시간은 쉘터버퍼를 사용하였을 때 Whetstone에서는 19.78%, FFT에서는 19.67%, FM에서는 23.93%, Linpack에서는 8.65%의 향상을 얻을 수 있다.

그림 8은 시뮬레이터에서 사용 가능한 실행유닛의 수를 충분히 하여 실질적으로 무한대로 설정하는 것과 같게 하는 경우의 시뮬레이션 수행 결과를 나타낸 것이다. 즉, 정수 연산 유닛과 부동소수점 연산 유닛의 개수를 충분히 설정을 하고 리오더버퍼에서 실행이 완료 된 명령어의 결과가 레지스터 파일에 갱신될 수 있는 개수를 충분히 설정하여 자원 충돌에 의한 성능 감소가 발생하지 않게 한 후 시뮬레이션을 수행한 결과이다. 명령어의 실행율은 실행유닛의 개수에 제한 두었을 때보다 커지고 수행시간은 줄어드는 것은 당연하다. 각각의 그래프를 보면 쉘터버퍼를 1개 사용하였을 때와 2개 이상 사용하였을 때의 성능 차이가 실행유닛의 수에 제한을 두는 경우보다 분명하게 나는 것을 알 수 있다. 이것은 실행유닛의 수를 충분히 많이 설정하면 긴 지연시간을 갖는 명령어가 동시에 수행될 확률이 높아지기 때문이다. 그러나 벤치마크 프로그램 자체가 가지고 있는 그러한 명령어 분포가 10%정도이므로 세 개 이상 동시에 수행될 확률은 높지 않다.

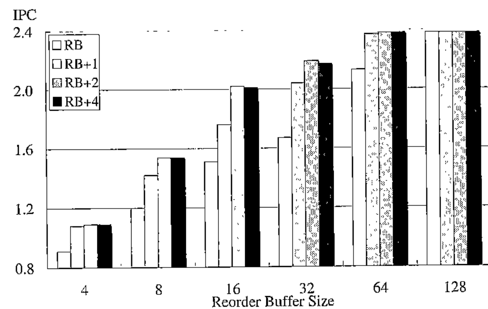
표 7은 실행유닛을 충분히 설정한 경우 최대 실행율을 유지하기 위하여 필요한 리오더버퍼의 크기와 쉘터



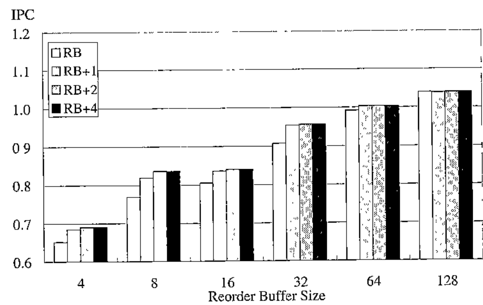
(a) Whetstone Benchmark



(b) FFT Benchmark



(c) FM Benchmark



(d) Linpack Benchmark

그림 8 실행유닛의 수를 충분히 설정한 경우의 IPC

표 7 실행유닛의 수를 충분히 설정한 경우의 쉘터버퍼에 의해 줄일 수 있는 리오더버퍼의 크기

벤치마크 프로그램	최대 실행율	리오더 버퍼만 사용한 경우(개)	쉘터버퍼를 사용한 리오더버퍼 (RB + SB)	사용된 리오더버퍼의 감소율(%)
Whetstone	2.13	40	16 + 2	55
FFT	3.35	56	32 + 2	39
FM	2.11	32	11 + 2	59
Linpack	0.82	24	8 + 2	58

표 8 실행유닛의 수를 충분히 설정한 경우의 프로그램 수행시간의 향상

벤치마크 프로그램	사용되어진 리오더버퍼의 크기 (개)	실행 시간 (사이클)		실행 시간의 향상율 (%)
		리오더버퍼만 사용한 경우	쉘터버퍼를 사용한 리오더버퍼의 경우	
Whetstone	16	1724484	1519056	13.52
FFT	16	19239807	17294551	11.25
FM	16	3604697	2908205	23.95
Linpack	8	13905977	12798201	8.66

버퍼에 의해 줄일 수 있는 리오더버퍼의 비율을 나타낸 것이다. 표 8은 두 개의 쉘터버퍼를 사용하였을 때 프로그램 수행시간의 향상율을 나타낸 것이다. 프로그램 수행시간의 향상율은 실행유닛에 제한을 두는 경우보다 낮게 나타난다. 이것은 리오더버퍼에서 레지스터파일로 갱신되는 명령어의 개수를 충분히 크게 설정하여, 스테그네이션 때문에 완료되지 못한 명령어들이 그 수에 관계없이 동시에 리오더버퍼를 빠져 나올 수 있기 때문에, 쉘터버퍼에 의하여 얻을 수 있는 성능 향상은 실행유닛의 수에 제한을 두는 경우와 비교하여 상대적으로 낮다.

5. 결 론

비순차로 이슈하는 수퍼스칼라 프로세서에서 리오더버퍼의 출구 부분에 지연시간이 긴 명령어가 위치할 때, 이미 수행이 종료된 명령어들이 완료되지 못하고 리오더버퍼에 머무르게 되는 명령어의 정체현상이 발생한다. 본 연구에서는 명령어의 스테그네이션을 제거하고 리오더버퍼의 사용을 효과적으로 줄이기 위한 새로운 리오더버퍼 구조를 제안하였다. 제안된 리오더버퍼 구조에서 긴 지연시간을 갖는 명령어는 다른 명령어들의 진행을 막지 않기 위해 쉘터버퍼로 옮겨진다. 이 쉘터버퍼는 비순차로 이슈하는 수퍼스칼라 프로세서 내의 파이프라인에서 명령어의 흐름을 균일하게 하여 성능을 향상시킬 수 있고 리오더버퍼의 지나친 하드웨어 비용을 줄일 수 있다. 쉘터버퍼는 명령어의 정체현상을 자주 일으키는 부동소수점 연산을 포함하는 응용 프로그램에서 높은 성능 향상을 보여주었다.

4개의 벤치마크 프로그램에 대한 명령어 트레이스를 이용하여 쉘터버퍼를 사용한 리오더버퍼를 시뮬레이션 하였다. 대부분의 경우 리오더버퍼의 엔트리 개수가 8개에서 32개 사이일 때 쉘터버퍼는 단지 1개 또는 2개만 사용하여도 현저한 성능 향상을 얻을 수 있었다. 쉘터버퍼를 4개 사용했을 때는 2개만 사용했을 경우와 비교하여 주목할만한 성능 향상은 없었다. 스테그네이션을 일으키는 명령어의 분포는 시뮬레이션에서 사용된 벤치마크 프로그램에서 약 10%를 차지한다. 따라서 긴 지연시간을 갖는 명령어가 세 개 또는 그 이상 연속하여 수행되는 경우는 매우 드물어 쉘터버퍼의 세 번째나 네 번째 엔트리는 거의 사용되지 않기 때문이다.

32 엔트리를 갖는 리오더버퍼는 전형적인 1.0μm 트리플 메탈(triple-metal) CMOS 기술을 사용하여 구현하는 경우 그 크기는 2mm × 2.85mm 이다[6]. 칩 다이의 면적은 사용된 리오더버퍼의 엔트리 개수에 비례한다고 가정할 수 있으며, 제안된 쉘터버퍼를 사용함으로서 실질적으로 줄일 수 있는 칩 다이의 크기는 표 5와 7로부터 얻을 수 있다. 고정된 다이 크기 경우에는 감소된 공간에 명령어 캐쉬나 데이타 캐쉬 같은 다른 구성요소들을 크기를 늘려 프로세서의 성능을 향상시킬 수 있다. 또한 긴 지연시간을 갖는 명령어를 쉘터버퍼로 보냄으로써 제거된 스테그네이션은 파이프라인의 각 스테이지를 따라 이동하는 명령어 흐름을 순조롭게 하여 명령어의 실행율을 높일 수 있다.

참 고 문 헌

[1] M. Johnson, "Superscalar Microprocessor Design," Prentice Hall, 1991.

[2] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," Proc. Third Int. Conf. Architectural Support Programming Language Oper. Sys., pp. 414-424, IEEE Computer Society Press, April 1989.

[3] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple-instruction issue," Proc. Third Int. Conf. Architectural Support Programming Language and Oper. Sys., pp. 290-302, April 1989.

[4] A. Inoue and K. Takeda, "Performance evaluation for various configuration of superscalar processors," Computer Arch. News, pp. 4-11, 1994.

[5] M. Butler, T. Y. Yeh, and Y. Patt, "Single instruction stream parallelism is greater than two," Proc. Third Int. Conf. Architectural Support Programming Languages and Oper. Sys., pp. 290-302, April 1989.

[6] J. Lenell, S. Wallace, and N. Bagherzadeh, "A 20 MHz CMOS reorder buffer for a superscalar microprocessor," 4th NASA Symposium on VLSI Design, Oct. 1992.

[7] S. Wallace, N. Dagli, and N. Bagherzadeh, "Design and implementation of a 100MHz reorder

buffer," 37th Midwest Symposium on Circuit and Systems, Aug. 1994.

[8] Steven Wallace and Nader Bagherzadeh, "Modeled and measured instruction fetching performance for superscalar microprocessors," IEEE Trans. Parallel and Distributed Systems, vol. 9, no. 6, pp. 570-578, Jun. 1998.

[9] P. K. Dubey, G. B. Adams III, and M. J. Flynn, "Instruction window size trade-offs and characterization of program parallelism," IEEE Trans. Computers, vol. 43, no. 4, pp. 431-442, April 1994.

[10] Y. H. Pyun, C. S. Park, and S. B. Choi, "The effect of instruction window on the performance of superscalar processors," IEICE Trans. Fundamentals, vol. E81-A, no. 6, pp. 1036-1044, June 1998.

[11] UltraSPARC User's Manual, July 1997.

[12] Standard Performance Evaluation Corporation, "SPEC Benchmark," http:// www.specbench.org, Sep. 1999.

[13] G. Irlam, "Spa" Personal Communication, http:// www.base.com/gordoni/spa/cat1 /spy.1, 1995.

장 문 석

1997년 건양대학교 컴퓨터공학과(학사). 2000년 인하대학교 전자공학과(석사). 2000년 ~ 현재 주식회사 모텍스 연구소 연구원. 관심분야는 컴퓨터 구조, 병렬 및 분산처리 시스템

이 정 우

2000년 인하대학교 전자공학과(학사). 2000년 ~ 현재 인하대학교 전자공학과 석사과정. 관심분야는 병렬 및 분산처리 시스템, 컴퓨터 네트워크

최 상 방

1981년 한양대학교 전자공학과(학사). University of Washington(석사). 1990년 University of Washington(박사). 1981년 ~ 1986년 LG 정보통신(주) 근무. 1991년 ~ 현재 인하대학교 전자공학과 교수. 관심분야는 컴퓨터 구조, 컴퓨터 네트워크, 병렬 및 분산처리 시스템, Fault-tolerant computing