In-order Completion을 지원하는 SMT(Simultaneous Multithreading) 프로세서 구조의 최적화

김문경, 문병인, 김재억, 정우경, 이승필, 이용석
연세대학교 전가전자공학과
전화 : 02-2123-2872 / 핸드폰 : 019-415-8633

Optimization of a SMT(Simultaneous Multi-Threading Architecture) with In-order Completion

Moon-Gyung Kim, Byung-In Moon, Jae-Uk Kim, Woo-Kyeong Jeong, Seoung-Pil Lee and Yong-Surk Lee
Department of Electric and Electronic Engineering Yonsei University
E-mail : bungae@yonsei.ac.kr

Abstract

This paper presented a optimized configuration for ARM-compatible, simultaneous multi-threading archi-tecture with in-order issue/completion. In spite of the penalty of ARM architecture, performance of the architecture has been increased. In in-order completion scheme, the SMT architecture was optimized to enhance performance with appending minimal hardware cost. This modified architecture supports dynamic sharing of resources including a single fetch queue and decode unit, and thus minimizes resource waste and greatly improves utilization of functional units.

I. 서문

오늘날, 고성능 마이크로프로세서들이 개발되고 있으며 그들의 속도는 점차 더 빨라지고 있다. 그들 중 대부분은 성능의 향상을 위하여 명령어 단위 병렬성(ILP)을 이용한 슈퍼스칼라 구조를 채택하고 있는데, 이는 그 자체에 한계점을 지니고 있다.[1] 이러한 시스템에서의 성능의 향상을 위해서, 쓰레드 단위 병렬성(TLP)을 이용하기 위한 여러 가지 방안들이 제시되어 왔다. 그러나, TLP를 이용하기 위한 대표적인 구조의 하나인 멀티프로세서의 경우에는 자원의 공유가 원활하지 않기 때문에 단일 쓰레드만이 존재할 경우에는 동작 유닛의 활용도가 크게 저하하게 된다.

이러한 문제를 극복하기 위하여 simultaneous multi-threading(SMT) 구조가 제안되었는데,[2][3] 이러한 SMT 구조는 ILP와 TLP의 특성을 최대한으로 활용함으로써 그 성능의 향상을 꾀하고 있다. 그러나, 현재까지 소개된 SMT 구조를 구현하기 위해서는 하드웨어상의 비용이 적지 않을 것으로 예상되고 있다. 따라서 본 논문에서는 SMT 구조를 최소한의 하드웨어 비용으로 구현하기 위하여 in-order completion 형태를 가진 프로세서를 제안하고 이의 최적화 방안을 제시했다. 먼저 2장에서는 기준이 될 목표 구조의 선정에 대해 소개하고, 3장에서 목표구조가 SMT구조로 바뀌기 위해서 필요로 하는 구조의 변경 사항에 대하여 알아본다. 그리고, 4장에서 이러한 구조의 구현 및 최적화를 위한 성능 평가 결과를 보이고, 5장에서 결론을 맺는다.

II. 목표 구조의 선정

SMT 구조의 구현을 위하여, 독자적인 명령어 셋 구조(ISA)를 새로이 만들지 않고 기존의 ISA구조를 이용하여 구현하는 것으로 하였다. 이 경우 테스트와 연구 결과물의 실제적인 사용이 용이하기 때문이다. 이번 연구에서는 초기에 ALPHA, MIPS, ARM ISA를 후보로 두고 이중 하나를 선정하였다.

2.1 기본 ISA의 선택

ALPHA ISA는 기본적으로 고성능 슈퍼스칼라 구조를 가지고 있기 때문에 앞서 언급한 세 가지 ISA중에서 SMT구조를 적용하기에 가장 적합하게 보였다. 그리고 MIPS ISA는 분기 지연 슬롯을 가지고 있기 때문에 SMT 뿐만 아니라 슈퍼 스칼라 구조로 하는데도 어느 정도의 어려움을 감수해야 할 것으로 보였다. 마지막으로 ARM ISA는 이 세 가지 ISA 중에서 가장 SMT를 구현하기에 불리한 조건을 지니고 있었다. 우선, ARM ISA에서는 상태 레지스터가 존재하며 또한 거의 모든 명령어에서 조건부 실행을 지원하고 있다. 이는 RISC프로세서에서는 분기 명령어의 개수를 줄일 수 있는 좋은 방안이긴 하나, 슈퍼스칼라 이상으로 넘어가기에는 매우 불리한 조건으로 작용하게 된다. 상태 레지스터는 거의 모든 명령어가 목표 레지스터로 사용하게 되기 때문에 종속 관계에 놓이게 될 명령어들이 거의 대부분이 되게 되며, 또한 조건부 실행은 명령어간의 종속관계가 그 조건에 따라 달라질 수 있게 되기에 종속관계의 정확한 예측이 불가능해지게 된다. 그러나, ARM ISA는 앞서의 두 ISA보다 더 많이 쓰이고 있으며 더 많은 응용프로그램들이 존재하고 있다. 또한, ALPHA ISA나 MIPS ISA보다 훨씬 저렴한 해결책으로 제시되고 있으며, 시장경쟁력 또한 무시할 수 없는 상황이다. 따라서, 좀 더 저렴한 하드웨어 비용을 통하여 경쟁력 있는 SMT 프로세서를 개발하기 위하여 ARM ISA를 선택하게 되었다.

2.2 기본 구조의 개념

적은 하드웨어 비용을 통해 ARM ISA에 SMT 구조를 적용하기 위하여, 앞서 언급한 단점을 극복할 수 있도록 기존에 제안된 SMT구조와는 다른 형태로의 변형이 이루어졌다. Tullsen[2]이 제안한 SMT구조에서는 register renaming과 out-of-order completion방식을 지원하였다. 그러나, ARM ISA에서 이러한 SMT 구조를 수용할 경우에는 조건부 실행 등의 구조상의 단점으로 인하여 도리어 하드웨어 복잡도의 증가와 성능의 저하를 가져올 수 있기 때문에 in-order completion 방식을 채택하면서 reorder buffer대신 기본적인 기능의 in-order 버퍼만을 두었다. 그리고, Hirata[3]가 제안한 SMT구조에서는 쓰레드 별로 명령어 큐와 디코드 유닛을 가지고 있는 형태를 취했으나, 본 논문에서 제안한 구조에서는 하드웨어 자원의 효율적인 공유를 통한 비용절감을 꾀하였다.

III. SMT 구조의 설계

본 프로세서는 ARM ISA구조를 기반으로 한 기존의 슈퍼스칼라 구조에서 파생되었으며, 기존 구조에 약간의 변형을 통하여 만들어졌다. 이 프로세서는 in-order issue, in-order completion을 지원하도록 하여 하드웨어에서의 이득을 취하였다. 대개, 최대 4개 쓰레드를 지원할 경우의 프로세서는 그림 1과 같이 나타난다.

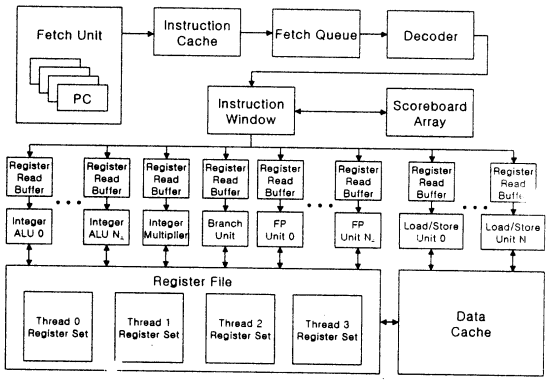

그림 1. 4개의 쓰레드를 지원하는 프로세서의 개략적인 구조

위에서 명령어 캐시 읽기 포트, 페치 큐, 디코더 유닛, 레지스터 파일의 읽기/쓰기 포트, 명령어 윈도우, 동작 유닛, 데이터 캐시, 결과 버스 등의 자원은 쓰레드 간에 동적으로 공유되도록 하였다.

페치유닛은 매 사이클마다 여러 쓰레드들로부터 명령어들을 가져오게 된다. 페치대역폭이 사이클 당 N일 경우, 페치 유닛은 매 사이클마다 페치 우선권 알고리즘에 따라 선택된 M개의 쓰레드들에 대해서 N/M개의 명령어를 가져오도록 하였다. 이를 위해서 명령어 캐시는 M개의 뱅크를 갖는 non-blocking 캐시를 사용하였다. 참고로, 프로그램 카운터(PC)의 개수는 동시에 지원 가능한 쓰레드의 개수 T와 동일해야 한다.

페치된 명령어들은 페치 큐의 끝단에 저장되며, 디코드 유닛은 페치 큐의 앞단에서 명령어들을 가져와서 해석한다. 명령어가 해석되는 동안에, 어떤 동작을 하는지에 따라서 필요한 동작 유닛과 동작에 필요한 레지스터의 주소 또는 즉치값의 존재를 인식하게 된다. 이렇게 순서대로 해석이 끝난 명령어들은 명령어 창에 저장이 된다.

명령어 창에 보관된 각 명령어는 원래 프로그램 상에서의 명령어 실행 순서대로 수행된다. 이를 통해 '쓰기 후 쓰기'의 의존성 문제를 해결할 수 있다. 그리고, 각 쓰레드 별로 자기 이전 명령어 중에 '쓰기 후 읽기' 문제, 자원충돌 문제가 없는 명령어들이 해당 동작 유닛으로 이슈되며, 스코어보드 상의 해당 레지스터 영역에 표시를 함으로써 해당 레지스터의 최신값에 대한 정보를 기록하게 된다.

이슈가 된 명령어들은 해당 연산자를 모든 쓰레드의 레지스터들을 가지고 있는 레지스터 파일로부터 읽어들이는 동작을 취하게 되는데, 이것이 레지스터 읽기 단계이다. 이 단계에서는 다중 사이클 명령의 경우에는 스테이트 머신을 가동시켜 다음 사이클에서도 이 유닛을 점유하면서 필요한 레지스터 값을 읽고, 또 처리해 줄 수 있도록 해 준다.

이렇게 전달된 동작 정보들을 통해 실행 단계에서는 동작 유닛 안에서 주어진 동작이 실행된다. 명령어 실행의 결과는 결과 버퍼에 저장되어 '읽기 후 쓰기'문제를 해결해 준다, '쓰기 후 읽기' 문제를 해결하기 위해서 필요하다면 해당 동작 유닛들로 전달해 줌으로써 레지스터 파일의 읽기 포트 수를 줄일 수 있다.

다음으로 메모리 단계에서는 메모리 제어를 통해서 데이터를 읽거나 쓰게 되는데, 메모리 동작을 수행하지 않는 유닛의 경우에는 하나의 버퍼를 더 두어 전체적인 파이프라인 단계를 맞추어 준다.

끝으로, 모든 결과가 끝나 결과버퍼에 기록된 내용들은 각 동작 유닛별로 연결되어 있는 쓰기 포트를 통해 동시에 레지스터 파일로 기록되고, 해당 스코어보드 비트는 클리어 된다.

앞에서 설명한 파이프라인 구조를 도식적으로 표현하면 그림 2와 같으며, 각각의 상태를 개략적으로 설명하면 다음과 같다.

- 페치 단계(F): 다중 쓰레드들로부터 명령어들을 페치 하여 페치 큐에 저장한다.
- 디코드 단계(D): 명령어들이 해석된다.
- 이슈 단계(I): 종속 문제 또는 자원 충돌 문제가 없는 명령어들이 동작 유닛들로 보내어진다.
- 읽기 단계(R): 명령어 연산자들을 레지스터 파일에서 읽어 오며, 다중 사이클 명령의 경우에는 스테이트 머신을 통해 동작 유닛의 제어를 시작한다.
- 실행 단계(E): 동작 유닛에서 명령어들을 실행하고, 메모리 주소 등을 계산하며 그 결과는 결과 버퍼에 기록된다.
- 메모리 단계(M): 해당 메모리 주소값을 통해 데이터를 읽거나 쓴다.
- 기록 단계(W): 실행 결과가 순서대로 레지스터 파일에 기록된다.

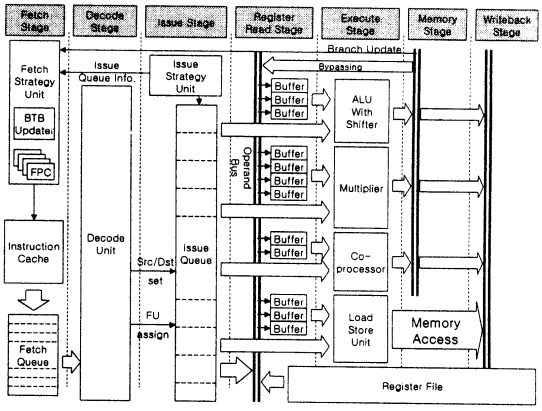

그림 2. 파이프라인 단계 구조도

이 구조에서는 in-order completion 형태를 갖기 때문에 분기 예측 오류 및 예외 발생에서의 복구가 매우 간단해진다. 복구가 필요할 경우에는 단지 분기예측오류 또는 예외가 발생한 명령어가 속한 해당 쓰레드에서 그 이후의 명령어들을 비워버리면 되기 때문이다. 여기서, out-of-order issue를 채택할 경우에는 in-order completion을 채택하였기 때문에 순환 종속 문제가 발생할 수 있다. 이는 같은 동작 유닛을 사용하는 명령어 중 아직 이슈되지 않은 명령어 이후의 명령어가 먼저 이슈될 경우에 발생할 수 있는데, 이는 동작 유닛 상에서 결과 버스를 통해 기록이 되는 구조상에서 이전 명령어가 먼저 결과 버스를 통해 기록이 된 후에야 다음 명령어가 기록할 수 있고, 그 전까지는 동작 유닛을 점유하면서 대기해야 하는 문제가 있기 때문에 발생한다. 이를 해결하기 위해서 결과 버스와 연결되는 동작 유닛 내의 결과 버퍼를 늘리는 방법이 있다. 그러나, 근본적인 해결을 위해서는 재순서 버퍼를 추가해야 하는데, 본 논문에서는 in-order issue를 채택하여 저 비용을 통한 SMT구조의 구현을 꾀하였다.

파이프라인 단계 내내 각 명령어들은 쓰레드 구분자와 함께 다니게 되며, 동시에 여러 쓰레드들의 수행을 지원하기 위해서 아래와 같은 변화가 이루어지게 된다.

- 명령어 캐시 및 데이터 캐시는 non-blocking형태를

가지며 여러 뱅크로 구성된다.
- 여러 개의 프로그램 카운터들과 페치 유닛 상에 쓰레드 선택을 위한 구조가 존재한다.
- 각 분기 내력 버퍼 항목에 쓰레드 구분자를 포함하게 한다.
- 종속관계의 확인, 명령어 비우기 수법을 수행할 때는 쓰레드 별로 수행한다.
- 레지스터 파일의 크기는 기존의 슈퍼스칼라 구조가 가진 레지스터 파일의 T배가 되게 한다.

위에서 언급한 변형 가운데, 레지스터 파일을 제외한 나머지들은 적은 하드웨어의 추가만으로 가능하다.

IV. 최적화 및 검증

이 구조의 성능을 최적화하기 위하여, 먼저 사이클 단위로 수행되며 프로그램을 직접 수행하는 종류의 시뮬레이터를 작성하였다. 기본 구조로는 ARM 버전 5 구조를 채택하였으며 C언어를 사용하여 작성하였다. 프로그램상의 파이프라인 흐름은 다음과 같다.

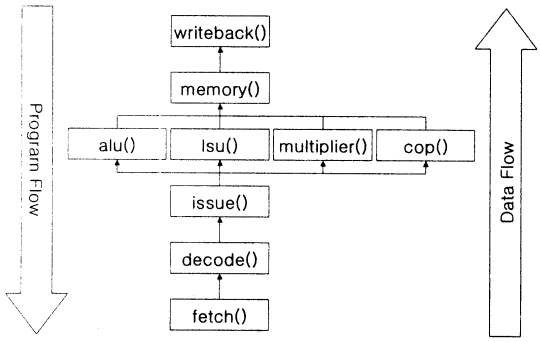

그림 3. 시뮬레이터 상의 프로그램 흐름과 데이터 흐름

이 시뮬레이터를 통한 모의실험에는 SPEC2000 벤치마크 모음집에 있는 개개의 독립적인 프로그램을 쓰레드로 사용하였고, 명령어 캐시 및 데이터 캐시에는 뱅크 충돌이 없는 것으로 간주하였다. 그리고, 성능 측정 지표로서 사이클 단위의 속도 증가율을 사용하였다.

페치 및 이슈 방침, 페치 및 이슈 대역폭, 지원 가능한 쓰레드의 수 등 여러 인자들의 변화를 통하여 여러 가지 프로세서 설정에 대한 모의실험 결과를 얻었다.

페치 쓰레드 수/사이클은 1, 2, 4개로 변화시켜 실험하였고, 아래와 같은 페치 전략의 종류를 실험하였다.
- 페치 큐와 이슈 큐 내의 명령어 수가 가장 적은 쓰레드에 우선권을 주는 방식
- 페치 큐와 이슈 큐를 포함해서 프로세서 내의 모든 파이프라인 상에 존재하는 명령어 수가 가장 적은 쓰레드에 우선권을 주는 방식
- 분기누적버퍼에 가장 많은 명령어를 갖는 명령어에게 낮은 우선권을 주는 방식
- 이슈 큐에서 가장 앞에 있는 명령어의 쓰레드에 대해서 우선권을 주는 방식
- Round Robin방식
- 대기로드리스트를 포함해서 로드 큐에 존재하는 명령어의 수가 가장 작은 쓰레드에 우선권을 주는 방식

이슈 전략의 종류로는 아래와 같은 방식을 테스트해 보았다.
- 전체 동작 유닛 내의 명령어 수가 가장 적은 쓰레드에게 우선권을 주는 방식
- 각 종류별 동작 유닛 내의 명령어수에 따라 가중치를 주어 쓰레드에게 우선권을 주는 방식

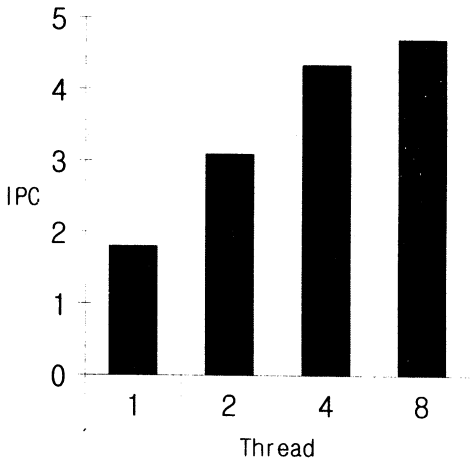

그림 4. 최적의 구조에서의 평균 IPC

그 중에서 성능 변화에 가장 중요한 역할을 하는 페치와 이슈 전략에 대하여 비교해 본 결과, 아래와 같은 설정을 가진 경우에 비용대비 최적의 성능을 얻을 수 있었다.
- 페치 쓰레드 수/사이클 : 매 사이클마다 전체 쓰레드들 중 2개에게 페치권한이 주어질 경우
- 페치 전략 : 페치 큐와 이슈 큐의 명령어 수에 기준을 두고서 명령어 수가 가장 적은 쓰레드에 최우선권을 주는 방식을 채택한 경우
- 이슈 전략 : 전체 동작 유닛 내의 명령어 수가 가장

적은 쓰레드에게 최우선권을 주는 방식을 채택한 경우

모의실험 결과, 본 논문의 구조를 지니는 4개 또는 8개의 쓰레드를 지원하는 프로세서의 성능은 동일한 자원과 이슈 방침을 지닌 일반적인 슈퍼스칼라 프로세서보다 두 배 이상인 것을 알 수 있었다.

V. 결론

본 논문에서는 명령어 단위 병렬성(ILP)과 쓰레드 단위 병렬성(TLP)을 통해 명령어 처리량을 향상시키는 in-order issue, in-order completion 형태의 SMT 프로세서 구조의 최적화에 대하여 설명하였다. 이 구조에서는 대부분의 시스템 자원들이 공유되고 동적으로 할당된다. 이 프로세서는 ILP와 TLP, 동적 자원 공유의 효용성을 높이기 위해서 여러 쓰레드로부터 동작 유닛들로 명령어들을 동시에 이슈하도록 하였으며, in-order completion을 사용하여 하드웨어 복잡도를 크게 감소시킬 수 있었다. 이러한 병렬 수행은 기존의 슈퍼스칼라 프로세서들에 최소한의 추가적인 하드웨어 비용을 들여서 성능과 자원 활용도를 향상시켜주었다. 여기서, 시스템 자원 및 내부 알고리즘의 변형 등을 통하여 최적화된 SMT 구조에 맞는 프로세서 설정을 알아보았다. 모의실험결과, 최대 4개, 8개 쓰레드를 지원하는 프로세서들은 동일한 자원과 이슈방침을 지닌 기존의 슈퍼스칼라 프로세서보다 두 배 이상의 성능향상을 보이는 것으로 나타났다.

참고문헌(Reference)

[1] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, Dean M. Tullsen, "SIMULTANEOUS MULTITHREADING: A Platform for Next-Generation Processors", September/October 1997 IEEE Micro, p.p. 12~19

[2] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," Proc. 23rd Annual International Symposium on Computer Architecture, pp.191-202, May 1996.

[3] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa, "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," Proc. 19th Annual International Symposium on Computer Architecture, pp.136-145, May 1992.