UltraSPARC을 위한 명령어 스케쥴링

윤한샘　문수묵
서울대학교 전기공학부

요 약

근래에 개발된 마이크로프로세서는 대부분 여러 개의 명령어들을 동시에 수행시킬 수 있는 슈퍼스칼라 구조를 채택하고 있는데 Sun의 UltraSPARC, Intel의 Pentium-Pro/P6, IBM의 PowerPC 계열 등이 그 예이다. UltraSPARC와 같이 하드웨어가 명령어 스케쥴링을 하지 않는 경우 프로세서의 자원을 최대한으로 활용하기 위해서는 컴파일러에 의한 명령어 스케쥴링이 필요한 것으로 알려져 있다. 소프트웨어 파이프라이닝 (software pipelining)은 여러 iteration의 명령어들 사이에 존재하는 명령어 수준 병렬성 (instruction-level parallelism) 을 이용하는 루프 스케쥴링 기법이다. EPS (Enhanced Pipeline Scheduling) 은 소프트웨어 파이프라이닝 기법의 하나로 정수 프로그램 (integer program) 에서 흔히 볼 수 있는 불규칙적인 루프 (irregular loop) 에 특히 적합한 것으로 알려져 있다. 이 논문에서는 EPS를 이용한 UltraSPARC 명령어 스케쥴링 기법을 소개하고 SPECint95 벤치마크를 대상으로 한 실험결과를 보여준다.

제 1 절　서론

VLIW와 inorder 슈퍼스칼라 프로세서처럼 하드웨어가 명령어를 재배치하지 않는 프로세서의 자원을 최대한으로 활용하기 위해서는 고도의 컴파일러 기법이 필요하다는 것은 널리 알려져 있다. 이와 같은 프로세서들은 설계하기 간단하지만 컴파일러는 복잡해질 수 밖에 없는데 컴파일러가 정적으로 프로그램을 분석하고 명령어들의 순서를 정확하게 결정해야 하기 때문이다. 재배치되지 않은 코드는 수행 도중 자주 파이프라인이 멈추게 되는데 이는 어떤 명령어가 멈추게 되었을 때 프로세서가 앞으로 수행될 명령어를 순서를 바꾸어서 먼저 수행할 수 없기 때문이다.

UltraSPARC 프로세서는 동시에 4개의 명령어까지 수행시킬 수 있는 슈퍼스칼라 구조를 채택하고 있는데 하드웨어 스케쥴링을 하지 않는 inorder 슈퍼스칼라 프로세서이므로 컴파일러에 의한 스케쥴링이 아주 중요하다. 가장 최근의 Sun 컴파일러는 (version 5.0) modulo scheduling을 이용하여 UltraSPARC에 적합한 코드를 생성하고 있지만 정수 프로그램 (non-numerical code) 쪽에는 여전히 개선의 여지가 있다. 정수 프로그램들은 많은 조건 분기문과 포인터에 기반한 자료구조를 가지고 있는데 이들의 "hot-spot"은 주로

복잡한 제어구조를 가지고 반복 횟수가 상수값이 아닌 루프들이다. 이러한 불규칙한 루프 (irregular loop) 들은 스케쥴링하기 까다로운데 Sun에서 채택한 modulo scheduling도 이러한 루프들을 스케쥴링하는 것을 포기했다.

이 논문에서는 EPS (Enhanced Pipeline Scheduling)[1] 를 이용한 UltraSPARC을 위한 명령어 스케쥴링 기법을 소개한다. EPS는 명령어 이동에 기반한 소프트웨어 파이프라이닝(software pipelining) 기법의 하나로 불규칙한 루프들도 간단하게 소프트웨어 파이프라인 할 수 있어서 특히 정수 프로그램에 적합하다. EPS는 DAG 스케쥴링을 여러번 반복하여 루프를 점점 조밀하게 변화시키는데 명령어를 루프의 역연결선 (back edge) 을 지나 이동시킴으로써 소프트웨어 파이프라인하는 효과를 낸다. EPS를 위한 DAG 스케쥴링 기법으로 selective scheduling[2] 을 이용한다.

이 논문의 나머지는 다음과 같이 구성되어 있다. 2장에서는 UltraSPARC의 그룹화 규칙(grouping rule)을 간략히 정리하고 3장에서는 EPS와 selective scheduling을 이용한 UltraSPARC을 위한 스케쥴링 기법을 설명하고 4장에서는 SPEC95 벤치마크 프로그램에 대한 실험결과를 보여준다.

제 2 절 UltraSPARC의 그룹화 규칙

UltraSPARC은 명령어 스트림을 그룹화 규칙에 위해 그룹단위로 나누어서 한 그룹에 속한 명령어들을 동시에 수행시킨다. 그룹화 규칙은 크게 자원 제약 조건 (resource constraint) 와 의존성 제약 조건 (dependency constraint) 로 분류할 수 있다.

2.1 자원 제약 조건

UltraSPARC는 동시에 최대 4개의 명령어까지 수행할 수 있는데 2개의 정수 명령어, 1개의 분기 명령어, 1개의 메모리 인출/저장 명령어를 수행할 수 있다.[1] 정수 명령어 실행 유니트는 IEU0, IEU1 두개로 구분되고 정수 명령어는 shift 명령어, compare 명령어, 산술/논리 명령어 세가지로 구분되는데 앞의 두 부류는 각각 IEU0, IEU1에 의해 수행되고 마지막 부류는 어느 것에 의해서도 수행될 수 있다. 단 산술/논리 명령어와 shift 명령어는 이 순서로 같이 수행될 수 없다. [2] 조건 분기 명령어와 그것의 뒤에 등장하는 명령어들[3]도 같이 수행될 수 있어서 컴파일러는 병렬화 명령어를 트리 구조로 모델화했다. 이러한 제약 조건외에도 명령어 캐쉬 정렬 (I-Cache alignment) 에 의해 그룹이 부서지는 경우도 있는데 다소 복잡하므로 생략한다.

[1] 여기서는 정수 프로그램만을 대상으로 했으므로 부동 소수점 명령어는 생각하지 않기로 하자.
[2] 컴파일러가 두 명령어의 순서를 바꿔주기만 하면 같이 수행될 수 있다.
[3] 분기 예측에 의해 결정된 명령어 스트림으로부터 인출된 명령어들

2.2 의존성 제약 조건

그룹 내의 의존성 제약 조건과 복수 지연 명령어 (multi-latency operation) 들로 인한 의존성 제약 조건으로 나누어 설명한다. 그룹내에 flow 의존성과 output 의존성은 허용되지 않고 역 의존성 (anti-dependency)은 허용된 다. 그룹내에서의 flow 의존성이 허용되는 특별한 두 경우가 있는데 compare 명령어와 조건 분기문은 같은 그룹에 있을 수 있고 정수 명령어의 목적지 레지스터가 메모리 저장 명령어의 주소로 사용될 때 이들은 동시에 실행될 수 있다. 복수 지연을 가지는 명령어로는 메모리 소환 명령어와 곱셈/나눗셈 명령어가 있는데 전자의 경우 지연 시간이 변할 수 있다. 부호있는 메모리 소환 명령어 (signed load operation)는 항상 지연 시간이 3 싸이클이고 부호없는 메모리 소환 명령어 (unsigned load operation) 는 2 싸이클인데 다음과 같은 경우 3 싸이클이 된다. 부호있는 메모리 소환 명령어가 일단 인출되면 메모리 소환 명령어가 인출되지 않는 싸이클이 나타날 때까지 뒤따르는 모든 부호없는 메모리 소환 명령어의 지연시간은 3 싸이클이 된다.

제 3 절 UltraSPARC을 위한 스케쥴링 기법

순차적 프로그램 (sequential program) 의 각 프로시져 (procedure) 에 대한 구간 분석 (interval analysis)을 통하여 얻은 루프의 트리 형태의 계층 구조를 post-order로 돌면서 각 루프를 EPS(Enhanced Pipeline Scheduling)과 selective scheduling 기법을 이용하여 스케쥴링한다. EPS는 다음의 두 과정을 반복하여 루프를 소프트웨어 파이프라인한다. 먼저 루프의 어떤 간선을 끊어서 DAG을 만든 후 끊은 간선의 자리에 병렬화 그룹 (parallel group) 을 DAG 스케쥴링을 적용시켜 만든다. 나중에 생성되는 병렬화 그룹은 앞서 만들어진 병렬화 그룹을 흡수하여 점점 조밀하게 된다. 그 결과 최종적으로 생성되는 파이프라인 커널 (pipeline kernel) 은 아주 조밀하다. EPS를 위한 DAG 스케쥴링 기법으로는 selective scheduling을 사용하는데 이 기법을 통해 수행 가능한 모든 경로에서 많은 유용한 명령어들을 찾아내어 스케쥴링할 수 있다. selective scheduling의 큰 장점중 하나는 flow 의존성만 없으면 레지스터가 부족하지만 않으면 항상 코드 이동을 할 수 있다는 것인데 false 의존성은 partial renaming으로 제어 의존성은 speculative 코드 이동을 통해서 극복한다.

그림1,2에는 SPEC95 벤치마크 perl의 “eval” 함수의 한 루프를 소프트웨어 파이프라인 한 예를 보여주고 있다. 루프는 두 개의 경로를 가지고 있는데 스케쥴링을 전혀 하지 않으면 각각의 경로의 길이는 11, 10 싸이클이다. 명령어 이동 범위를 기본 블럭 내로 제한하면 10, 9 싸이클, DAG내로 제한하면 7, 6 싸이클 까지만 줄일 수 있다. 소프트웨어 파이프라인을 허용하지 않을 경우 루프 독립 의존성 사슬 (loop-independent dependency cahin)

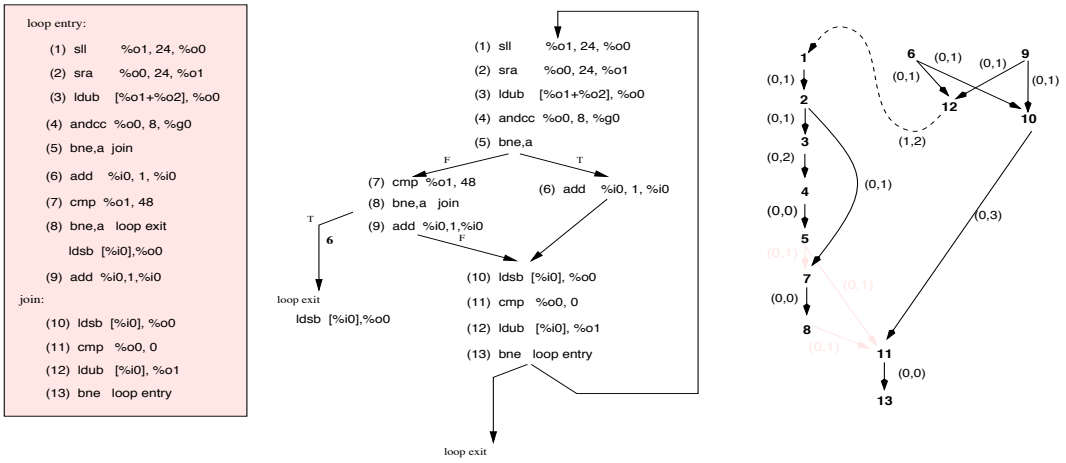

그림 1: Sequential representation, CFG representation and dependence graph of an example loop

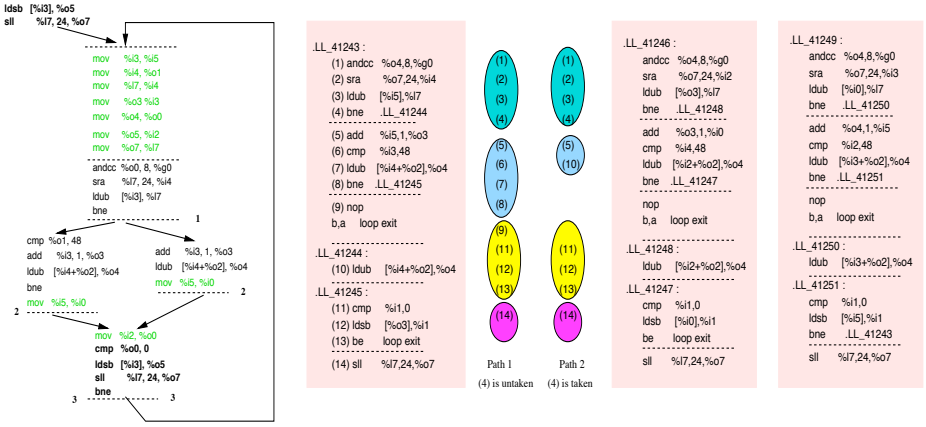

그림 2: Software pipelined loop after scheduling and unroll & copy elimination

`sll %o1,24,%o0 => sra %o0,24,%o1 => ldub [%o1+%o2],%o0 => andcc %o0,8,%g0`

으로 경로의 길이가 정해지는데 소프트웨어 파이프라인을 통해 이 긴 사슬을 끊어주면 경로의 길이를 더욱 줄일 수 있다. 소프트웨어 파이프라인으로 인해 명령어가 여러 iteration에 걸쳐 이동하게 되면 그것의 유효 범위 (live range) 가 여러 iteration에 걸쳐 있을 수 있는데 이 때 cross-iteration anti-dependency는 반드시 생기게 되어 있고 selective scheduling은 partial renaming을 통해 스케쥴링할 때는 복사 명령어 (copy operation) 로 여러 iteration에 걸친 긴 유효 범위를 분할해 놓는다. 프로세서의 자원이 풍부하면 이 복사 명령어를 제거하지 않아도 성능이 크게 나빠지지 않지만 UltraSPARC와 같이 자원이 VLIW 프로세서에 비

해 적은 경우 성능이 크게 나빠질 수 있기 때문에 복사 명령어들을 제거해야 한다. 이 복사 명령어들은 파이프라인 커널을 적당히 펼쳐 주면 (unroll) 대부분 융합 (coalesce) 하여 없앨 수 있다.[3] 그림 2에는 복사 명령어가 많이 쌓여 있는데 루프를 두 번 펼치면 모두 융합하여 없앨 수 있다. 그 결과 4, 4싸이클까지 경로의 길이를 줄일 수 있다.

외각 루프(outer loop) 의 경우 루프를 펼치기 힘들기 때문에 소프트웨어 파이프라인을 최내각 루프처럼 복사 명령어를 남기면서 하게 되면 융합되지 않는 (uncoalescible) 많은 복사 명령어를 남길 수 있어서 외각 지역에 대해서는 스케쥴하기 전에 무한개의 의사 레지스터 (pseudo register) 를 할당하여 거짓 의존성(false dependency)을 많이 제거한 후 복사 명령어를 남기지 않고 스케쥴한 후 하드웨어 레지스터를 할당한다. 이 때 spill이 발생하지 않기 위해 스케쥴할 때는 레지스터 압력 (register pressure) 을 적절히 조절한다.

제 4 절 실험 결과

벤치마크로는 SPEC95 정수 프로그램 8개를 사용했다. 기본 컴파일러 (base compiler) 로는 gcc 컴파일러 (version 2.8.1) 최적화 레벨 -O4 을 사용하고 Ultra5 270-MHz, Solaris 2.6, 단일 사용자 (single-user) 모드에서 실험했다. 똑같은 실행파일이 디스크의 다른 위치에 있을 때 그림 3 과 같이 성능의 변화가 심한 것을 확인 할 수 있었고 성능 측정은 10개의 똑같은 실행파일을 만들어서 이들의 평균 수행시간과 가장 빠른 수행시간을 비교 했다. 그림 4에는 평균 수행시간의 비교가 나타나 있는데 평균 5.4% 빨라졌음을 확인 할 수 있고 그림 5에는 가장 빠른 수행시간을 비교한 결과가 있는데 2.9%로 성능 향상폭이 줄어들었다.

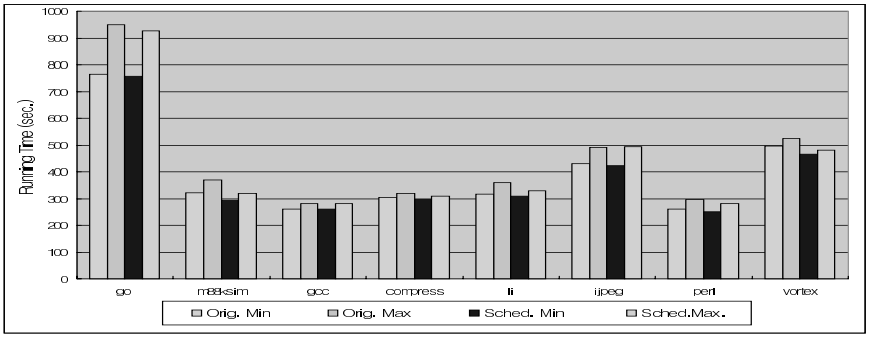

그림 3: Fluctuation of running time

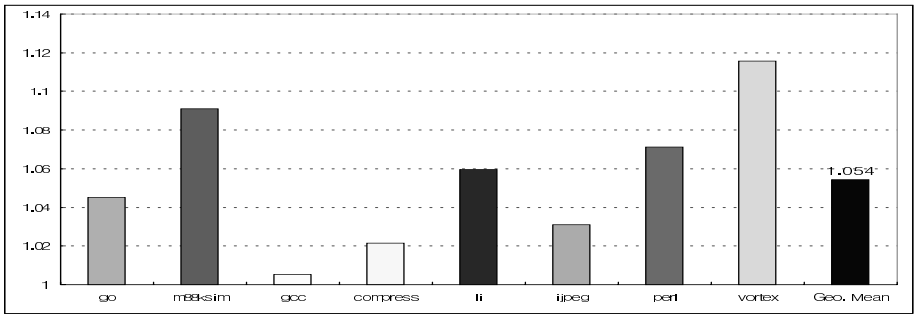

그림 4: Comparison of average running time

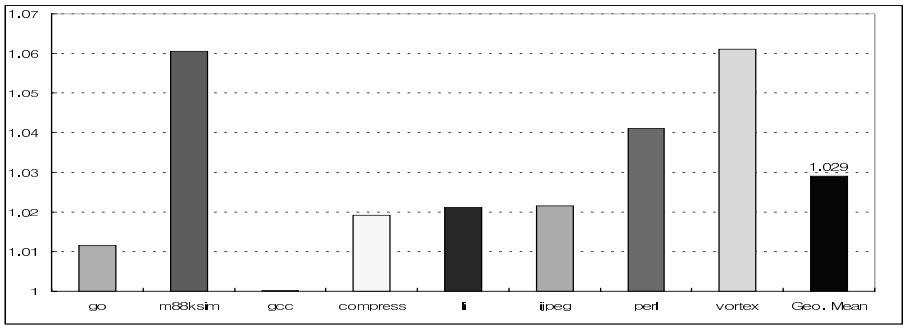

그림 5: Comparison of fastest running time

6에는 기본 블럭 스케쥴링, DAG 스케쥴링, 소프트웨어 파이프라이닝의 세가지 코드 이동 범위에 대한 스케쥴링 효과가 나타나 있다. 대체로 코드 이동 범위를 넓혀감에 따라 성능이 향상되었으나 gcc의 경우는 코드 이동 범위가 기본 블럭을 넘어가면 항상 성능이 나빠지고 go, perl, vortex는 DAG 스케쥴링은 효과가 있으나 소프트웨어 파이프라인은 효과가 없었다.

코드 이동 범위를 넓힌 것이 큰 효과를 나타내지 못하는 것은 다음과 같은 과감한 스케쥴링의 역효과 때문인 것으로 추정된다. 먼저, 그림 7에 나타나 있는 코드 팽창으로 인한 명령어 케쉬 미스 빈도의 증가인데 특히 gcc와 같이 working set이 큰 프로그램은 상당히 영향을 받을 것으로 생각된다. 다음 speculative bookeeping copy 코드들이 문제가 될 수 있는데 이들로 인하여 프로세서는 스케쥴링 하지 않은 코드보다 더 많은 코드를 수행해야 하는데 UltraSPARC과 같이 병렬화 그룹이 캐쉬 미스나 분기 예측 미스에 의해

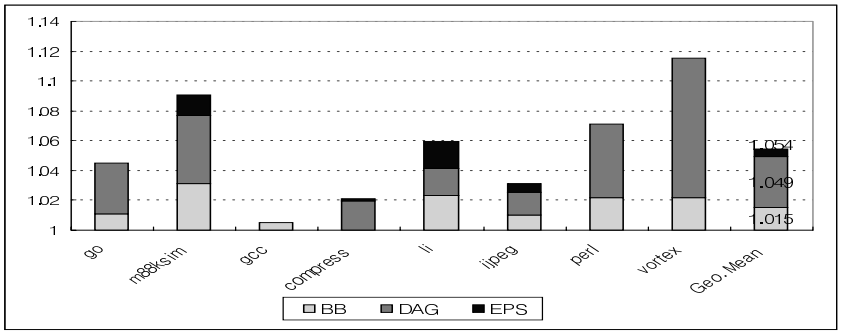

그림 6: Comparison of average running time

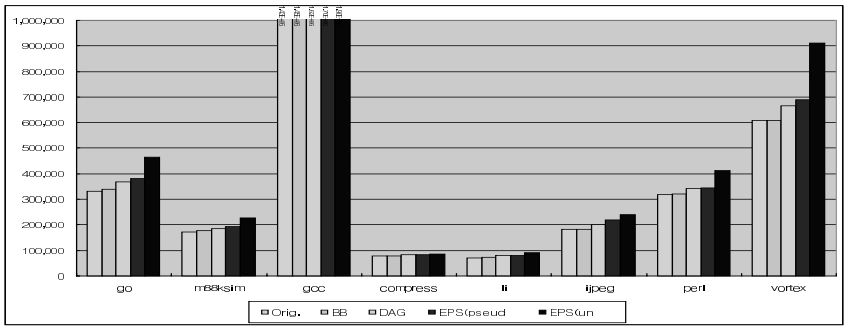

그림 7: Code expansion ratios

부서지기 쉬운 경우에 오히려 나빠질 수도 있을 것이다. 그 외에 융합되지 않는 복사 명령어와 복잡해진 제어 구조로 인해 추가로 필요해지는 무조건 분기 명령어 (unconditional branch) 도 부담이 될 수 있다.

제 5 절 결론

이 논문에서는 EPS와 selective scheduling 기법을 이용하여 UltraSPARC 의 자원을 최대한 활용하게 해주는 명령어 스케쥴링 기법을 소개하였다. SPEC95 정수 프로그램을 벤치마크로 한 실험에서는 유망한 결과가 나왔다. 앞으로는 시뮬레이터를 통한 과감한 스케쥴링의 역효과 분석및 조정 (tuning) , EPS의 발전된 형태인 Split-path EPS[4], 명령어 캐쉬 정렬, 스케쥴 후의 최적화 기법 (post-schedule optimization) 등을 통해 더욱 높은 성능을 얻을 수 있도록 할 것이다.

참고 서적

[1] K. Ebcioğlu and T. Nakatani. A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW architecture. In *Languages and Compilers for Parallel Computing*, pages 213–229. MIT Press, 1989.

[2] S.-M. Moon and K. Ebcioğlu. Parallelizing Non-numerical Code with Selective Scheduling and Software Pipelining. *ACM Transactions on Programming Languages and Systems*, 1997.

[3] S. Kim, J. Park, and S.-M. Moon. Unrolling Based Copy Coalescing. In *Submission to Proceedings of the SIGPLAN 1997 Conference on Programming Language Design and Implementation*, 1997.

[4] SangMin Shim and Soo-Mook Moon. Split-Path Enhnaced Pipeline Scheduling for Loops with Control Flows. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (Micro-31)*, pages 93–102, Dec. 1998.