ILP 마이크로프로세서에서 메모리 주소 모호성 제거의 성능 영향

정 회 목, 양 병 선, 문 수 묵
서울대학교 전기공학부

Effect of Memory Disambiguation for ILP Microprocessors [1]

HoeMok Chung, ByungSun Yang, Soo-Mook Moon
School of Electrical Engineering, Seoul Nat'l Univ.

요 약

ILP 마이크로프로세서를 위한 스케쥴링 과정에서 메모리 명령어가 프로그램의 임계 경로에 존재할 경우에 이의 스케쥴링은 성능 향상에 중요한 문제 중에 하나이다. 메모리 명령어의 원활한 코드 이동을 위해서는 장애가 되는 명령어들의 메모리 주소 간의 의존성의 분석을 필요로 한다. 본 논문에서는 컴파일 시간에 메모리 주소 간의 의존성 분석을 통한 성능 향상도를 VLIW 환경 하에서 비교한다. 실험 결과, 컴파일 시간에 메모리 주소 모호성 제거기를 사용한 경우 16 ALU 프로세서에서 정수 벤치마크 프로그램에 대해서 기하 평균으로 약 3 6%의 성능 향상이 가능하다

제 1 절 서 론

ILP (Instruction Level Parallelism) 마이크로프로세서의 성능을 효과적으로 이용하기 위해서는 강력한 스케쥴링 컴파일러가 필수적이다. 본 논문에서는 강력한 전역 스케쥴링 기법으로 알려진 '선택 스케쥴링 (Selective scheduling) 기법'과 '향상 파이프라인 스케쥴링 (Enhanced pipeline scheduling, EPS) 기법'을 이용한다 [1, 2] 스케쥴링 과정에서 메모리 로드 명령어가 프로그램 상의 임계 경로(critical path) 상에 존재하는 경우에 이들 메모리 명령어들에 대한 재배치(reordering)가 제대로 되지 않을 경우에는 충분한 성능 향상을 기대하기 어렵다.

메모리 명령어 간의 재배치를 위해서 이들 명령어 간의 의존성을 분석하여야 한다. 즉 store 명령어와 로드 명령어간의 의존성 분석을 필요로 한다. 그러나 일반 명령어와는 달리 메모리 명령어 간의 의존성 분석은 용이하지 않다 [3, 4] 본 논문에서는 스케쥴링 컴파일러에서의 메모리 주소 의존성 분석 기법을 살펴보고 그 성능 영향을 분석한다 본 논문의 구성은 먼저 2절에서 메모리 주소 의존성 분석 기법에 대해 살펴보고 실험 결과를 3절에서 보이며 4절에서 결론을 짓는다

제 2 절 메모리 주소 의존성 분석

메모리 로드 명령어는 일반적으로 프로그램의 임계 경로에 있는 경우가 많으므로 충분한 ILP를 추출하기 위해서는 메모리 로드 명령어의 자유로운 코드 이동이 필수적이다 이러한 메모리 명령어 간의 재배치는 그들 사이의 데이터 의존성이 없을 경우에만 가능하므로 스케쥴링 과정에서 이러한 의존 관계를 분석해야 한다. 다음의 그림 1에서 (a)의 경우 명령어 (2)가 명령어 (1)의 결과를 사용한다는 의존성 관계를 쉽게 확인할 수 있다. 이에 비해 (b)의 경우에는 ld와 st 간의 의존성을 파악하기 위해서는 이들이 접근하는 메모리 주소 '%r2 + 4'와 %r3 + %r4' 간이 같은지 다른지를 분석해야 한다. 따라서 두 메모리 명령어가 같은 메모리 주소를 접근하는지를 분석하기 위해서는 복잡한 작업을 필요로 하는 데, 이를 **메모리 주소 모호성 제거**(memory disambiguation)이라 한다.

```
add %r1, %r2, %r3  (1)        st %r1, [%r2 + 4]     (1)
slt %r3, %r4, %cc0 (2)        ld [%r3 + %r4], %r5   (2)
  (a) ALU operation             (b) Memory operation
```

그림 1 ALU 명령어와 메모리 명령어의 의존성

메모리 주소 모호성 제거기는 우선 주어진 코드를 분석하여 각 메모리 명령어들의 접근 주소를 주소식(address derivation)이라는

[1]본 연구는 1996년 한국과학재단의 연구비 지원 (961-0908-044-1)을 받아 수행되었습니다

문자식으로 표현하고 이를 비교하여 두 메모리 명령어 간의 의존성을 분석한다.

2.1 메모리 주소식 (Address Derivation)

주소식은 메모리 명령어가 접근하는 메모리 주소를 나타내기 위해 전역 변수의 주소, 스택 포인터, 정수, 루프의 유도 변수(induction variable), 그리고 고유 문자 상수와 같은 변수를 이용하여 표현되는 문자식이다. 그림 2에는 메모리 주소식을 구하는 예를 보여주고 있다. (a)의 ld의 경우 '%r5 - 1'은 '_a + 4'가 됨을 알 수 있고 st는 '%r1 + %r7'에서 %r7이 call에 의해 정의되므로 새로운 변수가 필요하고 이를 편의상 unknown이라 하면 주소식은 '_a + unknown'이 된다. (b)는 (a)와는 달리 조건 분기가 있어서 한 레지스터를 정의하는 명령어가 여러 개 존재할 수 있어 ld의 경우 '%r1 + %r4'가 '_a + 2' 또는 '_a + 1'이 됨을 알 수 있다. (c)는 루프 안에 존재하는 메모리 명령어의 예를 보여준다. %r2가 유도 변수이고 그 초기값이 0이고 매 반복(iteration)마다 1씩 증가한다면 ld의 주소식은 '_a + ind0 * 1 + 2'로 나타낼 수 있다. 여기서 ind0는 루프의 유도변수를 나타내고 '1'은 유도 변수가 1씩 증가 증가함을 나타낸다.

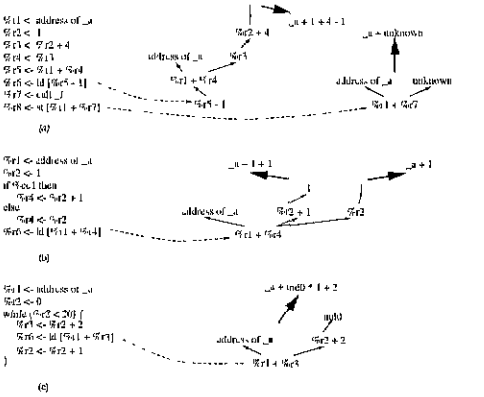

그림 2. 메모리 명령어와 주소식

2.2 모호성 제거 검사(Disambiguation Test)

두 메모리 명령어가 접근하는 메모리 주소가 겹치는 경우는, 각 명령어의 주소식은 aderiv1, aderiv2라 하고 각 명령어가 접근하는 메모리 데이터의 크기를 len1, len2라 할 때에, $-len1 < aderiv1 - aderiv2 < len2$와 같은 관계가 성립할 때이다. 이 때에 핵심이 되는 검사는 두 주소식의 차이가 0이 될 수있는지를 알아내는 것이다. 이 차이식은 유도 변수와 같은 정수형 변수와 정수형 상수, 계수를 사용하는 식이므로 이러한 검사는 다차 정수 방정식 (polynomial diophantine equation)의 정수해가 존재하는 지를 검사하는 문제로 볼 수 있다. 즉 차이식을 0로 만드는 정수해가 존재하면 두 메모리 명령어 사이에 의존 관계가 존재한다고 볼 수 있다. 현재 본 연구팀의 스케쥴링 컴파일러에는 차수가 일차인 경우에 대해서만 고려하고 있는 데 이는 대부분의 프로그램에서 2차이상의 방정식을 필요로하지 않기 때문이다.

따라서 메모리 주소 모호성 제거를 위해서 다음 방정식의 정수해가 존재하는 지를 판단해야 한다.

$$a_1 i_1 + a_2 i_2 + ... + a_n i_n + c = 0 \ (a_1, a_2, \ ., a_n, c : integer) \quad (1)$$

이 때 변수 $i_1, i_2, ..., i_n$은 유도 변수에 해당하고 모두 하한과 상한을 가진다. 즉 각 i_n은 다음과 같은 조건을 가정한다.

$$lb_k \le i_k \le ub_k, 1 \le k \le n \quad (2)$$

이러한 제한 조건과 선형 정수 방정식을 만족하는 정수해의 존재 여부를 검사하기 위해서 메모리 주소 모호성 제거기에서는 크게 **GCD 검사**와 **Banerjee 검사**를 수행한다. GCD 검사는 선형 정수 방정식이 정수해를 가질 필요충분 조건인 상수항이 변수의 계수들의 최대 공약수로 나누어 떨어진다는 것을 이용한다. GCD 검사는 식 2에 대해서는 아무런 고려를 할 수 없으므로 GCD 검사를 통하여 의존 관계가 존재하는 것으로 판정하여도 실제로는 의존 관계가 없을 경우도 발생한다. 이러한 부정확성의 결과는 추출할 수 있는 ILP의 양을 줄어들게 한다. Banerjee 검사를 위해서는 식 1에서 실수 함수 $g(x_1, x_2, ..., x_n) = a_1 x_1 + a_2 x_2 + ..., a_n x_n$를 만들고 식 2에서 정의되는 닫힌 공간에서 최소값 min과 최대값 max를 다음과 같이 구한다.

$$min = \sum_{i=1}^{n} (a_i^+ lb_i - a_i^- ub_i), \quad max = \sum_{i=1}^{n} (a_i^+ ub_i - a_i^- lb_i)$$

$$a^+ = \begin{cases} a & \text{if } a \ge 0 \\ 0 & \text{otherwise} \end{cases} \qquad a^- = \begin{cases} -a & \text{if } a \le 0 \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

그 결과 min이 음수이고 max가 양수이면 중간값 정리에 의해 식 1과 식 2를 만족하는 실수해가 존재함을 알 수 있다. Banerjee 검사는 두 식을 동시에 고려한다는 장점이 있지만 정수해가 아닌 실수해의 존재 여부를 판단하는 부정확한 면이 있다. 본 메모리 모호성 제거기에는 두 검사 방법의 장점을 상호 보완적으로 이용하기 위해서 먼저 GCD 검사를 행한 후에 정수해가 존재하는 것으로 판단한 후에 다시 Banerjee 검사를 수행하여 제한 조건도 고려한다.

제 3 절 실험 결과

본 논문의 실험은 본 연구실에서 개발한 SPARC 기반의 VLIW 실험 환경에서 수행되었고 [2] 사용한 벤치마크는 표 1과 같으며 사용된 프로세서 모델은 16 ALU, 8 ALU, 4 ALU, 2 ALU를 가진 것으로 가정한다. 이들 프로세서의 ALU 중에서 반은 메모리 명령어를 처리할 수 있고 또한 ALU 수만큼의 분기 목표를 갖는 다중 분기가 가능하다.

Benchmarks	Lines	Descriptions
eqntott	3616	Truth table generation
espresso	13639	Logic table optimization
li	7481	A lisp interpreter
compress	1421	File compression utility
yacc	6482	A parser generator
sed	3289	A stream editor
gzip	7388	GNU file compression utility
cmp	1755	Comparison of two files

표 1 Benchmark description.

프로세서의 메모리 모델은 Perfect Cache를 가정하고 모든 명령어가 한 싸이클에 수행된다고 가정하였고 speedup은 $\frac{total\ execution\ cycles\ of\ sequential\ execution}{total\ execution\ cycles\ of\ VLIW\ execution}$로 표시되고 ALU의 수에 따른 성능 비교는 그림 3, 4, 5, 6에 표시하였다. 실험 결과에서 메모리 주소 모호성 제거기를 사용하였을 경우에 16 ALU, 8 ALU, 4 ALU, 2 ALU에서 각각 3.6%, 2.0%, 0.8%, 0.1%의 성능 향상이 있음을 알 수 있다.

제 4 절 결 론

메모리 주소 모호성 제거기를 사용시의 성능 향상도는 크게 높지 않다. 이는 GCD 검사와 Banerjee 검사가 기본적으로 수치 계산프로그램에서 배열 주소 접근시에 효과적이기 때문에 정수 프로그램에서는 큰 효과를 나타내지 못하기 때문이다. 앞으로 이를 위한 정수 프로그램에서 효과적인 기법의 개발이 필요하며 또한 컴파일 시간뿐 아니라 수행 시간에 동적으로 의존성를 극복할 수 있는 기법의 적용이 필요할 것이다 [4]

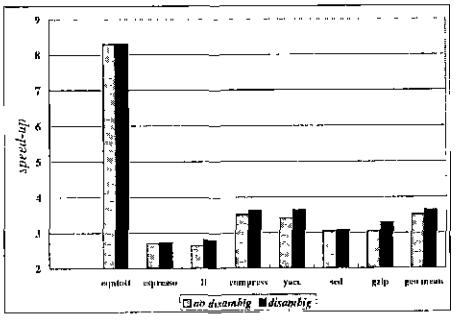

그림 3 Speedup comparison on 16-ALU machine.

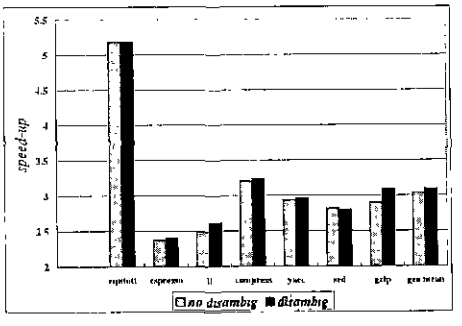

그림 4. Speedup comparison on 8-ALU machine.

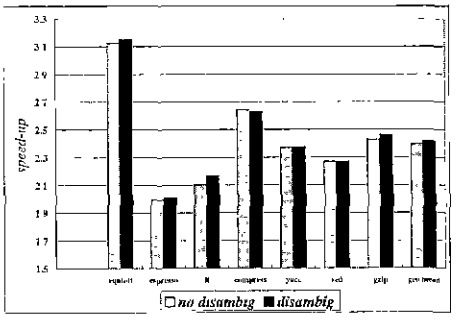

그림 5 Speedup comparison on 4-ALU machine

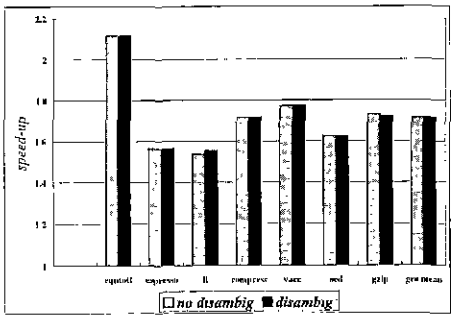

그림 6. Speedup comparison on 2-ALU machine.

참고 서적

[1] S-M. Moon and K Ebcioğlu. Parallelizing Non-numerical Code with Selective Scheduling and Software Pipelining *ACM Transactions on Programming Languages and Systems*, 19(6), November 1997.

[2] Soo-Mook Moon, HoeMok Chung, Jinpyo Park, SangMin Shim, and Jae-Woo Ahn SPARC-based VLIW Testbed. *IEE Proceedings Computers and Digital Techniques*, 145(3), May 1998.

[3] U Banerjee *Loop Transformation for Restructuring Compilers: the Foundations.* Kluwer Academic Publishers, 1993

[4] D.M. Gallagher and et al. Dynamic Memory Disambiguation using The Memory Conflict Buffer. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183 – 193, Oct 1994.