수퍼스칼라 프로세서를 위한 컴파일러에서 조건부 분기의 최적화

김 명 호†, 최 완††

요 약

본 논문에서는 수퍼스칼라 프로세서를 위한 컴파일러에서 조건부 분기 명령을 제거하는 최적화 기법을 제시하였다. 분기를 제거하는 단계적 방법으로 먼저 대수적 규칙을 사용하여 분기를 산술식의 형태로 변형하고, 그 식에 대응하는 명령 수순을 Granlund/Kenner의 GSO를 사용하여 완전 탐색한 후 목적 프로세서에서 실행시 최소의 동적 계수를 갖는 명령 수순을 선택하였다. 제안한 분기 최적화 기법을 SuperSPARC 프로세서와 GNU C 컴파일러를 사용하여 실험한 결과 입력 프로그램에서 최적화 패턴과 대응하는 조건부 분기의 경우 원래의 컴파일러가 생성하는 최적 코드 수순에 비하여 25% 이상의 추가적인 수행시간 개선 효과를 얻을 수 있었다.

Conditional Branch Optimization in the Compilers for Superscalar Processors

Myung Ho Kim† and Wan Choi††

ABSTRACT

In this paper, a technique for eliminating conditional branches in the compilers for superscalar processors is presented. The technique consists of three major steps. The first step transforms conditional branches into equivalent expressions using algebraic laws. The second step searches all possible instruction sequences for those expressions using GSO of Granlund/Kenner. Finally an optimal sequence that has the least dynamic count for the target superscalar processor is selected from the GSO output. Experiment result shows that for each conditional branch in the input program matched by one of the optimization patterns, the proposed technique outperforms more than 25% speedup of execution time over the original code when the GNU C compiler and the SuperSPARC processor are used.

1. 서 론

컴퓨터 하드웨어의 발전은 컴파일러 제작 기술에도 많은 영향을 미치게 되었다. 특히 최근에는 명령 축소형 프로세서(RISC)의 발전으로 인하여 최적화 컴파일러의 중요성이 한층 더 강조되고 있다. 명령 축소형 프로세서의 성능은 프로세서의 하드웨어 측면에서의 성능 향상 뿐만 아니라 이를 활용할 수 있는 최적화 컴파일러에 더욱 의존하기 때문이다. 명령 축소형 프로세서를 위한 컴파일러는 레지스터 할당, 파이프라인 조정, 캐쉬의 활용 등과 같은 기능을 최적화할 수 있어야 한다[1, 2].

수퍼스칼라(superscalar) 프로세서[3, 4]의 기술 발전으로 인하여 마이크로프로세서의 성능이 놀라울 정도로 발전하고 있다. 수퍼스칼라 프로세서는 클럭 사이클당 수행할 수 있는 스칼라 명령의 분량을 증가시켜 전체적인 작업 수행의 성능 향상을 도모한 것이다. 수퍼스칼라 프로세서에서 어떤 프로그램의 성능은 명령의 갯수(정적 계수)보다 한 클럭 사이클에 동시 실행될 수 있는 명령 그룹의 갯수(동적 계수)에 더욱 크게 좌우된다.

이상적인 경우 병행처리가 가능한 n개의 스칼라 프로세서로 구성된 수퍼스칼라 프로세서의 성능은 하나의 스칼라 프로세서로 구성된 경우에

• 본 논문은 '94 한국전자통신연구소 「ATM용 CHILL 컴파일러 연구」 지원에 의한 것임.
† 정 회 원 : 동아대학교 컴퓨터공학과 부교수
†† 정 회 원 : 한국전자통신연구소 책임연구원
논문접수 : 1994년 10월 29일, 심사완료 : 1995년 3월 17일

비하여 n배의 성능 향상이 있어야 한다. 그러나 실제적인 경우 프로그램의 명령 수순에 존재하는 여러가지 종속성으로 인하여 최대 성능을 보장할 수는 없다.

수퍼스칼라 프로세서를 위한 최적화 컴파일러는 스칼라 프로세서를 위한 컴파일러에 비해 명령의 적절한 선택, 명령의 수행 비용 모델에 근거한 명령 수순의 조정, 데이타나 레지스터 종속성의 제거, 분기(특히 조건부 분기) 명령으로 인한 제어 종속성 제거에 치중하여야 한다[5]. 특히 조건부 분기 명령은 동시 실행 가능한 명령 그룹의 크기를 격감시키며, 명령 수순의 조정을 어렵게 할 뿐만 아니라 캐쉬 미스의 확률을 높이는 등의 여러가지 부작용이 있으므로 가능한 한 제거하는 것이 바람직하다.

동적 분기 예측[6, 7] 등과 같은 고급의 마이크로프로세서 제작 기술을 사용하면 조건부 분기의 문제점을 해결할 수도 있지만 잘못된 예측으로 인한 부담 또한 대단히 큰 편이므로 이를 소프트웨어적으로 해결하는 것이 바람직하다. 소프트웨어에 의한 해결은 동적 분기 예측 기능을 지원하지 않는 프로세서에서의 분기 제거를 위한 유일한 방법이기도 하다.

본 논문에서는 수퍼스칼라 프로세서를 위한 컴파일러에서 분기 명령을 제거하는 최적화 기법에 대하여 연구하였다. 분기를 제거하는 단계적 방법으로 먼저 분기 명령을 대수적으로 변형하고, 변형된 분기에 대응하는 명령 수순을 Granlund/Kenner의 GSO[8]를 사용하여 완전 탐색한 후 목적 프로세서를 위한 최적의 명령 수순을 선택하였다. 이러한 최적화 기법은 기존의 파이프라인 조정에 의한 최적화와는 달리 명령 수순을 임의적으로 변경하지 않는다.

본 연구에서는 수퍼스칼라 명령 축소형 프로세서인 SuperSPARC(*SSP*)[9]을 목적 프로세서로 사용하였으며, GNU C 컴파일러(gcc)[10]의 SPARC 후단부에서 실험하였다. 실험 결과 제안한 분기 최적화 기법은 많은 경우 조건부 분기를 수퍼스칼라 특징을 활용하는 최적의 명령 수순으로 변경함으로써 동적 계수를 감소시키는 효과를 얻을 수 있음을 확인하였다. 입력 프로그램에서 최적화 패턴과 대응하는 조건부 분기의 경우 원래의 컴파일러가 생성하는 최적 코드 수순에 비하여 25% 이상의 추가적인 수행시간 개선 효과를 얻을 수 있었다.

2. 수퍼스칼라 프로세서와 분기의 최적화

수퍼스칼라 프로세서는 한 클럭 사이클마다 둘 이상의 스칼라 출력을 생성할 수 있는 프로세서이다. 이러한 프로세서는 어떤 프로그램을 실행하기 위한 명령의 정적 계수로부터 동적 계수를 쉽게 유추할 수 없는 특징이 있으며, 이는 결과적으로 성능 측정을 어렵게 하는 요인으로 작용한다. 따라서 최적화 컴파일러의 역할도 명령 수순의 정적 계수를 축소하기 보다는 동적 계수를 축소하는데 주력하여야 한다[1, 2].

2.1 SuperSPARC 수퍼스칼라 프로세서

본 연구의 목적 프로세서인 SSP는 명령 축소형 프로세서인 SPARC의 수퍼스칼라 확장으로 기존의 SPARC을 위한 프로그램을 수정없이 실행할 수 있으면서도 한 클럭당 1.4-1.6개의 명령을 실행할 수 있는 특징이 있다. 그러나 수퍼스칼라 특징을 활용하는 최적화 컴파일러가 수반될 경우 한층 더 효율을 개선할 수 있다.

SSP는 한 클럭당 최대 세개의 스칼라 명령을 수행할 수 있으며, 현재 명령에 후속하는 몇 개의 명령을 동적으로 검조하여 동시 실행 가능성을 판정한다. 그러므로 SSP에서 실행되는 프로그램의 성능은 수퍼스칼라 처리를 위하여 명령들이 어떻게 그룹을 형성하는가에 크게 좌우된다.

SSP는 가용한 하드웨어 자원에 의거하여 동시 실행이 가능한 명령의 그룹을 동적으로 형성하므로 이에 대한 자세한 이해가 필요하다. 명령의 동시 실행을 저해하는 SSP 자원의 대표적인 제한사항 중에서 본 연구와 관련된 것은 다음과 같다.

(1) 정수 레지스터 출력 포트
정수 레지스터 화일을 위하여 두개의 출력 포트가 제공된다. 모든 산술 연산은 이들 중 하나의 포트를 사용한다. 적재 명령도 하나의 포트를 사용한다.

(2) 정수 계산을 위한 ALU
SSP는 하나의 명령 그룹에 두개까지의 산술 연산을 수행할 수 있다. 쉬프터는 하나만 있으며, 한 명령 그룹 내에서 다른 연산의 결과가 쉬프터의 입력으로 연결될 수 없다.

SSP에서 어떤 명령을 동시 실행을 위한 명령의 그룹에 포함시킬 것인지 혹은 제외할 것인지에 관한 것은 여러가지 규칙에 따른다. 명령들의 그룹을 판정하는 규칙은 "후분리 규칙"과 "전분리 규칙"의 두가지 유형이 있다. 이들 규칙은 어떤 그룹이 특정 명령 이후 혹은 이전에 분리되어야 함을 의미한다.

다음의 (1)-(3)에 해당하는 명령이 발견되면 후속하는 명령은 현재 그룹에 속하는 명령과의 동시 실행이 불가능하다.

(1) 임의의 제어 이동 명령 직후
임의의 분기 명령과 그 명령에 후속하는 지연 명령은 명령 그룹에서 서로 분리되어야 한다.

(2) 명령 수순의 연결에서 조건 코드가 설정된 직후
ALU 연산의 연결 수순에서 두번째 명령이 조건 코드를 설정하면 더이상의 명령을 동일 그룹에 추가할 수 없다.

```
add %l5,%l3,%g2
subcc %g2,%g0,%g0          ! 분리
bge .LL1
```

(3) 무효화 분기(annulled branch) 이후 첫번째 명령의 직후
무효화 분기를 위한 지연 그룹은 하나의 명령만 포함할 수 있다. 무효화 분기의 지연 그룹은 분기가 발생하는 경우에만 실행되어야 하며, 이 때 분기 명령 직후의 첫번째 명령만이 실행된다. 분기가 발생하지 않으면 지연 명령은 전혀 실행되지 않는다.

한편 다음의 (1)-(6)에 해당하는 명령이 발견되면 그 명령 직전까지의 명령만이 동시 실행을 위한 그룹에 포함된다.

(1) 정수 레지스터 입력 포트가 부족할 때
SSP에는 레지스터 화일 입력 포트가 4개 있으므로 그보다 많은 포트를 요구하는 명령의 그룹을 생성할 수 없다.

(2) 정수 레지스터 출력 포트가 부족할 때
SSP에는 레지스터 화일 출력 포트가 2개 있으므로 그보다 많은 포트를 요구하는 명령의 그룹을 생성할 수 없다.

```
add %l0,%l1,%l2
sub %o1,%o2,%o3            ! 분리
and %o3,%o4,%o5
```

모든 ALU 연산(sethi와 결과를 %g0)에 기억하는 모든 연산 포함)은 하나의 출력 포트를 사용한다.

(3) 두번째 이동 연산 직전
SSP에 여러개의 ALU가 있기는 하지만 이동 연산은 한 사이클에 하나만 실행할 수 있다.

(4) 두번째 연결 연산 직전

(5) 이동 연산으로의 연결 직전
다음의 예에서와 같이 이동 연산은 연결 연산의 두번째 이후 명령이 될 수 없다.

```
add %l0,%l1,%l2
sll %l2,2,%l2              ! 분리
```

(6) 조건 코드가 설정되는 그룹에서 확장된 산술 명령이 실행되기 직전

```
addcc %g3,%g2,%g3
addx %o0,%g0,%o0           ! 분리
```

2.2 조건부 분기의 변환과 제거

SSP에서 분기는 명령 수순의 실행 시간을 두 사이클 정도 증가시키는 효과가 있다. 그러므로 프로그램에서 분기 명령을 원천적으로 제거하는 것은 대단히 중요하다. 일반적으로 하나의 분기 명령 대신 다른 유형의 명령을 네개까지 사용하여도 더 우수한 성능을 보인다. 또한 연속적인 분기 보다는 산술 연산의 조합이 성능 향상에 훨씬 바람직하다.

많은 프로그램에서 분기는 단순히 제어의 이동을 위한 것이기 보다는 비교 판단의 결과를 다른 산술식이나 배정문에 적용하고자 하는 것이다.

예를 들어 다음과 같은 C 프로그램 문장의 경우

```
if (a > 0) b++;
```

비교의 근본적인 목적이 분기에 있지 않고, 그 결과를 b의 증분치로 활용하는 데 있다. SSP의 경우 이 문장을 gcc를 사용하여 번역하면 아래와 같은 어셈블리 명령 수순을 얻을 수 있다.

```
cmp     %o0,0
bg,a    .LL1            ! 후분리규칙 (1)
mov     1,%o0           ! 후분리규칙 (3)
mov     0,%o0
.LL1:
add     %o1,%o0,%o0
```

이와 같은 경우를 위하여 많은 프로세서에서는 결과와 함께 조건 코드를 설정하는 연산(set condition code, scc)을 포함하고 있다. 상기한 프로그램 문장은 적절한 데이타 흐름 분석을 사용하여 아래와 같은 산술식으로 변환할 수 있으며,

```
b + (a > 0)
```

이는 (a > 0)의 여부를 검사하기 위한 scc 연산과 b의 가산으로 변환할 수 있다.

(a > 0)과 같은 특정 기능을 위한 명령 수순을 검색하는 데는 GSO를 활용할 수 있다. GSO는 SICS의 Granlund와 NewYork 대학의 Kenner에 의하여 개발된 특정 목표 함수를 위한 최적화된 명령 수순을 검색하는 컴파일러 개발 도구이다. GSO는 콜럼비아 대학의 Henry Massalin에 의하여 개발된 superoptimizer[11]를 이식성과 사용의 편이성을 제고하여 재작성한 것이다. GSO를 사용하여 SPARC에서 (a > 0)을 위한 명령 수순을 검색하면 길이 3인 수순이 45개나 있었다. b와의 가산을 고려하면 명령 수순의 길이는 4가 되며, 이는 분기를 사용한 경우에 비해 정적 계수 5을 4로 감소시키는 효과가 있다.

주어진 예에 대하여 GSO를 적용한 결과 특이한 경우를 발견할 수 있었다. (a > 0)을 위한 명령 수순을 검색하는 대신 b + (a > 0)을 위한 명령 수순을 검색한 결과 명령 수순의 길이가 4 대신 3인 수순이 다수 존재하였다. 그 중에서 대표적인 예는 다음과 같다.

```
sra     %i0,31,%i2
subcc   %g0,%i0,%i3     ! 전분리규칙 (6)
addx    %i2,%i1,%i4
```

이러한 변환은 원래의 프로그램에 비해 명령의 정적 계수를 5에서 3으로 감소시켰으며, 조건부 분기 명령을 제거하였을 뿐만 아니라, 명령의 동적 계수까지도 3에서 2로 감소시키는 효과를 얻을 수 있다. 조건부 증감을 위한 프로그램 패턴이 흔한 것을 고려하면 이와 같은 변환은 대단히 유용한 최적화임을 알 수 있다.

예시한 경우는 임의의 비교 연산 @에 대해서 $a^{+}(b@c)$와 같은 형태의 특수한 경우로 볼 수 있으므로 최적화의 잠재성이 대단히 높은 패턴이다.

조건부 분기를 포함하는 프로그램 패턴을 scc 명령을 사용하는 식으로 변환할 수 있는 예는 많이 있다. 예를 들면 다음과 같은 일련의 변환 과정은 쉽게 추론할 수 있다.

```
if (b@c) a = 1 ; else a = 0 ; →
    a = (b@c)? 1:0 →
    a = b@c
```

이 과정에서 밑줄친 부분은 ((b@c)? a:0)과 같은 패턴에서 a가 1인 특수한 경우로 볼 수 있으며, 이 식은 대수적으로 (a&-(b@c))와 동일한 의미이다. -(b@c)와 같은 형태의 문장은 프로그래머가 직접 사용할 가능성은 거의 없으나 조건부 분기를 변환하는 과정에서 흔히 나타나는 특수 패턴이다.

3. 분기 최적화 패턴의 개발

분기 최적화의 대상은 비교 판단의 결과를 다른 산술식이나 배정문에 사용하는 경우이다. 제2장에서 설명한 것과 같이 산술식이나 배정문과 연관된 비교 판단은 대수적 규칙을 사용하여 분기가 제거된 형태로 변환될 수 있다. 변환 결과에는 프로그램에 직접적으로 표현되지 않았던 특수한 패턴이 나타날 수 있다. 그러므로 분기 최적화 패턴의 개발은 이러한 특수 패턴에 대응하는 최적화된 명령 수순을 검색하는 과정으로 볼 수 있다.

3.1 분기 최적화 패턴 개발시의 고려사항

비교 판단의 결과를 필요로 하는 산술식이나 배정문에서 모든 분기를 완전히 제거할 수 있는 것은 아니다. SSP의 경우 하나의 분기 명령을 위한 수행 노력을 정수 연산 4개를 위한 노력과 비슷하게 볼 수 있으므로[9] 산술 연산이나 배정을 위한 추가적인 노력을 감안하더라도 정적 계수가 5 보다 큰 경우는 최적화의 원리에도 위배된다. GSO를 사용한 검색은 프로세서가 제공하는 명령의 종류가 m개이고 명령 수순의 정적 계수가 n일 경우 검색 시간의 복잡도가 $O(mn^{2n})$이나 되므로 긴 명령 수순을 검색하기 위한 효과적인 도구가 될 수 없다. (실제 실험 결과 n이 6인 경우 SSP/SunOS 4.1.3 환경에서 30 시간 이상의 검색 시간이 요구되었다.)

이와 같은 이유로 인하여 본 연구에서는 특정 기능을 위한 명령 수순의 길이가 5 이하인 경우만 고려하였다. 특히 추가적인 연산을 적용하더라도 정적 계수를 그대로 유지하거나 크게 증가시키지 않는 경우를 최적화의 주된 대상으로 하였다.

비교 판단과 관련된 분기의 제거를 위한 일반적 패턴으로 임의의 비교 연산 @에 대하여 (b@c), a^{+}(b@c), -(b@c), a&-(b@c)를 고려하였다. 본 논문에서는 이들 패턴을 각각 @, p@, n@, na@으로, 각 패턴을 위한 명령 수순의 정적 계수 s와 동적 계수 d를 $(s \mid d)$로 표현하였다. 또한 최적화 대상으로 채택된 패턴은 [패턴]으로, 제외된 패턴은 [패턴]으로 구별하여 표시하였다.

3.2 0과의 비교를 위한 패턴

부호가 없는 정수는 항상 0 이상이므로 이와 연관된 분기는 쉽게 제거할 수 있다. 부호를 고려한 경우는 $>$, $\geq$, $<$, $\leq$, $=$, $\neq$ 등과 같이 6가지의 비교 연산이 가능하므로 각각에 대한 최적 수순을 GSO를 사용하여 조사하였다.

(1) eq0와 ne0

임의의 정수가 0과 같은지의 여부를 검사하는 SSP 명령 수순은 (2 | 2)를 만족하는 8가지 경우가 있으며 그 대표적인 예는 다음과 같다.

```
addcc %i0,-1,%i1
subx  %i0,%i1,%i2          ! 전분리규칙 (6)
```

명령 수순의 정적 계수가 2임에도 불구하고 동적 계수 또한 2인 이유는 첫번째 명령이 조건 코드를 설정하고 두번째 명령이 이를 연결 연산의 입력으로 사용하므로 전분리규칙 (6)에 의하여 두 개의 명령 그룹으로 분리되기 때문이다.

peq0를 위한 SSP 명령 수순은 비교 이후에 가산을 수행함에도 불구하고 (2 | 2)를 만족하는 경우가 있으며 그 대표적인 예는 다음과 같다.

```
addcc %i0,-1,%i2
subx  %i1,-1,%i3           ! 전분리규칙 (6)
```

neq0를 위한 SSP 명령 수순도 peq0와 마찬가지로 (2 | 2)를 만족하는 경우가 있으며 그 대표적인 예는 다음과 같다.

```
addcc %i0,-1,%i1
addx  %g0,-1,%i2           ! 전분리규칙 (6)
```

naeq0를 위한 SSP 명령 수순은 논리곱을 위한 and 혹은 andn 연산을 회피할 수 없으며 (3 | 2)를 만족하는 명령 수순만 존재한다. 그러나 neq0와 and/andn 연산의 합성으로도 동일한 의미를 얻을 수 있으므로 최적화의 대상에서 제외한다.

이상을 종합하면 0과 같은지의 여부를 비교한 후 그 결과에 산술문 혹은 배정문에 결합하는 프로그램의 실행에는 (3 | 2)의 노력만 요구되므로 비교와 분기를 혼합한 (5 | 3)의 경우에 비해 정적 계수와 동적 계수를 공히 최적화할 수 있음을 알 수 있다.

임의의 정수가 0과 동일하지 않은지의 여부를 검사하는 ne0의 경우에도 명령 수순의 정적 계수

와 동적 계수가 eq0의 경우와 유사한 최적화 패턴을 찾을 수 있다.

(2) gts0와 les0

어떤 정수가 0보다 큰지의 여부를 검사하기 위한 명령 수순은 (3 | 2) 혹은 (3 | 3)을 만족하는 경우가 있다. (3 | 2)과 (3 | 3)의 예를 각각 보이면 다음과 같다.

(3 | 2)의 예

```
sub     %g0,%i0,%i1
andn    %i1,%i0,%i2     ! 전분리규칙 (4), (5)
srl     %i2,31,%i3
```

(3 | 3)의 예

```
addcc   %i0,%i0,%i1
subx    %i0,%i1,%i2     ! 전분리규칙 (6)
srl     %i2,31,%i3      ! 전분리규칙 (5)
```

예에서 보면 동일한 기능을 구현하는 명령 수순이고 정적 계수가 3으로 동일함에도 불구하고 이들간의 동적 계수가 다른 경우가 있음을 알 수 있다. 이는 SSP의 수퍼스칼라 특징에서 기인하는 현상으로 GSO를 사용하여 검색한 명령 수순에서 SSP를 위한 최적의 경우를 선정하는 단계의 중요성을 알 수 있다.

pgts0를 위한 경우에도 (3 | 2) 혹은 (3 | 3) 특성을 보이는 명령 수순이 있으며 이중에서 (3 | 2)를 만족하는 다음 수순을 최적 수순으로 선택하였다.

```
addcc   %i0,-1,%i2
sra     %i0,31,%i3      ! 전분리규칙 (6)
addx    %i3,%i1,%i4
```

주어진 예에서 addcc와 sra 명령 사이에서 분리가 발생하지 않은 이유는 이들 명령 사이에 데이타 종속성이 전혀 없으므로 SSP의 동적 그룹 판별기능에 의하여 sra와 addcc의 자동적인 순서 조정이 발생하기 때문이다.

ngts0의 경우에도 (3 | 2)를 만족하는 명령 수순이 존재한다. 그러나 nagts0의 경우에는 and 연산의 추가에도 불구하고 동적 계수가 여전히 2로 유지되는 (4 | 2)인 명령 수순이 다음과 같이 존재하였다.

```
addcc   %i0,-1,%i2
or      %i2,%i0,%i3
sra     %i3,31,%i4      ! 전분리규칙 (4), (5)
andn    %i1,%i4,%i5
```

임의의 정수가 0 이하인지의 여부를 검사하는 les0의 경우에도 명령 수순의 정적 계수와 동적 계수가 gts0의 경우와 유사한 최적화 패턴을 찾을 수 있다.

(3) lts0와 ges0

임의의 정수가 0보다 작은가를 검사하기 위한 lts0의 SSP 명령 수순은 "srl %i0,31,%i1"과 같이 (1 | 1)을 만족하는 수순이 존재한다. nlts0의 경우에도 "sra %i0,31,%i1"과 같이 (1 | 1)을 만족하는 수순이 존재하여, 수치적 부호 변환을 위한 추가적인 노력이 전혀 요구되지 않음을 알 수 있다. 한편 plts0와 nalts0의 경우에도 (2 | 1)을 만족하는 최적 수순이 존재한다.

임의의 정수가 0 이상인지의 여부를 검사하는 ges0의 경우에도 명령 수순의 정적 계수와 동적 계수가 lts0의 경우와 유사한 최적화 패턴을 찾을 수 있다.

3.3 부호를 가진 정수의 비교를 위한 패턴

(1) eq와 ne

두 정수 a와 b가 동일한 값인지의 여부를 확인하기 위한 eq의 명령 수순은 (3 | 2)를 만족하는 50가지 경우가 있었다. peq와 neq의 경우 추가적인 연산에도 불구하고 (3 | 2)를 만족하는 수순이 존재하며 그 대표적인 예는 다음과 같다.

peq의 경우

```
sub     %i0,%i1,%i3
addcc   %i3,-1,%i4          ! 전분리규칙 (6)
subx    %i2,-1,%i5
```

neq의 경우

```
sub     %i0,%i1,%i2
addcc   %i2,-1,%i3          ! 전분리규칙 (6)
addx    %g0,-1,%i4
```

naeq의 경우 and 연산의 추가로 인하여 정적 계수는 4로 증가하였으나 동적 계수는 여전히 2로 유지되어 (4 | 2)를 만족하는 수순이 된다.

두 정수가 서로 다른 값인지의 여부를 확인하기 위한 ne의 경우도 eq와 유사한 결과를 얻을 수 있다.

(2) gts와 lts

두 정수 a와 b 사이에 (a > b)인지의 여부를 확인하기 위한 gts의 명령 수순은 (4 | 2)를 만족하는 12가지 경우가 있었다. 그러나 pgts를 위한 (4 | 2)를 만족하는 명령 수순은 존재하지 않으며 최소한 (5 | 3)의 노력을 요구하는 것을 확인하였다.

한편 ngts를 위한 명령 수순 중에는 (4 | 2)를 만족하는 수순이 존재하였다. 이들 중 최적으로 판단되는 수순은 다음과 같다.

```
subcc   %i1,%i0,%i2
sra     %i0,31,%i3
sra     %i1,31,%i4          ! 전분리규칙 (3)
subx    %i4,%i3,%i5
```

(4 | 2)를 만족하는 ngts 패턴의 존재는 추가적인 and 연산을 고려하더라도 (5 | 3)을 만족하는 nagts 구현이 가능함을 의미한다.

두 정수 a와 b 사이에 (a < b)인지의 여부를 확인하기 위한 lts의 경우에도 gts와 유사한 결과를 얻을 수 있다.

(3) ges와 les

두 정수 a와 b 사이에 (a ≥ b)인지의 여부를 확인하기 위한 ges의 명령 수순은 (5 | 3)과 (5 | 4) 수순을 합하여 1000여 경우가 있었다. (5 | 3)을 만족하는 대표적인 수순은 다음과 같다.

```
subcc   %i0,%i1,%i2
srl     %i1,31,%i3      ! 전분리규칙 (2),(3)
srl     %i0,31,%i4
add     %i3,1,%i5       ! 전분리규칙 (2)
subx    %i5,%i4,%i6
```

이 수순은 명령 계수가 (5 | 3)이며, 임시 레지스터를 4개나 사용하는 등 바람직하지 못한 특성을 가진 수순이다. %i2의 값을 사용하는 후속 명령이 없으므로 이를 %g0(항상 0으로 정의되는 전역 레지스터)로 변환하고 가능한 명령 수순 조정을 하더라도 효과는 마찬가지이다. pges나 nges의 경우에는 추가적인 연산에도 불구하고 명령의 계수가 (5 | 3)으로 유지되었으나 nages의 경우에는 (6 | 3)이 되었다. 이 수순은 분기를 사용하는 경우에 비하여 동적 계수를 감소시키지 않을 뿐만 아니라 정적 계수를 오히려 증가시키는 바람직하지 못한 수순이다.

두 정수 a와 b 사이에 (a ≤ b)인지의 여부를 확인하기 위한 les의 경우에도 ges와 유사한 결과를 얻을 수 있다.

3.4 부호가 없는 정수의 비교를 위한 패턴

부호가 없는 정수의 비교는 부호 비교가 필요없는 특징으로 인하여 부호를 가진 정수의 비교보다 수행 노력이 훨씬 적게 요구된다. 그러므로 최적화의 효과를 극대화할 수 있는 대신 최적화 패턴의 활용도는 부호를 가진 정수의 비교보다 적은 편이다.

부호를 가진 정수의 비교에서 gts와 ges의 수

행 노력이 큰 차이를 보인 것에 반해 부호가 없는 정수의 비교에서는 모든 비교의 경우 동일한 수행 노력을 요구하므로 여기에서는 대표적으로 gtu의 경우만 소개하기로 한다.

SSP에서는 부호가 없는 두 정수 a와 b 사이에 (a > b)인지의 여부를 검사하기 위한 (2 | 2)를 만족하는 수순이 있으며, 대표적인 예는 다음과 같다.

```
subcc   %i1,%i0,%i2
addx    %g0,%g0,%i3        ! 전분리규칙 (6)
```

한편 pgtu와 ngtu를 위한 명령 수순의 경우 추가적인 연산에도 불구하고 (2 | 2)를 만족하는 명령 수순이 다음과 같이 존재한다.

pgtu의 경우

```
subcc   %i1,%i0,%i3
addx    %i2,%g0,%i4        ! 전분리규칙 (6)
```

ngtu의 경우

```
subcc   %i1,%i0,%i2
subx    %g0,%g0,%i3        ! 전분리규칙 (6)
```

nagtu를 위한 최적의 명령 수순은 and 연산의 추가로 인하여 정적 계수가 3인 수순만 존재한다. 그러나 새로이 추가된 명령에도 불구하고 동적 계수는 여전히 2를 유지하여 (3 | 2)인 수순이 된다.

```
subcc   %i1,%i0,%i3
subx    %g0,%g0,%i4        ! 전분리규칙 (6)
and     %i4,%i2,%i5
```

부호가 없는 정수의 동등 비교를 위한 기능은 부호를 고려한 정수의 경우와 동일하므로 eq 혹은 ne의 경우에 준하는 패턴을 사용하면 된다.

3.5 특수 패턴

직접적인 분기의 표현은 아니지만 프로그램에서 빈번히 조건부 식으로 표현되는 예로 절대치를 구하기 위한 abs가 있다. abs(x)는 x에 부작용이 존재하지 않는 한 (x > 0? x:-x)와 동일하므로 비교 판단과 분기의 합성으로 볼 수 있다. 이를 직접 번역한 SSP 명령 수순은 다음과 같다.

```
cmp     %o0,0
bl,a    .LL1               ! 후분리규칙 (1)
sub     %g0,%o0,%o0        ! 후분리규칙 (3)
.LL1:
```

여기에서 .LL1 레이블에 후속하는 명령은 별도의 그룹에 포함되게 되므로 동적 계수는 3 이상이 된다.

한편 분기를 사용하지 않는 최적 명령 수순을 검색하면 (3 | 2)를 만족하는 다음과 같은 수순을 구할 수 있다.

```
sra     %i0,31,%i1
add     %i1,%i0,%i2        ! 전분리규칙 (4)
xor     %i2,%i1,%i3
```

이와 같은 수순은 xor 명령과 후속하는 다른 명령을 동일 그룹에 포함시킬 수 있는 가능성이 높으므로 비교와 분기를 사용한 직접적인 번역에 비하여 동적 계수를 감소시키는 효과를 얻을 수 있다.

어떤 정수의 부호를 판정하기 위한 sgn도 최적화가 가능한 기능의 예이다. sgn(x)는 x에 부작용이 존재하지 않는 한 (x > 0? 1:(x < 0? -1: 0))과 동일한 의미이다. 비교와 분기를 사용하여 sgn을 직접적으로 번역하면 (4 | 3)이 최적 수순이지만 분기를 사용하지 않는 최적 수순을 검색하면 (3 | 2)를 만족하는 수순을 얻을 수 있다.

4. 분기 최적화 패턴의 실험

제3장에서 개발한 최적화 패턴을 실험하기 위하여 gcc의 SPARC 후단부를 변형하였다. 현재 gcc는 SPARC V8 호환 코드를 생성하므로 변종 프로세서들을 위한 공통의 후단부로 활용될 수는 있지만 SSP를 위하여 특별히 조정된 기능은 거의 없다.

4.1 GNU C 컴파일러 gcc의 개요

Gcc[10]는 이식성과 하드웨어 적응성이 대단히 우수한 최적화 컴파일러이다. 가장 최근에 발표된 gcc-2.6.3은 현재 사용되고 있는 거의 모든 범용 프로세서를 지원하며, C, C++와 Objective-C를 위한 전단부를 포함하고 있다. Gcc가 생성하는 코우드는 상용 C 컴파일러를 능가할 정도로 대단히 우수하다.

다양한 하드웨어를 위하여 고정된 중간언어를 사용하는 대신 gcc는 목적 하드웨어의 명령을 직접 표현할 수 있는 RTL(register transfer language)이라는 표현을 사용한다. 원시 프로그램은 목적 프로세서의 명령과 무한 갯수의 레지스터를 사용하는 RTL로 일단 번역된다. 컴파일러의 나머지 단계들은 최적화와 레지스터 할당 기능을 수행한다.

Gcc에서 명령(insn)은 여러 RTL 연산자, 이전 및 이후 명령을 가리키는 포인터 및 데이타 흐름 정보로 구성된 식(RTL expression, rtx)의 트리로 표현된다. RTL은 LISP과 유사한 문법으로 출력된다. 덧셈 연산을 위한 명령의 형태는 다음과 같다.

```
(insn 11 10 12 (set (reg/i:SI 0)
                    (plus:SI (reg:SI 22)
                             (const_int 10)))
    152 {addsi3} (insn_list 9 (nil))
    (expr_list:REG_DEAD (reg:SI 22) (nil)))
```

상기한 예는 레지스터 22와 상수 10의 합이 레지스터 0번에 기억됨을 의미한다. 이 명령에 부여된 고유한 번호는 11번이며 이후 명령은 12, 이전 명령은 10번으로 번호가 부여된 명령이다. 이 명령은 프로세서 설명 테이블의 152번 패턴 "addsi3"에 대응하였다. 이 명령에서 사용되는 값(레지스터 22)은 9번 명령에서 설정되었으며, 레지스터 22는 이 명령에서 DEAD 상태가 된다는 의미이다.

프로세서 설명 테이블은 일련의 패턴을 포함하며, 각 패턴은 프로세서가 제공하는 명령과 기본적인 연산을 위한 코우드 생성 방법 등을 설명한다. 명칭을 가진 전형적인 패턴은 특수한 연산에 대하여 어떤 RTL을 생성할 것인지를 표현한다. 예를 들어 addsi3는 4 바이트 정수에 대하여 3개의 피연산자를 사용한 덧셈을 의미한다. 모든 패턴은 어떤 명령이 정당하며 그 명령을 위한 어셈블리어를 어떻게 생성하는지를 명시하는데 사용된다.

생성되는 각 명령은 반드시 어떤 패턴에 대응하여야 한다. 이를 위하여 컴파일러는 최적화 과정의 일부로서 명령 체인에 대한 모든 가능한 변환을 실시한다. SPARC 프로세서를 위한 설명 테이블에서 addsi3를 위한 패턴은 다음과 같다.

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "register_operand" "=r")
        (plus:SI (match_operand:SI 1 "arith_operand" "%r")
                 (match_operand:SI 2 "arith_operand" "rI")))]
  ""
  "add %1,%2,%0")
```

명령들이 rtx의 트리로 표현되므로 한 명령의 결과를 다음 명령에서 사용할 수 있도록 직접 대체할 수도 있다. 이렇게 대체한 명령이 프로세서 설명 테이블의 어느 패턴과 대응하면 번역된 프로그램에서 하나의 명령을 축약할 수 있다. 이를 위하여 gcc에서는 combiner라는 별도의 최적화 단계를 두고 있다.

4.2 분기 최적화 패턴의 반영

제3장에서 개발한 최적화 패턴을 gcc에 반영하는 것은 극히 쉬운 일이다. 먼저 특정 목표함수를 위한 패턴들을 GSO로 검색한 후 SSP의 특징을 고려하여 최적의 명령 수순을 선택한다. 최적 수순의 선택에는 SSP의 여러가지 제한사항과 명령 그룹의 분리 규칙을 적용하여 동적 계수가 제

일 작은 것을 선택한다.

다음 단계는 선택된 코드 수순을 생성할 수 있도록 명령 패턴을 gcc의 프로세서 설명 테이블에 첨가하여야 한다. 예를 들어 -(b > c)에 대응하는 SPARC 명령 수순 중에서 최적으로 판단되어 선택된 것은 다음과 같으며

```
subcc   %i1,%i0,%i2
sra     %i0,31,%i3
sra     %i1,31,%i4          ! 전분리규칙 (3)
subx    %i4,%i3,%i5
```

이를 위한 패턴을 RTL을 사용하여 gcc의 SPARC 프로세서 설명 테이블에 첨가한 예는 다음과 같다. Gcc에서 SPARC을 위한 프로세서 설명 테이블은 "gcc-directory/config/sparc/sparc.md"라는 경로명의 화일에 수록되어 있다.

```
(define_insn ""
  [(set (match_operand:SI 0 "register_operand" "+r")
        (neg:SI (lt:SI (match_operand:SI 1
                         "register_operand" "r")
                       (match_operand:SI 2
                         "arith_operand" "rI"))))
   (clobber (match_scratch:SI 3 "=&r"))
   (clobber (reg:CC 0))]
  "TARGET_SUPERSPARC"
  "![>>> ngts\;cmp %2,%1\;sra %1,31,%3\;!!\;
   sra %2,31,%0\;subx %3,%0,%0\;!<<< ngts]")
```

최적화 패턴에서 "subcc %i1,%i0,%i2"로 표현된 SSP 명령을 "cmp %2,%1"로 변경하였다. 이는 %i2를 후속하는 명령에서 전혀 사용하지 않는 점에 착안하여 %g0를 결과로 하는 cmp 명령으로 대체함으로써 임시 레지스터의 필요성을 감소시킨 것이다. 최적화 패턴에서 "sra %i1,31,%i4; subx %i4,%i3,%i5"를 표현한 부분도 %i4가 연결 연산에서 사용되는 점에 착안하여 %0를 공유하도록 하였다.

프로세서 설명 테이블을 변경하는 작업 이외에도 "if (c) a^{++}"와 같은 문장을 "a+c"와 같은 패턴으로 변환하기 위해서 Granlund/Kenner가 제안한 분기의 변형[4]을 gcc의 "jump.c" 화일에 반영하였다.

4.3 변경된 컴파일러를 사용한 실험

변경된 gcc의 기능을 호출하기 위해서는 컴파일러 사용시 명령어 라인에 "-msupersparc" 옵션을 사용하여 TARGET_SUPERSPARC 플래그를 설정하여야 한다. 다음은 ((b > c)? a:0)을 위한 함수로서 이는 대수적으로 (a&-(b > c))와 동일한 의미의 식임을 앞서 소개한 바 있다.

```
# ifndef INLINE
# define INLINE
# endif
INLINE int nagts(int b, int c, int a)
{return (b>c? a: 0);}
```

변경된 gcc를 사용하여 주어진 프로그램을 번역하는 과정에서 combiner를 거친 후 rtx의 일부분을 보이면 다음과 같다.

```
(insn 36 35 13 (parallel[
      (set (reg:SI 75)
           (neg:SI (gt:SI (reg:SI 24 %i0)
           (reg:SI 25 %i1))))
      (clobber (scratch:SI))
      (clobber (reg:CC 0 %g0))
      ]) 43 {abssi2-17} (nil) (nil))))))

(insn 24 37 25 (set (reg/i:SI 24 %i0)
      (and:SI (reg:SI 26 %i2)
              (reg:SI 75))) 255 {andsi3}
      (insn_list 36 (nil)) (nil))))
```

36번 명령은 -(b > c)를 구하여 의사 레지스터 75번에 할당하는 기능을 위한 rtx를, 24번 명령은 a와 75번 의사 레지스터, 즉 36번 명령에서 계산된 -(b > c)의 논리곱을 위한 rtx를 표현하고 있다. -(b > c)를 위한 패턴을 프로세서 설명 테이블에 첨가한 결과 combiner 단계의 변환을 통하여 상기한 프로그램에서 (b > c? a:0)으로 표현된 식을 (a&-(b > c))로 변환하였음을 알 수 있다. 변환된 결과로 부터 함수 nagts를 위한 목적 코드를 생성한 결과를 보이면 다음과 같다.

```
!>>> ngts
cmp %o1,%o0
srl %o0,31,%g2
!!                  ! 전분리규칙 (3)
```

```
srl %o1,31,%o0   ! %o0를 임시 레지스터로 사용함
subx %o0,%g2,%o0
!<<< ngts
and %o2,%o0,%o0
```

>>>와 <<<로 표시된 것은 그 내부가 ngts 패턴을 위하여 최적화된 부분임을, !!는 명령 그룹의 분리 규칙에 따라 분리가 예상되는 위치를 의미한다. 이와 같이 명령 계수를 위한 주석을 첨가하여 주어진 수순이 (5 | 3)을 만족함을 쉽게 확인하도록 하였다.

반면 이와 같은 변환을 수행하지 않은 원래의 gcc가 생성하는 명령 수순은 다음과 같다.

```
cmp    %o0,%o1
bg,a   .LL4                ! 후분리규칙 (1)
mov    1,%o0               ! 후분리규칙 (3)
mov    0,%o0
.LL4:
sub    %g0,%o0,%o0         ! 전분리규칙 (4)
and    %o2,%o0,%o0
```

변경된 gcc가 생성한 코드가 (5 | 3) 특성을 보이는 반면 원래의 gcc가 생성한 코드는 (6 | 4) 특성을 보임을 알 수 있다. 원래의 gcc가 생성한 명령 수순은 또한 조건부 분기 명령을 포함하고 있어서 명령 수순의 조정을 어렵게 하며, 캐쉬 미스로 인한 추가적인 지연 발생 효과도 있다.

주어진 함수를 호출하기 위한 main 함수를 첨가하여 ms 이상의 수행 시간을 누적한 결과는 〈표 1〉과 같다. 각 실험은 gcc에 -O2 옵션을 사용하여 번역한 결과를 사용하였으며, 각각을 10회씩 수행하였다. 최적화가 코드 inline 확장에 미치는 영향을 확인하기 위하여 -DINLINE =inline 옵션을 사용한 결과는 〈표 2〉와 같다. 표에서 gcc는 원래의 gcc를, gcc'은 본 논문에서 제시한 최적화 패턴을 반영한 gcc를 의미한다.

이상에서 볼 때 inline 확장을 수행하지 않으면 조건의 참, 거짓에 관계 없이 평균적으로 9%의 실행 시간 개선 효과가 있음을 알 수 있다. 이론적으로는 25%의 성능 향상이 있어야 함에도 불구하고 9%로 관측된 이유는 main 함수나 함수의 호출 및 귀환 수순과 같은 최적화 대상 이외의 코드가 포함된 프로그램을 실행하므로 발생한 부담 때문이다.

한편 inline 확장의 경우에도 최적화 패턴으로 인한 효율의 감소는 관측되지 않았으며 이로부터 분기의 제거가 다른 최적화 기능에 악영향을 미치지 않음을 확인할 수 있다. 그러나 abs의 경우 inline 확장을 수행하지 않으면 수행에 필요한 시간이 3.0ms에서 2.7ms로 10%의 미세한 효율 향상이 있으나, inline 확장을 수행한 경우에는 1.65ms에서 0.8ms로 51.5%의 수행 시간이 단축되었다. 이는 주어진 예에서 분기 제거를 위한 최적화 패턴이 함수 호출의 inline 확장과 상승작용을 일으켰음을 의미한다.

조건부 분기를 대체할 수 있는 최적 수순으로 판단된 모든 패턴에 대하여 실험한 결과 상기한 예와 유사한 결과를 얻을 수 있었으며 〈표 3〉은

〈표 1〉 nagts의 수행 시간
〈Table 1〉 Performance results of nagts

조건이 거짓		조건이 참	
gcc	gcc'	gcc	gcc'
3.6	3.3	3.6	3.3
3.6	3.3	3.6	3.3
3.6	3.3	3.6	3.3
3.6	3.4	3.6	3.3
3.6	3.3	3.6	3.3
3.6	3.3	3.6	3.3
3.6	3.3	3.6	3.3
3.6	3.3	3.6	3.3
3.6	3.3	3.6	3.4
3.6	3.3	3.6	3.3

〈표 2〉 inline 확장시 nagts의 수행 시간
〈Table 2〉 Performance results of inlined nagts

조건이 거짓		조건이 참	
gcc	gcc'	gcc	gcc'
1.7	1.7	1.7	1.6
1.6	1.7	1.7	1.7
1.7	1.6	1.7	1.7
1.7	1.6	1.7	1.6
1.7	1.7	1.7	1.7
1.7	1.6	1.7	1.7
1.7	1.6	1.7	1.6
1.6	1.7	1.7	1.7
1.6	1.7	1.6	1.7
1.7	1.6	1.7	1.6

이를 종합적으로 정리한 것이다. 표에서 유사한 패턴은 대표적인 패턴 하나만 제시하였다.

〈표 3〉 모든 최적화 패턴의 정적/동적 계수와 성능 개선
(Table 3) Static/dynamic counts and speedup of all optimization patterns

패 턴	(정적계수|동적계수)		성능 개선 (%)	
	gcc	gcc'	이 론	실 험
eq0	(2|2)	(2|2)	0	0
peq0	(3|2)	(2|2)	0	3
neq0	(3|2)	(2|2)	0	4
naeq0	(4|3)	(3|2)	33	12
gts0	(4|3)	(3|2)	33	11
pgts0	(5|4)	(3|2)	50	19
ngts0	(5|4)	(3|2)	50	19
nagts0	(6|4)	(4|2)	50	21
lts0	(1|1)	(1|1)	0	0
plts0	(2|1)	(2|1)	0	0
nlts0	(2|1)	(2|1)	0	0
nalts0	(3|2)	(2|1)	50	20
eq	(4|3)	(3|2)	33	11
peq	(5|4)	(3|2)	25	8
neq	(5|4)	(3|2)	50	19
naeq	(6|4)	(4|2)	50	19
gts	(4|3)	(4|2)	33	10
pgts	(5|4)	(5|3)	25	10
ngts	(5|4)	(4|2)	50	18
nagts	(6|4)	(5|3)	25	9
gtu	(2|2)	(2|2)	0	0
pgtu	(3|2)	(2|2)	0	3
ngtu	(3|2)	(2|2)	0	3
nagtu	(4|3)	(3|2)	33	9
abs	(3|3)	(3|2)	33	10
sgn	(4|3)	(3|2)	33	11

이상의 실험 결과를 종합해 보면 본 논문에서 제시한 최적화 패턴은 프로그램의 정적 계수를 전혀 증가시키지 않으면서도 동적 계수를 4→3 (혹은 3→2)으로 감소시켰으며, 이로 인하여 25 % 이상의 성능 개선 효과를 얻을 수 있었다. 성능 개선 효과의 실험치를 보면 main 함수의 추가로 인한 부담을 고려하더라도 최적화 패턴이 조건부 분기를 많이 포함한 프로그램의 최적화에 충분히 활용될 수 있음을 알 수 있다.

4.4 분기 최적화의 경험

분기 최적화 패턴을 개발하고 이를 실험한 결과 최적화 컴파일러 작성의 관점에서 SSP의 몇 가지 특이성을 발견할 수 있었다.

먼저 SSP에서는 유사 연산을 구현하기 위한 노력에 차이를 보이는 경우를 다수 발견하였다. 예컨데 부호를 가진 정수의 비교 연산의 경우 lts는 (4 | 2), les는 (5 | 3)을 만족하는 수순이 있었다. 정적 계수 측면에서 보면 4와 5는 그다지 큰 차이가 아니지만 동적 계수 측면에서 2와 3의 차이는 수퍼스칼라 특징을 감안할 때 간과할 수 없는 것이다.

유사 연산을 구현할 수 있는 방법의 수에도 많은 차이를 보이는 경우를 발견하였다. 예컨데 lts의 경우에는 12가지 방법이 있었으나 les의 경우에는 무려 1000여 방법이나 존재하였다. 이같은 다양성은 최적 수순을 선정하는 데 많은 어려움을 부과하므로 바람직하지 않다. 이러한 노력을 자동화하기 위해서는 GSO가 생성한 명령 수순을 입력으로 하여 각각의 경우를 위한 정적, 동적 계수를 자동적으로 검사하고, 전체 수순에서 최적의 수순을 선정하기 위한 프로그램의 개발이 요구된다.

한편 RS/6000[12]과 같은 프로세서의 경우에는 이러한 현상이 거의 발견되지 않았다. 이 프로세서가 제공하는 doz(difference or zero) 명령은 어셈블리어 프로그램에 직접적으로 사용되는 경우가 거의 없지만 최적화 패턴에 활용될 경우 극도로 효율적인 패턴을 생성할 수 있는 잠재성이 있다. 이는 -(b@c)와 같은 형태의 프로그램 문장이 실제로는 거의 사용되지 않지만 어떤 변수의 조건부 설정을 위한 최적화 패턴에는 자주 나타나는 것과 비슷한 이유이다. 그러므로 실제 프로세서를 대상으로 GSO를 적용하는 대신 가상의 프로세서를 위한 명령을 GSO에 첨가한 후 최적 수순을 검사하도록 하면 최적화를 고려한 명령 집합을 설계하는 방법으로 사용될 수도 있을 것이다.

5. 결 론

본 논문에서는 수퍼스칼라 프로세서를 위한 컴파일러에서 조건부 분기 명령을 제거하는 최적화 기법에 대하여 연구하였다. 분기를 제거하는 방법으로 먼저 분기 명령을 대수적으로 변형하고, 변형된 분기에 대응하는 명령 수순을 탐색한 후 목적 프로세서를 위한 최적의 명령 수순을 선택

하였다. 명령 수순의 탐색에는 GSO를 활용하였으며, 분기 제거의 실제적인 효과를 감안하여 정적 계수가 5 이하인 경우만 고려하였다.

실험 결과 본 논문의 최적화 기법은 많은 경우 조건부 분기 명령을 수퍼스칼라 특징을 활용하는 최적의 명령 수순으로 변환함으로써 동적 계수를 감소시키는 효과를 얻을 수 있음을 확인하였다. 입력 프로그램에서 최적화 패턴과 대응하는 조건부 분기의 경우 원래의 컴파일러가 생성하는 최적 코드 수순에 비하여 25% 이상의 추가적인 수행시간 개선 효과를 얻을 수 있었다.

본 연구에서는 목적 프로세서로 수퍼스칼라 명령 축소형 프로세서인 SSP를 사용하였으며, 분기 제거를 위한 패턴을 gcc의 SPARC 후단부에 적용하여 그 효과를 실험하였다. 그러나 이와 같은 기법은 원칙상 임의의 수퍼스칼라 프로세서에도 적용할 수 있다.

명령 수순 검색에서 GSO와 같은 도구의 사용은 필수적이나 검색 기법의 문제점으로 인하여 정적 계수가 6 이상인 경우에는 효과적인 사용이 불가능하였다. 또한 SSP와 같이 최적화에 대하여 특이성을 보이는 프로세서의 경우에는 극도로 많은 명령 수순이 생성되는 경우가 빈번하여 최적 수순의 수동적인 판정에 많은 어려움이 있다. 이러한 문제점을 해결하기 위해서는 GSO의 검색 기법을 개선할 필요가 있으며, GSO의 출력을 자동적으로 분석하는 프로그램도 요구된다.

참 고 문 헌

[1] A. Lane, "Optimizing for Today's CPUs," BYTE Magazine, Feb. 1994.

[2] K. Dowd, High Performance Computing, O' Reilly & Associates, 1993.

[3] H. S. Stone, High-Performance Computer Architecture, Addison-Wesley, 1993.

[4] D. A. Patterson and J. L. Hennessy, Computer Architecture : A Quantitative Approach, Morgan, Kaufmann Publishers Inc., 1990.

[5] K. Y. Wang, "Precise Compile-Time Performance Prediction for Superscalar-Based Computers," Proc. of PLDI '94, pp. 73-84, 1994.

[6] T. Bill and J. R. Larus, "Branch Prediction For Free," Proc. of PLDI '93, pp. 300-313, 1993.

[7] J. A. Fisher and S. M. Freudenberger, "Predicting Conditional Branch Directions From Previous Runs of a Program," ACM SIGPLAN Notices, 27(9), pp. 85-95, 1992.

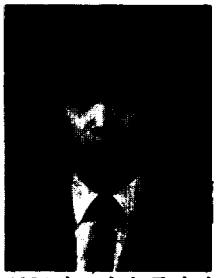
[8] T. Gränlund and R. Kenner, "Eliminating Branches using a Superoptimizer and the GNU C Compiler," Proc. of PLDI '92, pp. 341-352, 1992.

[9] SuperSPARC User's Guide, Texas Instruments, 1993.

[10] R. Stallman, Using and Porting GNU CC, Free Software Foundation, 1992.

[11] H. Massalin, "Superoptimizer : A Look at the Smallest Program," Proc. of ASPLOS II, pp.122-126, 1987.

[12] H. B. Bakoglu, G. F. Grohoski and R. K. Montoye, The IBM RISC system/6000 processor : Hardware overview, IBM Journal of Research and Development, 34(1), pp. 12-22, 1990.



김 명 호

1984년 경북대학교 전자공학과 졸업(학사)
1986년 한국과학기술원 전산학과 졸업(공학석사)
1989년 한국과학기술원 전산학과 졸업(공학박사)
1989년~94년 동아대학교 컴퓨터공학과 조교수
1994년~현재 동아대학교 컴퓨터공학과 부교수
관심분야 : 프로그래밍 언어(특히 추상화, multiparadigm 프로그래밍, 정형적 의미론), 최적화 컴파일러.

최 완

1981년 경북대학교 전자공학과 졸업(학사)
1983년 한국과학기술원 전산학과 졸업(공학석사)
1985년 한국과학기술원 전산학과(조교)
1988년 정보처리기술사 자격 취득
1985년~현재 한국전자통신연구소 책임연구원
관심분야: 소프트웨어공학, 프로그래밍 환경, 컴파일러.