15장 코드 개선

14장에서는 컴파일러 후단의 목표 코드 생성, 어셈블리, 링킹에 대해 알아봤다. 14장에서 살펴본 기술들은 올바르지만 전혀 최적화되지 않은 코드를 생성하는데, 예를 들면 많은 수의 중복 계산, 레지스터, 다중 기능 단위, 현대 마이크로프로세서의 비효율적인 사용이 있다. 15장에서는 코드 개선을 살펴본다. 코드 개선은 좋은 코드를 생성하기 위한 컴파일 단계다. 대부분의 경우 "좋은" 코드는 빠른 코드를 의미한다. 일부 경우에는 메모리 요구 사항을 줄이는 프로그램 변환도 고려한다. 때때로 실제 컴파일러는 전력 소비, 특정 회계 시스템에서의 실행 비용, 기타 다른 자원에 대한 필요 등을 최소화하려고 할 수 있는데, 이러한 쟁점은 여기서 다루지 않는다.

코드 개선에는 다양한 수준의 "적극성"이 존재한다. 매우 간단한 컴파일러나 좀 더 정교한 컴파일러의 "비최적화" 실행에서는 이미 생성된 목표 코드에서 이웃한 명령어들의 명백한 준최적 나열을 정밀 검사하기 위해 **국소적 최적화**(peephole optimizer)를 사용할 수 있다. 약간 더 높은 수준인 상품 수준 컴파일러의 전형적인 기본 수준에서는 기본 블록에 대해 최적에 가까운 코드를 생성할 수 있다. 14장에서 설명했듯이 기본 블록은 (실행된다고 가정했을 때) 항상 전부 실행되는 최대 길이의 명령어 나열이다. 지연 분기가 없을 때 어셈블리어나 기계 코드로 된 기본 블록은 분기의 목표 지점이나 조건 분기 다음의 명령어로 시작해서 분기나 분기 목표 지점 이전의 명령어로 끝난다. 결과적으로 하드웨어적인 처리가 없다면 제어는 항상 기본 블록의 시작부분으로 진입해서 끝부분으로 나온다. 기본 블록 수준의 코드 개선을 **지역 최적화**라고 한다. 지역 최적화는 중복 연산의 제거(예를 들어 불필요한 불러오기나 공통 하위 수식의 계산)와 효과적인 명령어 스케줄링과 레지스터 할당에 초점을 둔다.

좀 더 높은 수준의 적극적 코드 개선에서 상품 수준의 컴파일러는 더 많은 속도 개선을 위해 전체 서브루틴을 분석하는 기술을 사용한다. 이러한 기술을 **전역 최적화**라고

한다.[1] 전역 최적화 기술에는 중복 제거, 명령어 스케줄링, 레지스터 할당의 다중 기본 블록 버전과 루프의 성능 개선을 위해 설계된 코드 수정 등이 있다. 14.1.1절에서 살펴봤듯이 대개 중복 제거와 루프 개선 모두 프로그램의 제어 흐름 그래프 표현을 사용하며, 기본 블록 경계 간의 정보 흐름을 추적하기 위해 자료 흐름 분석 계열의 알고리즘을 채택한다.

다수의 최근 컴파일러는 가장 높은 수준의 적극적 코드 개선에서 다양한 형태의 프로시저 간 코드 개선을 수행한다. 프로시저 간 개선은 두 가지 이유에서 어렵다. 우선 한 프로그램에서 서브루틴은 여러 위치에서 호출될 수 있기 때문에 모든 호출 위치에서 유지가 보장되는 조건(이용 가능한 레지스터, 공통 하위 수식 등)을 찾기란(또는 만들어 내기란) 어렵다. 둘째로 많은 서브루틴이 분리 컴파일되기 때문에 일반적으로 프로시저 간 코드 개선기는 링커 작업의 일부를 포괄해야 한다.

15장에서는 국소, 지역, 전역 코드 개선을 살펴본다. 프로시저 간 개선은 다루지 않는다. 이에 관심이 있는 독자는 다른 책(15장 끝부분의 참고자료를 보자)을 참조하면 된다. 더욱이 여기서 다루는 주제의 경우에도 그 과정을 자세히 설명하기보다는 코드 개선을 "알기 쉽게 설명하는" 데 목적을 두었다. ©(심화학습에 있는) 15.3절에서 시작하는 대부분의 설명은 단일 서브루틴에 대한 코드의 연속적인 정제를 중심으로 전개된다. 이 확장된 예를 통해 일부 코드 개선의 주요 형태들이 어떻게 수행되는지에 대한 상세를 자세히 설명하지 않고도 이들이 내는 효과를 이해할 수 있을 것이다. 코드 개선만을 다루는 책은 계속 출간되고 있으며, 코드 개선은 여전히 매우 활발한 연구 주제다.

대부분의 책에서와 마찬가지로 이 책에서도 코드 개선을 때때로 **최적화**라고 하기도 한다. 그러나 최적화라는 용어는 매우 잘못된 명칭이다. 거의 모든 경우에 코드 개선 기술이 최적 코드를 이끌어 낸다고 보장할 수 없다. 이미 밝혀진 바와 같이 코드 개선의 비교적 간단한 측면(예를 들어 기본 블록 내에서의 레지스터 사용의 최소화) 조차 NP−난해하게 보일 수 있다. 진정한 최적화는 소규모의 특수 목적 프로그램 조각에 대해서만 현실적인 선택 사항이다[Mas87]. 책의 설명은 명령형 프로그램의 코드 개선에 초점을 둔다. 함수형 언어나 논리형 언어에 특정적인 최적화는 이 책의 범위 밖이다.

©(심화학습에 있는) 15.1절에서는 코드 개선의 여러 단계를 좀 더 자세히 살펴본다. 그 다음 ©(심화학습에 있는) 15.2절에서는 국소적 최적화를 다룬다. 원하는 경우 국소적 최적화는 다른 최적화가 없는 경우에 실행할 수도 있다. ©(심화학습에 있는) 15.2절에서는 몇 가지 유용한 용어도 소개한다. ©(심화학습에 있는) 15.3절과 ©(심화학습에 있는) 15.4절에서는 지역 중복 제거와 전역 중복 제거를 살펴본다. ©(심화학습에 있는) 15.5절과 ©(심화학습에 있는) 15.7절은 루프에 대한 코드 개선을 다룬다. ©(심화학습에 있는) 15.6절에서는 명령어 스케줄링을 다루며, ©(심화학습에 있는) 15.8절에서는 레지스터 할당을 살펴본다.

주1. 형용사 전역이 표준으로 사용되긴 하지만 전역 개선은 프로그램을 총체적으로 고려하지 않기 때문에 이 문맥에서는 다소 오해의 소지가 있다. 프로시저 간 개선이 좀 더 정확한 전역 개선일 수 있다.

15.1 코드 개선의 여러 단계

예 15.1

코드 개선의 여러 단계

14장에서 살펴봤듯이 후단의 구조는 컴파일러마다 상당히 다르다. 여기서는 일관성을 유지하기 위해 14.1절에서 소개한 구조에 초점을 둔다. (1.6절에서와 같이) 14.1절에서는 기계 독립적 코드 개선과 기계 특화 코드 개선을 목표 코드 생성에 의해 분리되는 개별적인 컴파일 단계로 나타냈다. 이는 사실 지나친 단순화였다. 사실 코드 개선은 훨씬 더 복잡한 과정으로 가끔 매우 많은 수의 단계로 구성된다.

어떤 경우 최적화는 서로 의존적이며 특정 순서로 실행되어야 한다. 다른 경우에는 독립적이며 아무 순서로나 실행될 수 있다. 다른 어떤 경우에는 다른 최적화가 적용될 때까지는 알 수 없었던 새로운 개선 가능성을 인식하기 위해 최적화를 반복하는 것이 중요할 수 있다.

15장에서는 실행 속도를 가장 많이 증가시키는 경향이 있으며, 가장 널리 쓰이는 코드 개선의 형태를 집중적으로 살펴본다. 이러한 개선을 구현하는 컴파일러 단계는 ©(심화학습에 있는) 그림 15.1과 같다. 이 구조에서 후단의 기계 독립적 부분은 중간 코드 생성으로 시작한다. 이 단계는 기본 블록에 대응되는 구문 트리 조각을 식별한다. 그 다음에는 각 노드가 대개 무한한 가상 레지스터를 가지는 이상적 기계를 위한 3-주소 명령어의 선형 나열을 포함하는 제어 흐름 그래프를 생성한다. 후단의 기계 특화 부분은 목표 코드 생성으로 시작한다. 이 단계는 각 블록을 타겟 머신의 명령어 집합으로 변환하고 제어 흐름 그래프의 호에 대응되는 분기 명령어를 생성하면서 기본 블록들을 하나의 선형 프로그램으로 결합한다.

©(심화학습에 있는) 그림 15.1에서 기계 독립적 코드 개선은 세 개의 단계로 나타냈다. 첫 번째 단계는 각 기본 블록 내의 중복 불러오기, 중복 저장, 중복 계산을 식별하고 제거한다. 두 번째 단계는 기본 블록 경계에 걸쳐(그러나 단일 서브루틴의 경계 안에서) 유사한 중복을 처리한다. 세 번째 단계는 루프에 특화된 여러 개선을 수행한다. ©(심화학습에 있는) 15.4, 15.5, 15.7절에서는 전역 중복 제거와 루프 개선을 실제로 별도의 여러 단계로 나눌 수 있다는 것을 알아본다.

기계 특화 코드 개선은 4개의 단계로 나타냈다. 첫 번째와 세 번째 단계는 본질적으로 동일하다. 5.5.2절에서 알아봤듯이 레지스터 할당과 명령어 스케줄링은 상충된다. 파이프라인 일시 중지(stall)를 가장 잘 최소화하는 명령어 스케줄은 구조 레지스터의 필요량(이를 보통 레지스터 압력이라고 한다)을 증가시키는 경향이 있다. 15장의 설명에서 가정하는 일반적인 방법은 우선 명령어를 스케줄링하고 구조 레지스터를 할당한 다음에 다시 명령어를 스케줄링하는 것이다. 구조 레지스터가 충분하지 않은 것으로 판명되면 레지스터 할당기는 레지스터를 일시적으로 메모리에 내려 보내기 위해 추가적인 불러오기와 저장하기 명령어를 생성한다. 두 번째 명령어 스케줄링에서는 추가적인

불러오기로 인한 지연을 채우게 시도한다.

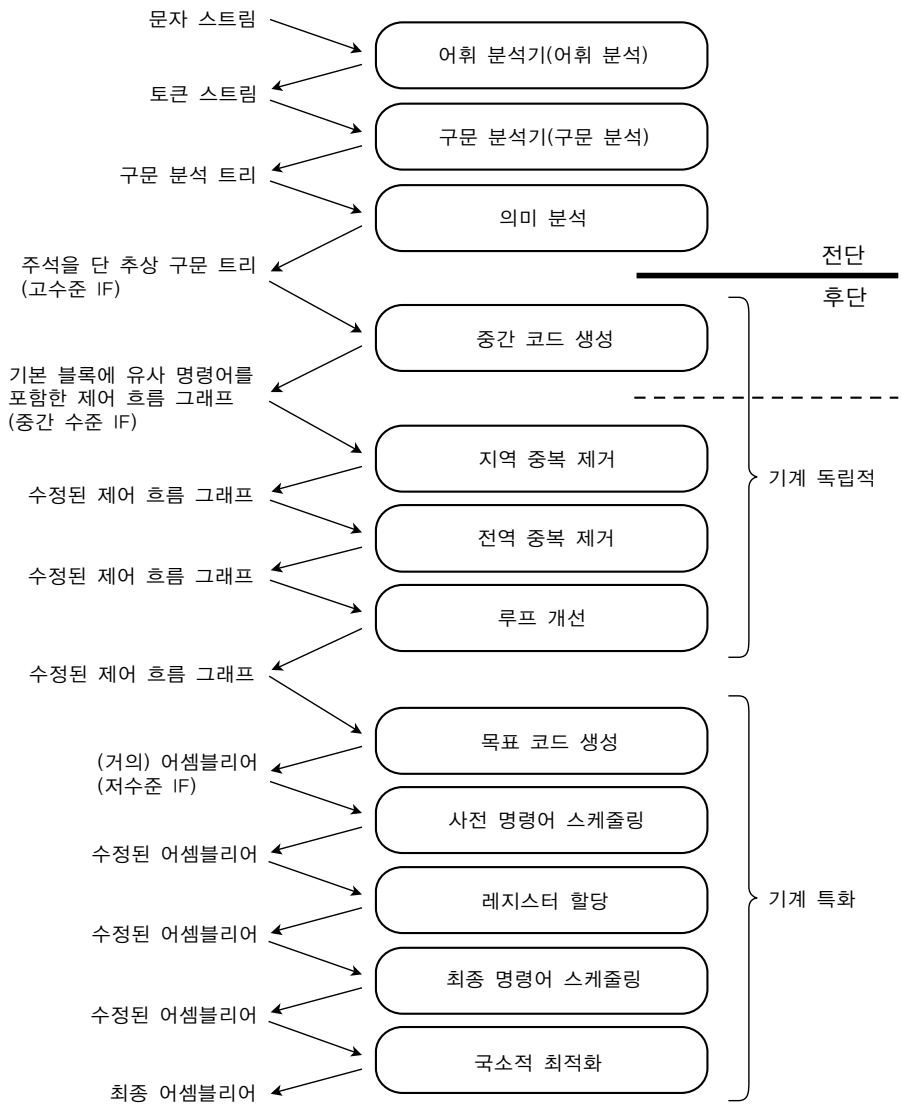


그림 15.1 | 그림 14.1의 컴파일러 구조를 좀 더 자세히 나타낸 그림. 기계 독립적 코드 개선과 기계 특화 코드 개선 모두 여러 단계로 나누었다. 그림 14.1에서와 마찬가지로 점선은 2-패스 컴파일러의 일반적인 "분리점"을 나타낸다.

15.2 국소적 최적화

기계 독립적 코드 개선을 수행하지 않는 간단한 컴파일러의 코드 생성기는 단순히 추상 구문 트리를 탐색하며 파일이나 전역 리스트로의 출력 또는 트리의 주석으로서 나이브한 코드를 생성한다. 그러나 1장과 14장에서 봤듯이 이렇게 생성된 코드의 질은 매우 떨어진다(예 1.2의 코드와 그림 1.5의 코드를 비교해보자). 특히 변수를 r-value로 사용하는 모든 경우가 불러오기를 유발하며 모든 대입문은 저장하기를 초래한다.

나이브한 코드의 질을 상당히 개선하는 비교적 간단한 방법은 목표 코드에 국소적 최적화기를 실행하는 것이다. 국소적 최적화기는 명령어 몇 개로 이뤄진 창(들여다보는 구멍, peephole)을 이동시키면서 명령어의 준최적 패턴을 찾는다. 찾고자 하는 패턴 집합은 휴리스틱이다. 일반적으로 특정 코드 생성기가 생성한 공통적인 준최적 프로그래밍 어구와 일치되거나 주어진 기계상에서 이용할 수 있는 특수 명령어를 활용하는 패턴을 생성한다. 몇 가지 예는 다음과 같다.

예 15.2
중복된 불러오기와 저장하기의 제거

■ **중복된 불러오기와 저장하기의 제거:** 국소적 최적화기는 가끔 불러오기 명령어가 생성한 값이 이미 레지스터에 있음을 인식할 수 있다. 예를 들어 다음의 왼쪽은 오른쪽과 같이 최적화된다.

```
r2 := r1+5                    r2 := r1+5
i := r2          ⟶           i := r2
r3 := i                       r3 := r2×3
r3 := r3×3
```

이와 유사하지만 약간 덜 일반적인 맥락에서 최적화기의 구멍 내에 동일한 위치에 대한 두 개의 저장하기가 있는 경우(그리고 이 위치로부터 사이에 일어나는 불러오기가 절대 없을 때)에는 일반적으로 첫 번째 저장하기를 삭제할 수 있다.

예 15.3
상수 선계산

■ **상수 선계산:** 나이브한 코드 생성기는 사실 컴파일 시점에 수행할 수 있는 계산을 실행 시간에 수행하는 코드를 생성할 수 있다. 국소적 최적화기는 가끔 이러한 코드를 인식할 수 있다. 예를 들어 왼쪽은 오른쪽과 같이 최적화된다.

```
r2 := 3×2          ⟶          r2 := 6
```

예 15.4
상수 전파

■ **상수 전파:** 때때로 변수는 프로그램의 특정 지점에서 상수 값을 가질 수 있다. 이런 경우 변수를 상수로 대체할 수 있다. 예를 들어 왼쪽은 중간과 같이 최종화되며 다시 오른쪽과 같이 최적화 된다.

```
r2 := 4                  r2 := 4                  r3 := r1+4
r3 := r1+r2      ⟶      r3 := r1+4       ⟶      r2 := ...
r2 := ...                r2 := ...
```

r2에 대한 마지막 대입을 통해 r2의 이전 값(4)이 유효하지 않음(dead), 즉 절대 필요치 않음을 알 수 있다(예상할 수 있듯이 앞으로의 계산에서 필요할 수 있는 값은 유효하다(live)고 한다). 유효하지 않은 값의 불러오기는 제거할 수 있다. 이와 유사하게 다음과 같은 최적화도 가능하다. 왼쪽이 중간과 같이 최적화되며 다시 오른쪽과 같이 최적화된다.

```
r2 := 4                 r3 := r1+4              r3 := *(r1+4)
r3 := r1+r2      ⟶      r3 := *r3        ⟶
r3 := *r3
```

(역시 r2는 유효하지 않다고 가정했다)

가끔 상수 선계산 후에 추가적인 상수 전파가 가능한 경우도 있다. 때로는 다음과 같이 그 반대도 일어난다. 왼쪽이 중간과 같이 최적화되며 다시 오른쪽과 같이 최적화된다.

```
r1 := 3                 r1 := 3                 r1 := 3
r2 := r1×2       ⟶      r2 := 3×2       ⟶      r2 := 6
```

r1의 3이 유효하지 않다면 초기 불러오기도 제거할 수 있다.

예 15.5
공통 하위 수식의 제거

■ **공통 하위 수식의 제거:** 최적화기의 구멍 내에서 동일한 계산이 두 번 일어나면 가끔 두 번째 계산을 제거할 수 있다. 왼쪽은 오른쪽으로 최적화된다.

```
r2 := r1×5              r4 := r1×5
r2 := r2+r3      ⟶      r2 := r4
r3 := r1×5              r3 := r4
```

여기서 보인 대로 공통 값을 저장하기 위해 가끔 추가적인 레지스터가 필요할 수 있다.

예 15.6
복사 전파

■ **복사 전파:** 때때로 레지스터 b의 내용이 상수일지는 알 수 없더라도 레지스터 b가 레지스터 a와 동일한 값을 포함한다는 것은 알 수 있는 경우가 있다. 이 경우 a와 b가 수정되지 않는 한 b를 a로 대체할 수 있다. 왼쪽이 중간과 같이 최적화되며 다시 오른쪽과 같이 최적화된다.

```
r2 := r1                r2 := r1                r3 := r1+r1
r3 := r1+r2      ⟶      r3 := r1+r1      ⟶      r2 := 5
r2 := 5                 r2 := 5
```

복사 전파를 코드 개선의 초기에 수행함으로써 레지스터 압력을 줄일 수 있다. 국소적 최적화기에서는 복사 전파를 통해(r2에 있는 r1의 복사본이 유효하지 않은 이 경우에서와 마찬가지로) 하나 이상의 명령어를 제거할 수 있을 수 있다.

예 15.7
강도 축소

■ **강도 축소:** 때때로 비교적 비용이 높은 명령어를 비용이 낮은 명령어로 대체하는 데 숫자 항등식을 사용할 수 있다. 특히 2의 제곱수에 의한 곱셈이나 나눗셈은 덧셈이나 시프트로 대체할 수 있다. 다음과 같이 최적화된다.

```
r1 := r2×2    ⟶  r1 := r2+r2    ⟶  r1 := r2 << 1
r1 := r2/2    ⟶  r1 := r2 >> 1
```

(이 마지막 대체는 `r2`가 음수일 경우 올바르지 않을 수 있다. ©심화학습에 있는 연습문제 15.1을 보자). 이와 유사한 맥락에서 산술 항등식을 통해 다음과 같은 단순화를 수행할 수 있다. 다음과 같이 최적화된다.

```
r1 := r2×0    ⟶  r1 := 0
```

예 15.8
불필요한 명령어의 제거

■ **불필요한 명령어의 제거:** 다음과 같은 명령어는 완전히 삭제할 수 있다.

```
r1 := r1 + 0
r1 := r1   1
```

■ **불러오기와 분기 지연 채우기:** 5.5.1절에서 지연 채우기 변환의 예를 몇 가지 살펴봤다.

예 15.9
명령어 집합 활용

■ **명령어 집합 활용:** 특히 CISC 기계상에서 간단한 명령어 나열은 가끔 좀 더 복잡한 명령어 몇 개로 대체할 수 있다. 예를 들어

```
r1 := r1 & 0x0000FF00
r1 := r1 >> 8
```

위 코드는 "바이트 추출" 명령어로 대체할 수 있다.

```
r1 := r2+8
r3 := *r1
```

끝의 `r1`이 무효한 위의 코드 나열은 기준 더하기 변위 주소 지정 방식을 사용해서 하나의 `r3` 불러오기로 대체할 수 있다.

마찬가지로 `*r2`가 4바이트 값인 위 수식도 자동 증가 주소 지정 방식을 사용해서 단일 불러오기로 대체할 수 있다. 많은 기계에서 연속적인 위치로부터의 불러오기들은 하나의 다중 레지스터 불러오기로 대체할 수 있다.

국소적 최적화기는 작은 고정 크기의 창을 이용하기 때문에 매우 빠르며 명령어당 적은 양의 오버헤드만 더해진다. 또 국소적 최적화기는 비교적 작성하기 쉬우며 나이브한 코드에 적용하면 큰 성능 개선을 얻을 수 있다.

그러나 앞서 살펴본 코드 개선 형태의 대부분이 국소적 최적화에만 국한되는 것은 아니라는 점을 알아야 한다. 사실 마지막(명령어 집합의 활용)을 제외한 모든 경우는 뒤에서 좀 더 일반적인 형태의 코드 개선을 다룰 때도 나온다. 좀 더 일반적인 형태의 코드

개선은 좁은 명령어 창만 이용하는 것이 아니기 때문에 더 나은 최적화를 수행한다. 좋은 기계 특화 코드 개선기와 기계 독립적 코드 개선기를 포함한 컴파일러에는 중복 명령어와 불필요한 명령어를 제거하거나, 상수를 미리 계산하거나, 강도 축소를 수행하거나, 불러오기와 분기 지연을 채우는 국소적 최적화기가 필요하지 않을 수 있다. 이러한 컴파일러에서 국소적 최적화기는 주로 타겟 머신만의 특징을 활용해서 후단의 나머지 부분에서 빠뜨릴 수 있는 특정 준최적 코드를 생성하는 역할을 한다.

설계와 구현

국소적 최적화

많은 경우 처음부터 더 나은 코드를 생성하는 것보다는 코드 개선기가 준최적 어구를 찾아서 수정하게 하는 것이 더 쉽다. 국소적 최적화기조차 1에 의한 곱셈이나 0에 의한 덧셈과 같은 일반적인 예는 찾아낸다. 코드 생성기가 이러한 경우를 특별하게 처리하게 하는 복잡도를 더할 이유는 전혀 없다.

15.3 기본 블록에서의 중복 제거

컴파일러는 지역 최적화를 구현하기 위해 우선 14.1.1절에서 설명한 대로 기본 블록에 대응되는 구문 트리 조각을 식별해야 한다. 대략 이러한 조각은 순차적(in-order) 탐색에 따라 이웃이며 선택이나 반복 구성소를 포함하지 않는 트리 노드로 구성된다. 그림 14.5에서는 단순한 구문 트리에 대한 선형(goto를 포함하는) 코드를 생성하기 위한 속성 문법을 보였다. 제어 흐름 그래프(연습문제 14.6)를 생성하는 데에도 유사한 문법을 사용할 수 있다.

제어 흐름 그래프 내에서 사용자 서브루틴에 대한 호출은 기본 블록 사이의 경계를 정의하는 한 쌍의 분기로 처리할 수도 있으나 호출이 복귀한다는 것을 아는 한 이 호출은 단순히 잠재적으로 범위가 넓은 부가 작용을 가진 명령어(즉, 많은 수의 레지스터와 메모리 위치를 덮어쓸 수 있는 명령어)로 처리할 수 있다. 8.2.5절에서 살펴봤듯이 컴파일러는 작은 서브루틴들을 인라인으로 확장할 수도 있다. 이 경우 "호출"의 동작을 완전히 볼 수 있다. 호출된 루틴이 단일 기본 블록으로 구성된다면 이 루틴은 호출 블록의 일부가 된다. 호출된 루틴이 여러 블록으로 구성된다면 이 루틴의 프롤로그와 에필로그는 호출 전 후에 있는 블록의 일부가 된다.

설계와 구현

기본 블록

프로그램 기본 블록의 대부분은 소스 코드에서 명확히 알 수 있다. 그러나 일부 기본 블록은 변환 과정 중에 컴파일러가 생성한다. 예를 들어 대규모의 레코드나 서브루틴 매개변수를 복사하거나 초기화하는 루프가 생성될 수 있다. 마찬가지로 실행시간 의미 검사가 많은 수의 묵시적 선택문을 유발할 수도 있다. 더욱이 ©(심화학습에 있는) 15.4.2, 15.5, 15.7절에서 보겠지만 많은 최적화가 코드를 한 기본 블록에서 다른 기본 블록으로 옮길 수 있으며 기본 블록을 생성하거나 파괴하거나 루프 중첩을 완전히 재구조화할 수 있다. 이러한 최적화의 결과로 최종적인 제어 흐름 그래프는 프로그래머가 나이브하게 예상했던 것과는 크게 다를 수 있다.

【15.3.1】 실제 예

예 15.10

combinations 서브루틴

15장의 나머지 부분에서는 대개 하나의 서브루틴, 명확히 말해서 모든 $0 \le m \le n$에 대한 2항 계수 $\binom{n}{m}$을 배열에 계산하는 서브루틴에 대한 코드 개선을 다룬다. 이 2항 계수들은 파스칼의 삼각형에서 n번째 행의 원소다. 각 행의 m번째 원소는 n개의 항목 중에서 선택될 수 있는 m개 아이템의 서로 다른 조합의 수를 나타낸다. 이를 C로 작성하면 다음과 같다.

```
combinations(int n, int *A) {
    int i, t;
    A[0] = 1;
    A[n] = 1;
    t = 1;
    for (i = 1; i <= n/2; i++) {
       t = (t * (n+1-i)) / i;
       A[i] = t;
       A[n-i] = t;
    }
}
```

이 코드는 모든 $0 \le m \le n$에 대해 $\binom{n}{m} = \binom{n}{n-m}$라는 사실을 이용한다. 정수 산술을 사용해도 반올림 오류가 발생하지 않는다는 것을 증명할 수 있다(©심화학습에 있는 연습문제 15.2).

예 15.11

구문 트리와 나이브한 제어 흐름 그래프

combinations 서브루틴에 대한 구문 트리는 ©(심화학습에 있는) 그림 15.2와 같다. 이 그림에는 기본 블록도 표시했다. 이에 대응되는 제어 흐름 그래프는 ©(심화학습에 있는) 그림 15.3에 담았다. 코드 개선 초기 단계에서 명령어 사이의 인위적인 충돌을

피하기 위해 계산된 모든 값이 별도의 레지스터에 저장되는 중간 수준 중간 형태 (IF)를 사용했다.

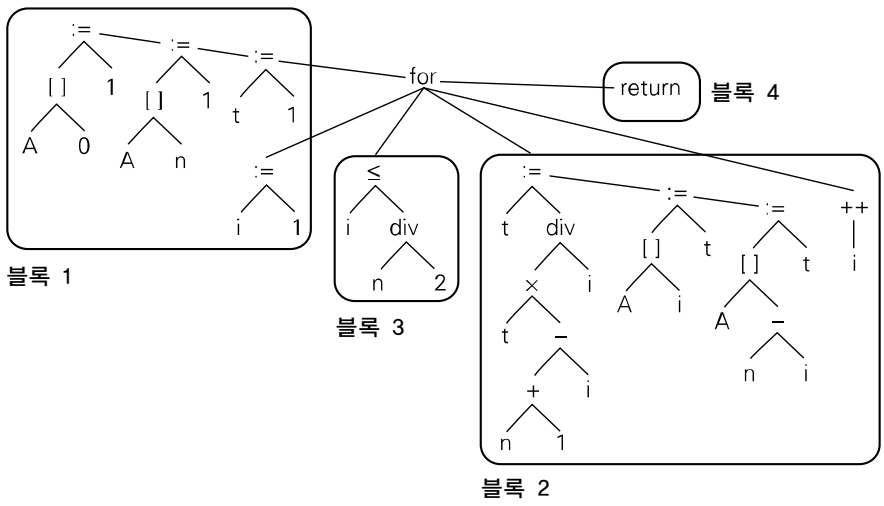


그림 15.2 | combinations 서브루틴에 대한 구문 트리. 기본 블록에 해당하는 부분을 상자로 묶었다.

이러한 레지스터는 가상 레지스터(무한한 가상 레지스터가 있다)임을 강조하기 위해 `v1`, `v2`, ...로 명명했다. ©(심화학습에 있는) 15.8절에서 구조 레지스터를 나타낼 때는 `r1`, `r2`, ...을 사용한다.

초기 제어 흐름 그래프에서 어떤 가상 레지스터도 둘 이상의 명령어로부터 값을 대입받지 않는다는 사실은 코드 개선 기술의 성공 여부에 아주 중요한 요소다. 대략적으로 말해서 이 사실은 결국 별도의 구조 레지스터에 저장되는 모든 값은 적어도 처음에는 별도의 가상 레지스터에 저장된다는 것을 의미한다. 물론 가상 레지스터에 대한 대입이 루프에 나온다면 해당 레지스터는 매 반복마다 다른 값을 취할 수 있다. 앞으로 다양한 코드 개선 단계를 거치면서 둘 이상의 위치에서 하나의 가상 레지스터에 값을 대입할 수 있게 규칙도 완화할 것이다. 중요한 것은 처음부터 가능할 때마다 새로운 가상 레지스터를 채택함으로써 코드 개선의 이후 단계들이 이용할 수 있는 자유도를 최대화했다는 점이다.

초기(진입) 블록과 최종(출구) 블록에는 서브루틴 프롤로그와 에필로그를 위한 코드를 포함시켰으며 ©(심화학습에 있는) 8.2.2절에서 설명한 MIPS 호출 규약을 가정한다. 또 컴파일러는 이 서브루틴이 리프며, 그러므로 복귀 주소(`ra`)나 프레임 포인터(`fp`) 레지스터를 저장할 필요가 없음을 인식했다고 가정한다. 모든 경우에 메모리의 `n`, `A`, `i`, `t`에 대한 참조는 스택 포인터(`sp`) 레지스터에 대한 적절한 변위 주소 지정을 수행하는 것으로 해석해야 한다.

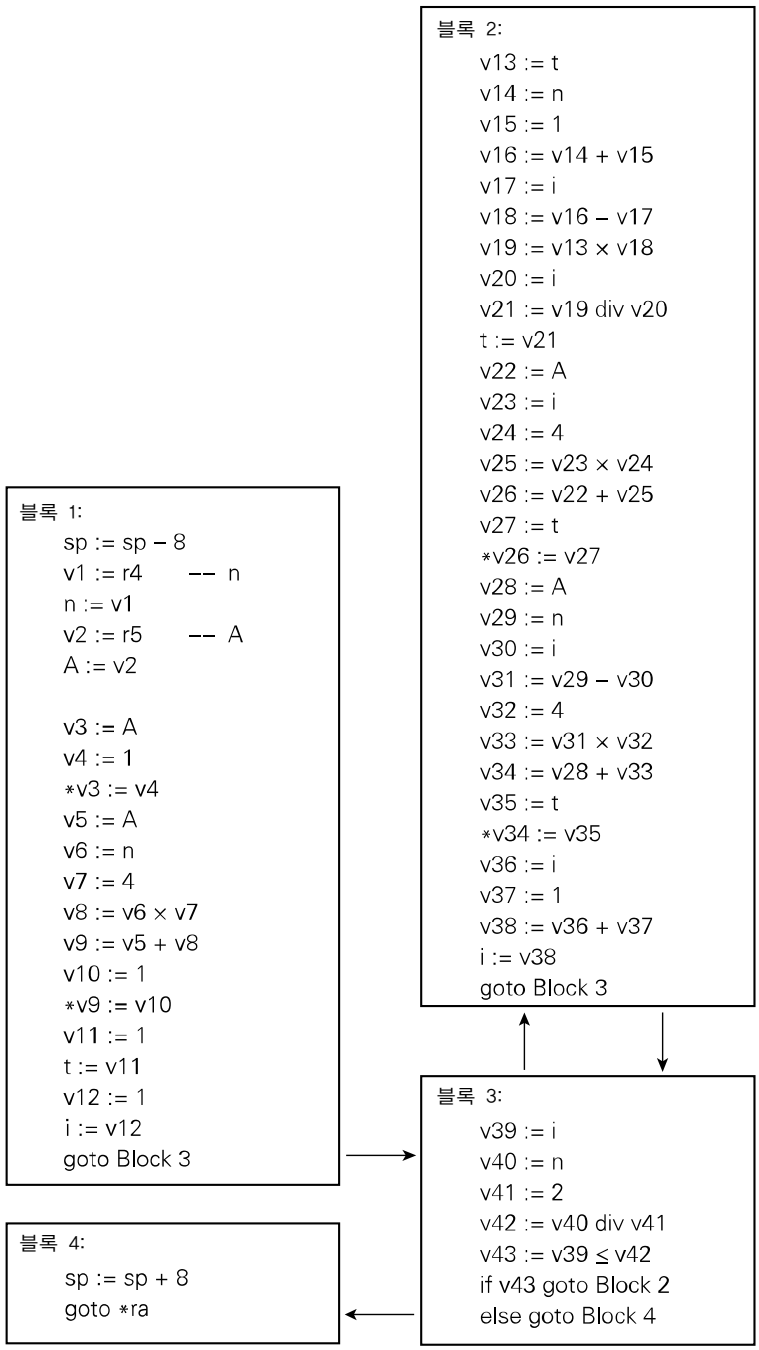


그림 15.3 | combinations 서브루틴에 대한 나이브한 제어 흐름 그래프. 참조 매개변수 A는 결과를 쓰는 배열의 주소를 포함한다는 것에 주의하자. 그러므로 `v3 := &A`가 아니라 `v3 := A`라고 쓴다.

매개변수 값이 레지스터(MIPS에서 구조 레지스터 `r4`와 `r5`)를 통해 전달된다고 가정하지만

(나이브한) 초기 코드는 매개변수 값을 메모리에 바로 저장하며, 그러므로 이에 대한 참조는 지역 변수에 대한 참조와 동일한 방식으로 처리할 수 있다. ©(심화학습에 있는) 15.4.1절에서 설명한 전역 값 번호 지정 알고리즘이 저장된 모든 값을 볼 수 있게 저장은 가상 레지스터를 통해 수행한다. 결국 몇 단계의 개선 후에는 매개변수와 지역 변수 모두 레지스터에 영구적으로 유지될 수 있어 다양한 불러오기, 저장, 복사 연산이 필요 없게 됨을 알게 될 것이다.

【15.3.2】 값 번호 지정

기본 블록 내의 코드를 개선하기 위해 불러오기와 저장하기를 최소화하고 중복 계산을 식별해야 한다. 이러한 작업의 일반적인 한 방법은 기본 블록에 대한 구문 트리를 수식 DAG(방향성 비순환 그래프, directed acyclic graph)로 변환하는 것이다. 수식 DAG에서 중복 불러오기와 중복 계산은 여러 부모를 가지는 개별 노드들로 병합된다[ASU86, 9.8절; FL88, 15.7절]. 지역 값 번호 지정이라는 기술을 통해 명시적으로 그림을 이용한 프로그램 표현 없이도 이와 유사한 기능성을 얻을 수 있다[Muc97, 12.4절].

값 번호 지정은 기호적으로 등가인 둘 이상의 계산("값")에 동일한 이름("번호")을 할당하며, 그러므로 중복된 인스턴스들은 공통 이름으로 식별할 수 있게 된다. 여기서 이름은 가상 레지스터로 가상 레지스터는 공통 값을 저장하고 있다는 것이 보장될 때마다 병합한다. 지역 값 번호 지정을 수행하는 동안 지역 상수 선계산, 상수 전파, 복사 전파, 공통 하위 수식 제거, 강도 축소, 불필요한 명령어 제거도 구현한다(이러한 최적화 간의 구별은 전역의 경우에 좀 더 명확해진다).

이미 불러오거나 계산된 값을 추적하는 사전을 유지하면서 기본 블록의 명령어를 순서대로 살펴본다. 불러오기 명령어 `vi := x`에 대해 `x`가 이미 어떤 레지스터 `vj`에 있는지 알아보기 위해 사전을 검색한다. 이미 `vj`에 있다면 단순히 `vi`의 사용을 `vj`로 대체해야 함을 나타내는 항목을 사전에 추가한다. `x`가 사전에 없다면 새 버전의 기본 블록에 불러오기를 생성하고 `x`는 `vi`에서 이용할 수 있음을 나타내는 항목을 사전에 추가한다. 상수 불러오기 `vi =: c`에 대해서는 `c`가 계산 명령어의 즉치 피연산자로 적합할 만큼 충분히 작은지 확인한다. 충분히 작다면 `vi`의 사용을 해당 상수로 대체해야 함을 나타내는 항목을 사전에 추가하지만 코드는 생성하지 않는다. 대신 `vi`가 나오면 적절한 명령어에 상수를 직접 포함시킨다. 상수가 크다면 `vi`를 이미 다른 레지스터 `vj`로 불러왔는지(또는 계산했는지) 알아보기 위해 사전을 검색한다. 이미 다른 레지스터로 불러왔다면 `vi`의 사용을 `vj`로 대체하게 명시한다. 상수가 크며 이미 이용할 수도 없다면 상수를 `vi`로 불러오는 명령어를 생성하고 적절한 사전 항목에 이용성을 기록한다. 모든 경우에 불러오기의 목표 레지스터에 대한 사전 항목을 생성하게 되며 이때 레지스터가 (1) 추후의 명령어에서 자기 자신의 이름으로 사용되어야 하는지 (2) 다른 레지스터로 대체되어야 하는지 또는 (3) 작은 즉치 상수로 대체되어야 하는지를 나타낸다.

계산 명령어 vi := vj op vk의 경우 우선 vj나 vk의 사용이 다른 레지스터나 작은 상수 vl과 vm으로 대체되어야 하는지 알아보기 위해 사전을 검색한다. 두 피연산자가 모두 상수라면 컴파일 시점에 연산을 수행해서 상수 선계산을 수행할 수 있다. 그 다음 위에서 불러오기에 대해 했던 것과 마찬가지로 상수를 처리한다. 즉, 상수가 작으면 그 값의 기록을 유지하고 값이 크면 상수가 상주하는 레지스터의 기록을 유지한다. 강도 축소나 불필요한 명령어 제거에 대한 가능성도 기록한다. 피연산자 중 적어도 하나가 상수가 아니라면(그리고 명령어가 불필요하지 않다면) (잠재적으로 수정된) 계산 결과를 이미 다른 레지스터 vn에서 이용할 수 있는지를 알아보기 위해 사전을 다시 검색한다. 이 최종적인 검색 연산의 키는 연산자 op와 피연산자 레지스터나 상수 vj(또는 vl)와 vk(또는 vm)의 조합으로 주어진다. 검색이 성공하면 vi의 사용을 vn으로 대체해야 함을 나타내는 항목을 사전에 추가한다. 검색이 실패하면 새로운 버전의 기본 블록에 적절한 명령어(예를 들어 vi : = vj op vk 또는 vi := vl op vm)를 생성하고 이에 대응되는 항목을 사전에 추가한다.

위 과정은 기본 블록을 통해 수행되기 때문에 사전에는 다음과 같은 네 종류의 정보가 포함된다.

1. **이미 계산된 가상 레지스터:** 자기 자신의 이름으로 사용될 것인지 아니면 다른 레지스터로 대체되거나 즉치 상수로 대체될 것인지에 대한 정보
2. **특정 변수:** 어떤 레지스터가 (현재) 값을 가지고 있는지에 대한 정보
3. **크기가 큰 특정 상수:** 어떤 레지스터가 값을 가지고 있는지에 대한 정보
4. **argi가 레지스터 이름이거나 상수일 수 있는 일부 트리플(op, arg1, arg2):** 어떤 레지스터가 이미 결과를 가지고 있는지에 대한 정보

저장하기 명령어 x := vi가 나오면 x에 대한 기존의 항목을 모두 제거하고 x를 vi에서 이용할 수 있음을 나타내는 항목을 추가한다. 또 (이 항목에) 메모리에 있는 x 값이 더 이상 유효하지 않음을 기록한다. x가 다른 변수 y의 별칭일 수 있다면 y에 대한 항목도 사전에서 삭제해야 한다(y가 x에 대한 별칭임이 확실하면 y 값을 vi에서 이용할 수 있음을 나타내는 항목을 추가할 수 있다). 앞선 설명에서는 다루지 않았지만 불러오기의 경우에도 이와 유사한 주의가 필요하다. x가 y의 별칭일 수 있고 사전에 메모리에 있는 y 값이 유효하지 않음을 나타내는 y에 대한 항목이 있다면 불러오기 명령어 vi := x 전에 y로의 저장이 선행되어야 한다. 블록의 끝부분에 도달하면 사전을 탐색하면서 메모리에 있는 값이 더 이상 유효하지 않은 모든 변수들에 대한 저장하기 명령어를 생성한다. 어떤 변수들이 서로 별칭일 수 있다면 값이 생성된 순서대로 저장하기 명령어를 생성하게 주의해야 한다. 저장하기를 생성한 후에는 블록을 끝내는 분기(존재하는 경우)를 생성한다.

지역 코드 개선

지역 값 번호 지정 과정에서는 몇 가지 중요한 연산을 자동으로 수행한다. 공통 하위 수식(앞선 예에서는 나오지 않았다)을 식별해 이러한 수식을 한 번만 계산할 수 있게 한다. 상수 선계산과 특정 강도 축소도 구현한다. 끝으로 지역 상수와 복사 전파를 수행하고 중복된 불러오기와 저장하기를 제거한다. 저장하기 명령어를 지연시키기 위한 사전의 사용은 (잠재적인 별칭이 없는 경우) 동일한 기본 블록 내에서 절대로 변수를 두 번 기록하거나 기록한 후 다시 읽지 않게 보장해준다.

찾을 수 있는 공통 하위 수식의 수를 증가시키기 위해 구문 트리를 선형화해서 어떤 종류의 정규 형태로 수식을 재배치하기 전에 이를 탐색하고자 할 수 있다. 예를 들어 교환 법칙이 성립하는 연산에 대해 필요한 경우 피연산자를 사전 순으로 배치하기 위해 하위 트리를 교환할 수 있다. 그러면 `a + b`와 `b + a`가 공통 하위 수식임을 식별할 수 있다. 어떤 경우(예를 들어 배열 주소 계산의 문맥이나 프로그래머의 명시적 허용이 있는 경우) 수식을 정규화하기 위해 결합 법칙이나 분배 법칙을 사용할 수 있다. 물론 6.1.4절에서 살펴봤듯이 이러한 변경은 일반적으로 산술 오버플로우나 절대치 불안정을 유발할 수 있다. 불행히도 이해하기 쉬운 정규화 기술은 `a + b + c`와 `a + c`에 존재하는 중복을 인식하지 못한다. 사전순 배열은 단순한 휴리스틱이다.

별칭에 대한 나이브한 접근 방법은 배열의 원소 `i`에 대한 대입이 임의의 `j`에 대해 원소 `j`를 변경할 수 있으며, 유형이 `t`인 객체에 대한 포인터를 통한 대입이 이 유형의 변수를 모두 변경할 수 있고 서브루틴 호출이 이 서브루틴의 유효 범위(최소한 모든 전역 변수를 포함)에서 볼 수 있는 모든 변수를 변경할 수 있다고 가정하는 것이다. 이러한 가정은 지나치게 조심스러운 것으로 컴파일러가 좋은 코드를 생성할 수 있는 능력을 크게 제한한다. 좀 더 적극적인 컴파일러는 배열 대입에 대한 잠재적인 별칭 집합을 좁히기 위해 배열 첨자의 광범위한 기호 분석을 수행한다. 이와 유사한 분석을 통해 특정 배열 항목이나 레코드 항목이 별칭화되지 않은 스칼라로 처리되어 레지스터로의 할당에 대한 후보가 될 수 있는지를 결정할 수 있을 수 있다.

예 15.12

지역 중복 제거의 결과

(심화학습에 있는) 그림 15.4는 지역 중복을 제거한 후의 `combinations` 서브루틴에 대한 제어 흐름 그래프다. (심화학습에 있는) 그림 15.3의 명령어 중 21개를 제거했으며 이는 모두 변수나 상수의 불러오기였다. 제거된 명령어 중 13개는 개선이 특히 중요한 루프의 몸체(블록 2와 3)에 있다. 레지스터를 상수 4로 곱하는 명령어 두 개와 레지스터를 2로 나누는 명령어 하나에 대해 등가의 시프트로 대체함으로써 강도 축소도 수행했다.

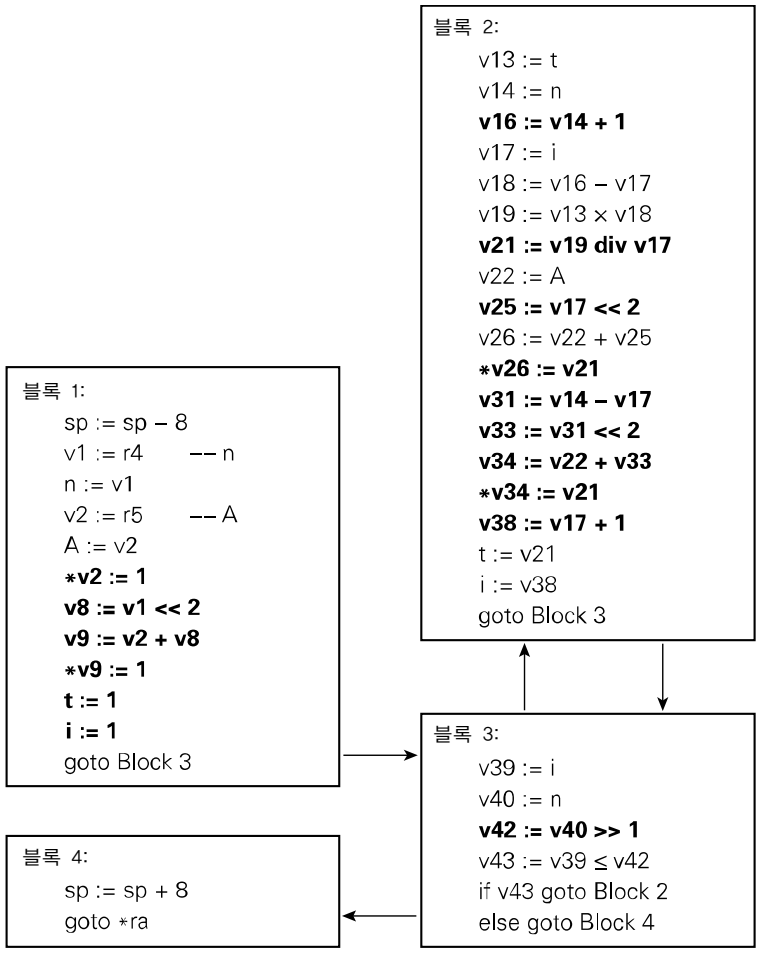


그림 15.4 | 지역 중복 제거와 강도 축소를 수행한 combinations 서브루틴의 제어 흐름 그래프. ©(심화학습에 있는)그림 15.3에서 변경한 부분은 굵은 글씨체로 나타냈다.

설계와 구현

포인터 분석

별칭을 도입하기 위해 포인터를 사용하는 경향은 전통적으로 포트란(Fortran) 컴파일러가 C 컴파일러보다 빠른 코드를 생성했던 이유 중 하나다. 최근까지 포트란에는 포인터가 없었으며 여전히 다수의 포트란 프로그램에서 포인터를 사용하지 않는다. 이와 달리 C 프로그램은 포인터를 많이 사용하는 경향이 있다. 포인터에 대한 별칭 분석은 활발한 연구 주제며 좋은 C 컴파일러가 포트란 컴파일러와 경쟁할 수 있는 위치에 도달하는 출발점이다. 이와 관련된 최신 기술을 알아보고자 한다면 마이클 힌드의 논문[Hin01]을 보자.

설계와 구현

공통 하위 수식

소스 코드 수준에서 공통 하위 수식을 제거할 수 있다고 생각하는 것은 당연하며 프로그래머는 때때로 이렇게 하고 싶을 수 있다. 예를 들어 다음의 수식을 보자.

```
x = a + b + c;
y = a + b + d;
```

이 수식은 다음으로 대체할 수 있다.

```
t = a + b;
x = t + c;
y = t + d;
```

그러나 이러한 변경이 항상 코드를 더 읽기 쉽게 해주지는 않으며 컴파일러가 자신의 작업을 수행하고 있다면 이러한 변경은 컴파일러의 작업을 빠르게 해주지도 않는다. 더욱이 공통 하위 수식의 많은 예는 소스 코드에서 전혀 볼 수 없다. 예를 들어 배열 첨자 계산(7.4.3절), 어휘적으로 둘러싸는 유효 범위에서의 변수 참조(8.2절), 복잡한 레코드에서 주변 필드에 대한 참조(7.3.2절)가 있다. 7.7.1절의 설계와 구현에서 설명한 포인터 산술과 마찬가지로 공통 하위 수식을 사람이 직접 제거하는 것은 코드를 더 읽기 쉽게 해주는 경우를 제외하면 보통 좋은 아이디어가 아니다.

✓ 확인문제

1. 코드 개선의 몇 가지 "적극성" 증가 수준을 설명하라.
2. 특정 순서로 수행해야 하는 코드 개선의 예를 세 가지 들어라. (다른 개선을 사이에 두고) 두 번 이상 수행해야 할 수 있는 코드 개선의 예를 두 가지 들어라.
3. 국소적 최적화란 무엇인가? 국소적 최적화기가 프로그램을 변경할 수 있는 방법을 최소한 네 가지 들고 설명하라.
4. 상수 선계산, 상수 전달, 복사 전달, 강도 축소란 무엇인가?
5. 레지스터에 있는 값이 유효하다는 것은 어떤 의미인가?
6. 제어 흐름 그래프란 무엇인가? 제어 흐름 그래프가 많은 전역 코드 개선 형태에서 중심이 되는 이유는 무엇인가? 제어 흐름 그래프는 서브루틴 호출을 어떻게 지원하는가?
7. 값 번호 지정이란 무엇이며 그 목적은 무엇인가?

8. 공통 하위 수식과 수식 재배치 간의 관계를 설명하라.

9. 일반적으로 프로그래머가 공통 하위 수식을 소스 코드 수준에서 제거하는 것이 실용적이지 않은 이유는 무엇인가?

15.4 전역 중복과 자료 흐름 분석

이 절에서는 기본 블록 간 경계에 걸친 중복된 불러오기와 계산의 제거를 다룬다. 전역 값 번호 지정을 수행하기 위해 기본 블록 코드를 정적 단일 대입문(SSA, static single assignment) 형태로 변환한다. 일단 값 번호가 지정되면 전역 공통 하위 수식 제거, 전역 상수 전파, 전역 복사 전파를 수행할 수 있게 된다. 컴파일러에서 SSA 형태로의 변환과 다양한 전역 최적화는 모두 자료 흐름 분석으로부터 나온다. 이 절에서는 우선 SSA 형태로의 변환을 매우 비형식적으로 다룬 후 몇 가지 전역 최적화(명확히 말하면 공통 하위 수식과 불필요한 저장하기 명령어를 식별하는 문제에 대한 전역 최적화)를 상세히 다룬다. ©(심화학습에 있는) 15.5절에서는 (특히) 불변 계산을 루프 밖으로 꺼내는 데 이용하는 도달 정의(reaching definition)의 계산을 위한 자료 흐름 방정식도 다룬다.

전역 중복 제거는 별도의 지역 패스 없이 지역 중복에서와 같은 방법으로 구조화할 수 있다. 그러나 지역 패스가 이미 수행되었다고 가정하면 전역 알고리즘의 구현과 설명은 좀 더 쉬워진다. 특히 지역 중복 제거는 (이 책의 설명에서는 다루지 않는 별칭이 없는 경우) 기본 블록에서 어떤 변수도 두 번 이상 읽거나 기록되지 않는다고 가정할 수 있게 해준다.

【15.4.1】 SSA 형태와 전역 값 번호 지정

©(심화학습에 있는) 15.3절에서 소개했듯이 값 번호 지정은 주어진 코드 몸체에서 불러오거나 계산한 모든 기호적으로 구별되는 값에 유일한 가상 레지스터 이름을 할당하며 이를 통해 중복된 불러오기나 계산을 인식할 수 있다. 전역 값 번호 지정의 첫 번째 단계는 서로 다른 기본 블록의 변수에 기록될 수 있는 값들을 구별하는 것이다. 이 단계는 정적 단일 대입문(SSA) 형태를 사용해서 수행한다.

중간 수준 IF로의 초기 변환은 모든 가상 레지스터가 유일한 명령어에 의해 값을 대입받게 보장했다. 이 유일성은 지역 값 번호 지정을 통해 얻었다. 그러나 변수는 둘 이상의 기본 블록에서 대입될 수 있다. 그러므로 SSA 형태로의 변환은 변수명에 첨자, 즉 각 저장 명령어마다 각기 다른 첨자를 추가하는 것으로 시작한다. 이러한 규약을 통해

좀 더 쉽게 전역 중복을 식별할 수 있다. 이는 SSA라는 용어도 설명해준다. SSA 프로그램의 첨자 달린 변수는 단일 정적(컴파일 시점) 대입, 즉 단일 저장 명령어를 가진다.

프로그램의 흐름을 따라가면서 대응되는 저장하기를 일치시키기 위해 불러오기 명령어의 변수에 첨자를 할당한다. 명령어 `v2 := x`가 명령어 `x3 := v1`에 의해 기록된 `x` 값을 읽는 것이 보장된다면 `v2 := x`를 `v2 := x3`로 대체할 수 있다. 어떤 버전의 `x`를 읽는지 알 수 없다면 가능한 대안들 중에서 선택하기 위해 가설 함수 Φ를 사용한다. 다행히 실행 시간에 실제로 Φ-함수를 계산할 필요는 없다. 가설 함수의 유일한 목적은 가능한 코드 개선의 식별을 돕는 것이며 가설 함수는 목표 코드 생성 전에 (첨자와 함께) 제거한다.

일반적으로 SSA 형태로의 변환(그리고 특히 Φ-함수의 식별)에는 자료 흐름 분석이 필요하다. 자료 흐름의 개념은 ◎(심화학습에 있는) 15.4.2절에서 전역 공통 하위 수식 제거를 설명할 때 다룬다. 이 부속 절에서는 비형식적으로 SSA 코드를 생성한다. 자료 흐름 형식화는 고급 컴파일러 교과서[CT04, 9.3절, AK 02, 4.4.4절, App97, 19.1절, Muc97, 8.11절]에서 찾아볼 수 있다.

예 15.13
SSA 형태로의 변환

`combinations` 서브루틴(◎심화학습에 있는 그림 15.4)에서는 블록 1의 끝부분에 있는 `t`와 `i`의 저장에 첨자 1을 할당한다. 블록 2의 끝부분에 있는 `t`와 `i`의 저장에는 첨자 2를 할당한다. 그러므로 블록 1의 끝부분에서는 t_1과 i_1이 유효하며 블록 2의 끝부분에서는 t_2와 i_2가 유효하다. 블록 3의 경우는 어떤가? 제어가 블록 1에서 블록 3로 진입하면 t_1과 i_1이 유효할 것이다. 하지만 제어가 블록 2에서 블록 3로 진입하면 t_2와 i_2가 유효할 것이다. 블록 1에서 블록 3로 진입하면 첫 번째 인자를 반환하고 블록 2에서 블록 3로 진입하면 두 번째 인자를 반환하는 함수 Φ를 만든다. 그리고 이 함수를 사용해서 새로운 이름 t_3과 i_3에 값을 기록한다. 블록 3은 `t`나 `i`를 수정하지 않기 때문에 t_3과 i_3이 블록의 끝부분에서 유효하리란 것을 알 수 있다. 더욱이 제어가 항상 블록 3에서 블록 2로 진입하기 때문에 t_3과 i_3은 블록 2의 시작 부분에서도 유효하다. 블록 2의 `v13`에 대한 불러오기는 t_3을 반환하게 보장되며 블록 2의 `v17`과 `v39`에 대한 불러오기는 i_3을 반환하게 보장된다.

SSA는 프로그램의 임의 지점에서 `vi`와 `vj`를 기호적으로 동일한 수식과 함께 불러온다면 이 두 값은 이전의 동일한 저장 명령어(의 동일한 실행)에 의해 생성되게 불러오기의 우편에 첨자와 Φ-함수로 주석을 단다. `combinations`는 간단한 서브루틴이기 때문에 하나의 Φ-함수만 필요하다. 이 함수는 제어가 블록 1에서 블록 3로 진입했는지, 블록 2에서 진입했는지를 나타낸다. 좀 더 복잡한 서브루틴에는 둘 이상의 이전 블록을 가지는 다른 블록들에 대해 추가적인 Φ-함수가 존재할 수 있다. `combinations` 서브루틴의 SSA 형태는 그림 15.5와 같다.

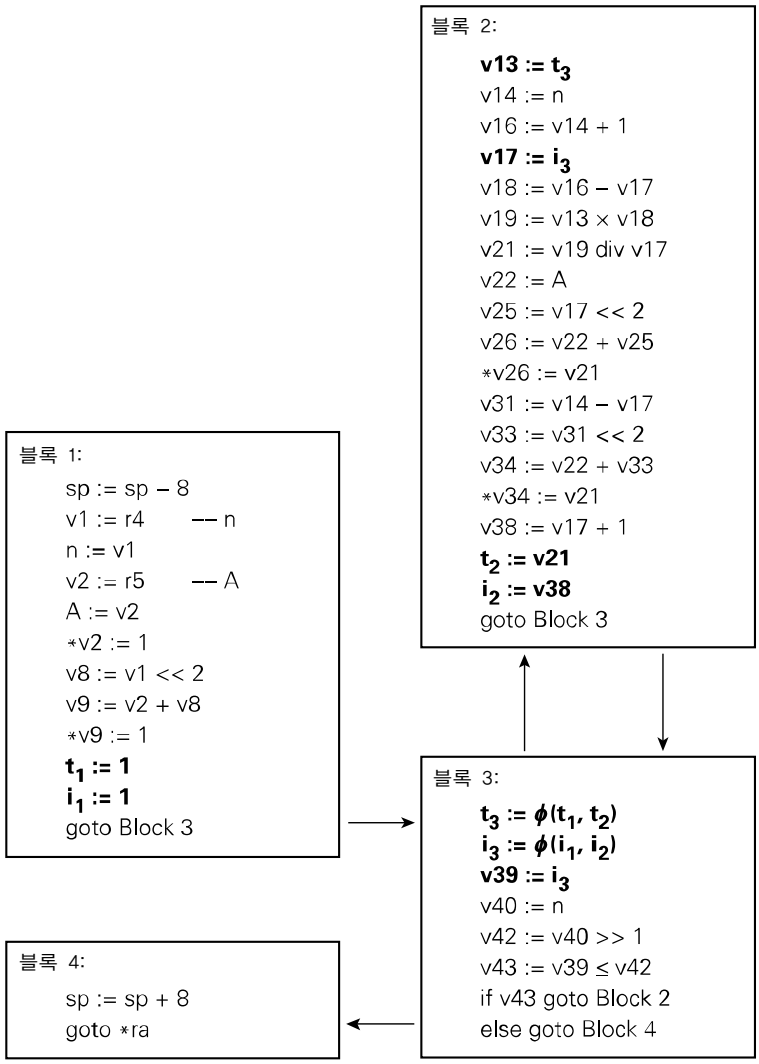


그림 15.5 | 정적 단일 대입문(SSA) 형태로 나타낸 combinations 서브루틴의 제어 흐름 그래프. 그림 15.4에서 변경한 부분은 굵은 글씨체로 나타냈다.

예 15.14
전역 값 번호 지정

Φ-함수에 의해 결정되는 흐름 의존적 값과 더불어 이제 전역 값 번호 지정을 수행할 차례가 되었다. 지역 값 번호 지정에서와 마찬가지로 목표는 기호적으로 동등한 수식을 가지게 보장되는 가상 레지스터들을 병합하는 것이다.

지역 값 번호 지정의 경우에는 불러온 수식과 계산된 수식과 이를 포함하는 가상 레지스터의 이름을 매핑하는 사전을 유지하면서 코드에 대한 선형 패스를 수행할 수 있었다. 전역 값 번호 지정의 경우에는 코드가 순환을 가질 수 있기 때문에 이 접근 방법으로 충분하지 않다. 일반적인 방법은 자료 흐름을 사용해서 형식화하거나 2.2.1절의 DFA 최소화 알고리즘과 유사한 방식으로 모든 수식을 동일한 최상위 수준 연산자로

통일한 후 피연산자가 구별되는 수식들을 반복적으로 분리하는 좀 더 간단한 알고리즘[Muc97, 12.4.2절]을 통해 얻을 수 있다. 여기서는 다시 combinations의 예를 통해 비형식적으로 전역 값 번호 지정 분석을 수행한다.

전역 값 번호 지정은 블록 1의 가상 레지스터 이름들을 채택하는 것으로 시작할 수 있다. 지역 중복은 제거했기 때문에 이 이름들은 이미 최대한 병합된 상태다. 블록 2에서 두 번째 명령어는 v14에 n을 불러온다. 블록 1에서 n에 대해 이미 v1을 사용했기 때문에 동일한 이름을 치환할 수 있다. 이 치환은 모든 가상 레지스터가 단일 정적 명령어에 의해 값이 주어진다는 가정에 최초로 위배된다. 그러나 이 "위배"는 안전하다. 두 n이 모두 동일한 첨자(이 경우에는 첨자가 없다는 것이 같음)를 가지므로 코드의 임의 지점에서 v1과 v14가 모두 주어진 값을 가진다면 그 값들이 동일하리란 것을 알 수 있다. (아직) 블록 1이 먼저 실행될 것을 모르기 때문에 (아직은) 블록 2의 불러오기를 제거할 수 없다. 일관성을 위해 블록 2의 세 번째 명령어에서 v14를 v1로 대체한다. 그 다음 비슷한 이유에서 8번째, 10번째, 14번째 명령어의 v22를 v2로 대체한다.

블록 3에서는 더 많은 대체를 수행한다. 첫 번째 실제 명령어($v39 := i_3$)에서는 블록 2의 v17로 동일한 우편을 불러온다는 것을 알 수 있고 그러므로 첫 번째와 네 번째 명령어의 v39를 v17로 대체한다. 이와 마찬가지로 두 번째와 세 번째 명령어의 v40은 v1로 대체한다. 블록 4에는 변경 사항이 없다.

combinations 서브루틴에 대한 전역 값 번호 지정의 결과는 ©(심화학습에 있는) 그림 15.6과 같다. 이 경우 유일하게 식별된 공통 값은 메모리에서 불러온 변수들이다. 좀 더 복잡한 서브루틴에서는 (존재하는 경우 어느 것이 중복인지는 아직 알 수 없더라도) 둘 이상의 블록에서 수행되는 동일하다고 알려진 계산들도 식별한다. 이때 불러오기에 대해 했던 것과 마찬가지로 기호적으로 동등한 모든 계산이 결과를 동일한 가상 레지스터에 위치시킬 수 있게 좌편을 재명명한다.

정적 단일 대입문 형태는 다양한 코드 개선에 유용하다. 이 책의 설명에서는 전역 값 번호 지정에만 정적 단일 대입문 형태를 사용했으며, 이후의 그림에서는 이를 생략한다.

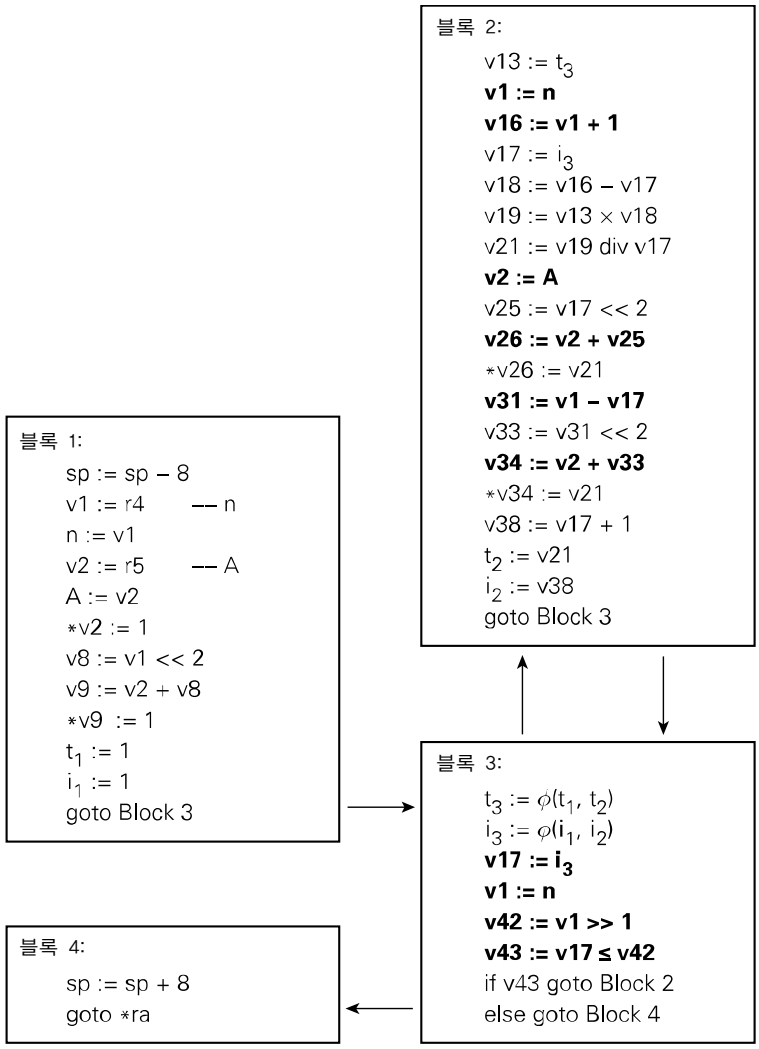


그림 15.6 | 전역 값 번호 지정 후 combinations 서브루틴의 제어 흐름 그래프. 그림 15.5에서 변경한 부분은 굵은 글씨체로 나타냈다.

【15.4.2】 전역 공통 하위 수식 제거

정적 단일 대입문 형태의 구성에서 자료 흐름 분석의 비형식적인 예를 살펴봤다. 이제 전역 공통 하위 수식 제거에 대한 좀 더 형식적인 예를 다룬다. 전역 값 번호 지정의 결과로서 모든 공통 하위 수식은 계산되는 모든 곳에서 동일한 가상 레지스터로 위치된다는 것을 알 수 있다. 그러므로 추후의 설명에서는 수식을 나타내는 데 가상 레지스터 이름을 사용한다. 전역 공통 하위 수식 제거의 목표는 주어진 가상 레지스터에 대한

값을 계산하는 명령어에 대해 이미 이 명령어 이전의 모든 제어 경로에서 해당 계산이 수행되었다고 확신할 수 있기 때문에 이를 제거할 수 있는 위치를 식별하는 것이다.

자료 흐름 분석의 많은 예가 다음과 같은 프레임워크로 나타낼 수 있다. (1) 각 기본 블록 B에 대한 In_B, Out_B, Gen_B, $Kill_B$라는 네 개의 집합, (2) Gen과 Kill 집합을 위한 값들, (3) 임의로 주어진 블록 B에 대한 집합들을 관련짓는 방정식, (4) 주어진 블록의 Out 집합을 이 블록을 계승하는 다음 블록들의 In 집합과 관련짓거나 블록의 In 집합을 이전 블록들의 Out 집합과 관련짓는 방정식, (5) (가끔) 특정 초기 조건. 분석의 목표는 방정식의 고정점, 즉 방정식과 초기 조건을 모두 만족시키는 In 집합과 Out 집합의 모순되지 않는 집합을 찾는 것이다. 어떤 문제는 단일 고정점을 가지는 반면 다른 문제는 둘 이상의 고정점을 가질 수 있다. 둘 이상의 고정점을 가지는 경우에는 대개 최소 고정점이나 최대 고정점(가장 작은 집합이나 가장 큰 집합) 중 하나를 구하면 된다.

예 15.15

이용 가능한 수식을 위한 자료 흐름 방정식

전역 공통 하위 수식 제거의 경우 In_B는 블록 B의 시작부분에서 이용 가능하다고 보장되는 수식(가상 레지스터)의 집합이다. 이러한 이용 가능한 수식들은 모두 이전 블록에 의해 설정된다. Out_B는 B의 끝부분에서 이용 가능하다고 보장되는 수식의 집합이다. $Kill_B$는 B에서 **무효화된**(killed, B에서 수식을 계산하는 데에 사용되었지만 그 이후로는 재계산되지 않는 변수의 하나에 대한 대입에 의해 무효화되는) 수식의 집합이다. 이용 가능한 수식 분석의 자료 흐름 방정식은 다음과 같다.[2]

$$Out_B = Gen_B \cup (In_B \setminus Kill_B)$$
$$In_B = \bigcap_{B\text{의 이전 블록 } A} Out_A$$

초기 조건은 $In_1 = \varnothing$, 즉 실행의 시작부분에서는 어떤 수식도 이용 가능하지 않다는 것이다.

이용 가능한 수식의 분석에서는 정보가 여러 분기를 걸쳐 전방으로 흐르기 때문, 즉 한 블록의 In 집합이 이전 블록들의 Out 집합에 의존적이기 때문에 이용 가능한 수식의 분석을 **전방 자료 흐름 문제**라고 한다. 이 절의 후반부에서 후진 자료 흐름 문제의 예를 살펴본다.

예 15.16

이용 가능한 수식에 대한 고정점

앞서 구한 방정식들에서 구하고자 하는 고정점은 2.3.2절에서 FIRST와 FOLLOW 집합을 계산했던 것과 매우 유사한 귀납적(반복적) 방식으로 계산한다. In_B에 대한 방정식에서는 수식이 B로 통하는 모든 경로상에서 이용 가능해야 함을 나타내기 위해 교집합을 사용한다. 이는 In_B가 반복적 알고리즘에서 이어지는 반복을 거치면서 줄어들 수밖에 없음을 의미한다. 이용 가능한 수식을 최대한 많이 찾고자 하기 때문에 초기에는 모든 수식이 첫 번째 블록을 제외한 모든 블록에 입력으로서 이용 가능하다고 가정

주2. 여기서 사용한 집합 표기법은 표준 표기법이다. $\cup_i S_i$는 모든 집합 S_i의 합집합을 의미하며 $\cap_i S_i$는 모든 집합 S_i의 교집합을 나타낸다. "A 빼기 B"라고 읽는 A\B는 A에는 있지만 B에는 없는 모든 원소의 집합을 의미한다.

한다. 즉, $In_{B,B\neq1}$ = {n, A, t, i, v1, v2, v8, v9, v13, v16, v17, v18, v19, v21, v25, v26, v31, v33, v34, v38, v42, v43}이다.

Gen과 Kill 집합은 각각의 기본 블록에 대한 한 번의 후진 패스에서 찾을 수 있다. 예를 들어 블록 3에서 마지막 대입문은 v43에 대한 값을 정의한다. 그러므로 v43이 Gen_3에 속한다는 것을 알 수 있다. 작업을 후진으로 진행하면 v42, v1, v17도 마찬가지로 Gen_3에 속한다는 것을 알 수 있다. 이러한 원소들을 하나하나 발견할 때 이것이 $Kill_3$에 미치는 영향도 고려한다. 가장 레지스터 v43은 프로그램에서 어떤 대입문의 우편에도 나오지 않으므로(v43은 가상 레지스터에 의해 명명된 수식의 일부가 아니다) 이 레지스터에 값을 부여함으로써 무효화되는 것은 아무 것도 없다. 가상 레지스터 v42는 v43에 의해 명명된 수식의 일부지만 v43이 블록의 후반부에서 값을 할당 받기 때문에(v43은 이미 Gen_3에 속한다) v42에 대한 대입문 때문에 v43이 $Kill_3$로 옮겨지지는 않는다. 가상 레지스터 v1의 경우는 다르다. v1은 v8, v6, v31, v42에 의해 명명된 수식의 일부다. v42는 이미 Gen_3에 있기 때문에 이를 $Kill_3$에 추가하지 않는다. 그러나 v8, v16, v31은 $Kill_3$에 추가한다. 이와 마찬가지 방식으로 v17에 대한 대입문은 v18, v21, v25, v38을 $Kill_3$에 추가한다. v8, v16, v18, v21, v25, v31, v38에 의존적인 가상 레지스터는 신경 쓸 필요가 없다는 점의 주의하자. 이는 반복적인 자료 흐름 알고리즘이 처리한다. 여기서 필요한 것은 한 수준의 의존성뿐이다. (예를 들어 블록 1과 2의 끝부분에서) 프로그램 변수에 대한 저장은 이에 대응되는 가상 레지스터를 무효화한다.

네 블록을 모두 후진 분석한 후에는 다음과 같은 Gen 집합과 Kill 집합을 얻게 된다.

Gen_1 = {v1, v2, v8, v9} $Kill_1$ = {v13, v16, v17, v26, v31, v34, v42}

Gen_2 = {v1, v2, v13, v16, v17, v18, v19, v21, v25, v26, v31, v33, v34, v38} $Kill_2$ = {v8, v9, v13, v17, v42, v43}

Gen_3 = {v1, v17, v42, v43} $Kill_3$ = {v8, v16, v18, v21, v25, v31, v38}

Gen_4 = ∅. $Kill_4$ = ∅.

모든 블록에 첫 번째 자료 흐름 방정식($Out_B = Gen_B \cup (In_B \setminus Kill_B)$)을 적용하면 다음과 같은 Out 집합을 얻을 수 있다.

Out1 = {v1, v2, v8, v9}

Out2 = {v1, v2, v13, v16, v17, v18, v19, v21, v25, v26, v31, v33, v34, v38}

Out3 = {v1, v2, v9, v13, v17, v19, v26, v33, v34, v42, v43}

Out4 = {v1, v2, v8, v9, v13, v16, v17, v18, v19, v21, v25, v26, v31, v33, v34, v38, v42, v43}

이제 첫 번째 방정식의 두 번째 반복이 뒤따르는 두 번째 방정식($In_B = \cap_A Out_A$)을 모든 블록에 적용하면 다음과 같은 집합들을 얻게 된다.

In_1 = ∅. Out_1 = {v1, v2, v8, v9}

In_2 = {v1, v2, v9, v13, v17, v19, v26, v33, v34, v42, v43} Out_2 = {v1, v2, v13, v16, v17, v18, v19, v21, v25, v26, v31, v33, v34, v38}

In_3 = {v1, v2}　　Out_3 = {v1, v2, v17, v42, v43}

In_4 = {v1, v2, v9, v13, v17, v19, v26, v33, v34, v42, v43}　　Out_4 = {v1, v2, v9, v13, v17, v19, v26, v33, v34, v42, v43}

각 방정식이 한 번 더 반복하면 다음과 같은 고정점을 얻는다.

In_1 = ∅.　　Out_1 = {v1, v2, v8, v9}

In_2 = {v1, v2, v17, v42, v43}　　Out_2 = {v1, v2, v13, v16, v17, v18, v19, v21, v25, v26, v31, v33, v34, v38}

In_3 = {v1, v2}　　Out_3 = {v1, v2, v17, v42, v43}

In_4 = {v1, v2, v17, v42, v43}　　Out_4 = {v1, v2, v17, v42, v43}

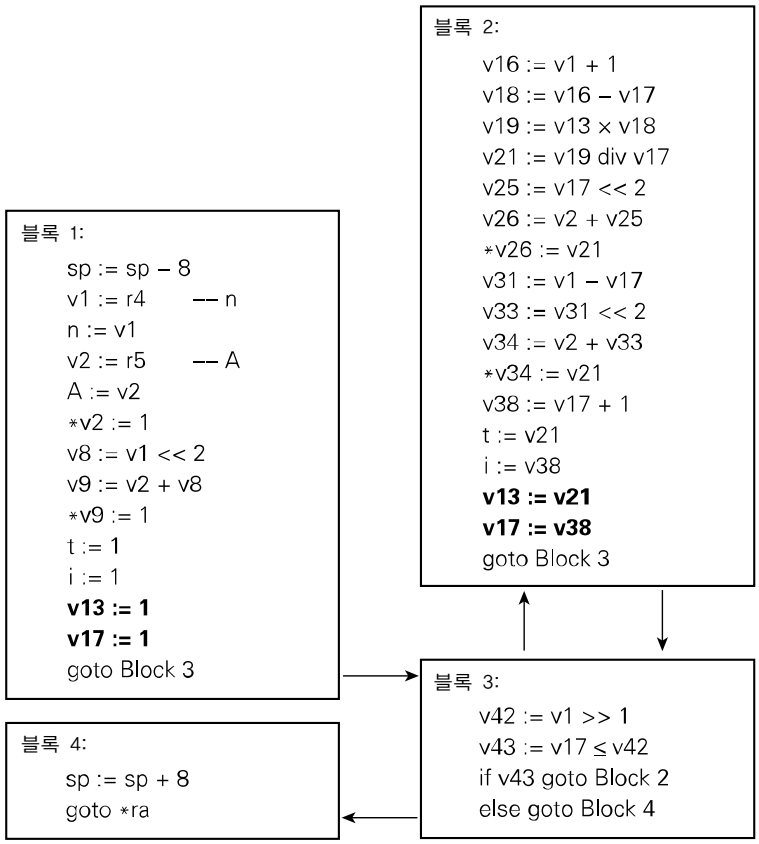


그림 15.7 | 전역 공통 하위 수식 제거를 수행한 후 combinations 서브루틴의 제어 흐름 그래프. 그림 15.6에 있던 불러오기의 다수가 제거되었음에 주목하자. 불러오기를 보충하는 레지스터-레지스터 이동은 굵은 글씨체로 나타냈다. 유효 변수 분석을 통해 이러한 이동 직전에 블록 1의 n과 A의 저장(v1과 v2)과 함께 두 쌍의 명령어를 제거할 수 있다. 유효 변수 분석은 서브루틴 프롤로그와 에필로그에서 스택 포인터에 대한 변경을 생략할 수도 있게 해준다. 지역 변수에 대한 저장 공간은 더 이상 필요 없기 때문이다.

예 15.17

전역 공통 하위 수식 제거의 결과

이제 지금까지 배운 것을 활용해보자. 가상 레지스터가 어떤 블록의 `In` 집합에 속할 때마다 해당 블록에서 이 레지스터의 대입문을 모두 삭제할 수 있다. 예로 살펴보고 있는 `combinations` 서브루틴에서는 블록 2에 있는 `v1`, `v2`, `v17`의 불러오기를, 블록 3에 있는 `v1`의 불러오기를 제거할 수 있다. 이와 더불어 변수에 대응되는 가상 레지스터가 어떤 블록의 `In` 집합에 속할 때마다 이 블록으로 들어오는 모든 잠재적 경로상에서는 이 변수의 불러오기를 레지스터-레지스터 이동으로 대체할 수 있다. `combinations`의 예에서는 블록 2에 있는 `t`의 불러오기와 블록 3에 있는 `i`의 불러오기를 대체할 수 있다(블록 2에 있는 `i`의 불러오기는 이미 제거했다). 이를 보충하기 위해 블록 1의 끝부분에서는 상수 1로 `v13`과 `v17`을 불러오고, 블록 2의 끝부분에서는 `v21`을 `v13`으로 이동시키고 `v38`을 `v17`로 이동시킨다. 최종 결과는 ©(심화학습에 있는) 그림 15.7과 같다(주의 깊은 독자는 엄격히 말해 `v21`과 `v38`이 필요하지 않음을 알아챘을 수 있다. `v13`과 `v17`로 새로운 값을 바로 계산하면 두 개의 레지스터-레지스터 이동을 제거할 수 있다. 이는 틀린 것은 아니지만 지금 수행할 필요는 없으며 ©심화학습에 있는 15.5.2절과 ©심화학습에 있는 15.8절에서 각각 설명할 귀납 변수 최적화와 레지스터 할당을 수행할 때 처리할 수 있다).

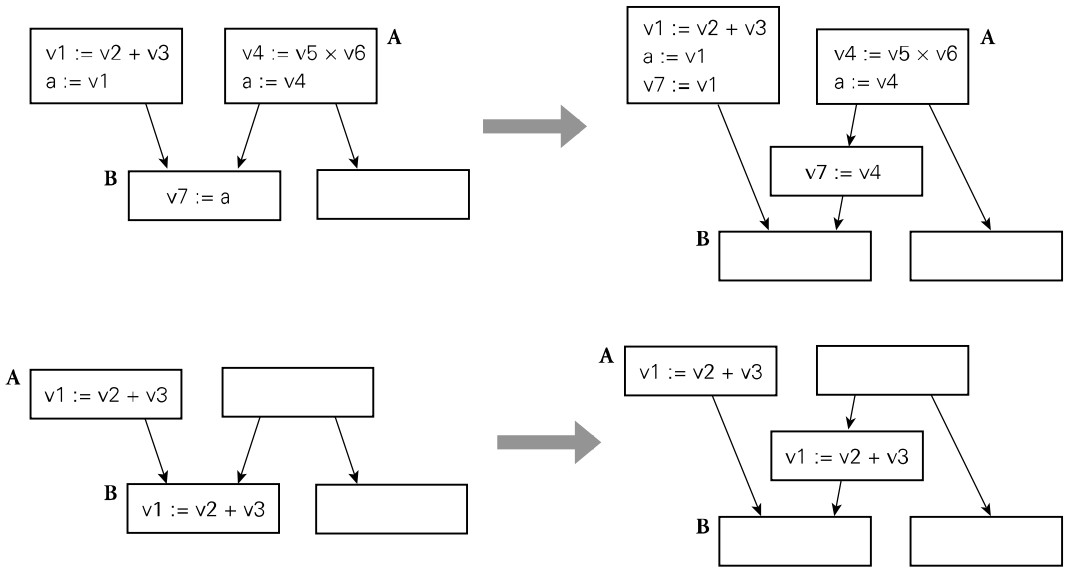


그림 15.8 | 중복된 불러오기를 제거하기 위한 제어 흐름 그래프의 연결선 분할(상단)과 부분적으로 중복된 계산을 제거하기 위한 제어 흐름 그래프의 연결선 분할(하단)

제어 흐름 연결선 분할

예 15.18

연결선 분할 변환

변수가 기록되는 어떤 블록(A라 하자) 뒤에 둘 이상의 블록이 오는데 이 중 하나의 블록(B라 하자)만이 중복된 불러오기를 포함하며 B가 둘 이상의 이전 블록을 가진다면 레지

스터-레지스터 이동을 유지하기 위해 A와 B 사이의 호에 새로운 블록을 생성해야 한다. 이를 통해 이동이 필요하지 않은 코드 경로에서는 이동이 실행되지 않게 한다. 이와 비슷한 맥락에서 수식이 A에서는 이용 가능하지만 B의 다른 이전 블록에서는 이용 가능하지 않다면 이것이 없는 이전 블록이나 이 이전 블록 뒤에 둘 이상의 블록이 온다면 연결 호상의 새로운 블록으로 수식의 불러오기나 계산을 이동시킬 수 있다. 이러한 이동은 A로 오는 경로상에 존재하는 중복을 제거한다. 이러한 "연결선 분할" 변환을 ©(심화학습에 있는) 그림 15.8에 나타냈다. 일반적으로 불러오기나 계산이 흐름 그래프의 일부 경로상에서는 이전의 불러오기나 저장하기의 반복이지만 다른 경로에서는 반복이 아닌 경우 이러한 불러오기나 계산은 부분적으로 중복된다고 한다. combinations 예에서는 연결선 분할이 필요치 않다.

공통 하위 수식 제거는 레지스터 압력에 복잡한 영향을 미친다. 수식 v10 + v20이 프로그램의 앞부분에서 레지스터 v30으로 계산되었으며, 이를 이용해 이 수식의 재계산을 v30의 직접 이용으로 대체하면 v30의 유효 범위(live range, 해당 값이 필요한 명령어의 범위)를 확장할 수 있다. 이와 동시에 v10과 v20이 프로그램의 중간 부분에서 다른 목적으로 사용되지 않는다면 이 레지스터들이 유효한 범위를 줄일 수 있다. 레지스터가 많이 필요한 서브루틴에서 좋은 컴파일러는 때때로 유효 범위를 줄이기 위해 공통 하위 수식 제거의 역(전방 치환이라고 함)을 수행한다.

유효 변수 분석

공통 하위 수식 제거와 같이 상수 전파와 복사 전파도 자료 흐름 분석의 예로 형식화할 수 있다. 이러한 분석은 combinations 예에서 어떠한 개선도 산출하지 않으며 이 책에서는 다루지 않는다. 대신 유효 변수 분석을 살펴본다. 유효 변수 분석은 현재 다루고 있는 예뿐만 아니라 일반적으로 전역 공통 하위 수식 분석이 불러오기 명령어를 제거한 모든 서브루틴에서 매우 중요하다.

유효 변수 분석은 앞서 언급한 후진 흐름 문제로 어떤 명령어가 앞으로 필요한 값을 생성하는지 결정함으로써 유효하지 않은(불필요한) 명령어를 제거할 수 있게 해준다. combinations 서브루틴 예에서는 메모리에 기록된 값과 유효하지 않은 저장의 제거만 다룬다. 유효 변수 분석은 가상 레지스터에 있는 값에도 적용할 수 있으며, 이때 유효 변수 분석은 유효하지 않은 다른 명령어를 식별하는 데 도움을 준다(combinations 예에서는 이러한 명령어 중 어떤 것도 이렇게 초기에는 나오지 않는다).

예 15.19

유효 변수를 위한 자료 흐름 방정식

자료 흐름 분석의 유효 변수 분석 예에서 In_B는 블록 B의 시작부분에서 유효한 변수들의 집합이다. Out_B는 블록 B의 끝부분에서 유효한 변수들의 집합이다. Gen_B는 B에서 처음으로 기록되지 않고 B에서 읽은 변수들의 집합이다. $Kill_B$는 처음으로 읽히지 않고 B에서 기록된 변수들의 집합이다. 자료 흐름 방정식은 다음과 같다.

$$In_B = Gen_B \cup (Out_B \setminus \text{Kill}_B)$$
$$Out_B = \bigcup_{successors\ C\ of\ B} In_C$$

이용 가능한 수식 분석에 대한 방정식과 비교해 볼 때 In과 Out의 역할이 반대로 되었으며(이것이 바로 유효 변수 분석이 후진 문제인 이유다) 두 번째 방정식의 교집합 연산자는 합집합으로 변경되었다. 교집합("모든 경로") 문제는 블록 간 모든 경로의 정보 흐름을 필요로 한다. 이와 달리 합집합("임의 경로") 문제는 어떤 경로를 따르는 흐름이 존재할 것을 필요로 한다. 좀 더 자세한 자료 흐름의 예는 ©(심화학습에 있는) 연습문제 15.7과 15.8에 담았다.

예 15.20

유효 변수에 대한 고정점

combinations 예에서는 다음과 같은 집합을 얻을 수 있다.

$Gen_1 = \varnothing$. $Kill_1 = \{n, A, t, i\}$

$Gen_2 = \varnothing$. $Kill_2 = \{t, i\}$

$Gen_3 = \varnothing$. $Kill_3 = \varnothing$.

$Gen_4 = \varnothing$. $Kill_4 = \varnothing$.

분석은 모든 블록 B에 대해 $In_B = Out_B = \varnothing$로 시작한다(서브루틴이 비지역[예를 들어 전역] 변수를 기록했다면 이러한 변수는 Out4의 초기 멤버가 된다). 자료 흐름 방정식의 반복이 한 번 돌면 모든 블록 B에 대해 $In_B = Gen_B$와 $Out_B = \varnothing$를 얻는다. 그러나 모든 B에 대해 $Gen_B = \varnothing$이기 때문에 이것이 고정점이 된다! 공통 하위 수식 제거를 거치면 어떤 매개변수나 지역 변수도 유효하지 않은 상황에 놓이게 된다. A, n, t, i의 저장이 모두 제거될 수 있다(그림 15.7을 보자). 이와 더불어 t와 i가 온전히 레지스터에 저장될 수 있다는 사실은 서브루틴 프롤로그와 에필로그에서 스택 포인터를 갱신할 필요가 없다는 것을 의미한다. 사실 어떤 스택 프레임도 존재하지 않는다.

별칭은 공통 하위 수식 제거와 유효 변수 분석 모두에서 보수적인 방식으로 다뤄야 한다. 저장하기 명령어가 변수 x를 수정할 수 있다면 공통 하위 수식 제거의 목적을 위해 저장하기가 x에 의존적인 모든 수식을 무효화한다고 간주해야 한다. 불러오기 명령어가 x에 접근할 수 있으며 x가 이 불러오기를 포함하는 블록의 앞부분에서 기록되지 않는다면 x는 블록 시작부분에서 유효한 것으로 간주해야 한다. combinations 예에서는 컴파일러가 참조 매개변수의 경우와 마찬가지로 배열 A가 값 매개변수 n이나 지역 변수 t와 i에 별칭을 부여할 수 없음을 검증할 수 있다고 가정했다.

✓ 확인문제

10. 정적 단일 대입문(SSA) 형태란 무엇인가? 전역 값 번호 지정에는 SSA 형태가 필요하지만 지역 값 번호 지정에는 필요치 않은 이유는 무엇인가?

11. SSA 형태의 문맥에서 Φ-함수란 무엇인가?

12. 자료 흐름 분석의 세 가지 다른 예를 들어라. 전방 흐름과 후진 흐름의 차이를 설명하라. 모든 경로 흐름과 임의 경로 흐름의 차이를 설명하라.

13. 다수의 자료 흐름 분석 예에 공통적인 `In`, `Out`, `Gen`, `Kill` 집합의 역할을 설명하라.

14. 부분적으로 중복된 계산이란 무엇인가? 부분 중복을 제거하는 알고리즘이 제어 흐름 그래프의 연결선을 분할할 필요가 있는 이유는 무엇인가?

15. 이용 가능한 수식이란 무엇인가?

16. 전방 치환이란 무엇인가?

17. 유효 변수 분석이란 무엇이며 그 목적은 무엇인가?

18. 코드 개선 알고리즘이 별칭의 가능성을 고려해야 하는 예를 최소 세 가지 설명하라.

15.5 루프 개선 I

프로그램이 대부분의 시간을 루프에서 소비하기 때문에 루프 속도를 개선하는 코드 개선은 특히 중요하다. 이 절에서는 두 종류의 루프 개선을 살펴본다. 한 종류의 루프 개선은 불변 계산을 루프의 몸체 밖으로 꺼내 헤더로 옮기는 방법이며, 다른 한 종류는 귀납 변수를 유지하면서 소비하는 시간을 줄이는 방법이다. ©(심화학습에 있는) 15.7절에서는 루프 몸체가 본래 루프의 반복 일부를 둘 이상 포함하게 재구조화함으로써 명령어 스케줄링을 개선하는 변환과 캐시 성능을 개선하고 병렬화 기회를 늘리기 위해 다중으로 중첩된 루프를 조작하는 변환을 다룬다.

【15.5.1】 루프 불변식

루프 불변식은 매 반복마다 동일한 결과가 보장되는 루프 내의 명령어(즉, 불러오기나 계산)다. 루프가 n번 실행되며 불변 명령어를 루프 몸체 밖으로 꺼내 헤더로 이동시킬 수 있다면(이때 몸체에서의 사용을 위해 레지스터에 결과를 저장한다) 프로그램에서 n-1번의 계산을 제거하게 되며, 이는 잠재적으로 매우 큰 시간 절약이다.

명령어가 불변식인지 아닌지를 알기 위해 루프의 몸체를 식별하고 피연산자 값이 정의된 위치를 추적해야 한다. 구조화된 제어 흐름에만 의존하는 언어(예를 들어 에이다나

자바)에서는 첫 번째 작업(루프 식별)이 쉽다. 구문 트리를 선형화할 때 단순히 적절한 표시자를 저장하면 된다. goto문을 사용하는 언어에서는 덜 구조화된 제어 흐름 그래프로부터 루프를 구성(복구)해야 할 수도 있다.

설계와 구현

루프 불변식

많은 루프 불변식이 주소 계산, 특히 배열을 위한 주소 계산에서 나온다. 194–CD의 보충설명에서 논한 공통 하위 수식과 마찬가지로 루프 불변식도 가끔 프로그램 소스에서 명시적이지 않으며 그러므로 사람이 직접하는 최적화로는 불변식을 루프에서 꺼낼 수 없다.

설계와 구현

제어 흐름 분석

구조화된 제어 흐름을 가지는 현대 언어에서는 대부분의 루프가 구문 트리에서 명시적인 구성소와 바로 일치된다. 일부 루프는 묵시적일 수 있는데, 이런 예로는 대규모의 레코드나 서브루틴 매개변수를 초기화하거나 복사해야 하는 루프, 꼬리 재귀를 표현하기 위한 루프 등이 있다. 좀 더 오래된 언어의 경우에는 제어 흐름 분석이라는 기술을 통해 구조체 복구를 수행한다. 제어 흐름 분석에 대한 자세한 내용은 표준 컴파일러 교과서[AK02, 4.5절, App97, 18.1절, Muc97, 7장]에서 찾아볼 수 있으며 이 책에서는 더 이상 다루지 않는다.

예 15.21

도달 정의를 위한 자료 흐름 방정식

피연산자가 정의됐을 수 있는 위치를 추적하는 것은 결국 도달 정의의 문제가 된다. 형식적으로 말해서 값 v를 위치(변수나 레지스터) l에 대입하는 명령어는 v가 코드 지점 p에서 여전히 l에 존재할 수 있는 경우 코드 지점 p에 도달한다. ©(심화학습에 있는) 15.4.1절에서 비형식적으로 다룬 정적 단일 대입문 형태에서의 규약과 마찬가지로 도달 정의 문제도 전방, 임의 경로 자료 흐름 방정식들의 집합으로 구조화할 수 있다. Gen_B를 블록 B의 최종 대입문(B에서 더 이상 덮어쓰지 않는 대입문)들의 집합이라고 하자. B의 각 대입문에 대해 동일한 위치에 대한 (모든 블록에 있는) 나머지 대입문은 모두 $Kill_B$에 위치시킨다. 그러면 다음을 얻을 수 있다.

$$Out_B = Gen_B \cup (In_B \setminus Kill_B)$$
$$In_B = \bigcup_{predecessors\ C\ of\ B} Out_C$$

일단 In_B(블록 시작부분에 있는 도달 정의들의 집합)가 주어지면 코드의 간단한 선형 조사를 통해 B 내부에서 사용되는 모든 값의 도달 정의를 결정할 수 있다.

도달 정의가 주어지면 어떤 명령어에 대해 이 명령어의 피연산자가 (a) 상수거나 (b)

전부 루프의 외부에 있는 도달 정의들을 가지거나 (c) 하나의 도달 정의(이러한 도달 정의가 루프 내부에 위치한 명령어 d라 하더라도 d가 그 자체로 루프 불변식인 한)를 가지면 이러한 명령어를 루프 불변식으로 정의한다(특정 변수에 대해 둘 이상의 도달 정의가 있다면 모든 정의가 동일한 값을 대입한다는 것을 알지 않는 한 불변성을 보장할 수 없으며 대부분의 컴파일러는 이러한 추론을 시도하지 않는다). 이전 분석들에서와 마찬가지로 명백한 경우에서 시작해서 고정점에 도달할 때까지 귀납적으로 진행한다.

예 15.22

루프 불변식 이동의 결과

combinations 예의 코드를 눈으로 살펴보면 두 개의 루프 불변식을 볼 수 있다. 하나는 블록 2에 있는 v16에 대한 대입문이고 다른 하나는 블록 3에 있는 v42에 대한 대입문이다. 이러한 불변식을 루프 밖으로 이동시킴으로써(그리고 ©심화학습에 있는 그림 15.7의 유효하지 않은 저장하기와 스택 포인터 갱신을 제거함으로써) ©(심화학습에 있는) 그림 15.9의 코드를 얻을 수 있다.

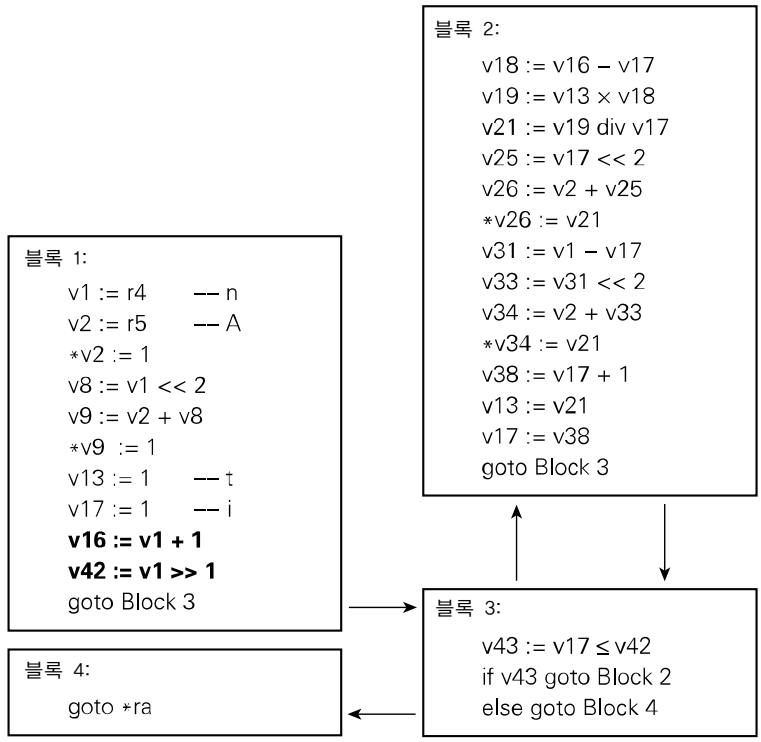


그림 15.9 | v16과 v42(굵은 글씨체로 나타냄)의 불변식 계산을 루프 밖으로 이동시킨 후 combinations 서브루틴의 제어 흐름 그래프. ©(심화학습에 있는) 그림 15.7의 유효하지 않은 저장도 제거했으며 이제는 전적으로 레지스터에 상주하는 t와 i를 위한 스택 공간을 제거했다.

새로운 버전의 코드에서 v16과 v42는 루프가 한 번도 실행되지 않더라도 계산된다. 일반적으로 이러한 선계산은 좋은 아이디어가 아닐 수 있다. 불변식 계산의 비용이 크고 루프가 실제로 실행되지 않으면 프로그램을 더 느리게 만든 것이 된다. 더욱이

불변식 계산이 실행 시간 오류(예를 들어 0으로 나누기)를 발생할 수 있다면 프로그램을 틀리게 만든 것일 수도 있다. 안전하고 효율적인 범용 해결책은 불변식 계산 전에 반복이 실행되는지에 대한 초기 검사를 삽입하는 것이다. 이는 ©(심화학습에 있는) 연습문제 15.4에서 다룬다. `combinations` 서브루틴의 경우에는 좀 더 나이브한 변환이 안전하며 (일반적인 경우) 효율적이다.

【15.5.2】 귀납 변수

귀납 변수(또는 레지스터)는 루프의 연속적인 반복에서 값의 간단한 연쇄를 취하는 변수다. 여기서는 산술 연쇄로 범위를 제한한다. 좀 더 정교한 예는 ©(심화학습에 있는) 연습문제 15.10과 15.11에 담았다. 일반적으로 귀납 변수는 루프 색인, 첨자 계산이나 루프 몸체 내에서 명시적으로 증가되거나 감소되는 변수에서 볼 수 있다. 귀납 변수는 주로 두 가지 이유에서 중요하다.

예 15.23
귀납 변수의 강도 축소

- 보통 귀납 변수는 강도 축소의 기회를 제공하며 특히 곱셈을 덧셈으로 대체함으로써 강도 축소를 할 수 있게 해주는 경우가 많다. 예를 들어 `i`가 루프 색인 변수라면 `i>a`에 대해 `t := k × i + c`와 같은 형태의 수식은 `ti := ti−1 + k`로 대체할 수 있다. 여기서 `ta = k × a + c`다.

예 15.24
귀납 변수 제거

- 보통 귀납 변수는 중복된다. 많은 경우에 루프의 모든 반복에 걸쳐 여러 귀납 변수를 레지스터에 저장하지 않고 좀 더 작은 수만 레지스터에 저장하고 필요할 때 나머지를 계산한다(이 계산이 충분히 저비용이라고 가정한다). 이를 통해 가끔 계산 비용의 증가 없이(때로는 계산 비용이 감소하면서) 레지스터 압력의 감소를 얻을 수 있다. 특히 다른 귀납 변수를 강도 축소한 후에 가끔 종료 검사를 적절히 변경함으로써 루프 색인 변수 자체를 제거할 수 있다(이에 대한 예는 ©심화학습에 있는 그림 15.10에 담았다).

```
A : array [1..n] of record
      key : integer
      // 기타 코드
for i in 1..n
      A[i].key := 0
```

(a)

```
v1 := 1
v2 := n
v3 := sizeof(record)
v4 := &A − v3
L: v5 := v1 × v3
v6 := v4 + v5
*v6 := 0
v1 := v1 + 1
v7 := v1 ≤ v2
if v7 goto L
```

(b)

```
v1 := 1
v2 := n
v3 := sizeof(record)
v5 := &A
L: *v5 := 0
v5 := v5 + v3
v1 := v1 + 1
v7 := v1 ≤ v2
if v7 goto L
```

(c)

```
v2 := &A + (n−1) × sizeof(record)
    −−>1 명령을 가질 것임
v3 := sizeof(record)
v5 := &A
L: *v5 := 0
v5 := v5 + v3
v7 := v5 ≤ v2
if v7 goto L
```

(d)

그림 15.10 | 귀납 변수의 코드 개선. 고수준 의사코드 소스는 (a)와 같다. 귀납 변수 최적화 이전의 목표 코드는 (b)와 같다. (c)에서는 v5가 더 이상 v1(i)에 의존적이지 않은 지점에서 v5, 배열 색인에 대한 강도 축소를 수행하고 v4를 제거했다. (d)에서는 v1 대신 v5를 사용하게 종료 검사를 수정하고 v1을 제거했다.

귀납 변수를 식별, 강도 축소, (때에 따라) 제거하기 위한 알고리즘은 다소 이해하기 쉽지만 매우 지루하고 장황하며[AK02, 4.5절, App97, 18.3절, Muc97, 14장] 여기서는 이에 대한 상세를 다루지 않는다. 많은 응용 프로그램에서 배열과 서브레인지 경계 확인을 제거하는 데 유사한 알고리즘을 사용할 수 있다.

예 15.25

귀납 변수 최적화의 결과

combinations 예에 대해 귀납 변수 최적화를 수행한 후의 코드는 ◎(심화학습에 있는) 그림 15.11과 같다. 두 개의 귀납 변수(배열 포인터 v26과 v34)에 대해 강도 축소를 수행하며 v25, v31, v33에 대한 필요성을 제거했다. 이와 유사하게 v18을 v17에 독립적이게 하며 v16에 대한 필요성을 제거했다. 한 번 사용되는 5번째 귀납 변수(v38)는 이를 계산하는 덧셈으로 대체함으로써 제거했다. 블록 1에서 지역 중복 제거를 반복함으로써 v34의 초기화가 v9에 상주한다고 알려진 값을 활용할 수 있다고 가정한다.

표현상의 목적을 위해 나눗셈 연산을 바로 v13으로 계산해 넣었으며 이를 통해 v21과 더불어 뒤에 있는 v13으로의 대입문을 제거했다. 실제 컴파일러는 나눗셈 시점에 v13의 이전 값이 유효하지 않다는 것을 검증하는 컴파일의 레지스터 할당 단계에 이르러서야 이러한 변경을 수행할 수도 있다(v21은 귀납 변수가 아니며 v21 값의 연쇄는 그렇게 간단하지 않다). 이 변경을 함으로써 이제 블록 2에 있는 마지막 중복 명령어를 제거하고 다른 쟁점과 비교적 독립적으로 명령어 스케줄링을 논할 수 있다.

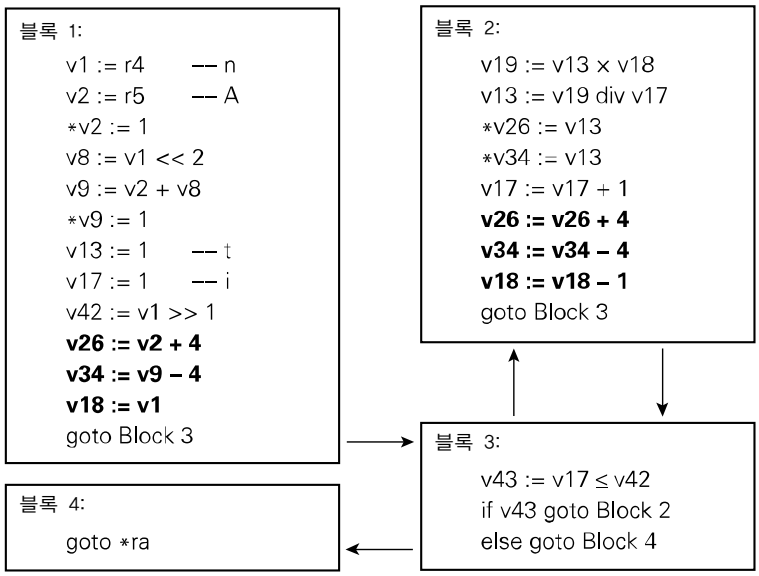


그림 15.11 | 귀납 변수 최적화 후 combinations 서브루틴의 제어 흐름 그래프. 레지스터 v26과 v34에 대해 강도 축소를 수행해 v25, v31, v33을 제거했다. 레지스터 v38과 v21은 v17과 v13으로 병합되었다. v18의 갱신도 단순화해 v16을 제거했다.

15.6 명령어 스케줄링

©(심화학습에 있는) 그림 15.1의 컴파일러 구조 예에서 루프 최적화의 다음 단계는 목표 코드 생성이다. 14장에서 살펴봤듯이 이 단계는 제어 흐름 그래프를 선형화하고 중간 수준 중간 형태의 명령어를 타겟 머신 명령어로 대체한다. 이러한 대체는 가끔 자동으로 생성되는 패턴 매칭 알고리즘에 의해 수행된다. 여기서는 유사어셈블러 "명령어 집합"을 계속 사용하므로 눈에 보이는 유일한 변화는 선형화뿐이다. 명확히 말해서 프로그램의 블록들이 그 이름을 통해 알 수 있는 순서로 연결되어 있다고 가정한다. 루프의 마지막 반복에서 제어는 블록 2에서 블록 3로, 블록 3에서 블록 4로 "떨어진다".

여기서는 두 번의 명령어 스케줄링을 수행하며 그 사이에서 레지스터 할당을 수행한다. 유사 어셈블러의 사용을 가정하고 국소적 최적화는 더 이상 상세히 다루지 않는다. 그러나 ©(심화학습에 있는) 15.7절에서는 목표 코드 생성에 앞서 적용할 수 있는 루프를 위한 추가적인 형태의 코드 개선을 다룬다. 이러한 개선의 설명은 명령어 스케줄링을 다룬 후에 더 분명해지기 때문에 뒤로 미룬다.

파이프라이닝된 기계에서 성능은 컴파일러가 파이프라인을 가득 차게 유지할 수 있는 범위에 매우 의존적이다. 5.5.1절에서 설명했듯이 지연은 (1) 명령어가 이전 명령어에 의해 여전히 사용 중인 기능 단위를 필요로 할 때, (2) 명령어가 이전 명령어에 의해

여전히 계산 중인 자료가 필요할 때, (3) 분기의 결과나 목표가 정해지기 전까지는 실행할 명령어를 선택할 수조차 없을 때 초래될 수 있다. 이 절에서는 기본 블록 내의 명령어들을 재정렬함으로써 해결할 수 있는 (1)과 (2)를 다룬다. (3)에 대한 좋은 해결책에는 일반적으로 하드웨어의 도움을 받는 분기 예측이 필요하다. 컴파일러는 다소 이해하기 쉬운 방식으로 분기 지연을 채우는 하위 문제를 해결할 수 있다[Muc97, 17.1.1절].

예 15.26

파이프라인 지연의 재명명

`combinations` 예의 루프 몸체를 자세히 살펴보면 이제까지 설명한 최적화들은 (심화학습에 있는) 그림 15.3의 블록 2를 명령어 30개로 된 나열에서 (심화학습에 있는) 그림 15.11의 명령어 8개로 된 나열(마지막 `goto`는 세지 않았다)로 변환해왔음을 알 수 있다. 불행히도 명령어를 재정렬하지 않는 파이프라이닝된 기계상에서 이 코드는 여전히 명백한 준최적이다. 특히 두 번째와 세 번째 명령어의 결과는 바로 사용되지만 곱셈과 나눗셈의 결과는 보통 여러 주기 동안 이용할 수 없다. 4주기의 지연을 가정한다면 이 블록은 실행하는 데 16주기가 걸리게 된다.

의존성 분석

파이프라인을 더 잘 사용하기 위해 명령어를 스케줄링하려면 우선 명령어를 방향성 비순환 그래프(DAG)로 정리한다. 5.5.1절에서 설명했듯이 방향성 비순환 그래프에서 각 노드는 명령어 하나를 나타내며, 각 호는 의존성을 나타낸다.[3] 대부분의 호는 흐름 의존성을 나타내는데, **흐름 의존성**이란 한 명령어가 이전 명령어에 의해 생성된 값을 사용하는 것을 말한다. 일부 호는 반의존성을 나타내는데, 반의존성이란 뒤에 오는 명령어가 이전 명령어에 의해 읽혀진 값을 덮어쓰는 것을 말한다. `combinations` 예에서 이러한 반의존성은 귀납 변수의 갱신과 일치한다. 구조 레지스터 할당 후에 명령어 스케줄링을 수행한다면 독립적인 값에 대한 동일한 레지스터의 사용이 반의존성의 수를 증가시킬 수 있으며 출력 의존성도 유발할 수 있다. 출력 의존성이란 뒤에 오는 명령어가 이전 명령어에 의해 기록된 값을 덮어쓰는 것을 말한다. 점차 많은 수의 기계에서 하드웨어 레지스터 재명명을 통해 반의존성과 출력 의존성을 숨길 수 있다.

예 15.27

값 의존성 DAG

공통 하위 수식 분석이 `combinations` 서브루틴에 있던 `i`, `n`, `t`의 불러오기와 저장하기를 모두 제거했으며, `A`의 원소에 대한 불러오기가 없기 때문에(저장만 함) 여기서의 의존성 분석은 온전히 레지스터에 있는 값만 다룬다. 일반적으로는 메모리에 있는 값도 처리해야 하며, 두 명령어가 언제 동일한 위치를 접근해 의존성을 공유하는지 결정하기 위해 별칭 분석에도 의존해야 한다. 타겟 머신이 (5.3절에서 설명한대로) 상태 코드를 가진다면 이를 흐름 의존성, 반의존성, 출력 의존성을 추적하면서 명시적으로 모델화해야 한다.

주3. 여기서 말하는 것은 의존성 DAG다. 이는 15.3절에서 설명한 수식 DAG와 관련이 있기도 하지만 명백히 구별된다. 특히 의존성 DAG는 수식에 가상 레지스터를 할당한 후에 구성되며 의존성 DAG의 노드는 변수와 연산자가 아니라 명령어를 나타낸다.

combinations 예의 블록 2에 대한 의존성 DAG는 ©(심화학습에 있는) 그림 15.12와 같다. 그림에서 알 수 있듯이 이 경우 DAG는 트리다. 이 DAG는 코드를 위에서 아래로 조사하면서 각 명령어 i와 i가 기록한 레지스터를 읽거나(실선) i가 읽은 레지스터를 기록하는(점선) 추후 명령어 j로 연결함으로써 생성한 것이다.

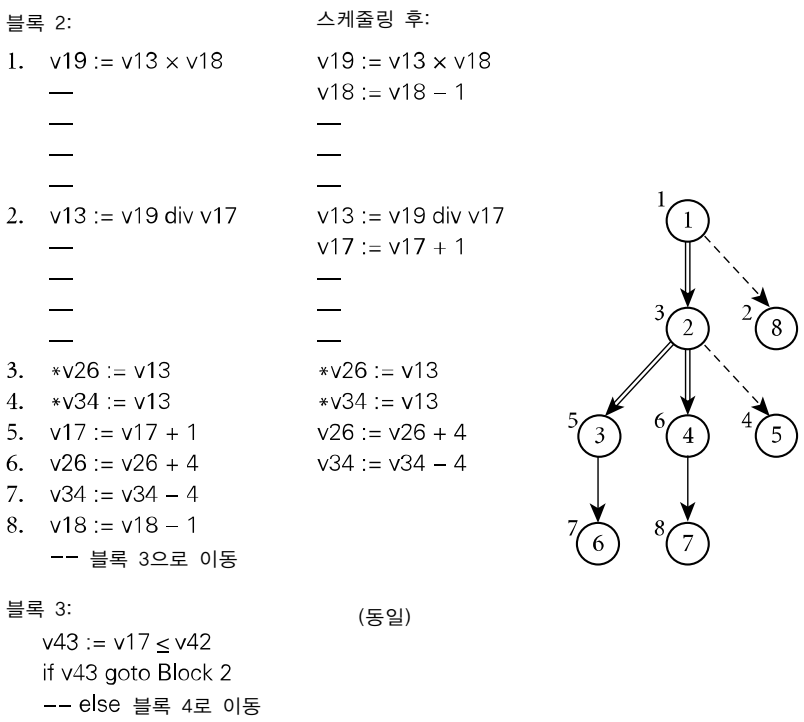


그림 15.12 | 그림 15.11의 블록 2에 대한 의존성 DAG. 명령어 스케줄링 전(왼쪽)과 후(오른쪽)의 루프 전체의 의사코드를 함께 나타냈다. DAG에서 원 안의 번호는 루프의 최초 버전에 있는 명령어와 일치한다. 바로 옆의 좀 더 작은 번호는 새로운 루프에서의 스케줄 순서를 의미한다. 실선은 흐름 의존성을 나타내며 점선은 반의존성을 나타낸다. 이중선은 가설적 기계상의 파이프라인 지연을 피하기 위해 네 개의 추가적인 명령어로 분리해야 하는 명령어 쌍을 의미한다. 지연은 블록 2에 명시적으로 보였다. 배열 색인 코드를 수정하지 않는 한(©심화학습에 있는 연습문제 15.19) 두 개의 명령어만 옮길 수 있다.

의존성 DAG에 대한 임의의 위상 정렬(즉, 각 노드가 자신의 자식 노드보다 앞에 오게 노드를 열거하는 것)은 올바른 스케줄을 나타낸다. 이상적으로 전체 지연을 최소화하는 정렬을 선택해야 한다. 코드 개선의 다른 많은 부분과 마찬가지로 이 작업 역시 NP-난해며 그러므로 실제 기술은 휴리스틱에 의존한다.

타이밍 정보를 표현하기 위해 동일한 파이프라인에서 일시 중지(stall) 없이 j가 i 다음에 실행될 때 명령어 i와 j의 스케줄링 사이에 필요한 주기 수를 반환하는 함수 지연(i, j)를 정의한다(기계 독립성을 유지하기 위해 코드 개선기의 이 부분은 기계의 특징을 담은 테이블로부터 얻어야 하며 이러한 기계 특징은 "하드-코딩"해선 안 된다). 자명하지 않은 지연은

자료 의존성이나 불완전하게 파이프라이닝된 기능 단위와 같은 물리적 자원 사용의 충돌에서 야기될 수 있다. 여기서는 모든 단위가 완전히 파이프라이닝되었다고 가정하며 그러므로 모든 지연은 자료 의존성 때문에 발생한다.

이제 루트에서 리프까지 DAG를 탐색해보자. 각 단계마다 우선 후보 노드들의 집합을 결정한다. 후보 노드란 모든 부모 노드가 스케줄링된 노드를 말한다. 그 다음 각 후보 노드 `i`에 대해 `i`가 일시 중지 없이 실행할 수 있는 가장 이른 시점을 결정하기 위해 이미 스케줄링된 노드와의 지연 함수를 사용한다. 또 `i`에서 리프까지의 모든 경로에 대해 호에 나온 지연 합의 최대값을 미리 계산한다. 이를 통해 `i`가 스케줄링된 후 기본 블록을 완료하는 데 필요한 시간의 하한을 알 수 있다. `combinations` 예에서는 후보 노드 중 하나를 선택하기 위해 다음과 같은 세 가지 휴리스틱을 사용한다.

1. 일시 중지 없이 시작될 수 있는 노드를 선호한다.
2. 1번으로 결정할 수 없으면 블록의 끝부분까지 최대 지연을 가지는 노드를 선호한다.
3. 2번으로도 결정할 수 없으면 본래 소스 코드에서 가장 먼저 나오는 노드를 선호한다(이 전략은 좀 더 직관적인 어셈블리어를 이끌어 디버깅을 돕는다).

그 외에 다음과 같은 스키줄링 휴리스틱이 가능하다.

DAG에서 자식 노드가 많은 노드를 선호한다(이는 스케줄링 알고리즘의 향후 반복에 대한 융통성을 증가시킨다).

- 레지스터를 마지막으로 사용하는 노드를 선호한다(이는 레지스터 압력을 줄여준다).
- 다중 파이프라인의 경우 최근에 명령어를 받지 않은 파이프라인을 사용할 수 있는 노드를 선호한다.
- 타겟 머신이 다중 파이프라인을 가진다면 각 명령어에 대해 이 명령어가 사용할 것으로 생각되는 파이프라인을 추적함으로써 현재 주기에서 시작할 수 있는 후보와 다음 주기까지 실행할 수 없는 후보 노드를 구별할 수 있어야 한다(정밀하지 않은 기계 모델, 캐시 부적중, 기타 예측 불가능한 지연 때문에 때로는 추측이 틀릴 수 있다).

예 15.28

명령어 스케줄링의 결과

불행히도 `combinations`의 DAG에는 선택의 여지가 거의 없다. 유일하게 가능한 개선은 명령어 8을 곱셈이나 나눗셈 지연 빈 공간 중 하나로 이동시키고 명령어 5를 나눗셈 지연 빈 공간 중 하나로 이동시킴으로써 블록 2의 전체 주기 수를 16에서 14로 줄이는 것이다. (1) 타겟 머신이 루프의 끝부분에서 후진 분기를 올바르게 예측하며 (2) 분기의 무효화 지연 빈 공간에 블록 2의 첫 번째 명령어를 중복시킬 수 있다면(마지막 반복을 제외하고는) 블록 3에서 추가적인 지연을 초래하지 않는다. 그러므로 루프의 전체 주기 수는 스케줄링 전에 반복당 18이며, 스케줄링 후에는 반복당 16이다. 이는 11%의 개선이다. ©(심화학습에 있는) 15.7절에서는 재스케줄링을 통해 훨씬 더 빠

른 코드를 얻는 다른 버전의 블록을 살펴본다.

(심화학습에 있는) 15.1절의 거의 끝부분에서 언급했듯이 전역 코드 개선과 레지스터 할당 후에 명령어 스케줄링을 반복하는 것이 바람직할 수 있다. 유용한 값을 가지는 가상 레지스터의 수가 타겟 머신의 구조 레지스터 수를 초과하는 때가 존재하면 일부 값을 메모리로 내려 보내고 추후에 다시 불러오는 코드를 생성해야 한다. 재스케줄링은 이러한 불러오기로 인한 지연을 처리하기 위해 필요하다.

✓ 확인문제

19. 루프 불변식이란 무엇인가? 또 도달 정의란 무엇인가?
20. 때때로 불변식을 루프 밖으로 꺼내는 것이 안전하지 않을 수 있는 이유는 무엇인가?
21. 귀납 변수란 무엇인가? 강도 축소란 무엇인가?
22. 제어 흐름 분석이란 무엇인가? 제어 흐름 분석이 과거에 비해 덜 중요해진 이유는 무엇인가?
23. 레지스터 압력이란 무엇인가? 레지스터 메모리로 내려 보내기란 무엇인가?
24. 명령어 스케줄링은 기계 독립적 코드 개선 기술인가? 설명하라.
25. 왼존성 DAG의 생성과 사용을 기술하라. 흐름 의존성, 반의존성, 출력 의존성 차이를 설명하라.
26. 명령어 스케줄링과 레지스터 할당 간의 관계를 설명하라.
27. 스케줄링할 명령어의 우선순위를 정하는 데 사용할 수 있는 휴리스틱을 몇 개 나열하라.

15.7 루프 개선 II

(심화학습에 있는) 15.5절에서 언급했듯이 대부분의 프로그램이 대부분의 시간을 보내는 곳이 바로 루프기 때문에 루프의 속도를 개선하는 코드 개선은 특히 중요하다. 이 절에서는 본래 루프의 반복을 둘 이상 포함하게 루프 몸체를 재구조화함으로써 명령어 스케줄링을 개선하고 캐시 성능을 개선하거나 병렬화 기회를 늘리기 위해 다중으로 중첩된 루프를 조작하는 변환을 다룬다. 루프 변환과 의존성 이론에 대한 광범위한 내용은 앨렌과 케네디의 교과서[AK02]에서 찾아볼 수 있다.

【15.7.1】 루프 펼치기와 소프트웨어 파이프라이닝

예 15.29
루프 펼치기의 결과

루프 펼치기란 두 개 이상의 소스 수준 루프의 반복을 좀 더 긴 새로운 단일 반복에 내장시켜 스케줄러가 본래 반복들의 명령어들을 섞을 수 있게 해주는 변환이다. `combinations` 예의 반복 2개를 펼치면 ©(심화학습에 있는) 그림 15.13과 같은 코드를 얻는다. 루프의 처음 절반에서 기록된 레지스터에 대해서는 별도의 이름(여기서는 't'로 시작하는 이름)을 사용했다. 이러한 관례를 통해 반의존성과 출력 의존성을 최소화해 스케줄링의 범위를 넓힐 수 있다. 루프 오버헤드를 최소화하려고 하다 보면 두 번째 저장하기 명령어들에서 변위 주소 지정을 사용한다면 배열 포인터 귀납 변수(`v26`과 `v34`)를 루프의 각 반복마다 한 번씩만 갱신되면 된다는 것을 알 수 있다. 블록 1의 끝부분에 추가된 새로운 명령어들은 n/2(본래 루프의 반복 수)가 짝수가 아닌 경우를 위한 것이다.

다시 블록 3의 분기를 지연 없이 스케줄링할 수 있다고 가정하면 (스케줄링 이전에) 펼친 루프의 전체 시간은 32주기나 본래 루프의 반복당 16주기다. 스케줄링 후 이 숫자는 본래 루프의 반복당 12주기로 감소한다. 불행히도 8주기(본래 반복당 4)는 여전히 일시 중지로 인해 손실된다.

예 15.30
소프트웨어 파이프라이닝의 결과

루프를 두 번이 아니라 세 번 펼치면(©심화학습에 있는 연습문제 15.20을 보자) (재스케줄링도 사용하면) 비용을 본래 반복당 11.3 주기까지 낮출 수 있지만 이는 그렇게 많은 개선은 아니다. 기본적인 문제를 ©(심화학습에 있는) 그림 15.14의 상단에 나타냈다. 본래 버전의 루프에서 두 개의 저장하기 명령어는 나눗셈 지연이 끝날 때까지 시작할 수 없다. 루프를 펼치면 내부 반복들의 명령어들을 섞을 수 있지만 마지막 나눗셈 이후에는 여전히 6주기의 "종료" 비용(4개의 지연 빈 공간과 두 개의 저장하기)이 필요하다.

`combinations` 서브루틴의 소프트웨어 파이프라이닝된 버전은 ©(심화학습에 있는) 그림 15.14의 하단의 도표와 ©(심화학습에 있는) 그림 15.15의 제어 흐름 그래프와 같다. 아이디어는 내부적인 구동이나 종료 비용 없이 몸체가 본래 루프의 연속적인 반복들 여럿으로 구성된 루프를 만드는 것이다. `combinations` 예에서 소프트웨어 파이프라이닝된 루프의 각 반복은 본래 루프의 반복 3개에 기여한다. 새로운 반복들(수직 막대 사이에 나타냄) 내부에서는 어떤 것도 나눗셈이 완료될 때까지 대기하지 않는다. 지연을 피하기 위해 몇 가지 방법으로 코드를 변경했다. 첫째로 새로운 루프의 각 반복이 본래 루프의 여러 반복에 기여하므로 새로운 루프를 최소한 한 번 실행할 수 있는 충분한 반복들이 존재하게 보장해야 한다(이는 새로운 블록 1에서 검사의 목적이다). 둘째로 "파이프라인"을 "준비"하고 "플러시(flush)"하기 위한 코드(첫 번째 반복의 처음 부분과 마지막 반복의 마지막 부분을 실행하기 위한 코드)를 루프 앞과 뒤에 추가했다. 루프 펼치기를 수행했을 때와 마찬가지로 새로운 "파이프라인 플러시" 코드에 기록된 레지스터에는 별도의 이름(이 경우에는 `t13`)을 사용했다. 셋째로 필요한 준비의 양을 최소화하기 위해 파이프라이닝된 루프의 첫 번째 루프가 "0번째" 본래 반복의 일부로서 `v26`과 `v34`를 "갱

신"할 수 있게 이들을 하나의 빈 공간 전에 초기화했다.

```
블록 1:
        …                  -- 그림 15.11, 블록 1의 코드
        v44 := v42 & 01
        if !v44 goto 블록 3
        -- else 블록 1a로 이동
블록 1a:
        *v26 := 1
        *v34 := 1
        v17 := 2
        v26 := v26 + 4
        v22 := v22 – 4
        v18 := v18 – 1
        goto 블록 3
```

블록 2:	스케줄링 후:
1. t19 := v13 × v18	t19 := v13 × v18
—	t18 := v18 – 1
—	t17 := v17 – 1
—	v18 := t18 – 1
—	—
2. t13 := v19 div v17	t13 := t19 div v17
—	v17 := t17 + 1
—	—
—	—
—	—
3. *v26 := t13	v19 := t13 × t18
4. *v34 := t13	*v26 := t13
5. t17 := v17 + 1	*v34 := t13
6. v26 := v26 + 8	v26 := v26 + 8
7. v34 := v34 – 8	v34 := v34 – 8
8. t18 := v18 – 1	v13 := v19 div t17
9. v19 := t13 × t18	—
—	—
—	—
—	—
—	*(v36+4) := v13
10. v13 := t19 div t17	*(v34+4) := v13
—	
—	
—	
—	
11. *(v26 – 4) := v13	
12. *(v34 + 4) := v13	
13. v17 := t17 + 1	
14. v18 := t18 – 1	
-- 블록 3으로 이동	

```
블록 3:                                (동일)
        v43 := v17 ≤ v42
        if v43 goto 블록 4
        -- else 블록 4로 이동
```

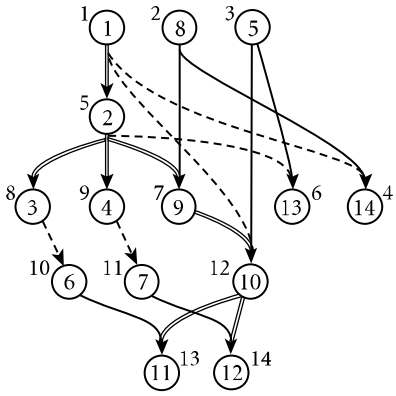


그림 15.13 | 루프 몸체의 반복 2개를 펼친 후 combinations 서브루틴의 블록 2에 대한 의존성 DAG. 명령어 스케줄링 전(왼쪽)과 후(오른쪽)의 전체 루프에 대한 선형화된 의사코드도 나타냈다. 블록 1의 끝부분에 추가된 새로운 명령어들은 본래 루프의 반복 수가 2의 배수가 아닌 경우를 위한 것이다.

끝으로 준비 코드가 블록 1의 끝부분에서 v13을 유효하지 않은 상태로 남겨두기 때문에(이 사실을 알아내기 위해 가상 레지스터에 대한 유효 변수 분석을 사용했다) 블록 1에서 v13의 초기화를 제거했다.

루프의 본래 버전과 파이프라이닝된 버전 모두가 반복 간의 경계를 걸쳐 5개의 비상수 값을 전달하지만 이 중 하나는 서로 다르다. 본래 루프가 나눗셈 결과를 레지스터 v13의 다음 곱셈으로 전달하는 반면 파이프라이닝된 루프는 곱셈 결과를 레지스터 v19의 나눗셈으로 전달한다. 좀 더 복잡한 루프에서는 파이프라이닝된 루프의 반복 간 경계에 걸쳐 두 가지 또는 심지어 세 가지 버전의 (본래 루프의 두 개 이상의 반복에 대응되는) 단일 레지스터를 전달해야 할 수 있다. 추가적인 값을 저장하기 위해 (combinations 예의 펼친 버전에 있는 새로운 t13과 t 레지스터들과 유사한) 새로운 가상 레지스터를 도입해야 한다. 이러한 경우 소프트웨어 파이프라이닝은 레지스터 압력을 증가시키는 부가 작용을 가지게 된다.

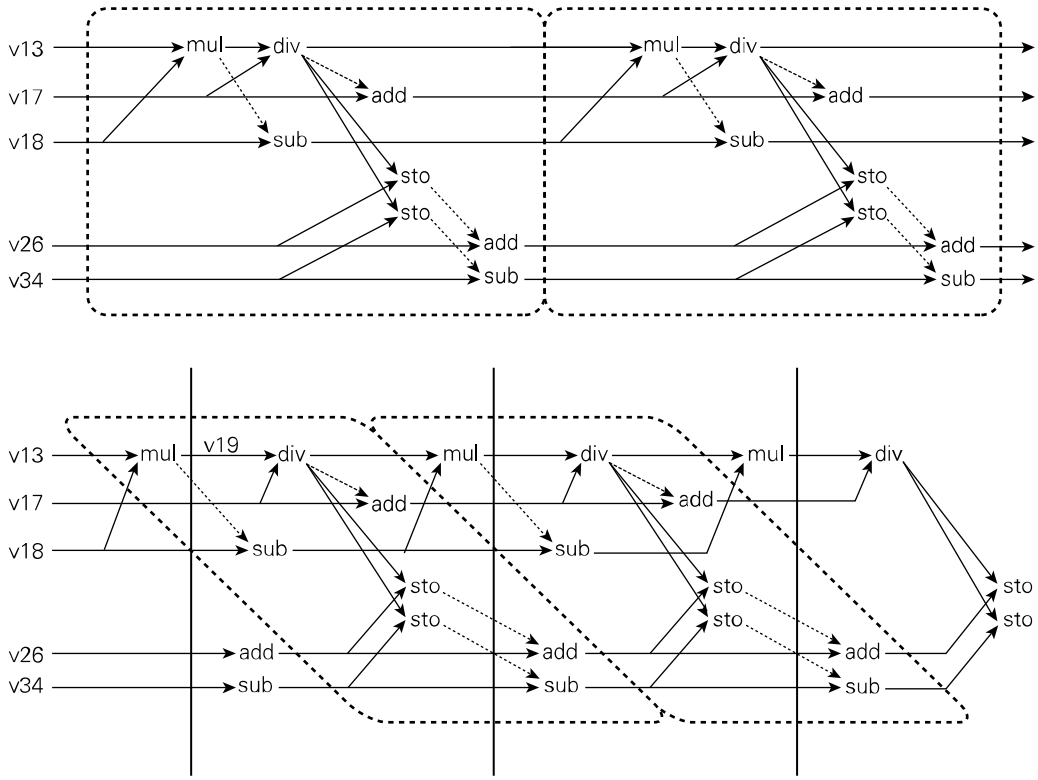


그림 15.14 | 소프트웨어 파이프라이닝. 상단 그림은 본래(파이프라이닝하지 않은) 루프의 실행을 나타낸다. 하단 그림에서는 본래 루프의 각 반복이 파이프라이닝된 루프의 반복 세 개에 걸쳐 펼쳐져 있다. 본래 루프의 반복들은 점선 상자로 둘러쌓았으며 파이프라이닝된 루프의 반복들은 수직 막대로 구분했다. 하단 그림에는 첫 번째 반복 이전에 파이프라인을 준비하는 코드와 마지막 반복 후에 파이프라인을 플러시하는 코드도 보였다.

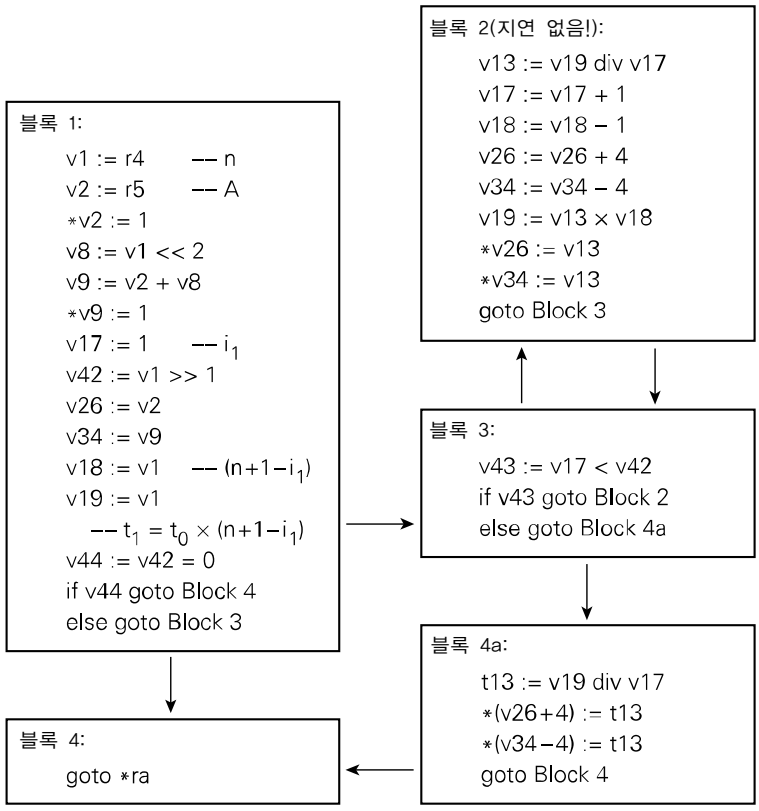


그림 15.15 | 소프트웨어 파이프라이닝 후 combinations 서브루틴의 제어 흐름 그래프. 블록 1 끝부분의 추가적인 코드와 검사, 블록 3에 있는 검사의 변경(≤가 아닌 〈), 새로운 블록(4a)은 파이프라인을 지원하고 초기 반복의 시작부분을 준비하고 마지막 반복의 끝부분을 플러시하기에 충분한 반복이 존재함을 보장해준다. 블록 1 주석의 변수명에 달린 접미어는 루프 반복을 나타낸다. 즉, t1은 루프의 첫 번째 반복에서의 t 값이며 t0는 파이프라인을 준비하기 위한 "0번째" 값이다.

`combinations` 서브루틴의 파이파라이닝된 버전의 루프에서 각 명령어는 지연 없이 진행될 수 있다. 반복당 전체 주기 수는 10으로 준다. 루프 펼치기와 소프트웨어 파이프라이닝을 함께 사용하면 이보다 더 나은 결과를 얻을 수도 있다. 예를 들어 각 반복에 두 개의 곱셈-나눗셈 쌍(수반하는 명령어들과 함께 본래 루프의 명령어 3개가 아닌 4개로부터 이끌어 낸)을 내장시킴으로써 배열 포인터 갱신과 종료 조건 검사 주기를 2배로 늘려 본래 루프의 반복당 전체 주기 수를 8주기로 줄일 수 있다(©심화학습에 있는 연습문제 15.21을 보자).

요약하면 루프 펼치기는 루프 오버헤드를 줄이며 명령어 스케줄링의 기회를 증가시킬 수 있다. 소프트웨어 파이프라이닝은 스케줄링을 용이하게 하는 작업은 좀 더 잘 수행하지만 루프 오버헤드는 해결하지 않는다. 합리적인 코드 개선 전략은 반복당 오버헤드가 전체 작업의 특정 임계치 아래로 떨어질 때까지는 루프를 펼친 후 스케줄링 지연을 제거해야 하는 경우 소프트웨어 파이프라이닝을 사용하는 것이다.

【15.7.2】 루프 재정렬

이제까지 살펴본 코드 개선 기술들은 주로 두 가지 목적(중복되거나 불필요한 명령어를 제거하는 것과 파이프라이닝된 기계상에서 일시 중지를 최소화하는 것)을 위한 것이었다. 최근에는 다른 두 가지 목표가 점차적으로 중요해지고 있다. 첫째로 프로세서 속도 개선이 메모리 지연 개선을 계속 능가하면서 캐시 부적중을 최소화하는 것이 점차 더 중요해졌다. 둘째로 병렬 기계를 위해 코드에서 병행으로 실행할 수 있는 부분을 식별하는 것이 중요해졌다. 다른 최적화와 마찬가지로 루프 동작을 변경함으로써 가장 큰 개선을 달성할 수 있다. 여기서는 몇 가지 쟁점을 간단히 다룬다. 좀 더 자세한 내용에 대한 참고문헌은 15장의 끝부분에서 찾아볼 수 있다.

캐시 최적화

예 15.31
루프 교환

캐시 최적화의 가장 간단한 예는 다차원 행렬(배열)을 탐색하는 코드에서 볼 수 있을 것이다.

```
for i := 1 to n
    for j := 1 to n
       A[i, j] := 0
```

A가 행 우선 순서로 배열되어 있으며 각 캐시 라인이 A의 원소를 m개 포함한다면 이 코드에서는 n^2/m번의 캐시 부적중이 발생한다. 한편 A가 열 우선 순서로 배열되어 있으며 캐시가 A의 라인 n개를 저장하기에 너무 작다면 메모리로부터 전체 배열을 m번 인출(fetch)하면서 n^2번의 캐시 부적중이 발생하게 된다. 이러한 차이는 성능에 엄청난 영향을 미친다. 루프 재정렬을 수행하는 컴파일러는 중첩된 루프를 서로 교환하는 방법으로 이 코드를 개선할 수 있다.

```
for j := 1 to n
    for i := 1 to n
       A[i, j] := 0
```

예 15.32
루프 타일화 (블록화)

좀 더 복잡한 예에서는 루프 교환이 배열 하나에 대한 참조 집약성은 개선하지만 나머지 배열에 대한 참조 집약성은 떨어뜨릴 수 있다. 2차원 행렬을 전치하는 다음 코드를 보자.

```
for j := 1 to n
    for i := 1 to n
       A[i, j] := B[j, i]
```

A와 B가 메모리에 동일한 방식으로 배열되어 있다면 하나는 캐시 라인을 따라 접속되겠지만 다른 하나는 캐시 라인과 엇갈리게 접속될 것이다. 이 경우 루프를 타일화하거

나 블록화함으로써 참조 집약성을 개선할 수 있다.

```
for it := 1 to n by b
    for jt := 1 to n by b
       for i := it to min(it + b − 1, n)
          for j := jt to min(jt + b − 1, n)
             A[i, j] := B[j, i]
```

여기서 min 계산은 n이 b로 나눠떨어지지 않을 가능성을 대비한 것으로 n이 b의 배수라는 것을 안다면 생략할 수 있다. 가장 안쪽 루프의 코드를 중복하고자 한다면 각 루프의 최종 반복에 대해 다른 코드를 생성할 수도 있다(◎심화학습에 있는 연습문제 15.24).

새 코드는 ◎(심화학습에 있는) 그림 15.16과 같이 A와 B의 b × b 블록들에 대해 하나는 열 우선 순서로, 다른 하나는 행 우선 순서로 반복한다. 캐시가 b × b 블록 2개의 자료를 동시에 유지할 수 있는 m의 배수로 b를 선택한다면 메모리로부터 정확히 한 번에 모든 것을 인출하면서 A와 B 모두에서 m개의 배열 원소당 한 번의 캐시 부적중만이 발생한다.[4] 타일화는 많은 다차원 배열 알고리즘에서 유용하다. ◎(심화학습에 있는) 연습문제 15.22에서는 행렬 곱셈을 살펴본다.

캐시 집약성을 개선할 수 있는 나머지 두 개의 변환은 루프 분배(분열[fission]이나 쪼개기[splitting]라고도 함)와 그 반대인 루프 융합(재밍이라고도 함)이다. 분배는 하나의 루프를 이 루프의 문장을 일부 포함하는 여러 개의 루프로 쪼개는 것이다. 융합은 별개의 루프들을 모아 합치는 것이다.

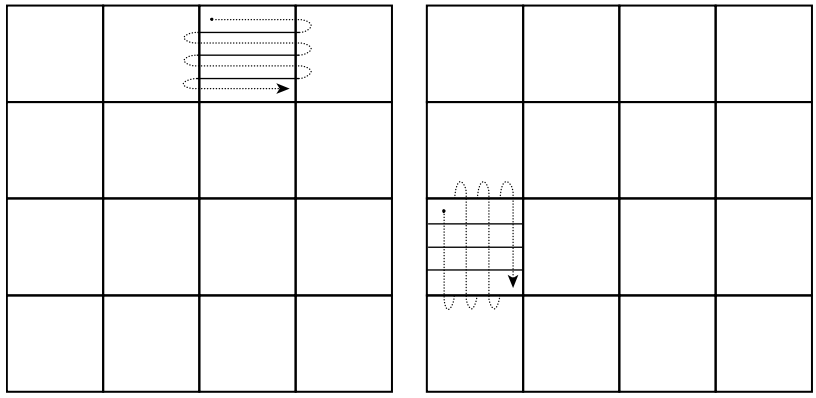


그림 15.16 | 행렬 연산의 타일화(블록화). A의 타일 하나와 B의 타일 하나가 캐시에 동시에 올라갈 수 있는 한 m번의 접근 중 단 한 번의 접근만이 한 번의 캐시 부적중을 일으킨다(m은 캐시 라인당 원소의 개수다).

주4. B가 기록만 되고 읽히지는 않지만 하드웨어는 라인으로의 첫 번째 기록 시점에 B의 각 라인을 메모리에서 인출하므로 수정된 단일 원소는 캐시 안에서 갱신될 수 있다. 하드웨어는 라인을 메모리에 다시 기록하기 전까지는 전체 라인이 수정된다는 것을 알지 못한다.

예 15.33

루프 분배

예를 들어 한 쌍의 배열을 재구조화하는 다음의 코드를 보자.

```
for i := 0 to n—S1
    A[i] := B[M[i]];
    C[i] := D[M[i]];
```

여기서 M은 B나 D에 있는 위치로부터 A나 C에 있는 위치로의 매핑을 정의한다. B나 D 모두는 아니지만 둘 중 하나가 캐시에 한 번에 올라갈 수 있다면 루프 분배를 통해 더 빠른 코드를 얻을 수 있다.

```
for i := 1 to n
    A[i] := B[M[i]];
for i := 1 to n
    C[i] := D[M[i]];
```

예 15.34

루프 융합

한편 다음의 코드에서는 분리된 루프가 집약성을 떨어뜨릴 수 있다.

```
for i := 1 to n
    A[i] := A[i] + c
for i := 1 to n
    if A[i] < 0 then A[i] := 0
```

A가 한번에 캐시에 올라가기에 너무 크다면 위의 두 루프는 전체 배열을 메모리로부터 두 번 인출하게 된다. 그러나 두 루프를 융합하면 A를 한 번만 인출해도 된다.

```
for i := 1 to n
    A[i] := A[i] + c
    if A[i] < 0 then A[i] := 0
```

두 루프가 동일한 경계를 가지지 않는다 하더라도 귀납 변수를 변환하거나 한 루프에서 상수 번의 반복을 벗겨낸다면(peel) 여전히 두 루프를 융합할 수 있다.

예 15.35

완벽한 루프 중첩 얻기

루프 분배는 "불완전한" 루프 중첩을 "완벽한" 루프 중첩으로 변환함으로써 다른 변환(예를 들어 루프 교환)을 용이하게 할 수 있다.

```
for i := 1 to n
    A[i] := A[i] + c
for j := 1 to n
    B[i, j] := B[i, j] × A[i]
```

이 중첩은 바깥쪽 루프가 안쪽 루프 외의 것을 포함하기 때문에 완벽하지 않다고 한다. 분배는 가장 바깥쪽 루프를 두 개 생성한다.

```
for i := 1 to n
    A[i] := A[i] + c
```

```
for i := 1 to n
    for j := 1 to n
       B[i, j] := B[i, j] × A[i]
```

이제 중첩된 루프는 완벽하며 원하는 경우 교환할 수 있다.

루프 최적화에 대한 이전 설명과의 연결선상에서 루프 분배는 레지스터 압력을 줄일 수 있으며 루프 융합은 루프 오버헤드를 줄일 수 있다는 것을 알아두자.

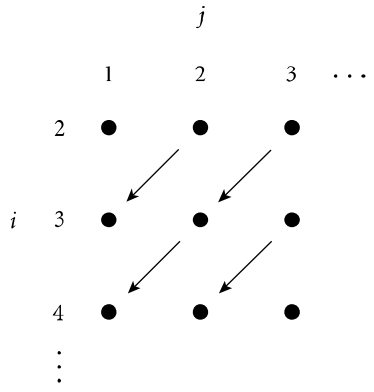
루프 의존성

예 15.36
루프 운반 의존성

루프를 재배열할 때는 모드 자료 의존성을 따르게 매우 주의해야 한다. 특히 반복이 일어나는 순서를 강제하는 루프 운반 의존성이라는 것에 주의해야 한다. 예를 들어 다음의 코드를 보자.

```
for i := 2 to n
    for j := 1 to n-S1
       A[i, j] := A[i, j] - A[i-i1, j+1]
```

여기서 반복 (i, j)에서 A[i, j]의 계산은 반복 (i−1, j+1)에서 계산된 A[i−1, j+1]에 의존적이다. 가끔 반복 공간 그림을 통해 이러한 의존성을 나타낼 수 있다.



이 그림에서 i와 j 차원은 배열 첨자가 아니라 루프 색인을 나타낸다. 호는 루프 운반 흐름 의존성을 나타낸다.

이 코드의 i와 j 루프를 서로 교환하고자 하면(예를 들어 캐시 집약성을 개선하기 위해) 의존성 때문에 교환이 불가능함을 알 수 있다. 결국 A[i−1, j+1]을 기록하기 전에 A[i, j]를 기록하려고 하게 된다.

루프 운반 의존성을 분석하기 위해 고성능의 최적화 컴파일러들은 다른 배열 참조의 첨자 수식이 동일한 값으로 값 계산되게 하는 색인 값들의 집합을 표현할 수 있는

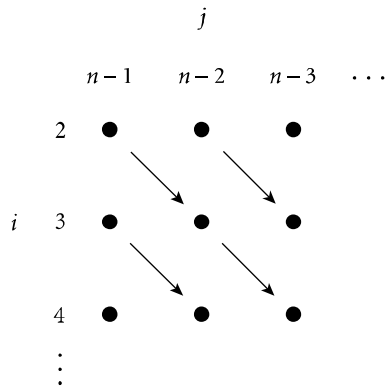
기호 수학을 사용한다. 이러한 분석의 정교함은 컴파일러마다 다소 차이가 있다. 대부분의 컴파일러가 루프 색인의 선형 조합을 처리할 수 있다. 물론 일반적 수식의 동치는 계산 불가능하기 때문에 어떤 컴파일러도 모든 수식을 처리하지는 못한다. 첨자를 완전히 특징지을 수 없다면 컴파일러는 보수적으로 최악의 상황을 가정하고 안전성을 증명할 수 없는 변환을 모두 제거해야 한다.

예 15.37
루프 반전과 교환

많은 경우에 하고자 하는 변환을 미리 배제시키는 의존성을 완벽히 특징지을 수 있는 루프는 그러한 의존성을 제거하는 방식으로 수정할 수 있다. ©(심화학습에 있는) 예 15.36에서는 의존성을 해치지 않고 j 루프를 뒤집을 수 있다.

```
for i := 2 to n
    for j := n-S1 to 1 by -1
       A[i, j] := A[i, j] - A[i-i1, j+1]
```

이러한 변경은 반복 공간을 변형시킨다.



이제는 루프를 안전하게 교환할 수 있다.

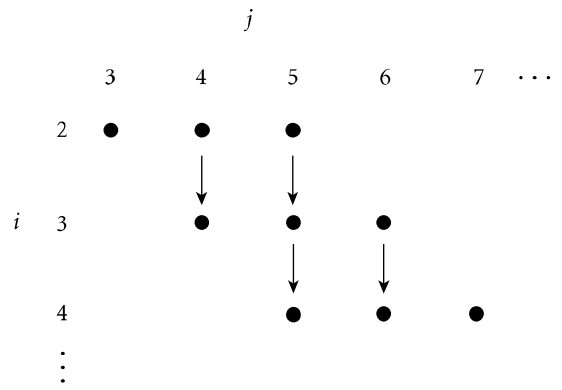
```
for j := n-S1 to 1 by -1
    for i := 2 to n
       A[i, j] := A[i, j] - A[i-i1, j+1]
```

예 15.38
루프 기울이기

때때로 의존성을 제거하는 다른 변형으로 루프 기울이기가 있다. 간단히 말해서 루프 기울이기는 바깥쪽 루프 색인을 안쪽 루프 색인에 추가한 후 적절한 첨자로부터 뺌으로써 정사각형의 반복 공간을 평행사변형으로 변형시킨다.

```
for i := 2 to n
    for j := i+1 to i+n-1
       A[i, j-ii] := A[i, j-i] - A[i-i1, j+1-i]
```

잠깐만 생각해보면 이 코드가 이전과 완전히 동일한 순서로 완전히 동일한 원소들을 접근한다는 것을 알 수 있다. 그러나 이 코드의 반복 공간은 아래와 같다.



이제 루프를 안전하게 교환할 수 있다. 변환은 반복 공간의 경사면에 대한 지원의 필요성 때문에 복잡하다. min과 max 함수를 사용하지 않기 위해 각자의 루프 중첩을 가지는 두 개의 삼각형으로 반복 공간을 나눌 수 있다.

```
for j := 3 to n+1
    for i := 2 to j−1
       A[i, j−ii] := A[i, j−i] − A[i−i1, j+1−i]
for j := n+2 to 2 n−1
    for i := j−n+1 to n
       A[i, j−ii] := A[i, j−i] − A[i−i1, j+1−i]
```

기울이기를 하면 j 루프를 뒤집는 것보다 코드가 복잡해지긴 하지만 루프 반전을 사용할 수 없는 의존성이 존재하는 경우에 사용할 수 있다.

루프 분배 등과 같은 몇 가지 다른 루프 변환도 루프 운반 의존성을 제거하는 데 사용할 수 있으며 캐시 집약성을 개선하거나 (바로 아래서 다룰 것과 같이) 벡터 기계나 다중 프로세서상의 코드의 병렬 실행을 가능하게 해주는 기술을 적용할 수 있게 해준다. 물론 어떤 집합의 변환도 의존성을 모두 제거하지는 못한다. 일부 코드는 절대 개선할 수 없다.

병렬성

(적어도 비재귀적 프로그램에서) 루프 반복은 병렬로 실행할 수 있는 연산을 쉽게 구할 수 있는 부분이다. 이상적으로는 루프 운반 의존성이 없는 독립적인 루프 반복을 찾아야 한다(어떤 경우 반복에 의존성이 있더라도 연산을 적절히 동기화함으로써 반복을 병렬로 실행할 수도 있다). 예 12.9와 12.3.6절에서는 프로그래머가 병렬 실행을 명시할 수 있는 루프 구성소를 살펴봤다. 컴파일러는 이러한 특수 구성소가 없는 언어에서도 가끔 최대한 적은 루프 운반 의존성을 가지는 루프를 식별(또는 생성)함으로써 코드를 병렬화할 수 있다. 앞서 다룬 변환들은 이때 매우 중요한 도구로 쓰인다.

예 15.39

세밀하지 않은 (coarse-grain) 병렬화

컴파일러는 병렬화 가능한 루프가 주어지면 좋은 성능을 보장하기 위해 몇 가지 기타 사항을 고려해야 한다. 이 중 가장 중요한 것의 하나가 병렬성의 세분성(granularity)이다. 매우 간단한 예로 2차원 배열을 "전부 0으로 채우는" 문제를 생각해보자. 이 배열에서 색인은 0에서 n-1까지며 행 우선 순서로 배열되어 있다.

```
for i := 0 to n-S1
    for j := 0 to n-1
       A[i, j] := 0
```

여러 개의 범용 프로세서를 포함하는 기계상에서는 다음과 같이 바깥쪽 루프를 병렬화할 수 있다.

```
-- 프로세서 pid상에서:
for i := (n/p   pid) to (n/p   (pid + 1) -1)
    for j := 1 to n
       A[i, j] := 0
```

이 코드에서는 각 프로세서에 초기화할 행들을 분배했다. 여기서는 프로세서들이 0에서 p-1까지의 번호를 부여 받으며 n이 p로 나눠떨어진다고 가정했다.

예 15.40

스트립 마이닝(strip mining)

벡터 기계에서의 병렬화 방법은 크게 다르다. 벡터 기계는 v개의 원소를 가지는 벡터 레지스터 집합과 벡터 자료를 불러오고 저장하고 계산하기 위한 명령어들을 가진다. 벡터 명령어는 깊게 파이프라이닝되어 벡터 기계가 높은 수준의 세밀한(fine-grain) 병렬성을 활용할 수 있게 해준다. 컴파일러는 하드웨어 특성을 이용하기 위해 안쪽 루프를 병렬화한다.

```
for i := 0 to n-S1
    for j := 0 to n/v
       A[i, j:j+v-1] := 0        -- 벡터 연산
```

여기서 `A[i, j:j+v-1]`은 A에서 v개의 원소로 된 단편(slice)을 나타낸다. 상수 v는 벡터 레지스터의 길이(역시 n은 이 길이로 나눠떨어진다고 가정한다)로 설정해야 한다. 좀 더 긴 루프에서 v개의 원소로 된 연산을 추출하는 코드 변환을 **스트립 마이닝**(strip mining)이라고 한다. 이는 본질적으로 타일화의 1차원 형태다.

컴파일러 병렬화에서 중요한 기타 사항으로는 통신과 부하 균형이 있다. 참조 집약성이 단일 프로세서상에서 캐시와 주 메모리 사이의 통신을 줄이는 것과 마찬가지로 병렬 프로그램들의 집약성은 프로세서 간 통신과 프로세서와 메모리 사이의 통신을 줄인다. 단일 프로세서에서 캐시 미스를 줄이기 위해 사용한 것과 비슷한 최적화 기법들은 다중 프로세서상의 통신 트래픽을 줄이는 데도 사용할 수 있다.

부하 균형은 병렬 기계에서 프로세서 간의 일을 분배하는 것을 말한다. 프로그램의 작업을 16개의 프로세서로 분배하는 경우 각 프로세서가 각자의 작업을 수행하는 데

동일한 시간을 소요할 때만 16배에 가까운 속도 증가를 얻을 수 있다. 실수로 15개의 프로세서에는 각기 5%의 작업만을 할당하고 나머지 25%의 작업을 16번째 프로세서에 할당하면 4배 이상의 속도 증가를 얻기 힘들다. 간단한 루프의 경우에는 컴파일 시점에 프로세서들에게 작업을 분배하기 충분한 정확도로 성능을 예측하는 것이 가끔 가능하다. 반복들이 서로 다른 양의 작업을 수행하거나 서로 다른 캐시 동작을 보이는 좀 더 복잡한 루프의 경우에는 실행 시간에 작업을 분배하는 자가 스케줄링된 코드를 생성하는 것이 더 나은 경우가 많다. 가장 간단한 형태의 자가 스케줄링은 12.2절에서 설명한 것과 같이 "태스크 가방"을 생성한다. 각 태스크는 루프 반복의 집합으로 구성된다. 이러한 태스크들의 수는 프로세서의 수보다 훨씬 더 큰 값으로 결정한다. 프로세서는 주어진 태스크를 종료하면 다른 태스크를 얻기 위해 태스크 가방으로 돌아간다.

15.8 레지스터 할당

전역 최적화를 하지 않는 간단한 컴파일러에서 레지스터 할당은 모든 기본 블록에서 독립적으로 수행될 수 있다. 자주 접근하는 변수를 많은 블록의 끝부분에서 메모리에 저장하고 다시 다른 블록에서 읽어오는 분명한 비효율성을 피하기 위해 간단한 컴파일러는 보통 그런 변수를 식별하고 이에 서브루틴보다 긴 유효 범위를 가지게 레지스터를 할당하기 위해 휴리스틱을 적용한다. 전용 레지스터를 가질만한 분명한 후보로는 루프 색인, 파스칼 계열어에 있는 `with`문의 묵시적 포인터(7.3.3절), 스칼라 지역변수와 매개변수 등이 있다.

1970년대 초반 레지스터 할당은 그래프 색 지정 NP-난해 문제와 동치라는 것이 알려졌다. 1980년대 초반 샤틴 등의 업적[CAC+81]을 따라 그래프 색 지정의 휴리스틱(비최적) 구현이 적극적으로 최적화를 수행하는 컴파일러에서 레지스터 할당에 대한 일반적인 접근 방법이 되었다. 여기서는 기본적인 아이디어를 설명한다. 좀 더 자세한 내용은 쿠퍼와 토르크존의 교과서에 나와 있다[CT04, 13장].

예 15.41
가상 레지스터의 유효 범위

첫 번째 단계는 동시에 유효한 값들을 포함하기 때문에 구조 레지스터를 공유할 수 없는 가상 레지스터들을 식별하는 것이다. 이 단계를 수행하기 위해서는 도달 정의 자료 흐름 분석(©심화학습에 있는 15.5.1절)을 사용한다. `combinations` 서브루틴의 소프트웨어 파이프라이닝된 버전(©심화학습에 있는 그림 15.15)에 대해 ©(심화학습에 있는) 그림 15.17과 같이 가상 레지스터들의 유효 범위(live range)를 나타낼 수 있다. `v19`의 유효 범위가 블록 2의 끝부분에 있는 후방 분기에 걸쳐있음에 주목하자. 비록 인쇄상으로는 끊어져 있지만 시간상으로 보면 한 덩어리의 유효 범위다.

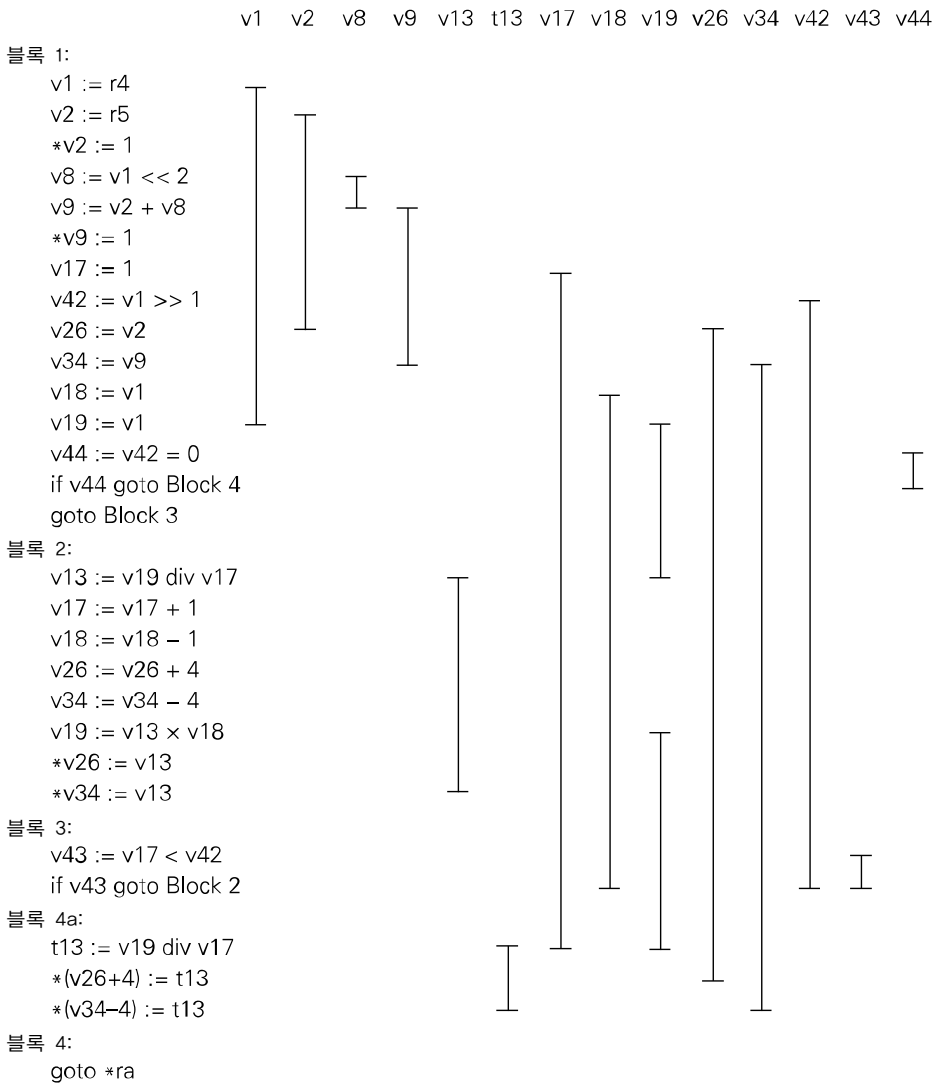


그림 15.17 | combinations 서브루틴의 소프트웨어 파이프라이닝된 버전(©심화학습에 있는 그림 15.15)에 대한 가상 레지스터들의 유효 범위

예 15.42
레지스터 색 지정

이러한 유효 범위가 주어지면 레지스터 간섭 그래프를 구성한다. 이 그래프의 노드는 가상 레지스터를 나타낸다. 레지스터 `vi`와 `vj`는 동시에 유효한 경우 호로 연결된다. ©(심화학습에 있는) 그림 15.17에 대응되는 간섭 그래프는 ©(심화학습에 있는) 그림 15.18과 같다. 가상 레지스터를 가능한 한 가장 적은 수의 구조 레지스터로 매핑하는 문제는 이제 이 그래프의 최소 색 지정을 찾는 것이 되었다. 즉, 어떤 호도 동일한 색의 두 노드를 연결하지 않는 노드로의 "색" 할당 작업이 되었다.

`combinations` 예를 잘 살펴보면 몇 가지 최적해를 찾을 수 있다. 그림 중앙에 있는

6개의 레지스터는 군집(완전히 연결된 하위 그래프)을 구성한다. 그러므로 각기 별도의 구조 레지스터에 매핑해야 한다. 또 한 레지스터가 코드의 어딘가에서 다른 레지스터로 복사되지만 이 두 레지스터가 동시에 유효한 때는 절대 없는 경우도 세 가지가 있다(레지스터 v1와 v9, v2와 v26, v9와 v34). 이러한 경우에는 공통 구조 레지스터를 사용해서 복사 명령어를 제거할 수 있다. 이러한 최적화를 **유효 범위 합병**이라고 한다. 레지스터 v13, v43, v44는 군집의 모든 멤버와 연결되어 있지만 서로는 연결되어 있지 않으므로 7번째 구조 레지스터를 공유할 수 있다. 레지스터 v8은 v1, v2, v9과 연결되어 있지만 그밖에는 어떤 것과도 연결되어 있지 않다. 여기서는 임의로 v8과 v13이 우측의 세 레지스터와 공유하는 것으로 선택했다.

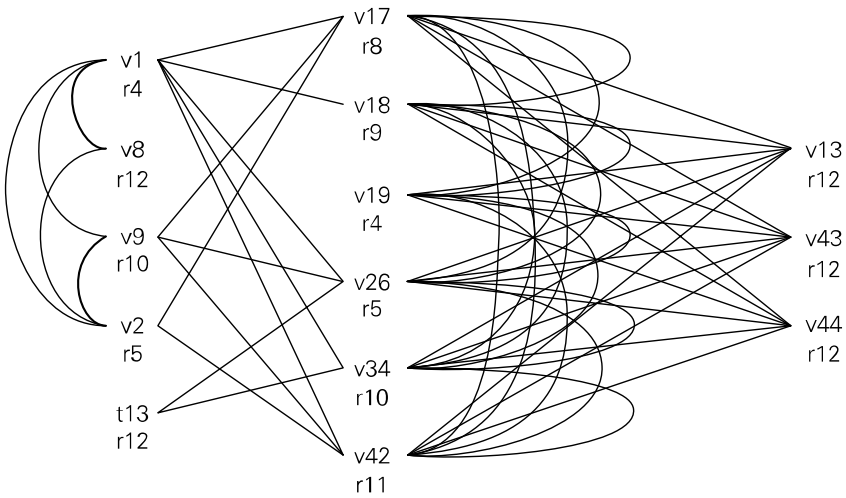


그림 15.18 | combinations 서브루틴의 소프트웨어 파이프라이닝된 버전에 대한 레지스터 간섭 그래프. 구조 레지스터 이름을 사용해서 가능한 7색 색 지정 중 하나를 나타냈다.

예 15.43

최적화된 combinations 서브루틴

combinations 서브루틴의 최종 코드는 ◎(심화학습에 있는) 그림 15.19와 같다. v1/v19와 v2/v26은 초기값이 전달된 레지스터인 r4와 r5에 할당했다. 이 서브루틴은 리프기 때문에 이러한 레지스터들이 다른 할당에 필요한 경우는 없다. MIPS의 관례(5.4.4절)를 따라 레지스터 r8~r12는 부가적인 임시 레지스터로 사용했다.

```
Block 1:
    *r5 := 1
    r12 := r4 << 2
    r10 := r5 + r12
    *r10 := 1
    r8 := 1
```

그림 15.19 | 구조 레지스터 할당과 불필요한 복사 명령어 제거를 마친 combinations 서브루틴의 최종 코드(이어짐)

```
        r11 := r4 >> 1
        r9 := r4
        r12 := r11 = 0
        if r12 goto Block 4
        goto Block 3
    Block 2:
        r12 := r4 div r8
        r8 := r8 + 1
        r9 := r9 —1
        r5 := r5 + 4
        r10 := r10 —4
        r4 := r12   r9
        *r5 := r12
        *r10 := r12
    Block 3:
        r12 := r8 < r11
        if r12 goto Block 2
    Block 4a:
        r12 := r4 div r8
        *(r5+4) := r12
        *(r10—14) := r12
    Block 4:
        goto *ra
```

그림 15.19 | 구조 레지스터 할당과 불필요한 복사 명령어 제거를 마친 combinations 서브루틴의 최종 코드

이제까지 두 가지 중요한 쟁점을 간략히 알아봤다. 우선 거의 모든 실제 기계에서 구조 레지스터는 균일하지 않다. 정수 레지스터는 부동점 실수 연산에 사용할 수 없다. 서브루틴 호출에 걸쳐 필요한 값을 가지는 변수에는 호출자-저장 레지스터를 사용하면 안 된다. 특수 명령어(예를 들어 CISC 기계의 바이트 검색)가 덮어쓴 레지스터는 이러한 명령어들에 걸쳐 필요한 값을 저장하는 데 사용할 수 없다. 이런 종류의 제약을 처리하기 위해서는 주로 가상 레지스터와 구조 레지스터에 대한 노드를 모두 포함하게 레지스터 간섭 그래프를 확장한다. 그 후 호는 각 가상 레지스터에서 이것이 매핑되어야 하는 구조 레지스터를 연결하게 그린다. 구조 레지스터는 모두 별도의 색을 가지게 강제하기 위해서 서로 연결한다. 이렇게 구한 그래프를 색 지정한 후 가상 레지스터를 동일한 색의 구조 레지스터에 할당한다.

여기서는 다루지 않은 두 번째 쟁점은 필요한 구조 레지스터가 충분치 않을 때 발생한

다. 이 경우에는 간섭 그래프를 색 지정하는 것이 불가능하다. 컴파일러는 (여기서 다루지 않은) 다양한 휴리스틱을 사용해서 유효 범위를 둘 이상의 하위 범위로 쪼갤 수 있는 가상 레지스터를 선택한다. 이렇게 하면 하위 범위의 끝부분에서 유효한 값을 메모리로 내려 보내고(저장하고) 다음 하위 범위의 시작부분에서 다시 불러올 수 있다. 또는 이 값을 생성했던 계산을 반복함으로써 재구체화(rematerialize)할 수도 있다(필요한 피연산자들은 여전히 이용 가능하다고 가정한다). 어느 것의 비용이 더 낮은지는 불러오기와 저장하기의 비용과 계산 생성 복잡도에 따라 달라진다.

충분한 수의 범위 쪼개기를 사용하면 세 가지 색만으로도 임의의 그래프를 색 지정할 수 있다는 것은 쉽게 증명할 수 있다. 비결은 메모리로 내려 보내기와 재구체화의 비용을 낮게 유지하는 쪼개기 집합을 찾는 것이다. 일단 레지스터 할당이 끝나면 ©(심화학습에 있는) 15.1절과 15.6절에서 설명했던 대로 새로 생성된 불러오기 지연을 채우기 위해 명령어 스케줄링을 반복한다.

✓ 확인문제

28. 루프 펼치기와 소프트웨어 파이프라이닝의 차이는 무엇인가? 소프트웨어 파이프라이닝이 레지스터 압력을 증가시킬 수 있는 이유를 설명하라.

29. 루프 교환과 루프 타일화(블록화)의 목적은 무엇인가?

30. 루프 분배와 루프 융합에서 얻을 수 있는 잠재적 이익은 무엇인가? 루프 벗겨내기(peeling)란 무엇인가?

31. 루프가 완벽하게 중첩되었다는 것은 어떤 의미인가? 완벽한 루프 중첩이 중요한 이유는 무엇인가?

32. 루프 운반 의존성이란 무엇인가? 때때로 이러한 의존성을 제거할 수 있는 세 가지 루프 변환을 설명하라.

33. 다중 프로세서를 위한 병렬화 전략과 벡터 기계를 위한 병렬화 전략의 기본적인 차이점을 기술하라.

34. 자가 스케줄링이란 무엇이며 언제 유용한가?

35. 레지스터의 유효 범위란 무엇인가? 유효 범위가 명령어들의 연속 범위가 아닐 수 있는 이유는 무엇인가?

36. 레지스터 간섭 그래프란 무엇이며 그 의미는 무엇인가? 컴파일러 상품들은 레지스터 할당을 왜 (정밀한 해결책 대신) 휴리스틱에 의존하는가?

37. 마이크로프로세서의 구조 레지스터들을 레지스터 할당을 위해 균일하게 다루는 것이 불가능할 수 있는 이유를 세 가지 나열하라.

15.9 정리

15장에서는 코드 개선("최적화")이라는 주제를 살펴봤다. 국소적 최적화, 지역과 전역(서브루틴 내) 중복 제거(상수 폴딩, 상수 전파, 복사 전파, 공통 하위 수식 제거), 루프 개선(불변식 이동, 강도 축소나 귀납 변수의 제거, 펼치기와 소프트웨어 파이프라이닝, 캐시 개선이나 병렬화를 위한 재정렬), 명령어 스케줄링, 레지스터 할당과 같이 가장 중요한 최적화 기법들을 대부분 살펴봤다. 이 밖에 열거하기에도 벅찰 만큼 많은 다른 기술들을 코드 개선 분야나 실제 쓰이는 상품에서 찾아볼 수 있다.

코드 개선을 용이하게 하기 위해 (명령어 스케줄링을 위한) 의존성 DAG, (값 번호 지정을 통한 전역 공통 하위 수식 제거를 포함한 많은 목적을 위한) 정적 단일 대입문(SSA) 형태, (구조 레지스터 할당을 위한) 레지스터 간섭 그래프를 포함한 몇 가지 새로운 자료 구조와 프로그램 표현을 도입했다. 많은 전역 최적화에 대해서는 자료 흐름 분석을 사용했다. 명확히 말해서 자료 흐름 분석을 사용해 (전역 공통 하위 수식 제거를 위해서) 이용 가능한 수식을 식별했고 (불필요한 저장하기를 제거하기 위해서) 유효한 변수를 식별했으며 (루프 불변식을 식별하기 위해) 도달 정의를 계산했다(도달 정의 계산은 가상 레지스터의 유효 범위를 찾는 데에도 유용하다). 전역 상수 전파, 복사 전파, SSA 형태로의 변환, 기타 다른 목적의 주최 역할로도 자료 흐름 분석을 사용할 수 있음을 살펴봤다.

컴파일러 작성자와 사용자 모두에게 분명한 질문은 가능한 다수의 코드 개선 기술 중 가장 "본전을 뽑을 수 있을 만한 가치"가 있는 기술은 무엇이냐는 것이다. 근래의 기계에서 명령어 스케줄링과 레지스터 할당은 확실히 가치가 있는 명단에 들어간다. 기본 블록 수준 스케줄링과 중복된 불러오기와 저장하기의 제거는 모든 상품 수준의 컴파일러에서 중요하다. 중요한 부가적인 이익은 루프 색인과 기타 많이 사용되는 지역 변수와 매개변수의 반복된 불러오기와 저장하기를 피하기 위해서일 때만 특정 종류의 전역 레지스터 할당에서 얻을 수 있다. 주로 하드웨어를 잘 사용하는 문제로 귀결되는 이러한 기본적인 기술을 넘어서서 폰 노이만 프로그램에서 가장 중요한 이익은 배열에 대한 참조, 특히 루프 내에서 배열에 대한 참조를 최적화하는 것으로부터 얻게 된다. 다수의 상품 수준 컴파일러가 (1) 최소한 배열의 중복 주소 계산을 식별하기에 충분한 정도의 공통 하위 수식 분석을 수행하며, (2) 불변식 계산을 루프로부터 꺼내고, (3) 귀납 변수에 대한 강도 축소를 수행해 가능한 경우 귀납 변수를 제거한다.

15장의 소개 부분에서 언급했듯이 코드 개선은 여전히 매우 활발한 연구 분야다. 이 연구의 다수가 전통적인 최적화 기술이 특히 효율적이지 못한 언어 기능과 계산 모델의 문제를 해결한다. 예로는 C의 포인터에 대한 별칭 분석, 객체지향 언어에서 가상 메소드 호출의 정적 결정(인라이닝과 프로시저 간 최적화를 허용하기 위한 것), 메시지 전달 언어에서의 스트림라이닝된 통신, 함수형과 논리형 언어의 다양한 쟁점들이 있다. 어떤 경우에는 새로운 프로그래밍 패러다임이 코드 개선의 목표를 변경할 수도 있다.

예를 들어 자바나 C# 프로그램의 적시(just-in-time) 컴파일에서는 코드 개선기의 속도가 코드 개선기가 생성하는 코드의 속도만큼 중요할 수도 있다. 다른 경우에는 새로운 정보원(예를 들어 실행 시간 프로파일링으로부터의 피드백)이 새로운 개선 기회를 만들기도 한다. 끝으로 프로세서 구조의 발전(다중 파이프라인, 매우 넓은 명령어 워드, 비순차적 실행, 구조적으로 볼 수 있는 캐시, 예측 명령어)으로 인해 새로운 난제가 계속 나오고 있으며 점차적으로 프로세서 설계와 컴파일러 설계가 서로 연관되고 있다.

15.10 연습문제

15.1 ©(심화학습에 있는) 15.2절에서는 명령어 `r1 := r2/2`를 명령어 `r1 := r2 >> 1`로 대체하게 제안하고 이러한 대체가 음수의 경우에는 올바르지 않을 수 있다고 살펴봤다. 문제가 무엇인지 설명하라. 논리 시프트 연산과 산술 시프트 연산 사이의 차이점을 아는 것이 좋을 수 있다(거의 모든 어셈블리어 사용설명서에서 찾아볼 수 있다). 반올림과 관련한 쟁점을 고려하는 것이 좋을 수도 있다.

15.2 combinations 서브루틴의 루프에서 나눗셈 연산(©심화학습에 있는 예 15.10)이 항상 0의 나머지를 생성한다는 것을 증명하라. 분자 주변이 괄호가 왜 필요한지 설명하라.

15.3 어떤 코드 개선은 때때로 프로그래머가 소스 언어 프로그램에서 수행할 수 있다. 예로는 (재계산할 필요가 없게) 공통 하위 수식을 저장하기 위한 추가적인 변수를 도입하는 것, 불변식 계산을 루프 밖으로 이동시키는 것, 귀납 변수나 2의 제곱수에 의한 곱셈에 강도 축소를 적용하는 것 등이 있다. 프로그래머가 제대로 수행할 수 없는 최적화를 몇 가지 기술하라. 그리고 어떤 이유에서 프로그래머가 수행할 수 있는 최적화 중 어떤 것은 컴파일러가 최적화하게 남겨두는 것이 가장 좋을 수 있는지 설명하라.

15.4 예 6.61(337쪽)에서는 다음과 같은 루프를

```
// 전
for (i = low; i <= high; i++) {
   // 도중
}
// 후
```

다음과 같이 변환할 수 있다는 것을 알아봤다.

```
    -- 전
    i := low
    goto test
top:
    -- 도중
    i +:= 1
test:
    if i ≤ high goto top
    -- 후
```

그리고 실제로 이 변환은 combinations 서브루틴에서 사용하던 것이다. 다음은 또 다른 변환의 예다.

```
    -- 전
    i := low
    if i > high goto bottom
top:
    -- 도중
    i +:= 1
    if i ≤ high goto top
bottom:
    -- 후
```

이 변환이 combinations에 대해 사용했던 변환보다 더 나을 수 있는 이유를 설명하라(힌트: 분기의 수, 루프 불변식의 이동, 지연 빈 공간을 채울 수 있는 기회 등을 생각해보자).

15.5 이전 연습문제의 변환에서 시작해서 15장에서 combinations 서브루틴에 대해 설명했던 코드 최적화들을 다시 적용해보라.

15.6 215-CD쪽에 나열한 세 가지 휴리스틱이 최적의 코드 스케줄을 도출하지 못하는 예를 들어보라.

15.7 어떤 변수의 사용을 유발하는 가능한 모든 제어 경로상에서 이 변수가 값을 대입 받았다는 것(이것이 6.1.3절에서 설명한 명백한 대입의 개념이다)을 검증하는 데 전방 자료 흐름 분석을 사용할 수 있음을 보여라.

15.8 매우 분주한 수식(향후의 모든 코드 경로상에서 계산될 것이 보장되는 수식)은 후진, 모든-경로 자료 흐름 분석을 통해 알아낼 수 있다. 이러한 수식에 대한 공간 절약 코드 개선을 제안하라.

15.9 각 가상 레지스터의 값에 기여하는 변수와 레지스터의 집합을 식별할 수 있게 해주는 지역 값 번호 지정 동안 정보는 어떻게 모으는지 설명하라(레지스터 vi의 값이 레지스터 vj나 변수 x에 의존적이면 B가 vj나 x에 대한 대입문을 포함하고 vi에 대한 향후 대입문은 포함하지 않는 경우 이용 가능한 수식 분석 동안 $vi \in Kill_B$라고 할 수 있다).

15.10 i가 루프 색인 변수일 때 루프 내에서 수식 i^2를 어떻게 강도 축소할 수 있는지 보여라. 로프 단계 크기는 1이라고 가정해도 무방하다.

15.11 나눗셈은 가끔 덧셈이나 뺄셈보다 비용이 훨씬 더 크다. for 루프 내부에 있는 i div c 형태의 수식을 어떻게 덧셈이나 뺄셈으로 대체할 수 있는지 보여라. 여기서 i는 루프 색인 변수며 c는 정수형 상수다. 루프 단계 크기는 1이라고 가정해도 무방하다.

15.12 다음과 같은 고수준 의사코드를 보자.

```
read(n)
for i in 1..100
   B[i] := n × i
   if n > 0
      A[i] := B[i]
```

n>0이라는 조건은 루프 불변식이다. 이를 루프 밖으로 이동시킬 수 있는가? 이동시킬 수 있다면 어떻게 할 수 있는지 설명하고 이동시킬 수 없다면 그 이유를 설명하라.

15.13 유효 변수 분석은 루프 불변식 제거 전이나 후에 수행해야 하는가(아니면 전과 후에 두 번 수행해야 하는가)? 작성한 답의 근거를 제시하라.

15.14 그림 1.5의 나이브한 gcd 코드에 대해 (값 번호 지정을 통한) 지역 중복 제거와 명령어 스케줄링을 수행한 결과를 보여라.

15.15 이전 연습문제의 결과 프로그램의 제어 흐름 그래프를 그리고 전역 값 번호 지정의 결과를 보여라. 다음으로 자료 흐름 분석을 사용해서 적절한 전역 최적화를 모두 수행하라. 그리고 전역 레지스터 할당을 수행하기 위해 레지스터 충돌 그래프를 그리고 색 지정하라. 끝으로 명령어 스케줄링의 최종 패스를 수행하라. 이렇게 얻은 코드는 예 1.2의 버전과 비교해서 어떤가?

15.16 ◎(심화학습에 있는) 15.6절에서는 하드웨어 레지스터 재명명이 가끔 반의존성과 출력 의존성을 숨길 수 있다고 알아봤다. 이것이 ◎(심화학습에 있는) 그림 15.12에 도움이 되는가? 설명하라.

15.17 다음의 코드를 보자.

```
v2 := *v1
v1 := v1 + 20
v3 := *v1
--
v4 := v2 + v3
```

v1의 갱신을 앞쪽에 있는 두 번째 불러오기의 지연 빈 공간으로 이동시킴으로써 이 코드에 필요한 시간을 어떻게 줄일 수 있는지 보여라(v1은 끝부분에서 여전히 유효하다고 가정하자). 이러한 종류의 변환을 적용하기 위해 유지해야 할 조건과 정확성을 유지하기 위해 개별 명령어에 취해야 하는 변경을 기술하라.

15.18 다음의 코드를 보자.

```
v5 := v2 × v36
--
--
--
--
v6 := v5 + v1
v1 := v1 + 20
```

v1의 갱신을 뒤쪽에 있는 곱셈의 지연 빈 공간으로 이동시킴으로써 이 코드에 필요한 시간을 어떻게 줄일 수 있는지 보여라. 이러한 종류의 변환을 적용하기 위해 유지해야 할 조건과 정확성을 유지하기 위해 개별 명령어에 취해야 하는 변경을 기술하라.

15.19 이전 두 연습문제의 맥락에서 (펼치기와 파이프라이닝에 앞서) v26과 v34의 갱신을 뒤쪽에 있는 지연 빈 공간으로 이동시킴으로써 combinations 서브루틴의 주 루프를 어떻게 줄일 수 있는지 보여라. 이러한 변경이 루프 성능에 미치는 영향은 몇 %인가?

15.20 ◎(심화학습에 있는) 그림 15.11과 15.13을 가이드로 사용해서 combinations 서브루틴의 루프를 세 번 펼쳐라. 그리고 새로운 블록 2에 대한 의존성 DAG를 구성하라. 끝으로 블록을 스케줄링하라. 이렇게 구한 코드는 본래(펼치지 않은) 루프의 반복당 몇 주기를 소비하는가? 루프의 소프트웨어 파이프라이닝된 버전(◎심화학습에 있는 그림 15.15)과 비교하면 어떤가?

15.21 루프 펼치기와 소프트웨어 파이프라이닝을 모두 수행한 버전의 combinations 서브루틴을 작성하라. 즉, ◎(심화학습에 있는) 그림 15.14의 이웃한 막대 사이에 있는 명령어들로부터 루프 몸체를 구성하지 말고 가장 왼쪽 수직 막대와 가장 오른쪽 수직 막대 사이에 있는 명령어들로부터 루프 몸체를 구성하라. 배열 포인터는 반복당 한 번만 갱신해야 한다. 작성한 코드는 본래 로프의 반복당 몇 주기를 소비하는가? 작성한 코드는 파이프라인을 "준비"하고 "플러시"하기에, 또 충분한 반복 횟수를 확인하기에 얼마나 까다로운가?

15.22 행렬 곱셈에 대한 다음의 코드를 보자.

```
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      C[i][j] = 0;
   }
}
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      for (k = 0; k < n; k++) {
         C[i][j] += A[i][k] * B[k][j];
      }
   }
}
```

행렬 A, B, C에 대한 접근 패턴을 기술하라. 배열이 큰 경우 메모리로부터 각 캐시 라인을 몇 번이나 인출하는가? 안쪽의 두 루프를 타일화하라. 그리고 이것이 캐시 부적중 횟수에 미치는 영향을 기술하라.

15.23 다음과 같은 가우스 소거법의 간단한 예를 보자.

```
for (i = 0; i < n-1; i++) {
   for (j = i+1; j < n; j++) {
      for (k = n-1; k >= i; k--) {
         A[j][k] -= A[i][k] * A[j][i] / A[i][i];
      }
   }
}
```

(가우스 소거법은 행렬을 삼각형 모양으로 변형시킨다. 이는 선형 방정식의 해를 구하는 데 있어 핵심적인 단계다) 이 코드에서 루프 불변식은 무엇인가? 루프 운반 의존성은 무엇인가? 코드를 어떻게 최적화할지 논해보자. 집약성을 개선하는 루프 변환은 반드시 고려해야 한다.

15.24 (심화학습에 있는) 예 15.32의 타일화된 행렬 전치를 수정해 안쪽 루프 경계의 `min` 계산을 제거하라. (심화학습에 있는) 연습문제 15.22의 답안에도 동일한 변경을 수행하라.

15.11 탐구문제

15.25 가장 즐겨 쓰는 컴파일러의 후단 구조를 알아보자. 어떤 수준의 최적화를 사용할 수 있는가? 각 수준에서는 어떤 기술을 사용하는가? 기본 수준은 무엇인가? 컴파일러가 어셈블리어나 목적 코드를 생성하는가?

몇 개의 프로그램 조각을 이용해서 최적화를 실험해보자. 컴파일러가 어셈블리어를 생성하게 하거나 디스어셈블러나 디버거를 사용해서 생성된 목적 코드를 조사해보자. 이 코드의 품질을 다양한 최적화 수준에서 평가해보자.

컴파일러가 별도의 어셈블러를 사용한다면 어셈블러 입력과 디스어셈블링된 출력을 비교해보자. 어셈블러가 수행한 최적화가 있다면 어떤 것인지 말해보자.

15.26 일반적으로 컴파일러는 프로그램 변환이 코드의 정확성을 보존하는 경우에만 해당 프로그램 변환을 적용할 수 있다. 그러나 어떤 상황에서는 변환의 정확성이 실행시간에야 비로소 알 수 있는 정보에 의존적일 수 있다. 이런 경우 컴파일러는 사용할 버전을 선택하거나 매개변수화된 일반적인 버전을 맞춤화하는 실행시간 검사와 함께 두 가지(혹은 그 이상의) 버전을 생성할 수 있다.

살츠 등이 연구한[SaMC91] "조사자–실행자" 컴파일 패러다임에 대해 알아보자. 이 기술은 얼마나 일반적인가? 어떤 상황에서 성능적 이익이 실행시간 검사의 비용과 코드 크기의 잠재적 증가보다 클 것으로 기대할 수 있는가?

15.27 어떤 프로그램 변환의 경우 정확성은 증명할 수 있는 반면 이 변환으로부터 얻을 수 있는 이익은 불확실한 경우가 있다. 응용 동작에 대한 자세한 지식 없이는 변환이 프로그램을 더 빠르게 만들지 아니면 느리게 만들지를 결정할 수 없을 수 있다. 일부 컴파일러는 높은 최적화 수준에서 자기 자신의 동작에 대한 통계를 모으는 코드를 생성할 수 있다. 이러한 통계는 이후 재컴파일의 코드 개선 단계를 수행하는 데 사용할 수 있다. 또 최근 몇몇 프로젝트에서는 지속적이고 투명하게(transparently) 프로그램 동작을 감시한 후 기존의 실행 가능한 기계 코드의 기대 성능을 개선하게 이 기계 코드를 수정하는("최적화"하는) 시스템을 개발해왔다.

HP Dynamo 프로젝트[BDB00]에 대해 알아보자. 인용 색인을 사용해서 이를 인용하는 좀 더 최신의 프로젝트를 찾아보자. 프로그램이 배치된 후에 이익이 있게 적용할 수 있는 최적화 목록을 작성해보자. 이런 방식의 코드 개선이 가질 수 있는 잠재적인 문제에 대해 논해보자.

15.28 최근 경향은 (확실하진 않지만) 프로그래밍 오류를 구성하기 쉬운 정보 흐름의 패턴을 검사하기 위한 정적 컴파일러 분석을 사용하는 것이다. 예를 들어 ◎(심화학습에 있는) 탐구문제 13.25에서는 펄과 루비의 오염 방식(taint mode)이라는 개념을 소개했다. 오염 방식은 믿을 수 없는 곳에서 시작되었으며 그러므로 컴퓨터 보안이 중요한 문맥에서 사용해선 안 되는 값들을 실행 시간에 추적하게 시도한다. 컴파일 시점에 이와 유사한 분석을 수행하는 가이어 등[GL03]의 연구를 조사해보자. 비슷한 맥락에서 고수준 오류를 잡기 위해 정적 모델 검사를 사용하는 양 등[YTEM04]와 첸 등[CDW04]의 연구도 조사해보자. 이러한 연구들에 대해 어떻게 생각하는가? 정적 분석이 부정 오류(분석이 놓치는 오류)와 긍정 오류(분석이 오류라고 판단하는 올바른 코드)를 모두 가지는 경우에도 이런 분석이 유용할 수 있는가?

15.29 무어의 법칙의 다소 비관적인 패러디로 마이크로소프트 연구소의 토드 프럽스팅(그 자신도 저명한 컴파일러 연구자임)은 프럽스팅의 법칙을 제안했다. 프럽스팅의 법칙은 "컴파일러의 발전이 18년마다 컴퓨팅 능력을 두 배씩 증가시킨다"는 것이다(이 법칙에 대한 지적은 research.microsoft.com/~toddpro/에서 볼 수 있다).

컴파일러 기술의 역사를 조사해보자. 주요 혁신은 어떤 것들인가? 속도 외에 다른 분야에서 중요한 발전이 있었는가? 프럽스팅의 법칙은 이 분야에 대한 공정한 평가인가?

15.12 참고자료

후단 컴파일러 기술에 대한 정보는 최신 컴파일러 교과서들(예를 들어 쿠퍼와 토르크존의 저서[CT04], 그루네 등의 저서[GBJL01], 아펠의 저서[App97])에서 쉽게 접할 수 있다. 15장에서 사용한 설명의 대부분은 자세한 정보와 관련 연구에 대한 인용을 풍부하게 포함하고 있는 머치닉의 고급 컴파일러 설계와 구현에서 영감 받은 것이다[Muc97]. 컴파일러 기술을 선도하는 연구의 대부분은 ACM 프로그래밍 언어 설계와 구현에 대한 컨퍼런스(ACM Conference on Programming Language Design and Implementation)에서 찾아볼 수 있다. 이 컨퍼런스의 최초 20년 동안의 "최고 논문"을 담은 모음집이 2004년에 출간되었다[McK04].

이 책에서는 코드 개선을 살펴볼 때 폰 노이만 계열의 언어를 중점적으로 살펴봤다. 이와 유사한 기술들이 함수형 언어[App91; KKR+86; Pey87; Pey92; WM95, 3장; App97, 15장; GBJL01, 7장], 객체지향 언어[AH95; GDDC97; WM95, 5장; App97, 14 장; GBJL01, 6장], 논리형 언어[DRSS96; FSS83; Zho96; WM95,4장; GBJL01, 8장]에도 존재하며 활발한 연구 주제지만 이 책의 범위 밖이다. 함수형 언어에서의 주요 난제는 루프 방식의 최적화를 수행할 수 있는 호출의 반복적인 패턴(예를 들어 꼬리 재귀)을 식별하는 것이다. 객체지향 언어에서의 주요 난제는 인라이닝과 프로시저 간 코드 개선을 가능케 하기 위해 가상 서브루틴 호출의 목표 지점을 정적으로 예측하는 것이다. 논리형 언어에서의 주요 난제는 목표 지시형 검색의 기본 과정을 더 잘 지시하는 것이다.

지역 값 번호 지정은 코크와 슈와르츠가 최초로 제안했으며[CS69] 15장에서 설명한 전역 알고리즘은 알펀, 웨그만, 자덱의 알고리즘[AWZ88]에 기반한 것이다. 샤틴 등[CAC+81]은 레지스터 할당을 위한 그래프 색 지정의 사용을 널리 퍼뜨렸다. 싸이트론 등[CFR+91]은 정적 단일 대입문 형태의 생성과 사용을 기술했다. 앨렌과 케네디[AK02, 12.2절]는 C에서의 별칭 분석에 대한 일반적인 문제를 논했다. 포인터는 이러한 분석에서 가장 어려운 부분이지만 최근 몇 년 간 상당한 발전이 있었다. 힌드[Hid01]는 광범위하고 이해하기 쉬운 조사를 수행했다. 기본 블록 의존성 DAG로부터의 명령어 스케줄링은 기븐스과 머치닉이 기술했다[GM86]. 범용 기술은 리스트 스케줄링이라고 하며 근래의 방법은 머치닉[Muc97, 17.1.2절]과 쿠퍼와 토르크존[CT04, 12.3절]의 교과서에서 찾아볼 수 있다. 마살린은 진정한 최적 프로그램을 생성하는 것이 바람직할 수 있는(그리고 가능할 수 있는) 상황에 대한 좋은 설명을 제공했다[Mas87].

루프 변환과 병렬화에 대한 정보는 앨렌과 케네디의 최신 교과서[AK02], 울페의 좀

더 오래된 교과서[Wof96], 베이컨, 그레이엄, 샤프의 훌륭한 조사[BGS94] 등에서 얻었다. 바너지는 루프 의존성 분석을 자세히 기술했다[Ban97]. 라우와 피셔는 벡터 프로세서, 넓은-명령어-워드프로세서, 슈퍼스칼라 프로세서가 활용할 수 있는 종류의 세밀한 명령어 수준 병렬성 등을 논했다[RF93].

최근 일부 연구 그룹은 (제3자로부터 바이너리 형태로 구입했기 때문에) 소스 코드를 구할 수 없는 프로그램을 분석해왔다. 범용 바이너리 인스트루멘테이션 도구[LS95, SE94, RVL+97]는 프로그램을 프로파일링하거나 추적하는 데 사용하거나 다른 메모리 구조를 에뮬레이트하는 데 사용할 수 있다[SFL+94, SG96]. 좀 더 적극적인 도구는 한 기계 구조의 객체 파일을 다른 기계 구조의 객체 파일로 변환하거나(예를 들어 x86 바이너리를 다른 업체의 하드웨어상에서 실행하는 것[SCK+93, HH97b]) 기존의 프로그램 바이너리에 대한 적극적 개선을 수행할 수 있다[BDB00, CDS03]. 저수준 코드로부터 모든 고수준 의미 정보를 복구하는 것이 불가능하다는 것은 증명되었지만 최근의 발전을 보면 놀랍게도 가끔 이용할 수 있는 정보의 수준이 매우 풍부하다.