튜플 코드 상에서의 최적화기 구현과 분석

송 진 국*

Implementation and Analysis of Optimizers on Tuple codes

Jin-Kook Song*

요약

컴파일러의 코드 최적화(code optimization)는 생성되는 기계 코드의 실행 시간을 감소시키고 기억장소 크기를 감소시키기 때문에 매우 중요한 단계이다. 본 연구에서는 튜플(tuple) 형태의 중간코드에서 제어 및 자료 흐름 분석(control and data flow analysis)을 수행하여 프로그램의 흐름 분석 정보를 얻어 각종 최적화를 수행하는 최적화기(optimizer)를 구현하였다. 또한, 각 최적화기들이 수행한 최적화 정도를 비교 분석하고, 최적화기들 상호간의 의존성을 분석하였다. 따라서, 컴파일러 최적화 단계에서 우선적으로 수행할 최적화기를 선택 문제를 해결하고, 상호간의 의존도로부터 최적화들간의 순서를 정할 수 있다.

Abstract

Code optimization phase in a compiler are very important because the phase reduces the running time and the storage size of machine codes. I developed flow analyzers and optimizers on intermediate codes. The flow analyzers generate control-flow and data-flow information. The optimizers transform the intermediate codes into the improved codes using this information. This paper describes the development of flow analyzers and optimizers. I also examined the execution performance, the cost and the dependency of each optimization.

Ⅰ. 서 론

RISC는 기계 명령어의 대부분이 한 싸이클에 처리될 수 있도록 단순하며 획일적인 구조로 설계되어 있으며, 다수의 레지스터를 보유하여 load-store 아키텍처로 구성되어 있다. 이러한 단순한 구조의

* 진주산업대학교 전자계산학과 교수
접수일자 : 1999년 11월 18일

CPU를 바탕으로 한 시스템의 효율을 극대화하기 위해서는 고도의 소프트웨어 기술이 필요하다. 즉, 프로그램 개발에 사용되는 컴파일러가 발생시키는 기계어 프로그램이 어느 정도 최적화(optimization) 되느냐에 따라 RISC의 성능은 결정된다.

따라서, 본 연구에서는 최적화된 기계 코드를 얻기 위해 중간 코드상에서의 흐름 분석기와 최적화 기들을 구현하였다. 컴파일러의 흐름 분석 및 최적화기의 위치는 그림 1과 같다. 컴파일러 전단부(front-end)의 결과인 중간코드로부터 흐름 분석을 수행하고, 최적화를 수행한다. 최적화된 중간코드는 코드 생성기로 전달된다.

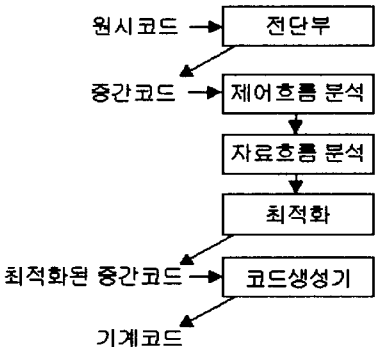

그림 1. 흐름 분석과 최적화의 위치
Fig. 1. Flow analysis phase and optimization phase

Ⅱ. 중간언어 TUP

본 연구에서는 튜플 형태의 중간언어인 TUP 코드를 사용하였다.

튜플에 기반한 중간언어는 최적화 컴파일러에 많이 사용되며, 보통 피연산자는 기억 장소나 이전 연산의 결과이 때문에, RISC와 같이 다수의 레지스터를 갖는 기계어 코드로 쉽게 번역될 수 있고, 중간코드 최적화기를 거치면 효율적인 기계어 생성도 가능하다. 전체 TUP 코드의 연산자와 의미는 표 1과 같다. 의미는 C 언어로 표기하였다.

표 1. TUP 코드
Table 1. TUP codes

operators	meaning	operators	meaning
(ADDI,a,b,c)	c = a+b	(AND,a,b,c)	c = a&&b
(SUBI,a,b,c)	c = a-b	(OR,a,b,c)	c = a|b
(MULI,a,b,c)	c = a*b	(NOT,a,b,)	b = !a
(DIVI,a,b,c)	c = a/b	(CALL,F,n,b)	b = F(a_1,..a_n)
(MODI,a,b,c)	c = a%b	(RET,a,,)	return(a)
(MINUSI,a,b,)	b = -a	(IF,a,L,)	if (a) goto L
(ADDU,a,b,c)	c = a+b	(IFNOT,a,L,)	if (!a) goto L
(SUBU,a,b,c)	c = a-b	(EQ,a,b,c)	c = a==b
(MULU,a,b,c)	c = a*b	(NE,a,b,c)	c = a!=b
(DIVU,a,b,c)	c = a/b	(GT,a,b,c)	c = a>b
(MODU,a,b,c)	c = a%b	(LT,a,b,c)	c = a<b
(ADDF,a,b,c)	c = a+b	(GE,a,b,c)	c = a>=b
(SUBF,a,b,c)	c = a-b	(LE,a,b,c)	c = a<=b
(MULF,a,b,c)	c = a*b	(BITAND,a,b,c)	c = a&b
(DIVF,a,b,c)	c = a/b	(BITOR,a,b,c)	c = a|b
(MINUSF,a,b,)	b = -a	(BITXOR,a,b,c)	c = a^b
(POINT,a,b,)	b = *a	(SHL,a,b,c)	c = a<<b
(ADDRESS,a,b,)	b = &a	(SHR,a,b,c)	c = a>>b
(LP,a,b,)	*b = a	(GOTO,L,,)	goto L
(CHAR,a,b,)	b = (char) a	(LABEL,L,,)	L:
(SHORT,a,b,)	b = (short) a	(FUN,F,,)	start of function
(INT,a,b,)	b = (int) a	(END,F,,)	end of function
(UNSIGNED,a,b,)	b=(unsigned)a	(FILE,FN,,)	source file
(LONG,a,b,)	b = (long) a	(FRAME,FS,,)	local frame
(FLOAT,a,b,)	b = (float) a	(GLOBAL,a,s,v)	global variable
(DOUBLE,a,b,)	b =(double) a	(EXTERN,E,,)	extern var, function
(POINTER,a,b,)	b = (*) a	(APARAM,a,,)	actual parameter
(COPY,a,b,)	b = a	(FPARAM,b,,)	formal parameter

Ⅲ. 흐름 분석

정확하고 효율적인 최적화를 위하여 흐름분석 과정이 필요하다. 그림 1에서와 같이 제어 흐름 분석과 자료 흐름 분석으로 구성되는 흐름 분석은 전단부로부터 생성되는 중간코드상에서 동작하여 코드 최적화 과정에서 필요한 정보를 생성한다.

제어 흐름 분석은 프로그램 실행시 제어의 흐름을 분석하는 과정으로 프로시져 사이의 제어 흐름 정보를 나타내는 호출 그래프(call graph)의 생성 과정과 기본블럭(basic block) 사이의 제어 흐름 정보를 나타내는 흐름 그래프(flow graph)의 생성 과정으로 구성된다[1][3]. 호출 그래프는 노드가 하나의 프로시져이며, 각 노드간에 에지는 프로시져 간에 호출을 나타내는 방향성 그래프이다. 흐름 그래프는 노드가 하나의 기본블럭이며, 각 노드간의 에지는

프로그램 실행 시에 제어가 전달되는 흐름 가능성을 나타낸다. 기본블럭이란 프로그램이 실행되면서 중지 또는 다른 위치로 제어가 분기하지 않고 연속적으로 실행되는 문장들의 집합이다[1][3].

자료 흐름 분석이란 실행 환경에서 프로그램의 변경(modification), 유지(preservation), 값들의 사용(usage) 등의 정보를 조사하는 단계로써 수준은 대상 단위에 따라 단일 문장(statement)이거나 기본블록(intrablock), 프로시져(intraprocedure), 프로그램(interprocedure)으로 구분한다. 일반적으로 단일 문장과 기본블럭을 대상으로 하는 자료 흐름 분석을 지역적(local)이라고 하며, 프로시져와 프로그램을 단위로 하는 경우를 전역적(global)이라고 한다[1][3]. 자료의 수집은 일반적으로 계층적으로 수행된다. 먼저 문장 단위의 자료 흐름 분석이 수행되어 이 분석 정보를 모아 각 기본블럭 단위의 정보를 계산하고, 기본블럭 단위의 정보가 계산되면 이 정보를 이용해 기본블럭 사이의 정보가 전달되는 과정을 거쳐 프로시져 단위의 정보를 계산하고, 또 이 정보를 모아 전체 프로그램을 단위로 하는 정보를 생성함으로써 자료 흐름 분석을 수행하게 된다. 자료 흐름 분석에서 대상이 되는 대표적인 문제로는 생존 변수(live variable) 분석과 도달 정의(reaching definition) 분석, 초기화되지 않은 변수(uninitialized variable) 분석, 이용가능한 수식(available expression) 분석, 이용빈도 높은 수식 분석 등이 있다.

흐름 분석의 종류에는 아래와 같이 계산된 정보가 다음 기본블럭에 전달되는 방향에 따라 전진(forward) 방식과 후진(backward) 방식이 있으며, 경로를 따라 전달되는 정보들을 합성하는 방법에 따라 모든 경로(All path) 방식과 임의 경로(Any path) 방식으로 구분된다. 전진 방식은 임의 블럭에서 생성된 정보가 후속 블럭에 전달되어 계산되는 방식이고, 후진 방식은 임의 블럭에서 생성된 정보가 그 블럭의 선행 블럭에 전달되어 계산되는 방식이다. 임의 경로 방식은 하나의 기본블럭에서 여러 기본블럭으로부터 정보를 입력받는 경우 모든 경로의 정보에 대해 합(union) 연산을 수행하고, 모든 경로 방식은 한 기본블럭에서 여러 기본블럭으로부터 정보를 입력받는 경우 모든 경로의 정보에 대해 곱(and) 연산을 수행한다[1].

자료흐름 방정식은 각 기본블럭에서 생성 및 변화되는 정보들을 계산하기 위한 알고리즘을 방정식으로 나타낸 것으로 구성은 다음과 같다.

$In(b)$:선행 기본블럭과 연결된 경로를 통해 블록b에 입력 또는 블록b로부터 출력되는 정보

$Out(b)$:후속 기본블럭과 연결된 경로를 통해 블록b에 입력 또는 블록b로부터 출력되는 정보

$Gen(b)$:기본블럭b에서 생성되는 정보

$Kill(b)$:기본블럭b에서 사라지는 정보

따라서 일반적인 자료흐름 방정식을 다음과 같이 표현할 수 있다.

$$Out(b) = Gen(b) \cup (In(b) - Kill(b)$$

또는

$$In(b) = Gen(b) \cup (Out(b) - Kill(b)$$

생존 변수 분석은 이전의 프로그램 문장에서 정의되어 현재 위치에서 사용 가능한 변수들을 분석하는 것으로 필요한 정보와 방정식은 다음과 같다.

$LiveIn(b)$:기본블럭b의 입구(*entry*)에서 생존변수 집합

$LiveOut(b)$:기본블럭b의 출구(*exit*)에서 생존변수 집합

$LiveUse(b)$:기본블럭b에서 사용되는 변수 집합

$Def(b)$:기본블럭b에서 정의되는 변수 집합

$$LiveIn(b) = LiveUse(b) \cup (LiveOut(b) - Def(b))$$

$$LiveOut(b) = \bigcup_{i \in s(b)} LiveIn(i)$$

도달 정의는 레지스터 할당(register targeting)과 생존 범위, 상수 전파(constant propagation), 복사 전파(copy propagation)의 분야에 적용시킬 수 있고, 임의 경로, 전진 방식으로 수행하므로 다음과 같은 방정식으로 표현된다.

$ReIn(b)$:기본블럭b의 입구에서 도달하는 정의 집합

$ReOut(b)$:기본블럭b의 출구에 도달하는 정의 집합

$ReGen(b)$:기본블럭b에서 정의되는 정의 집합

$Kill(b)$:기본블럭b에서 *kill*되는 정의 집합

$$ReOut(b) = ReGen(b) \cup (ReIn(b) - Kill(b))$$

$$ReIn(b) = \bigcup_{i \in s(b)} ReOut(i)$$

이용 가능한 수식이란 현재 계산하는 수식이 이

전에 이미 계산된 수식인지를 검사하는 과정으로서 전역 공통 수식(global common subexpression) 최적화 과정에서 이용되는 정보이고, 모든 경로, 후진 방식으로 수행하므로 다음과 같은 방정식으로 표현된다.

$AvailIn(b)$:기본블럭b입구에서 이용가능한 변수 집합

$AvailOut(b)$:기본블럭b출구에서 이용가능한 변수 집합

$Computed(b)$:기본블럭b내에서 계산된 수식 집합

$Killed(b)$:기본블럭b에서 이용가능한 수식중 새로 정의되어 이용불가능한 수식 집합

$$AvailIn(b) = \bigcap_{i \in s(b)} AvailOut(i)$$

$$AvailOut(b) = Computed(b) \cup (AvailIn(b) - Killed(b))$$

이용빈도 높은 수식이란 모든 경로에서 사용되는 수식으로 이에 관련된 정보는 레지스터를 할당하는 단계에서 레지스터를 받는 후보의 우선 순위를 계산하는데 이용되며, 또 루프 내의 코드를 루프 밖으로 이동시키는데도 사용된다. 모든 경로, 전진 방식으로 수행하므로 다음과 같은 방정식으로 표현된다.

$VerybusyIn(b)$:기본블럭b 입구에서 사용되는 수식 집합

$VerybusyOut(b)$:기본블럭b 출구에서 사용되는 수식 집합

$Used(b)$:기본블럭b 내에서 $kill$되기이전에 사용되는 수식 집합

$Killed(b)$:기본블럭b 내에서 사용되기이전에 $kill$되는 수식 집합

$$VerybusyIn(b) = Used(b) \cup (VerybusyOut(b) - Killed(b))$$

$$VerybusyOut(b) = \bigcap_{i \in s(b)} VerybusyIn(i)$$

자료 흐름에 관련된 여러 분야에서 지금까지 기술한 방법으로 동작을 하면 관련된 정보를 계산할 수 있다. 이러한 방법은 각 기본블럭을 단위로 기본블럭에 입력되는 정보들과 기본블럭 내부에서 생성되는 새로운 정보를 합성하여 출력으로 변경된 새로운 정보를 다른 기본블럭에 전달하게 된다. 그러면 각 분야에서 흐름 분석이 시작되는 초기 기본블럭(전진 방식에서는 프로시져의 시작 기본블럭이고, 후진 방식에서는 제어의 흐름이 끝나는 단말 기본블럭이다)은 입력되는 정보들이 없기 때문에 표 2와 같이 미리 결정된 초기값을 갖게된다. 즉, 초기화되지 않은 변수 분석에서는 자료 전달의 시작되는 기본블럭이 프로시져가 시작하는 첫부분의 기본블럭이므로 이 블럭의 In 값은 그 프로시져 내부에서 나타나는 모든 변수가 초기값으로 입력된다. 그리고 나머지 분야의 분석에서 자료 전달이 시작되는 기본블럭의 입력 값은 공집합으로 입력된다[1].

표 2. 각 문제들의 초기값

Table 2. Initialization of problems

방식 / 경로	전진 방식		후진 방식	
	분야	초기값	분야	초기값
임의경로	도달 정의	$\emptyset$	생존 변수	$\emptyset$
	초기화되지 않은변수	모든변수		
모든경로	이용가능한 수식	$\emptyset$	이용빈도 높은 수식	$\emptyset$
	복사 전파	$\emptyset$		

UD(use-definition)-체인은 변수의 사용에 대한 가능한 모든 정의문을 찾는 문제이고, 역으로 DU-체인은 변수의 정의가 사용될 수 있는 모든 사용문을 찾는 문제이다[5]. 이 두 정보는 최적화 단계에서 사용된다.

STAT은 프로시져 P가 가지는 문(statement)들의 집합이고, VAR는 프로시져에 사용된 단순변수(scalar)라고 하자. 변수 v∈VAR를 문 S∈STAT에서 발생한 변수라 할 때, 문 S는 변수 v의 정의(definition)라 한다. 즉 S의 수행결과 변수 v의 값이 새로이 정의된다. 또한, 문 S의 수행과정에서 변수 v의 값을 참조하면 문 S를 변수 v의 사용(use)이라 한다. 문 S는 v=v+1에서와같이 정의와 사용이 동시에 발생하는 경우도 있다.

모든 정의문들과 사용문들을 다음과 같이 나타낸다[5].

$$S_{DEFS} = \{S \in STAT : S\text{는 정의}\}$$

$$S_{USES} = \{S \in STAT : S\text{는 사용}\}$$

S_{DEFS}는 모든 정의의 집합, S_{USES}는 모든 사용의 집합으로 정의한다. 임의의 문 S가 정의하는 변수와 사용하는 변수를 다음과 같이 정의한다[5].

$$DEF(S) = \{v \in VAR : S\text{는 변수}v\text{의 정의}\}$$

$$USE(S) = \{v \in VAR : S\text{는 변수}v\text{의 사용}\}$$

임의의 기본블럭 n에서 변수 v에 대해 발생한 정의 S가 마지막 정의이면 S를 outward exposed definition이라 한다. 또한, 기본블럭 n에서 사용된 변수 v가 정의없이 사용된다면 이때의 사용을 outward exposed use라 한다[5].

이제 DEF와 USE의 정의를 기본블럭으로 확장하면 다음과 같다. BB를 프로시져 P의 기본블럭의 집합이고, n∈BB이라 하자.

$DEF(n) = \{v \in VAR : n \in BB$에 변수$v$의 *outward exposed definition* 존재.$\}$

$USE(n) = \{v \in VAR : n \in BB$에 변수$v$의 *outward exposed use* 존재.$\}$

$S_{DEF}(n) = \{S \in STAT : n \in BB$에서 S는 *outward exposed definition*$\}$

$S_{USE}(n) = \{S \in STAT : n \in BB$에서 S는 *outward exposed use*$\}$

DEF(n)은 n에서 정의된 변수 v, USE(n)은 n에서 정의 없이 사용된 변수 v를 나타낸다. 이들이 변수에 대해 정의된 반면에, S_{DEF}는 n에서 발생하여 n 밖으로 영향을 끼칠 수 있는 정의 S, S_{USE}는 n에서 outward exposed use가 발생한 사용 S를 각각 문 단위로 정의한다[5].

n, n'∈BB에 대해 n'에서 n으로 가는 경로가 존재할 때, n'의 변수 v에 대한 정의 S는 n에 도달(reach)한다고 한다. 변수 v의 정의 S가 n에 도달하고 v∈USE(n)이면 S는 n의 변수 v 사용에 도달한다고 말한다. n∈BB에 도달하는 도달 정의를 다음과 같이 정의한다[5].

$RD(n) = \{S : S \in S_{DEFS},\ S\ reaches\ n,\ n \in BB\}$

이때, RD(n)은 n에 도달하는 모든 S를 말한다. n에 대한 UD-체인은 RD(n)에 속하는 S들 중 v를 정의하는 S의 집합으로 정의할 수 있다.

$UD(n, v) = \{S' : S' \in RD(n),\ v \in DEF(S'),\ n \in BB\}$

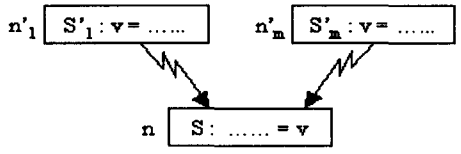

그림 2. UD(n,v) = {S'1,....,S'm}
Fig. 2. UD(n,v) = {S'1,....,S'm}

UD(n,v)는 기본블럭 n과 변수 v에 대한 UD-체인이다. DU-체인은 UD(n,v)에 의해 다음과 같이 정의된다[5]. N에 속하는 임의의 기본블럭 n'에 대해, S'∈ SDEF(n'), v ∈ DEF(S')가 있을 때,

$DU(n', v) = \{S : 어떤\ n에\ 대해\ S \in S_{USE(n)},\ v \in USE(S),\ S' \in UD(n, v)\}$

DU(n',v)는 기본블럭 n'에서 정의된 변수 v에 대한 DU(definition-use)-체인이다.

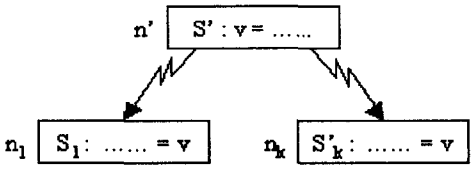

그림 3. DU(n',v) = {S1, ... Sk}
Fig. 3. DU(n',v) = {S1, ... Sk}

n,n'∈BB이고 S∈SUSE(n), S'∈ SDEF(n'), v∈ USE(S)∩DEF(S')에 대해 다음이 성립한다.

$S' \in UD(n, v) \Leftrightarrow S \in DU(n', v)$

Ⅳ. 최적화

프로그램 최적화에 있어 적은 노력으로 많은 이득을 얻는 프로그램 변환을 가장 좋은 최적화라 한다[1]. 프로그램의 변환은 프로그램 의미를 유지해야 하고, 속도를 개선하여야 한다[1]. 때때로 코드의 크기를 줄이기 위한 변환들이 있는데, 실행 속도가 코드 크기보다 더 중요하다[1]. 최적화의 종류는 기계 의존적 최적화(machine dependent optimization)와 기계 독립적 최적화(machine independent optimization)로 분류할 수 있다[1]. 기계 의존적 최적화는 레지스터의 갯수나 레지스터의 사용, 기계 연산자의 특성 등 목적 기계의 특성에 의존하여 최적화를 수행하는 것으로 레지스터 할당, 기계 의존적 핍홀 최적화 등이 있다. 기계 독립적 최적화는 기계의 특성과 관계없이 최적화를 수행하는 것으로 본 논문에서의 중간 코드에 대한 최적화기들이다. 특히, 기계 독립적 최적화 중, 제어 흐름 분석의 정보만 필요한 최적화가 있고, 제어 흐름 분석 및 자료 흐름 분석 정보가 필

요한 최적화가 있다. 제어 흐름 분석 정보만이 필요한 최적화로는 분기문 최적화(jump-to-jump optimization)와 도달 불가능한 기본블럭 제거(unreachable basic block elimination)가 있고, 자료 흐름 분석 정보도 필요한 최적화로는 복사 전파(copy propagation)와 공통 수식 제거(common subexpression elimination), 루프 최적화(loop optimization), 데드 코드 제거(dead code elimination), 후진 복사 전파(backward copy propagation)등이 있다.

하나의 기본블럭내에서 최적화가 가능한 것을 지역 최적화(local optimization)이라 하고, 기본블럭간의 정보를 이용하여 최적화를 수행하는 것을 전역 최적화(global optimization)라 한다. 지역 최적화로는 상수 중첩, 대수적 변환(algebraic transformation) 등이 있다.

각각의 최적화기들은 독립적으로 수행하지만 하나의 최적화가 수행된 후, 그 영향으로 다른 최적화가 수행할 것이 발생하게 된다. 각 최적화 기법 사이의 관계를 살펴보면 분기문 최적화와 도달 불가능 기본블럭 제거, 후진 복사 전파는 다른 최적화에 영향을 주지 않으며, 상수 접기와 공통 수식 제거, 루프 최적화는 복사 전파의 수행이 필요하다. 또한, 데드 코드 제거 및 루프 최적화는 분기문 최적화 수행이 필요하고, 복사 전파와 루프 최적화는 데드 코드 제거의 수행을 필요로 한다. 복사 전파의 경우 상수 중첩의 최적화를 필요로 한다. 이에 해당하는 영향을 표 3에 나타내었다.

표 3. 각 최적화 기법들의 영향
Table 3. Relations of optimizers

최 적 화	필요한 최적화
분기문 최적화	영향 없음
도달불가능한 기본블럭 제거	영향 없음
후진 복사 전파	영향 없음
상수 접기	복사 전파
강도 축약	영향 없음
데드 코드 제거	분기문 최적화, 데드 코드 제거
복사 전파	데드 코드 제거, 상수 접기, 공통 수식 제거
공통 수식 제거	복사 전파
루프 최적화	복사 전파, 데드 코드 제거, 분기문 최적화, 도달불가능한 기본블럭 제거

4.1 분기문 최적화

분기문 최적화는 연속된 분기(jump to jump), 연속된 레이블(label sequence), 도달 불가능한 레이블(unreachable label)의 처리로 분류된다. 연속된 분기는 프로그램에서 분기되어 도달한 곳의 코드가 또 분기문인 경우를 의미한다. 연속된 분기는 무조건 분기후 무조건 분기, 조건 분기후 무조건 분기, 무조건 분기후 조건 분기, 조건 분기후 조건 분기의 4종류로 분류된다. 분기문이 연속되는 경우 프로그램은 동일한 의미를 가지는 하나의 분기문으로 대치될 수 있다. 이중 무조건 분기후 무조건 분기의 예는 표 4와 같다.

표 4. 연속된 분기의 예
Table 4. Example of jump-to-jump branch

대치 전	대치 후	도달 불가능 레이블 제거 후
...	...	...
(GOTO, 100)	(GOTO, 200)	(GOTO, 200)
...	...	...
(LABEL, 100)	(LABEL, 100)	
(GOTO, 200)	(GOTO, 200)	(GOTO, 200)
...	...	...
(LABEL, 200)	(LABEL, 200)	(LABEL, 200)
...	...	...

4.2 도달 불가능한 블럭의 제거

이 최적화는 도달 불가능한 블럭을 제거하여 다른 모든 최적화가 빠른 시간내에 이루어질 수 있게 하는 것이 목적이다. 여기서 도달 불가능한 블럭은 시작 노드로부터 도달할 수 없는 노드의 블럭과 조건문이 항상 참, 또는 항상 거짓일 때 도달할 수 없는 노드의 블럭으로 구분한다. 시작노드로부터 도달 불가능한 블럭의 제거는 전체 최적화에 있어 맨 처음에 수행이 되며, 시작 노드에서 깊이 우선 탐색을 하여 도달하지 못하는 노드는 도달

불가능한 블럭이므로 제거한다. 예를 표 5에 나타내었다.

표 5. 도달 불가능 블럭 제거의 예
Table 5. Example of elimination of unreachable block

수행 전	수행 후
(IF, 1, 10)	(GOTO, 10)
(ADDI, $T1, 4, $T2) (SUBI, $T2, 1, $T3)	도달 불가능 블럭
(LABEL, 10) (MULI, $T3, 4, $T4)	(LABEL, 10) (MULI, $T3, 4, $T4)
(IFNOT, $T1, 10, 5)	(IFNOT, $T1, 10, 5)

4.3 지역 최적화

지역 최적화는 기본블럭 단위로 최적화를 수행하는 것으로 전역적인 흐름 분석 없이도 최적화가 가능하다. 구현된 지역 최적화 기법으로 상수 중첩(constant folding), 강도 축약(reduction in strength), 대수적 변환(algebraic transformation)이 있고, 추가적으로 단항 MINUS 연산, 형 변환, 비교 연산 등을 상수 중첩 단계에서 처리하고 있다. 지역 최적화기는 복잡한 연산을 단순하고 빠른 연산으로 대체함으로써 코드를 개선시킨다.

상수 중첩이란 수식의 피연산자가 상수이거나 또는 상수값을 갖고 사칙연산 또는 SHIFT 연산일 경우 컴파일 시간에 계산하는 기법이다. 예를 들어 (MULI, 1, 4, T3)라는 수식에 대해 컴파일 시간에 (COPY, 4, T3)라는 수식으로 대치해 준다. 그리고, MINUSI, MINUSF 연산자일 경우 컴파일 시간에 첫번째 피연산자에 -1을 곱한 후 연산자를 COPY 연산자로 바꾸어준다. 또한, 각종 형변환 연산을 컴파일 시간에 직접 형변환하여 COPY연산자로 대치한다. 끝으로, 연산자가 비교 연산자일 경우에 컴파일시 계산하여 부울 값을 첫번째 피연산자에 직접 할당한다.

강도 축약이란 연산 비용이 비싼 연산자 대신 가능한 경우에 한해서 같은 동작을 수행하면서 비용이 적은 연산자로 대치해 주는 기법을 의미한다. 예를 들어, (MULI, T1, 2, T2)와 같은 수식이 컴파일 시간에 존재하게 되면 (SHL, T1, 1, T2)의 수식으로 대치함으로써 강도 축약을 수행한다. 이렇게 함으로써 수행되는 연산을 보다 빠르고, 비용이 적은 형태로 변한 가능하게 된다.

대수적 변환의 한계는 정의할 수 없을 정도로 많다. 그러나 몇가지 대수적 항등식은 단순화 과정을 통해서 최적화 효과를 높일 수 있다. 예를 들어, (ADDI, T1, 0, T2) 또는 (MULI, T2, 1, T3)와 같은 수식이 컴파일 시간에 존재하게 되면, 변수에 0을 더하거나 1을 곱하는 것은 무의미하므로 (COPY, T1, T2), (COPY, T2, T3)의 수식으로 바꾸어 준다.

4.4 공통 수식 제거

공통 수식 제거란 수식 계산이 중복되어 있는 경우에, 한번만 수식을 계산하고 나머지 중복된 계산을 이미 계산된 수식 값으로 대체시켜 수식의 중복 계산을 제거하려는 것이 목적이다. 어떠한 수식 E가 있을 때, 만일 이 수식이 이미 계산되었고 수식 E의 모든 피연산자의 값이 변경되지 않았다면 공통수식으로 취급한다. 이미 계산된 수식의 값을 이용할 수 있다면 다시 계산하는 것을 피하려 한다. 예를 들어 표 6의 (a)와 같은 기본블럭에서, 문장 S_2와 S_4는 같은 수식을 계산하는 공통 수식이다. 따라서, 위의 문장들은 표 6의 (b)와 같이 변형될 수 있다.

표 6. 지역 공통 수식 제거의 예
Table 6. Example of local common subexpression elimination

(a)	(b)
S_1 : (ADDI, a, b, c)	S_1 : (ADDI, a, b, c)
S_2 : (SUBI, a, d, b)	S_2 : (SUBI, a, d, b)
S_3 : (ADDI, a, b, e)	S_3 : (ADDI, a, b, e)
S_4 : (SUBI, a, d, d)	S_4 : (COPY, b, d)

S_1과 S_3는 공통 수식으로 보이나, 공통 수식이 될 수 없다. 이유는 S_1과 S_3사이에 b의 값이 재정의(redefinition)되었기 때문이다. 따라서 이 수식은 변형될 수 없다. 이렇게 기본블럭 단위로 공통 수식

제거를 수행하는 것을 지역 최적화인 지역 공통 수식 제거(local common subexpression elimination)라 한다. 전역 공통 수식 제거(global common subexpression elimination)는 기본블럭 내에 발생한 수식이 다른 기본블럭에 중복되어 발생하였을 때 수행되는 것으로 기본블럭 간의 공통 수식을 계산하여 제거한다.

전역 공통 수식 제거의 예로써 그림 4와 같은 기본블럭들이 있을 때, 기본블럭 B_2의 문장 (ADDI, a, b, c), (SUBI, d, e, f)와 (EQ, c, f, g)는 공통 수식이다. B_4의 (SUBI, d, e, f)는 공통 수식이지만, 기본블럭 B_3에서 b가 재정의되었기 때문에 (ADDI, a, b, c)는 공통 수식이 아니다.

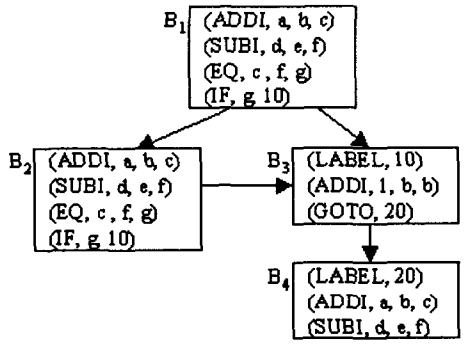

그림 4. 전역 공통 수식 제거의 예

Fig. 4. Example of global common subexpression elimination

전역 공통 수식 제거를 수행하려면 다음의 이용가능한 수식들을 계산한 이후에 공통수식 제거를 수행한다. 어느 한 점에서 수식 (ADDI, x, y, z)가 이용 가능(available)하다는 것은 초기 문장으로부터 그 점까지의 모든 경로에서 (ADDI, x, y, z)의 계산이 존재하고, 수식 (ADDI, x, y, z) 이후의 문장에서부터 그점까지의 모든 경로내의 문장들이 x나 y에 값을 할당하지 않는다는 의미이다. 이용가능한 수식에 대해, 기본블럭 내에서 x 또는 y에 값을 할당하면 그 블럭에서 수식 (ADDI, x, y, z)가 이용 불가능하게 된다(killed)고 하고, 기본블럭 내에서 x나 y에 값을 할당하지 않으면 수식 (ADDI, x, y, z)를 생성한다(generate)고 한다. 이용가능한 수식의 계산은 도달 정의를 구하는 규칙과 같다.

4.5 복사 전파

복사 전파란 복사문(copy statement)의 목적(target) 심볼을 뒤따르는 실행 문장의 피연산자로 전파시키는 최적화 기법이다. 예를 들면, 표 7의 (a)와 같은 문장을 (c)와 같은 문장으로 변형함으로써, 중간 코드 크기를 감소시키고 실행 속도를 증가시킨다.

표 7. 지역 복사 전파의 예

Table 7. Example of local copy propagation

(a)	(b)	(c)
(COPY, a, b) (ADDI, b, c, d)	(COPY, a, b) (ADDI, a, c, d)	(ADDI, a, c, d)

기본블럭(basic block) 내에서 복사 전파를 수행하는 것을 지역 복사 전파(local copy propagation)라 하고, 여러 기본블럭간에 복사 전파를 수행하는 것을 전역 복사 전파(global copy propagation)라 한다. 지역 복사 전파는 한 기본블럭 내에서만 복사 전파를 수행하는 것이다. 표 7 (b)에서 심볼 b의 사용(use)이 더 이상 존재하지 않으면, 복사 전파 수행이후에 데드 코드 제거를 수행하면 (c)와 같이 COPY 문이 제거될 수 있다.

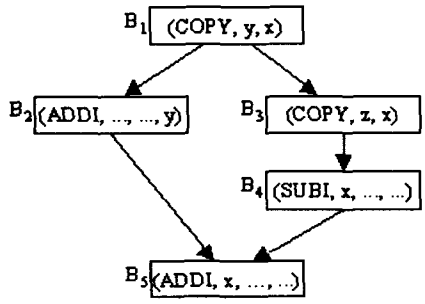

그림 5. 흐름 그래프 예

Fig. 5. Example of flow graph

$c_{gen}[B_1] = \{ (COPY,y,x) \}$
$c_{gen}[B_2] = \emptyset$
$c_{gen}[B_3] = \{ (COPY,z,x) \}$
$c_{gen}[B_4] = \emptyset$
$c_{gen}[B_5] = \emptyset$
$c_{kill}[B_1] = \{ (COPY,z,x) \}$
$c_{kill}[B_2] = \{ (COPY,y,x) \}$
$c_{kill}[B_3] = \{ (COPY,z,x) \}$
$in[B_1] = \emptyset$
$in[B_2] = in[B_3] = out[B_1] = \{ (COPY,y,x) \}$
$out[B_1] = \emptyset$
$out[B_3] = in[B_4] = out[B_4] = \{ (COPY,z,x) \}$
$in[B_5] = out[B_2] \cap out[B_4] = \emptyset$

문장 s의 (COPY, x, y)와 y를 사용하는 문장 u에 대해 복사 전파를 수행하려면, 문장 s는 y를 사용하는 문장 u에 도달하는 단 하나의 정의이어야 하고, 문장 s로부터 u에 이르는 모든 경로에 y에 값을 할당하는 문장이 없어야 한다. 이 방정식은 공통 수식 제거를 위한 이용가능 수식계산의 방정식과 같다. 예를 들어, 그림 5와 같은 흐름 그래프가 있을 때, 각 기본블럭의 정보를 구하면 다음과 같다.

4.6 데드 코드 제거

프로그램 내의 문장에서 변수의 값이 정의되고 그 변수의 값이 이후에 사용되면, 그 변수는 생존하다고 한다. 만일 사용되지 않으면, 그 변수는 데드(dead)하다고 한다. 데드되는 변수가 존재하는 문장을 데드 코드라 하고, 프로그램의 의미를 변경시키지 않고 제거될 수 있다. 데드 코드는 프로그래머가 고의로 작성하거나, 최적화의 프로그램 변형에 의해 나타난다. 예를 들어 표 8 (a)의 중간코드를 살펴보면, 변수 a는 문장 S_2에서 정의되고 문장 S_4에서 사용되어 생존하고, 변수 b는 문장 S_3에서 정의되고 문장 S_4에서 사용되어 생존한다. 하지만 문장 S_4에서 정의된 변수 c는 이후에 사용되지 않으므로 데드하다. 따라서 문장 S_4는 제거될 수 있다. 또, 문장 S_4가 제거되면 문장 S_2와 S_3에서 정의된 변수 a와 b는 데드하므로 문장 S_2와 S_3는 데드 코드가 된다. 따라서 데드 코드 제거이후 결과는 (b)와 같다.

표 8. 데드 코드 제거의 예
Table 8. Example of dead code elimination

(a)	(b)
S_1 : (FUN, f, 0) S_2 : (COPY, 1, a) S_3 : (COPY, 2, b) S_4 : (ADDI, a, b, c) S_5 : (END, f)	S_1 : (FUN, f, 0) S_5 : (END, f)

이러한 최적화를 위해서는 데이타 흐름 분석에 의존하는데, 그 중 변수 정의에 대한 변수 사용의 정보인 DU-체인을 이용한다.

4.7 후진 복사 전파(Backward copy propagation)

전단부로부터 생성되는 임시변수의 수가 많고 임시변수를 원시프로그램에서 작성한 심볼로 복사하는 튜플이 많아 이 복사문을 제거하는 최적화로서 UD-체인과 DU-체인 정보를 이용한다.

표 9. 후진 복사 전파의 예
Table 9. Example of backward copy propagation

(a)	(b)	(c)
S_1: (COPY, a, T_1) S_2: (COPY, b, T_2) S_3: (ADDI, T_1, T_2, T_3) S_4: (COPY, T_3, c)	S_1: (COPY, a, T_1) S_2: (COPY, b, T_2) S_3: (ADDI, T_1, T_2, c)	S_3: (ADDI, a, b, c)

예를들어, 표 9 (a)의 튜플들을 살펴보자. 여기서 사용된 심볼 T_n은 임시변수이다. 전단부에서 생성하는 임시변수는 단일 정의만 있으므로 임시변수에 대한 UD-체인은 유일하다. 이에 따라 문장 S_3의 임시변수 T_3를 c로 대체시키고 문장 S_4를 제거하려는 것이 아이디어이다. 문장 S_4의 T_3에 대한 UD-체인을 구하면 $\{S_3\}$가 되고, UD-체인에 해당하는 T_3의 DU-체인을 구하면 $\{S_4\}$가 된다. 다음에 S3의 정의 심볼을 c로 대체시키고 DU-체인의 문장들에 있는 튜플들의 T_3를 모두 c로 대체시킨다. 결과적으로 (b)와 튜플들이 생성된다. 이후에 복사 전파를 수행하면 (b)의 튜플들은 (c)와 같이 최적화된다.

이 최적화는 전역 또는 지역의 구분없이 각 튜플단위로 실행하고, 이 최적화의 영향은 루프 최적화에서 인덕션 변수를 찾는데 유용하다.

4.8 루프 최적화

일반적으로 프로그램 실행시간의 대부분은 내포된 루프들에서 소비되므로 루프 최적화는 전체 최적화 중에서 매우 중요한 위치를 차지한다. 그러므로 루프 밖의 명령어의 수가 증가한다 할지라도 일단 루프 내의 명령어 수를 줄일 수만 있다면 프로그램의 실행 시간을 향상시킬 수가 있다. 이러한

루프 최적화의 방법들에는 여러 가지가 있으며 그 중 중요한 코드 이동과 인덕션 변수 제거, 강도 축약의 기법을 구현하였다.

코드 이동은 루프내의 코드 수를 줄일 수 있는 변환 기법으로써 루프의 실행 횟수와 무관하게 항상 같은 값을 가지는 코드(이를 loop-invariant라 한다)를 선택하여 루프 위로 이동시킨다. 인덕션(induction) 변수란 루프 내에서 일정한 상수로 증가 또는 감소되는 변수로서 주로 루프의 종료 검사나 배열의 옵셋 계산에 사용된다. 루프 내에 인덕션 변수가 둘 이상 존재할 경우 하나를 제외한 모든 인덕션 변수들을 제거할 수 있는데 이러한 변환을 인덕션 변수 제거라고 한다. 인덕션 변수와 관련된 다른 변환은 강도 축약(reduction in strength)으로서 인덕션 변수를 이용하여 계산하는 값비싼 연산자를 보다 비용이 싼 연산자로 대치시킨다.

루프 최적화에서는 먼저 튜플들을 탐색하면서 자연 루프를 찾아내며 이를 실제적인 최적화 수행 단계에서 쉽게 이용할 수 있도록 내포순, 크기순 그리고 프로그램상의 루프의 위치순에 따라 정렬시킨다. 루프 탐색 시 변경된 정보로 인하여 다시 한번 자료 흐름 분석 작업이 필요하며 이 작업 후 인배리언트 탐색에 들어가게 된다. 정렬되어 넘겨지는 루프를 처음부터 하나씩 살펴가는데 이때 루프 내의 모든 튜플을 한번만 탐색하면 가능한 모든 인배리언트를 찾아낼 수 있다. 일단 모든 루프에 대해서 인배리언트 탐색이 끝나면 각 루프의 프리헤더로 탐색한 인배리언트를 이동시킨다. 이때 튜플 이동으로 인하여 튜플 테이블 내의 튜플 위치가 바뀌었으므로 이전에 탐색해 놓은 인배리언트 정보와 루프 정보를 새로이 변경시켜주어야 한다.

이렇게 인배리언트 탐색 및 이동 작업이 끝나면 튜플 위치와 블럭정보가 바뀌었으므로 다시 한번 자료 흐름 분석을 한 뒤 다음 단계로 넘겨진다.

인덕션 변수의 탐색 단계에서는 먼저 하나의 기본 인덕션 변수를 찾은 뒤 이 변수의 모든 가족(family)들을 깊이 우선 탐색 방법으로 찾아낸다. 즉 하나의 기본 인덕션 변수를 찾으면 그 변수의 가족이 더 이상 발견되지 않을 때까지 DU-체인을 따라가며 탐색한다. 이렇게 하여 하나의 기본 인덕션 변수에 대한 모든 가족을 찾을 수 있다. 강도 축약에서는 인덕션 변수를 사용하는 튜플 중에 곱하기 연산자가 있으면 이를 보다 싼 연산자인 복사문으로 바꾼다. 이 단계가 끝나면 기본 인덕션 변수 중에는 단순히 루프 종료 검사에만 쓰이는 인덕션 변수가 생길 수 있는데 이 변수를 가족 중의 한 변수로 대치시킴으로써 제거시킬 수 있다.

V. 테스트 및 성능분석

5.1 테스트

컴파일러 전체의 성능분석을 위해서는 6개의 benchmark프로그램을 사용하였다. 이 프로그램들은 가우스소거법을 이용하여 연립 방정식을 풀고 역행렬을 구하는 수학적 연산이 많은 프로그램, 고급 자료구조인 Deap에서 삽입하고 삭제하는 알고리즘을 수행하는 프로그램, NP-hard 문제인 0/1 Knapsack 문제를 다이나믹 프로그래밍 기법을 사용해서 해결하는 프로그램들로 구성되어 있다. 성능 분석은 각 최적화 단계가 중간코드의 크기에 어떤 영향을 미치는가에 중점을 두고 분석하였다.

분기문 최적화는 사용자의 실수로 만들어진 코드에 대해 행해지는 경우도 있지만, 주로 도달 불가능한 코드의 제거등과 같은 다른 최적화 기법들에 의해 코드가 삭제되었을 경우에 실행되게 된다. 예를 들면, 기존에 어떤 레이블(lable)과 프로그램 코드가 존재하였으나 데드 코드(dead code)의 제거 후에 모든 코드들이 삭제되어 바로 다음 레이블로 점프해야하는 경우 등이 발생하게 된다. 이 때 불필요하게 되는 중간의 레이블이 삭제되는가를 테스트하였다. C언어 상의 루프문도 중간코드에서는 특정 레이블로의 분기문에 해당하기 때문에 루프 내의 코드들이 모두 데드코드로 제거된 후 불필요한 루프문의 삭제도 테스트하였다.

도달 불가능한 코드 역시 사용자의 실수에 의해 발생하는 경우보다 다른 여타의 최적화 과정 중에서 발생되는 것이 일반적이다. 도달 불가능한 코드는 주로 상수 중첩 후에 발생할 수 있다. 'if'문이나 'while'문 등의 수식문(expression)이 상수 중첩(constant folding)에 의해 항상 거짓 값이 되는 경우가 생길 수 있다. 이런 경우에 if나 while등에 의해서 수행될 문장들(statements)은 제거되어야 한다.

복사 전파는 불필요한 복사문의 제거가 이루어지는 지의 여부와 여러 단계에 걸쳐서 일어나는 복사 전파의 경우에는 어떻게 처리되는가를 중심으로 테스트되었다. 또한 값이 변화된 변수에 대해서도 복사 전파를 행하는지 여부 등도 테스트의 주요 대상이 되었다. 특히 복사 전파는 단독적인 테스트보다는 다른 최적화 과정들과 같이 하는 테스트를 위주로 행해졌다. 예를 들면, 데드 코드의 삭제등이 일어난 후에 필요하게 되는 복사 전파가 이루어지는 지의 여부 그리고 도달 불가능한 코드 내에 그 값이 변하는 변수에 대해서, 도달 불가능한 코드의 제거 후에 다시 복사 전파가 이루어지는지 등도 테스트되었다.

상수 중첩은 가능한 모든 수식의 계산을 컴파일 시간에 행하는가, 그리고 그 범위는 어떻게 되는가 등을 주요 대상으로 테스트하였다. 어떤 변수가 한 번의 상수 중첩에 의해서 상수로 대체되었다면, 그와 관련된 다른 계산과 그 결과 또한 상수 중첩의 대상이 되는가의 여부 그리고 상수 중첩의 대상이 된다면, 몇 단계까지 상수 중첩이 이루어지는가가 주요 테스트 대상이 되었다. 상수만을 반환하는 함수에 의해, 변수의 값이 정의될 때 이 변수도 상수 중첩에 의해 제거되는지의 여부등도 테스트되었다.

데드 코드의 제거는, 사용되지 않는 변수의 정의가 제거되는가와 같은 아주 기본적인 테스트로부터 다른 최적화 기법들에 의해 만들어지는 데드 코드도 제거되는가 등이 테스트의 주요 대상이었으며, 최종 출력문(critical statement)으로부터 역방향으로 추적해서 사용되지 않는 모든 문장(주로 변수의 정의)을 데드 코드로 인정하고, 그 문장들을 제거하는지 여부를 테스트하였다. 또한 독립적인 프로그램에 불필요한 코드를 임의적으로 삽입하므로써, 데드코드의 제거가 실제 프로그램에서 어느 정도까지 가능한지도 테스트하였다. 데드 코드의 제거는 그 순서가 앞설 수록 다른 최적화 과정에 좋은 영향을 미칠 수 있으므로, 전체 최적화 과정 중 몇 번째 스텝에 이루어지는지, 그리고 몇번에 걸쳐서 이루어지는지도 테스트하였다.

공통 수식 제거는 각 문장(statement)단위로 테스트하는 것을 기초로 하여, 각 문장의 일부가 공통 수식으로 취급되어 삭제되는지 여부, 그리고 수식에 포함되어 있는 변수의 값이 변한 경우에는 어느 부분까지를 공통 수식으로 처리하여 중복을 제거하고 대신 사용하는가 하는 것이 주요 테스트 대상이었다.

루프 최적화는 독립적으로 테스트하기도 하였지만, 주로 앞의 다른 최적화과정들과 복합적으로 테스트하는 것을 기본으로 하였다. 코드의 이동은 주로 그 루프와는 무관한 수식이나 문장(loop-invariant expression)을 루프 안에 삽입하여 동일한 값의 계산이 여러번 반복시켜 그 문장이나 수식이 루프 앞이나 뒤의 적당한 곳으로 이동되는지 여부를 테스트하였다. 다중 내재 루프(multiple nested loop)의 경우에 루프와 무관한 수식과 문장은 어떤 범위까지 이동이 가능한지의 여부와 만약 모든 코드가 루프 밖으로 이동되었거나, 데드코드로 판단되어 제거되었을 경우에 그 루프 자체가 삭제되는가 여부 또한 테스트되었다. 다른 최적화 과정의 테스트와 마찬가지로, 루프 내의 무관한 코드의 이동도 여타 다른 최적화 과정들과 혼합되어 테스트되었다. 예를 들면, 상수 중첩에 의해 상수화 되거나 복사 전파에 의해 루프와 무관한 수식문이 되었을 경우에 루프 앞과 뒤의 적당한 곳으로 이동이 되었는지, 아니면 제거되었는지 여부등도 테스트되었다. 인덕션 변수 제거는 루프의 반복에 따라 계속적으로 값이 변화되는 인덕션 변수나 인덕션 문장(induction statement)의 경우에는 각각 얼마만큼의 변수나 문장만이 남는지가 주요 테스트 대상이었다. 강도 축약은 어떤 수식문에 대해서, 수학적으로 동일하며 비용(cost)이 적은 수식문으로 대체되는가를 테스트하였다. 특히 곱하기 연산과 나누기 연산이 그 주요 대상이었다. 곱하기 연산의 경우에 곱해지는 수가 2의 제곱수일 경우에는 왼쪽 쉬프트로 변환이 되는가, 2의 제곱수가 아닐 경우에 더하기 연산으로 대체되는가 등이 주요 테스트 대상이었으며, 나누기 연산의 경우도 마찬가지로 각각 오른쪽 쉬프트와 빼기 연산으로 대체되는가를 테스트하였다. 강도 축약은 루프 내에서의 수행 속도를 효율적으로 줄일 수 있으므로, 상수 중첩이나 복사 전파(copy propagation)등의 다른 최적화 과정이 수행된 후에 발생할 수 있는 경우도 테스트하였다. 또한 루프내에 존재하는 곱하기 연산이 동일한 값을 계산하는 여러 개의 더하기 연산으로

표 10. 최적화기에 의해 감소된 튜플 수
Table 10. Reduced tuples by optimizers

programs / optimizations		deap.c	gauss.c	knap.c	minicalc.c	scanner.c	symtab.c	total	rate
Before optimizations		2,259	1,439	1,193	1,468	1,554	4,109	12,022	100.00%
Jump-to-jump optimization	Reduced	75	191	103	66	286	244	965	-8.03%
Unreachable block elimination	Reduced	19	1	3	10	30	53	116	-0.96%
Backward copy propagation	Reduced	71	53	79	16	103	178	500	-4.16%
Dead code elimination	Reduced	8	7	8	6	1	12	42	-0.35%
Copy propagation	Updated	255	264	236	62	180	800	1,797	14.95%
	Reduced	240	256	230	57	158	766	1,707	-14.20%
Constant folding	Updated	3	1	6	0	3	252	265	2.20%
Copy propagation	Updated	3	1	6	0	2	251	263	2.19%
	Reduced	3	1	6	0	2	251	263	-2.19%
Algebraic transformation	Updated	0	1	1	2	9	33	46	0.38%
Copy propagation	Updated	0	1	1	2	18	42	64	0.53%
	Reduced	0	1	1	2	9	33	46	-0.38%
Common subexpression elimination	Updated	40	0	9	0	4	19	72	0.60%
	Reduced	-16	0	-5	0	-2	-9	-32	+0.27%
Copy propagation	Updated	58	0	16	0	4	21	99	0.82%
	Reduced	54	0	14	0	4	21	93	-0.77%
Loop optimization	Reduced	190	61	107	57	138	443	996	-8.28%
Jump-to-jump optimization (after loop optimization)	Reduced	3	0	4	1	6	8	22	-0.18%
Unreachable block elimination (after loop optimization)	Reduced	1	0	3	0	3	6	13	-0.11%
After optimizations		1,611	868	640	1,253	816	2,103	7,291	60.65%

대체되는 가도 테스트하였다. 이 경우는 루프의 반복수가 적은 경우와 많은 경우로 나누어서 테스트하였다. 대수적 변환은, 루프와 관계없는 다른 영역에서도 테스트되었지만, 특히 루프내에서 그 중요성을 가지므로 루프 내에 있는 수식문에 대해서 테스트되었다. 이미 증명된 수학적 정의를 적절히 이용하여 루프내에서의 연산을 얼마만큼 줄었는가가 주요 테스트 대상이 되었다.

5.2 성능 분석

6개의 benchmark 프로그램을 수행했을 때 각 최적화 단계별로 감소된 중간 코드 튜플의 수를 표 10에 나타내었다. Symtab.c의 경우 전체 튜플의 수는 4109에서 2103으로 49%가량 감소하였고, 6개 프로그램의 튜플수의 평균 감소율은 39.35%이었다.

몇몇 최적화별로 감소율을 살펴보면 분기문 최적화는 8.03% 감소되었고 후진 복사 전파는 4.16% 감소, 복사 전파는 14.95% 변경에 14.20% 감소, 루프 최적화는 8.28% 감소되었다. 따라서, 필수적인 최적화는 복사 전파와 루프 최적화, 분기문 최적화라 할 수 있다. 공통 수식 제거는 프로그래머가 고의적으로 공통 수식이 있도록 작성하지 않는 한 적은 감소율을 보였다. 하지만, 제어 흐름 분석에 의한 최적화는 구현하기 쉽고 실행 시간도 적은 반면, 흐름 분석에 의한 최적화는 많은 구현 시간과 실행 시간을 요구한다. 특히 루프 최적화가 가장 많은 시간을 요구한다. 따라서 프로그램의 실행 시간을 단축시키기 위해서는 컴파일 시간이 다소 증가하더라도 최적화 단계를 수행하는 것이 필요하다는 것을 알 수 있고, 최적화기 구현 시간에 따른 최적화기 선택을 할 수 있다. 표 11에는 각 테스트 프로그램의 컴파일 시간과 실행 시간을 나타내었다. 나타난 바와 같이 최적화를 수행할수록 컴파일 시간은 더 소요되지만 기계코드의 실행 시간은 매우 감소된다.

표 11. 테스트 프로그램의 컴파일 시간과 실행 시간
Table 11. Compile time and execution time of test programs

화일 이름	컴파일 시간 (단위: 초)				실행 시간 (단위:1/60초)			
옵션	-O0	-O1	-O2	-O3	-O0	-O1	-O2	-O3
deap.c	9.6	9.6	35.8	36.6	13	13	12	12
gauss.c	4.4	4.4	15.9	16.8	11	10	8	6
knap.c	4.4	4.4	20.4	21.5	117	116	88	87
minicalc.c	1.9	1.8	4.0	4.0	5	4	3	2
scanner.c	6.3	6.3	36.2	37.4	8	7	6	4
symtab.c	9.2	16.6	89.5	90.8	7	5	4	3

-O0 : 최적화가 없다.
-O1 : 분기문 최적화와 도달 불가능한 블럭 제거 최적화
-O2 : -O1과 복사 전파, 데드 코드 제거, 후진 복사전파, 공통 수식 제거.
-O3 : -O2와 루프 최적화

Ⅵ. 결 론

본 연구에서는 중간 코드 상에서의 기계 독립적인 최적화로써 분기문 최적화와 도달하지 않는 기본블럭 제거, 루프 최적화, 복사 전파, 상수 중첩, 강도 축약, 대수적 변환, 데드 코드 제거, 중복 수식 제거 등의 최적화기들을 구현하였다. 화일 단위로 최적화를 수행하면 자료 흐름 분석 등에서 기억장소를 많이 차지하게 되어 함수 단위로 최적화를 수행하게 하였다. 함수 단위로 정보를 유지하게 하면 함수간의 흐름 정보를 유지하기 어려운 반면에 적은 기억장소를 사용하게 되어 기억장소 사용면에서 효율적이다. 하지만 함수 단위로 수행을 하더라도 문제점은 있다. 즉, 하나의 함수가 매우 커지면 하나의 함수에 대한 정보를 저장하기 위한 기억장소가 매우 커지게 된다는 점이다. 하나의 함수내의 기본블럭들의 수가 매우 크면 정보를 저장하기 위한 기억장소 사용이 많이 필요하게 되고, 정보 계산 시간도 길어지게 된다. 따라서 하나의 함수 내에 기본블럭의 갯수가 500을 넘으면 자료 흐름 분석 및 최적화들을 수행하지 않도록 하였다. 기본블럭의 갯수가 크면 오히려 실행시간과 기억장소를 많이 소요하므로 노력에 비해 얻어지는 효과가 너무 작기 때문이다.

테스트 대상 프로그램들은 ANSI C의 참고 메뉴얼에 언급되어 있는 내용에 부합하도록 각각의 구문에 따라 테스트 프로그램을 설정하였고, 여러가지 구문을 조합하여 구문간의 관계에 따른 테스트도 가능하게 하였으며, 다양한 수치 해석 프로그램과 자료 구조 프로그램, 그리고 이미 알려져 있는 프로그램에 대하여 테스트를 수행하여 전체 600여개에 대한 테스트를 시행하였다.

분기문 최적화와 도달 불가능한 블록의 제거는 매우 적은 시간으로 높은 최적화를 얻을 수 있고, 후진 복사 전파와 복사 전파는 자료 흐름 분석 정보로부터 매우 높은 최적화를 얻었으며, 루프 최적화는 개발 시간 및 실행 시간도 길고 최적화 효과도 높다. 컴파일러 개발 초기에 적은 비용으로 큰 효과를 얻을 수 있는 최적화기를 먼저 개발하고, 그 결과에 대해 더 최적화가 필요한 경우에 최적화기를 개발하는 것이 시간이나 노력 낭비를 하지 않게 된다. 다시 말해, 개발 초기에 최적화 기법 개발 순서를 경제성을 고려하여 설정하여야 한다.

결론적으로, 본 연구에서는 컴파일러에서 필요한 최적화 기술을 개발하였다. 다수의 기존 최적화 기법들을 개선, 구현하였으며, 각 기법의 성능-비용 관계를 시험하여 기술적 자료와 경험을 축적하였다. 축적된 기술과 경험은 컴파일러뿐만 아니라, 여타의 대규모 소프트웨어 개발에 큰 도움이 될 것이다.

참고 문헌

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, *"Compilers : Principles, Techniques and Tools"*, Addison-Wesley, 1986.

[2] M. Burke. *"An interval-based approach to exhaustive and incremental interprocedural dataflow analysis"*, ACM Transactions on Programming Languages and Systems, 12(3):341-395, July 1990.

[3] Charles N.Fischer, Richard J.LeBlane, Jr, *"Crafting A Compiler"*, The Benjamin/Cummings Publishing Company, Inc., 1988.

[4] Michael E. Wolf, Monica S. Lam, *"A Data Locality Optimizing Algorithm"*, ACM SIGPLAN Notices, 1991.

[5] Hans Zima, Barbara Chapman, *"Supercompilers for Parallel and Vector Computers"*, ACM Press, 1991.

송 진 국(Jin-Kook Song)

1988년 2월 홍익대학교 전자계산학과 졸업

1990년 2월 홍익대학교 대학원 전자계산학과 졸업 (이학석사)

1998년 2월 홍익대학교 대학원 전자계산학과 졸업 (이학박사)

현재 국립 진주산업대학교 전자계산학과 교수

*관심연구분야 : 프로그래밍언어, 컴파일러, 시스템 소프트웨어