DFA 패턴 매칭을 위한 코드 최적화기의 자동적 생성

윤 성 림[†] · 오 세 만[††]

요 약

주어진 입력 프로그램과 의미적으로 동등하면서 좀 더 효율적인 코드로 바꾸는 것을 코드 최적화라 하며, 이런 과정은 코드 최적화기에 의해 수행된다. 본 논문에서는 코드 최적화기를 자동적으로 생성하는 도구인 코드 최적화기 생성기를 설계하고 구현하였다. 즉, 패턴 형식에 대한 표현을 입력으로 받아 기술된 형태의 최적화 코드를 찾아내는 DFA 패턴 매칭을 위한 코드 최적화기를 자동적으로 생성하는 것이다. DFA 패턴 매칭은 패턴들의 정규화 과정을 통해 패턴 검색 시 발생하는 중복 비교를 제거하여, 패턴 형태의 단순화 및 구조를 개선함으로 비용이 적게 든다. DFA 패턴 매칭을 위한 코드 최적화기의 자동적 생성은 다양한 형태의 중간코드로 바뀌더라도 해당하는 코드 최적화기를 만들어야 하는 수고를 덜어줌으로써 코드 최적화에 대한 정형화(formalism)를 할 수 있다. 또한, DFA로 구성되어 최적화를 하기 때문에 최적화 속도가 빠르고, 코드 최적화기를 만드는데 필요한 시간과 비용을 절약할 수 있는 장점을 가진다.

키워드 : 결정적 유한 오토마타, 코드 최적화기 생성기, 코드 최적화기, 패턴 매칭, 패턴 형식, DFA 정규화

Automatic Generation of Code Optimizer for DFA Pattern Matching

Sunglim Yun[†] · Seman Oh[††]

ABSTRACT

Code Optimization is converting to a code that is equivalent to given program but more efficient, and this process is processed in Code Optimizer. This paper designed and processed Code Optimizer Generator that automatically generates Code Optimizer. In other words, Code Optimizer is automatically generated for DFA Pattern Matching which finds the optimal code for the incoming pattern description. DFA Pattern Matching removes redundancy comparisons that occur when patterns are sought for through normalization process and improves simplification and structure of pattern shapes for low cost. Automatic generation of Code Optimization for DFA Pattern Matching eliminates extra effort to generate Code Optimizer every time the code undergoes various transformations, and enables formalism of Code Optimization. Also, the advantage of making DFA for optimization is that it is faster and saves cost of Code Optimizer Generator.

Key Words : Deterministic Finite Automata, Code Optimization Generator, Code Optimizer, Pattern Matching, Pattern Description, DFA Normalization

1. 서 론

코드 최적화란 주어진 입력 프로그램과 의미적으로 동등하면서 좀 더 효율적인 코드로 바꾸는 것을 의미한다. 따라서 코드 최적화기는 가급적 계산의 횟수를 줄이고, 보다 빠른 명령을 사용하여 실행 시간이 짧은 코드를 생성해야 하며, 기억 장소의 요구량을 최소화해야 한다. 일반적으로 최적화는 컴파일 과정에서 선택적인 단계로 필요시에만 컴파일러 옵션으로 선택하여 실행할 수 있다. 최적화를 수행하는 주요한 기준은 첫째, 반드시 입력과 의미가 동등해야 한다. 둘째, 평균적으로 속도가 빨라져야 한다. 셋째, 최적화에 들인 노력에 대한 비용이 줄어야 한다.

† 정 회 원 : 동국대학교 컴퓨터공학과 강사
†† 종신회원 : 동국대학교 컴퓨터공학과 교수(교신저자)
논문접수 : 2006년 10월 16일, 심사완료 : 2007년 1월 3일

기존의 코드 최적화 방법으로는 스트링 패턴 매칭과 트리 패턴 매칭을 이용하여 최적화를 수행하였다. 스트링 패턴 매칭[7, 8]은 중간 코드에 대응하는 최적의 패턴을 찾기 위한 방법으로 과다한 최적화 패턴 검색 시간으로 비효율적이다. 트리 패턴 매칭[2, 8]은 패턴 결정시 중복 비교가 발생할 수 있으며, 코드의 트리 구성에 많은 비용이 드는 단점을 가지고 있는 방법이다. 기존 최적화 방법들의 단점을 극복하기 위한 새로운 방법으로 오토마타(Automata)를 이용하여 패턴 매칭을 하였다. 이 방법은 다른 패턴 매칭 기법보다 결정적인 오토마타로 구성하기 때문에 결정적으로 패턴이 확정됨에 따른 패턴 선택 비용이 줄어들며, 최적화 패턴 검색 시간도 빨라지는 효율적인 방법이다.

본 논문에서는 의미적으로 동등한 효율적인 코드로 바꾸기 위한 코드 최적화 과정을 수행하는 코드 최적화기를 자동으로 생성할 수 있는 컴파일러 자동화 도구인 코드 최적

화기 생성기(Code Optimizer Generator)에 대해 제안하고자 한다. 다시 말하면, 코드 최적화기 자동 생성 도구인 COG를 구현하여 DFA 패턴 매칭을 위한 코드 최적화기를 만드는 것이다. COG는 코드 최적화기에서 필요한 패턴에 대한 모든 정보를 참조할 수 있는 DFA 테이블을 생성한다. 최적화 패턴 검색 시 DFA 최적화 알고리즘을 이용하여 최적화를 하기 때문에 최적화의 속도가 기존 최적화 알고리즘을 적용한 것보다 빠르다. DFA 패턴 매칭은 패턴들의 정규화 과정을 통해 패턴 검색 시 발생하는 중복 비교를 제거하여, 패턴 형태의 단순화 및 구조를 개선함으로 비용이 적게 든다. DFA 패턴 매칭을 위한 코드 최적화기[3]의 자동 생성의 장점으로는 각각의 중간 코드에 대한 케이스별로 코드 최적화기를 만들어야 하는 수고를 덜어줌으로써 코드 최적화에 정형화(formalism)를 할 수 있다. 또한, DFA로 최적화를 하기 때문에 최적화 속도가 빠르고, 코드 최적화기를 만드는데 필요한 시간과 비용을 절약할 수 있다. 최적화 패턴이 추가되거나 변경될 경우에 테이블만 변경되므로 확장성이 높으며, 새로운 언어에 대한 재목적성이 높은 장점을 가진다.

본 논문의 구성은 다음과 같다. 2장에서는 연구를 수행하기 위한 관련 연구로서 생성기-생성기[12, 14, 15]와 DFA 최적화에 대해 기술하고, 3장에서는 본 논문에서 제안하는 코드 최적화기 생성기를 제안한다. 4장에서는 제안한 코드 최적화기 생성기에 대한 실험을 통해 분석을 하고, 마지막으로 5장에서는 결론 및 향후 연구 과제를 서술한다.

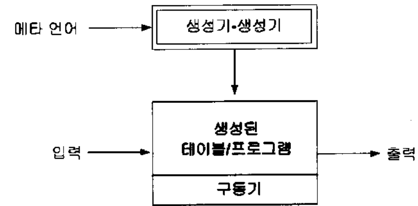
2. 관련 연구

프로그래밍 언어와 컴퓨터 구조가 다양해짐에 따라 많은 컴파일러가 필요하게 되었고, 컴파일러의 전과정이나 각 단계들을 자동적으로 생성하는 도구들에 대한 연구가 활발히 진행되고 있다. 이러한 도구중 컴파일러-컴파일러[14]는 특정한 프로그래밍 언어를 위한 언어 표현과 목적 기계에 대한 기계 표현(machine description)[11]을 입력으로 받아 하나의 컴파일러를 출력한다. 이 때 생성된 컴파일러는 언어 표현에 기술된 소스 프로그램을 입력받아 목적 기계의 코드로 번역해 주는 역할을 수행한다. 본 논문의 관련 연구로 컴파일러의 단계를 자동적으로 생성하는 도구인 생성기-생성기와 구문 분석 단계를 자동화 할 수 있는 도구인 파서-생성기에 대해 설명한다. 기존의 코드 최적화 방법으로 스트링 패턴 매칭과 트리 패턴 매칭에 대해 설명한다.

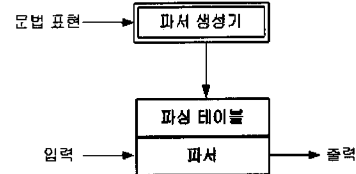
2.1 생성기-생성기(Generator-Generator)

생성기-생성기는 생성될 단계의 기능을 묘사하는 메타 언어(meta language)를 입력으로 받아 각 단계가 사용하게 될 테이블(또는 프로그램)을 출력한다. 그러면, 각 구동기(drive routine)는 이 테이블을 이용하여 그 단계에서 수행해야 할 일을 처리한다. (그림 1)은 컴파일러의 각 단계를 생성하는 자동화 도구를 기능적인 면에서 나타낸 것이다.

따라서 필요한 컴파일러 단계를 구현하기 위하여 프로그



(그림 1) 자동화 도구의 기능



(그림 2) 파서 생성기의 기능

램을 작성하는 것이 아니라 그 단계에서 처리하는 작업을 메타 언어로 기술하면 된다. 어휘 분석이나 구문 분석 단계에 대한 자동화 도구는 1970년에 나타났으나 후단부 작성을 도와주는 도구는 최근에 이르러서야 발표되기 시작하였다. 그 이유는 최적화나 코드 생성에 대한 정형화(Formalism)가 어려웠기 때문이다.

생성기-생성기의 한 종류인 파서 생성기(Parser Generator) [6, 9, 15]는 언어의 문법 표현으로부터 파서를 자동으로 생성하는 도구를 말하며, 그 기능은 (그림 2)으로 도시화 할 수 있다. 보통 파서 생성기를 파서 제작 시스템(Parser Generating System)이라 부른다.

언어의 문법 표현으로부터 파서 생성기는 파서를 제어하는 테이블을 생성한다. 파서는 이 테이블을 이용하여 주어진 문장에 대한 문법적인 검사를 하며 다음 단계에서 필요한 의미 정보(semantic informations)를 만든다. 모든 언어에 대하여 파서 부분은 동일하고 테이블만 다르다. 새로운 언어에 대한 파서를 만들기 위해서는 단지 문법 표현만 바꾸면 된다. 일반적인 표현 문법으로는 context-free 문법을 사용한다.

2.2 스트링 패턴 매칭

스트링 패턴 매칭[7]은 중간 코드를 입력 받아서 패턴으로 기술된 스트링을 찾아 최적화된 스트링 패턴으로 매칭시키는 방법이다. 다시 말하면, 몇 개의 명령어에 대한 윈도(window)를 가지고 입력을 검색하여 일련의 비효율적인 코드를 좀 더 효율적인 코드로 대치하는 방법이다.

스트링 패턴 매칭 알고리즘을 살펴보면, 텍스트와 패턴의 각 문자열을 비교한다. 각 문자열이 일치하게 되면 다음의 문자열로 이동하여 비교한다. 만약, 패턴의 우측 마지막 문자열까지 일치하면 패턴 검색은 완료된 것으로 간주한다. 그러나 일치하지 않은 문자열이 발견되면 패턴을 문자열만큼 우측으로 이동시켜 다시 패턴의 시작 문자열부터 다시 비교를 시작한다.

스트링 패턴 매칭의 단점은 패턴 결정 시에 반복적으로 많은 비교 동작(5000번의 리스캔)이 이루어지므로 최적화

패턴을 찾는데 걸리는 시간이 많아 비효율적이다.

예를 들면, ACK 중간 코드 최적화기는 중간 코드에 대한 기본 블록에 대해 최적화 패턴으로 대치하는 핍홀 최적화 동작을 수행하였다. 또한 중간 코드에 대응하는 최적의 패턴을 찾기 위한 방법으로 스트링 패턴 매칭을 적용하였다.

2.3 트리 패턴 매칭

트리 패턴 매칭[2]은 중간 코드를 입력 받아서 패턴으로 기술된 트리형태로 재구성이 가능하며, 패턴 분석 및 패턴 결정이 용이하다는 장점을 갖는다. 하향식(Top-Down)으로 이루어지는 패턴 결정 과정에서 중복 비교가 발생할 수 있으며, 코드의 트리 구성에 많은 비용이 드는 단점이 있다.

중간 표현으로써 트리 구조에 기반을 둔 TCOL을 사용하였으며, 목적기계 코드를 생성하기 위해 순환적으로 트리를 순회할 수 있는 알고리즘을 사용하였다. 즉, 트리 패턴의 중간 코드를 입력으로 받아 목적 코드를 생성하는 방법이다.

예를 들어, ACK 중간코드 최적화 알고리즘인 스트링 패턴 매칭 알고리즘 사용으로 인하여 발생되는 과다한 최적화 패턴 검색 시간이 걸리는 문제를 개선하기 위하여 트리 패턴 매칭을 사용하였다.

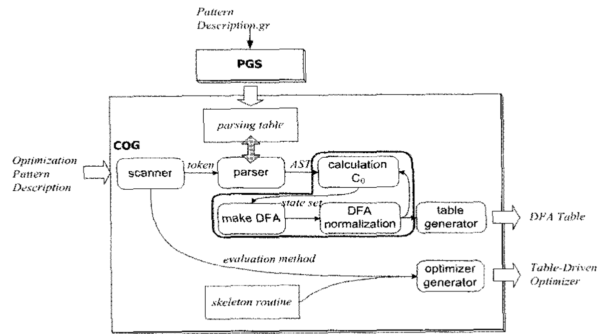
트리 패턴 매칭의 알고리즘은 중간코드 트리와 최적화 트리 패턴이 구성된 후 트리 패턴 매칭을 시작한다. 중간 코드 트리를 중위운행법으로 운행하면서 최적화 패턴 테이블에서 매칭 되는 하위 중간 코드 트리가 존재하는지의 여부를 확인하고 존재할 경우 조건식의 값이 참인지를 확인한다. 이 조건을 만족할 경우에 최적화된 트리로 재구성한다.

3. 코드 최적화기 생성기

3.1 시스템 구성도

코드 최적화기 생성기(Code Optimizer Generator)[16]는 최적화 패턴 형식(Pattern Description.gr)을 입력받아 (그림 3)과 같은 처리과정을 수행하여 DFA 테이블로 출력된다. COG의 구조는 스캐너(Scanner), 파서(Parser), DFA 생성기(DFA Generator), 테이블 생성기(Table Generator), 최적화기 생성기(Optimizer Generator)로 구성된다. DFA 생성기는 C_0계산(Caculation C_0), DFA 생성(Make DFA), DFA 정규화(DFA Normalization) 과정인 세 부분으로 구성된다.

COG를 구성하고 있는 각각의 모듈을 살펴보면, 스캐너는 최적화 패턴 형식을 입력 받아 문법적으로 의미 있는 최소 단위인 일련의 토큰을 생성하는 일을 한다. 파서는 패턴 문법을 통해 얻어진 파싱 테이블을 이용하여 패턴 형식을 구문 분석하고 그 결과로 AST를 생성한다. DFA 생성기는 파서로부터 생성된 AST를 이용하여 C_0를 계산하고, 패턴을 DFA 형태로 변환하고, 정규화 과정을 거쳐 DFA 테이블에 사용 가능한 상태 집합(state set)으로 구성한다. 테이블 생성기는 상태 집합을 입력으로 받아 테이블을 이용한 최적화기에서 사용가능한 doubly linked list의 자료구조(2차원 배열)로 변환한다. 최적화기 생성기는 shift, replace, advance



(그림 3) 코드 최적화기 생성기 구성도

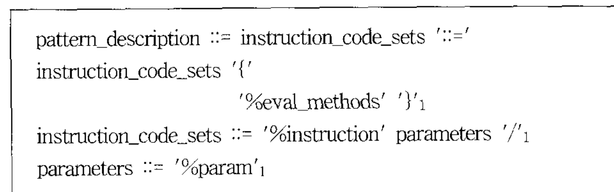
3개의 action 정보를 가지고 있는 스켈레톤 루틴과 패턴 기술에서 얻어진 evaluation_method를 결합하여 테이블을 이용한 최적화기를 생성한다.

3.2 패턴 형식 기술

패턴 형식 기술은 패턴 기술(Pattern Description)의 최적화 패턴 형식(pattern.gr)은 (그림 4)의 EBNF 형식으로 대치 부분과 패턴 부분으로 구성된다. 대치 부분과 패턴 부분에 대한 길이는 제한이 없으며, 하나의 패턴은 한 라인으로 구성된다.

대치 부분은 패턴에 대해 최적화된 명령어로 변환되는 부분을 나타낸다. 패턴 부분은 코드 최적화기에 의해 인식될 수 있는 최적화되지 않은 명령어의 열을 나타낸다. 기술된 패턴으로부터 코드 최적화기 생성기를 이용하여 패턴 기술의 내용을 파싱하고 (대치부 ::= 패턴부) 쌍으로 구성된 입력을 받아들여 DFA 테이블을 출력한다.

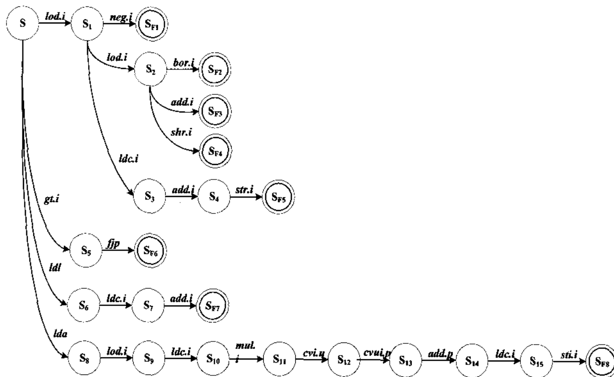
패턴 형식 기술의 기본적인 요소는 instruction_code_set, parameters, eval_methods로 패턴 형식을 구성한다. 패턴 형식은 명령어 코드 집합들의 대치 부분과 패턴 부분을 구분하기 위하여 구분자 '::='로 기술한다. 패턴 부분의 명령어 코드 집합과 eval_method로 기술한다. 명령어 코드 집합은 명령어와 파라메터 그리고 구분자 '/'로 기술한다.


```
pattern_description ::= instruction_code_sets '::='
instruction_code_sets '{'
                      '%eval_methods' '}'1
instruction_code_sets ::= '%instruction' parameters '/'1
parameters ::= '%param'1
```

(그림 4) 패턴 형식

3.3 DFA 테이블

DFA 테이블[13, 16, 17]의 행은 일련의 상태들과 열은 코드에 대한 명령어들의 집합으로 이루어진다. 결정된 패턴 DFA의 모든 정보를 가지고 있으며 코드 최적화기에서 패턴 DFA에 대한 정보를 참조하여 실질적으로 최적화된 *.sil 코드를 생성한다. 각 패턴들의 모든 정보는 오토마타 형태로 정보를 가지고 있으며 *.sil 코드와 더불어 코드 최적화기(Code Optimizer)에 의해 DFA를 이용하여 중간 코드를 매



(그림 5) 패턴 예제

〈표 1〉 DFA 테이블

codes / states	lod.i	neg.i	bor.i	add.i	shr.i	ldc.i	add.i	str.i	gt.i	fjp	ldl	lda	mul.i	cvi.u	cvui.p	add.p	sti.i
0	s1								s5		s6	s8					
1	s2	r1				s3											
2			r2	r3	r4												
3							s4										
4								r5									
5										r6							
6						s7											
7							r7										
8	s9																
9						s10											
10													s11				
11														s12			
12															s13		
13																s14	
14						s15											
15																	r8

칭할 수 있다. DFA 테이블 Action으로는 shift, replace, advance 3가지 Action들로 구성한다.

코드 최적화기는 중간코드인 *.sil 코드를 라인 단위로 입력 받아 코드 최적화기 생성기[5]에 의해 생성된 DFA 테이블 정보를 참조하여 실질적으로 최적화된 *.sil 코드를 생성한다. 각 패턴들의 모든 정보는 오토마타 형태의 정보를 가지고 있으며 *.sil 코드와 더불어 코드 최적화기(Code Optimizer)에 의해 DFA를 이용하여 중간 코드를 매칭할 수 있도록 한다.

DFA 테이블은 일련의 상태들과 코드에 대한 명령어들의 집합으로 구성된다. 즉, 한 상태와 입력 심벌(명령어)을 보고 다른 상태로 이동하는 것을 나타낸다. (그림 5) 패턴 예제 다섯 번째 패턴은 상태 0에서 코드 lod.i을 보고 상태 1로 shift한다. 다시 상태 1에서 코드 ldc.i를 보고 상태 3로 shift한다. 상태 3에서 add.i를 보고 상태 4로 shift한 후 상태 4에서 코드 str.i를 보고 replace를 한다. (그림 5) 패턴 예제를 가지고 <표 1>과 같이 DFA 테이블을 나타낼 수 있다.

3.4 DFA 테이블 구성 알고리즘

최적화 패턴 형식을 입력 받아 스캐너가 문법적으로 의미있는 최소 단위인 일련의 토큰을 생성한다. 그 토큰을 입력 받아 파서는 패턴 문법을 통해 얻어진 파싱 테이블을 이용하여 패턴 형식을 구문 분석하고 그 결과 추상 구문 트리(Abstract Syntax Tree)를 생성한다.

DFA 생성기는 생성된 AST를 이용하여 파싱 테이블을 구성하는 기본적인 자료가 되는 C_0를 계산하고, 패턴을 DFA 형태로 변환하며, 정규화 과정을 거쳐 DFA 테이블에 사용 가능한 상태 집합(State Set)으로 구성하는 반복적인 과정을 거친다.

코드 최적화기 생성기의 처리과정 중 DFA 생성기와 테이블 생성기는 테이블을 이용한 코드 최적화기에서 패턴에 대한 모든 정보를 참조할 수 있는 DFA 테이블을 생성하는 핵심적인 과정이다. 이 과정에서 사용하는 5개의 심벌의 구성은 C_0, Lookahead, State, Instruction, DFA 테이블로 구분된다. 각 심벌의 구성들을 살펴보면, C_0의 종류는 LR(0)[1, 10] 아이템들의 집합을 $LR(0)_{set}$으로 $LR(0)_{set}$들의 집합을 I로 나타내며, $LR(0)_{set}$의 i번째 엘리먼트는 I_i로 구분된다. 상태[4]는 4개의 상태로 구성되며 각 상태를 살펴보면, 상태들의 집합인 S_{set}과 시작 상태는 S_S, 현재 상태는 S_C와 종결 상태는 S_F로 구분된다. 명령어는 명령어들의 집합인 $Instr_{set}$로 나타내며, DFA 테이블은 상태와 DFAT(S, Instruction)상태와 명령어로 나타낸다.

```
Algorithm make_dfa
  begin
    // make start state and set first conditions
    make S_S
    stateCounter ← 0
    S_C ← S_S
    // make dfa for all LR(0)_set
    for all I_i ∈ I
        // check final state and split state
        S_Set ← Φ
        if I_i have reduce item and kernel item(s) then
            S_Set ∪ normalize_dfa(I_i)
            I ∪ calculation_C0(I_i)
        fi
        S_Set ∪ S_C
        // make all child state of current state
        if LA_set(i) = Φ then
            set_final_state(S_C)
        else
            for all S_C ∈ S_Set
                for all LA ∈ LA_set(i)
                    stateCounter ← stateCounter + 1
                    make_state(S_C+stateCounter)
                    set_child(S_C, S_C+stateCounter, LA)
                end for
            end for
        fi
          S_C ← S_i+1
    end for
  end.
```

(알고리즘 1) DFA 생성 알고리즘

DFA 테이블을 생성하기 위한 알고리즘은 과정은 첫 번째, C_0는 테이블을 구성하는 기본적인 자료가 되며 주어진 문법으로부터 C_0를 계산한다. 두 번째, DFA 생성은 C_0의 계산된 결과를 가지고 패턴을 DFA 형태로 변환하기 위한 알고리즘으로 (알고리즘 1)과 같이 기술한다. 마지막으로, DFA 정규화는 생성된 DFA들이 패턴 검색 시 발생하는 생

성된 상태 집합 중 정규화된 DFA 조건을 만족하지 못하는 상태들이 정규화된 DFA가 될 수 있도록 변환하는 과정이며, 정규화 이유는 DFA 패턴 검색시 처리가 간결하고 해당 패턴을 빠르게 찾는데 있다. 이 과정을 (알고리즘 2)와 같이 기술한다. 따라서, 위의 세 단계를 반복적으로 실행하여 DFA 테이블을 생성한다.

```
Algorithm normalize_dfa(LR(0)set L);
begin
  for all kernel item LA ∈ L
    stateCounter ← stateCounter+1
    make_state(S_C-stateCounter)
    S_Set ∪ S_C+stateCounter
    set_child(S_S, S_C-stateCounter reduce item)
  end for
  return S_Set
end.
```

(알고리즘 2) DFA 정규화 알고리즘

(알고리즘 2)는 모든 $LR(0)_{set}$을 위한 DFA를 생성하는 알고리즘으로 상태가 종결 상태와 분할 상태를 체크하기 위해 I_i가 reduce item과 kernel item을 모두 가지고 있는 경우 I_i에 대해 DFA 생성 알고리즘을 수행한다.

DFA 정규화 알고리즘은 LR(0) 아이템들의 집합에 모든 kernel item에 대해 새로운 상태를 생성하고, 현재 상태의 값과 상태 카운트 값을 계산한 후 상태 집합에 추가한다. 정규화를 한 후 대치부의 내용을 반영하기 위해서 시작 상태에서 reduce item을 보고 정규화된 상태로 가는 것을 추가한다.

3.5 DFA(Deterministic Finite Automata) 최적화

최적화 패턴과 COG(Code Optimizer Generator)를 이용하여 얻어진 테이블 정보를 이용하여 최적화를 수행한다. 최적화 패턴은 결정적인 유한 오토마타를 이용하여 한 상태에서 명령어를 보고 다른 상태로 이동하는 것을 상태전이도로 구성한다. DFA를 이용한 코드 최적화는 오토마타를 통해 결정적으로 패턴이 확정됨에 따른 패턴 선택 비용이 줄어들며, 최적화 패턴 검색 시간이 빨라진다.

첫 번째, 테이블은 기본적으로 행과 열로 구성되며, 상태정보로 구성된 행과 명령어 정보로 구성된 열로 구성되며 각 행과 열에 해당하는 아이템은 각각 shift, replace, advance의 세 가지 속성 중 하나의 속성을 가지고 있다.

두 번째, 구동기는 일련의 코드정보와 상태정보를 이용하여 테이블에 기록된 속성에 따라 액션을 하는데, shift일 경우에는 현재 상태를 shift에 해당하는 상태로 변경하며, replace일 경우에는 현재까지 매칭된 정보를 이용하여 기존 코드를 해당 최적화 코드로 변경하며, advance일 경우에는 상태 값을 초기화하고 다음 코드정보를 읽어온다.

테이블-구동 패턴 매칭 기법은 위에서 언급한 3가지 행동만으로 최적화를 수행하므로 최적화기 구성이 간단하며, 미리 구성된 테이블 정보를 이용하므로 최적화 수행이 기존의 최적화 방법에 비해 매우 빠르다. 또한 테이블에 해당 코드 최적화 정보가 기술되어 있고, 구동기는 코드와 독립적이므로, 최적화 패턴이 추가되거나 변경될 경우에 테이블만 변경되므로 확장성이 높으며, 새로운 언어에 대한 재목적성도 높다.

SIL(Standard Intermediate Language)코드에 대해 목적기계 독립적인 최적화 동작을 수행하여 최적화된 opt_sil 코드를 출력하는 DFA 최적화 부분은 패턴 형식, 코드 최적화기 생성기, 코드 최적화기로 구성된다.

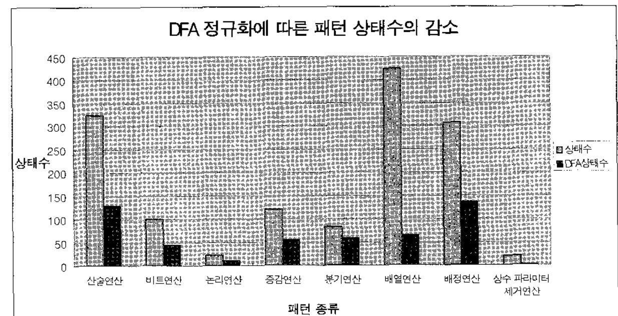
패턴 형식은 (대치::=패턴)의 쌍으로 구성되는데 패턴 부분은 코드 최적화기에 의해 인식될 수 있는 최적화 되지 않은 sil 코드의 열을 나타내며, 대치 부분은 패턴에 대해 최적화된 코드로 변환되는 부분을 나타낸다. 이러한 패턴 형식은 코드 최적화기 생성기의 입력으로 사용된다.

코드 최적화기 생성기는 sil 코드에 대한 패턴 표현을 입력으로 받아 구문 분석을 수행한 후 DFA 테이블을 생성한다. DFA 테이블은 배열 형태로 패턴의 정보를 가지고 있으며 코드 최적화기에 의해 참조된다. DFA 최적화는 입력으로 받은 sil 명령어와 DFA 테이블을 참조하여 최적화된 sil 코드를 출력한다. 이 때 코드 최적화기에서 사용되는 최적화 방법은 핍홀 최적화 방법을 이용한다. 코드 최적화 단계에서 수행될 수 있는 최적화 동작으로는 주로 컴파일러 상수 시간 연산, 중복된 load/store 명령문의 제거, 수식의 대수학적 간소화(algebraic simplification), 연산 강도 경감(strength reduction), 불필요한 일련의 블록 삭제 등의 효과를 얻을 수 있다.

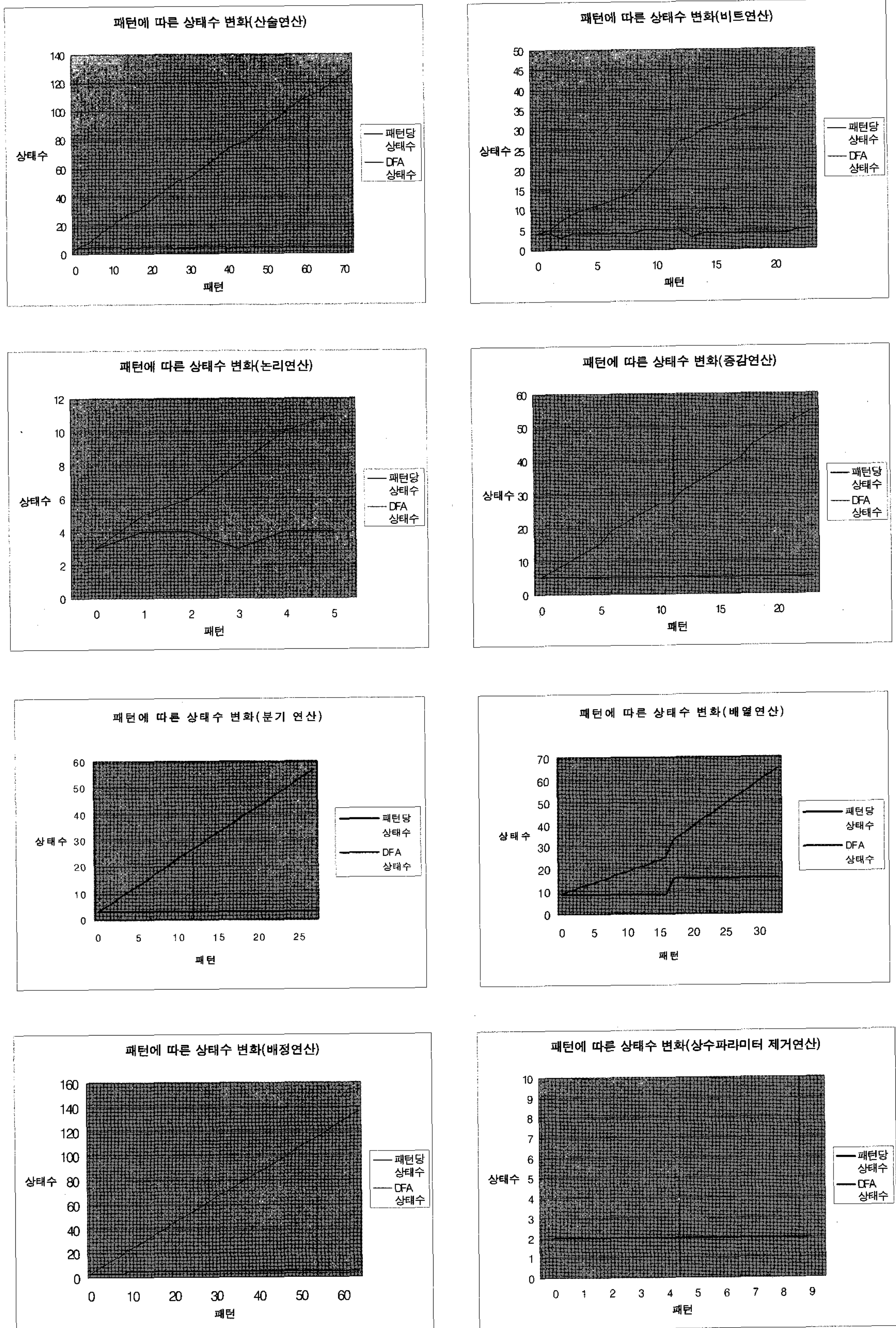
4. 실험 결과

〈표 2〉 패턴 연산 종류에 따른 상태수, DFA 상태수 비교

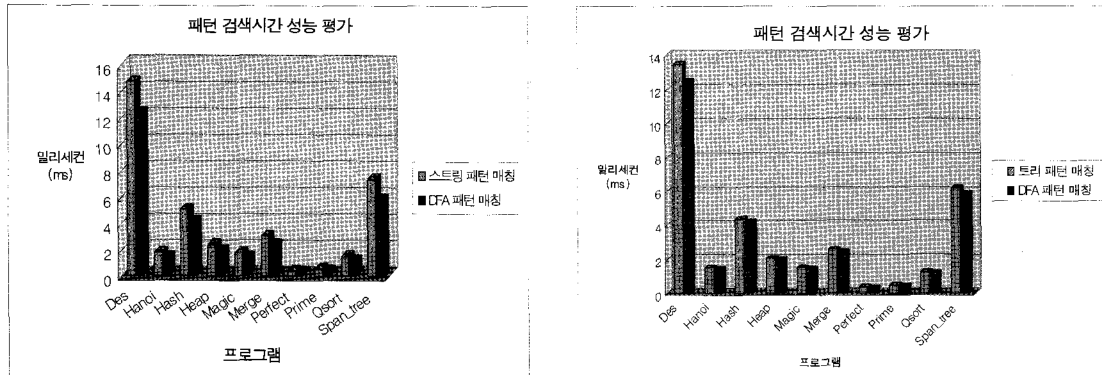
구분	산술연산	비트연산	논리연산	증감연산	분기연산	배열연산	배정연산	상수 파라미터 제거연산
패턴(개)	73	24	6	24	28	34	65	10
상태수(개)	325	100	22	120	84	425	307	20
상태수/패턴(%)	4.45%	4.17%	3.67%	5.00%	3.00%	12.50%	4.72%	2.00%
패턴(개)	73	24	6	24	28	34	65	10
DFA 상태수(개)	129	45	11	55	57	65	136	2
DFA 상태수/패턴(%)	1.77%	1.88%	1.83%	2.29%	2.04%	1.91%	2.09%	0.20%
감소율(%)	60.31%	55.00%	50.00%	54.17%	32.14%	84.71%	55.70%	90.00%



(그림 6) DFA 정규화에 따른 패턴 상태수의 감소



(그림 7) 연산 종류별 패턴에 따른 상태수 변화



(그림 8) 최적화 패턴 매칭 검색시간

본 논문의 DFA 패턴 매칭을 위한 코드 최적화기의 자동적 생성의 실험으로 첫 번째, 각 패턴을 연산 종류에 따라 상태수와 DFA 상태수 변화를 비교하는 실험과 두 번째, 중간 코드를 입력 받아 패턴으로 기술된 최적화 패턴을 기존의 최적화 방법인 스트링 패턴 매칭과 트리 패턴 매칭으로 실행시 소요되는 패턴 검색시간과 DFA 패턴 매칭으로 실행시 소요되는 패턴 검색시간을 비교하는 실험이다. 실험 환경으로는 Intel Pentium-IV 1.8GHz, RAM 512MB, Windows XP를 사용하였다.

첫 번째 실험 대상 데이터는 264개의 패턴들을 8개의 연산 종류로 패턴을 구분한다. 8개의 구분된 연산 종류로는 산술연산, 비트연산, 논리연산, 증감연산, 분기연산, 배열연산, 배정연산, 상수 파라미터 제거 연산으로 구성된다. 각 구분된 연산 종류에 따른 패턴의 상태수와 DFA 상태수 비교 실험은 <표 2>와 같은 실험 결과를 얻었다. 다시 말하면, <표 2>는 각 연산 종류로 구분된 패턴들은 상태(State)를 이용하여 표현하기 위해 사용된 상태수와 패턴들의 상태를 DFA 정규화 과정을 통해 표현된 DFA 상태수로 작성된다. 예를 들어, 각 연산 종류의 패턴 중 산술연산에 해당되는 73개 패턴들은 상태 325개로 표현되며, 이 패턴들의 상태들은 DFA 정규화 과정을 통해 129개의 DFA 상태로 표현된다. 감소율은 100-(DFA 상태수/상태수)*100 계산식을 이용하여 감소율을 얻을 수 있다. 즉, 산술연산 패턴들은 상태로 표현하는 것보다 DFA 상태로 표현할 경우 상태수가 60.31%로 감소된다. 8개의 연산 종류 패턴 중에서 상수 파라미터 제거 연산은 다른 연산 종류 패턴보다 상태수가 훨씬 감소된 결과를 확인할 수 있다.

(그림 6)은 264개 패턴을 각 연산 종류의 패턴을 상태를 이용하여 표현된 상태수와 표현된 상태들을 DFA 정규화 과정을 통해 중복된 상태를 제거하여 상태수의 감소와 패턴 형태의 단순화 및 구조를 개선함으로 비용이 적게 드는 결과를 얻을 수 있다. 예를 들어, 여러 개의 연산 종류 패턴 중에서 배열연산의 425개 상태가 DFA 정규화를 통하여 65개의 상태로 표현되기 때문에 감소된 상태수를 확인할 수 있다.

연산 종류별 패턴에 따른 상태수 변화에 대한 실험으로 결과는 (그림 7)과 같다. (그림 7)은 패턴당 상태수와 패턴 중에서 중복 상태를 비교 제거하는 DFA 정규화 과정을 통해 나온 상태수를 나타낸다. DFA 정규화 과정은 DFA 테이블 구성시 핵심적인 부분으로 DFA 패턴 매칭은 패턴들의 정규화 과정을 통해 패턴 검색 시 발생하는 중복 비교를 제거한다. 또한, 패턴 형태의 단순화 및 구조를 개선함으로 DFA 테이블을 생성하는데 비용이 적게 드는 결과를 얻을 수 있다. 8개의 연산 중 상수파라미터 제거 연산의 패턴에 따른 상태수 변화는 패턴당 상태수와 DFA 상태수가 동일한 결과를 얻었다. 또한, 연산 종류 중에서 패턴에 대한 DFA 상태수는 (그림 7)과 같이 패턴 당 상태수의 증가는 DFA 정규화 과정을 거치므로 산술연산, 배정연산, 배열연산이 다른 연산들보다 많은 상태수를 갖는 것을 확인할 수 있다.

두 번째 실험은 DFA 패턴 매칭 방법과 기존 최적화 방법인 스트링 패턴 매칭 및 트리 패턴 매칭의 최적화 패턴 검색 실행 시간에 대해 비교한다. 기존 최적화 방법의 문제점으로는 스트링 패턴 매칭의 패턴 결정시 과다한 최적화 패턴을 검색하는데 시간이 많이 걸리는 비효율적인 문제와 트리 패턴 매칭의 패턴 재구성 과정에 소요되는 많은 비용 문제가 있다. DFA 패턴 매칭 방법은 패턴들의 정규화 과정을 통해 패턴 검색 시 발생하는 중복 비교를 제거하며, 패턴 형태의 단순화 및 구조를 개선함으로 적은 비용으로 최적화를 수행할 수 있다. 이 실험결과는 스트링 패턴 매칭의 문제점을 개선한 트리 패턴 매칭도 비교적 빠른 검색을 하지만, (그림 8)과 같이 DFA 패턴 매칭 방법이 트리 패턴 매칭보다 패턴 결정 과정에서 최적화 패턴 매칭 검색시간이 조금 더 빠른 최적화 방법이라는 것을 확인할 수 있다. 다시 말해, (그림 8)의 성능 평가 결과를 분석해 보면, DFA 패턴 매칭 방법이 스트링 패턴 매칭 방법보다 패턴 검색시간을 평균 20% 정도 단축시켰으며, 트리 패턴 매칭 방법보다는 패턴 검색시간을 평균 6% 정도 단축시킨 결과를 얻을 수 있다. 본 논문에서 개발한 DFA 패턴 매칭 방법이 패턴 검색시간을 좀 더 빠르게 향상시킨 것을 확인할 수 있다.

5. 결론 및 향후 과제

원시 프로그램에 대한 컴파일 과정 중 최적화 단계에서는

프로그램의 실행 속도를 개선시키고 코드 크기를 줄일 수 있는 다양한 최적화 기법을 수행한다. 비효율적인 명령어의 순서를 구별해 내고 연속되는 명령어의 순서를 의미적으로 동등하면서 좀 더 효율적인 코드로 개선하는 방법이다.

본 논문에서는 의미적으로 동등한 효율적인 코드로 바꾸기 위한 코드 최적화 과정을 수행하는 코드 최적화기를 자동적으로 생성할 수 있는 컴파일러 자동화 도구인 코드 최적화기 생성기(Code Optimizer Generator)를 제안하였다. 즉, 코드 최적화기 자동적 생성 도구인 COG를 구현하여 DFA 패턴 매칭을 위한 코드 최적화기를 구현하였다. COG는 코드 최적화기에서 필요한 패턴에 대한 모든 정보를 참조할 수 있는 DFA 테이블을 생성한다. 최적화 패턴 검색시 DFA 최적화 알고리즘을 이용하여 최적화를 하기 때문에 최적화의 속도가 기존 최적화 알고리즘을 적용한 것보다 빠르다. DFA 패턴 매칭은 패턴들의 정규화 과정을 통해 패턴 검색 시 발생하는 중복 비교를 제거하여, 패턴 형태의 단순화 및 구조를 개선함으로 비용이 적게 든다. DFA 패턴 매칭을 위한 코드 최적화기의 자동적 생성의 장점으로는 각각의 중간코드에 대한 케이스별로 최적화기를 만들어야 하는 수고를 덜어줌으로써 코드 최적화에 정형화를 할 수 있다. 또한, DFA를 이용하여 최적화를 하기 때문에 최적화 속도가 빠르고, 코드 최적화기를 만드는데 필요한 시간과 비용을 절약할 수 있다. 최적화 패턴이 추가되거나 변경될 경우에 테이블만 변경되므로 확장성이 높으며, 새로운 언어에 대한 재목적성이 높은 장점을 가진다.

향후 연구과제로는 양질의 최적화된 코드를 생성하기 위해 DFA 테이블에 보다 효율적인 최적화를 위한 최적화 패턴을 개발하여 추가할 예정이다. 또한 최적화 패턴 수의 증가 및 감소가 코드 최적화 전·후 코드 경감율에 어떤 영향을 주는가에 대해 계속 연구할 예정이다.

참 고 문 헌

[1] A. V. Aho and S. C. Johnson, "LR Parsing," ACM Computing Surveys (CSUR), Vol.6, No.2, pp.99-124, Jun., 1974.

[2] Alfred V. Aho, Mahadevan Ganapathi and Steven W. K. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming," ACM TOPLAS, Vol.11, No.4., pp.491-516, Oct., 1989.

[3] C. W. Fraser, A. L. Wendt, "Automatic Generation of Fast Optimizing Code Generators," Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, Vol.23, No.7, pp.79-84, Jun., 1988.

[4] Christopher W. Fraser and Todd A. Proebsting, "Finite-State Code Generation," ACM SIGPLAN, Vol.34, No.5, pp.270-280, May, 1999.

[5] Christopher W. Fraser, David R. Hanson and Todd A. Proebsting, "Engineering a Simple, Efficient Code-Generator Generator," ACM LOPLAS, Vol.1, No.3, pp.213-226, Sept., 1992.

[6] David A. Workman, John B. Higdon, "The design of a parser generator," Proceedings of the 16th annual Southeast regional conference, pp.82-86, Apr., 1978.

[7] Diomidis and Spinellis, "Declarative Peephole Optimization using String Pattern Matching," ACM SIGPLAN Notices, Vol.34, No.2, pp.47-51, Feb., 1999.

[8] Fabrice Le Fessant and Luc Maranget, "Optimizing Pattern Matching," ACM SIGPLAN, Vol.36, No.10, pp.26-37, Oct., 2001.

[9] George H. Roberts, OPG: An Optimizing Parser Generators," ACM SIGPLAN Notices, Vol.23, No.6, pp.80-90, Jun., 1988.

[10] M. Ancona, G. Dodero, V. Gianuzzi, and M. Morgavi, "Efficient Construction of LR(k) States and Tables," ACM TOPLAS, Vol.13, No.1, pp.150-178, Jan., 1991.

[11] R. G. Cattell, "Automatic Derivation of Code Generators from Machine Descriptions," Source ACM TOPLAS, Vol.2, No.2, pp.173-190, Apr., 1980.

[12] Rudolf Landwehr, Hans-Stephan Jansohn and Gerhard Goos, "Experience with an Automatic Code Generator Generator," ACM SIGPLAN, Vol.17, No.6, pp.56-66, Jun., 1982.

[13] Sunglim Yun and Seman Oh, "Table-Driven Code Optimizer," International Conference on Parallel and Distributed, Computing, Applications and Technologies, pp.59-63, Dec., 2005.

[14] Thomas E. Cheatham, Thomas A. Standish, "Optimization Aspects of Compiler-Compilers," ACM SIGPLAN, Vol.5 No.10, pp.10-17, Oct. 1970.

[15] 오세만, 컴파일러 입문 개정판, 정익사, 서울, 2004.

[16] 윤성림, 오세만, "패턴 테이블을 이용한 코드 최적화", 한국멀티미디어학회 논문지, 제8권, 제11호, pp.1556-1564, 2005.

[17] 윤성림, 오세만, "DFA를 이용한 코드 최적화", 한국정보처리학회 춘계학술발표논문집, 제12권, 제1호, pp.523-526, 2005.

윤 성 림

e-mail : yslhappy@dgu.edu

1998년 한국방송통신대학교 전자계산학과(학사)

2000년 동국대학교 교육대학원 컴퓨터교육(교육학석사)

2006년 동국대학교 일반대학원 컴퓨터공학과(공학박사)

1990년~2002년 동국대학교 전자계산원 전산운영팀 근무

2004년~현재 동국대학교 컴퓨터공학과 강사

관심분야 : 프로그래밍 언어, 컴파일러, 임베디드 시스템

오 세 만

e-mail : smoh@dgu.edu

1977년 서울대학교 수학교육과(학사)

1979년 한국과학기술원 전산학과(공학석사)

1985년 한국과학기술원 전산학과(공학박사)

1985년~1989년 동국대학교 컴퓨터공학과 조교수

1989년~1994년 동국대학교 컴퓨터공학과 부교수

1994년~현재 동국대학교 컴퓨터공학과 교수

관심분야 : 프로그래밍 언어, 컴파일러, 모바일 컴퓨팅, 임베디드 시스템