DFA 를 이용한 코드 최적화

윤성림*, 오세만
동국대학교 컴퓨터공학과
e-mail: {yslhappy, smoh@dgu.edu}

Code Optimization Using DFA

Sung-Lim Yun*, Se-Man Oh
Dept of Computer Engineering, Dongguk University

요 약

원시 프로그램에 대한 컴파일 과정 중 최적화 단계에서는 프로그램의 실행 속도를 개선시키고 코드 크기를 줄일 수 있는 다양한 최적화 기법을 수행한다. 특히, 핍홀 최적화는 비효율적인 명령어의 순서를 구별해 내고 연속되는 명령어의 순서를 의미적으로 동등하면서 좀 더 효율적인 코드로 개선하는 방법이다.

최적화 패턴 매칭 방법 중 스트링 패턴 매칭 방법은 중간 코드에 대응하는 최적의 패턴을 찾기 위한 방법으로 과다한 최적화 패턴 검색 시간으로 비효율적이고, 트리 패턴 매칭은 패턴 결정시 중복 비교가 발생할 수 있으며, 코드의 트리 구성에 많은 비용이 드는 단점을 가지고 있는 방법들이다.

본 논문에서는 기존의 최적화 방법들의 단점을 극복하기 위한 방법으로 DFA(Deterministic Finite Automata)를 이용한 코드 최적화 방법을 제안한다. 이 방법은 다른 패턴 매칭 기법보다 오토마타(Automata)로 구성하기 때문에 비용은 적어지고, 오토마타를 통해 결정적으로 패턴이 확정됨에 따른 패턴 선택 비용이 줄어들며, 최적화 패턴 검색 시간도 빨라지는 효율적인 방법이다.

1. 서론

최근 원시 프로그램에 대한 컴파일 과정 중 최적화 단계에서는 프로그램의 실행 속도를 개선시키고 코드 크기를 줄일 수 있는 다양한 최적화 기법을 수행한다. 컴파일러 개발 과정에서 최적화 기법의 수행은 목적기계 독립적인 중간코드 최적화(intermediate code optimization) 와 목적기계 의존적인 목적코드 최적화(target code optimization)로 구분할 수 있다.

또한 최근의 컴파일러 개발에 대한 관심이 다양한 목적기계에 쉽게 적용할 수 있는 코드를 생성하도록 재목적이 가능한 최적화 컴파일러(retargetable optimization compiler)에 있으므로 목적기계와 독립적인 중간코드에서의 최적화는 더욱 필요하다.

특히, 핍홀 최적화는 최적화에 드는 비용에 비해 큰 효과를 얻을 수 있으며, 그 기법은 비효율적인 명령어의 순서를 구별해 내고 연속되는 명령어의 순서를 의미적으로는 동등하면서 좀 더 효율적인 코드로 개선하는 방법이다.

패턴 매칭 방법을 이용한 코드 생성 방법은 양질의 코드 생성을 위해 중간 언어에 대한 분석을 수행한 후 하나 이상의 명령어로 구성된 패턴에 대해 목적 코드를 생성하는 방식이다. 따라서 패턴 매칭을 이용한 코드 생성 방식에서는 먼저 중간언어의 패턴에 대한 변환될 목적 코드간의 관계가 기술되어 실질적인 코드 변환 시에 참조될 수 있도록 해야 한다. 패턴 매칭 방법중 스트링 패턴 매칭은 패턴 결정시에 반복적으로 많은 비교 동작(5000 번의 리스캔) 수행해야 함으로 최적화 패턴을 찾는데 걸리는 시간이 많아 비효율적이고, 트리패턴 매칭은 패턴 결정 과정에서 중복 비교가 발생할 수 있고, 코드의 트리 구성에 많은 비용이 드는 단점들이 있다.

본 논문의 구성은 2 장에서는 본 연구의 기반이 되는 스트링 패턴 매칭, 트리 패턴 매칭에 대한 관련 연구를 소개한다. 3 장에서는 기존의 패턴 매칭 방법인 스트링 패턴 매칭, 트리 패턴 매칭들의 단점을 보완하기 위한 새로운 매칭 방법으로 DFA 를 이용한 코드 최적화를 하기 위해 최적화 시스템의 구성도, 패턴 기술, DFA 최적화 알고리즘을 기술한다.마지막으로, 4 장에서는 본 연구의 결론과 향후 연구 과제에 대해서

기술한다.

2. 관련 연구

2.1 스트링 패턴 매칭 기법

스트링 패턴 매칭 기법은 중간 코드를 입력 받아서 패턴으로 기술된 스트링을 찾아 최적화된 스트링 패턴으로 매칭시키는 방법을 말한다. 다시말하면, 몇 개의 명령어에 대한 윈도(window)를 가지고 입력을 검색하여 일련의 비효율적인 코드를 좀 더 효율적인 코드로 대치하는 방법이다.

스트링 패턴 매칭 기법 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 매칭 기법의 단점은 패턴 결정 시에 반복적으로 많은 비교 동작(5000 번의 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 매칭 기법을 적용하였다[7].

2.2 트리 패턴 매칭 기법

트리 패턴 매칭 기법은 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 TCOL 을 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 매칭 방법을 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. [표 1]는 트리 패턴 매칭 알고리즘을 표로 도시화한 것이다.

[표 1] 트리 패턴 매칭 알고리즘

```
match_treepattern(node_ptr ppar, node_ptr pnode, bool flag)
/* ppar      parent 중간코드 node 포인터
   pnode     매칭할 중간코드 node 포인터
   flag      pnode 가 ppar 의 자노드/제노드 여부
*/
{
  if (pnode != NULL)
      if (매칭되는 하위 tree 존재)
          트리를 재구성.
          pnode 를 해당 위치로 변경.
    match_treepattern(pnode, pnode->son, true);
    match_treepattern(pnode,pnode->brother,  flase);
  end if
}
```

트리패턴 매칭 단계별로 간단히 살펴보면, 첫 번째 단계에서는 패턴 트리 검색, 두 번째 단계에서는 조건식 체크, 세 번째 단계에서는 대치(replacement) 마지막으로 최적화 트리 구성을 한다.

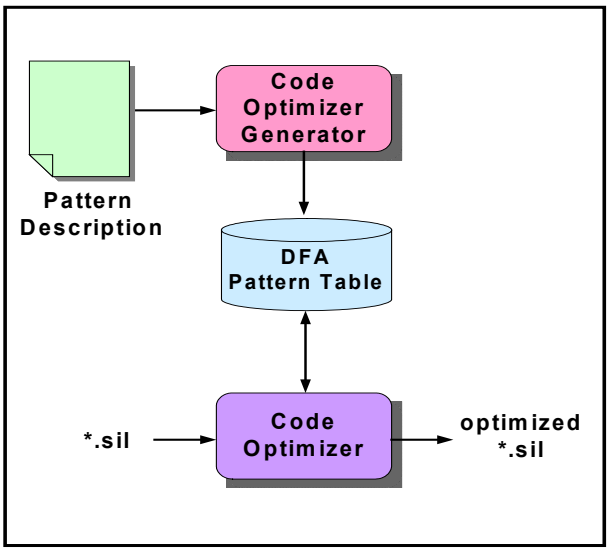
3. DFA 를 이용한 코드 최적화

3.1 최적화 시스템의 구성도

스트링 패턴 매칭의 최적화 패턴 검색 시간이 많이 걸리는 문제와 트리 패턴 매칭의 하향식(Top-Down)으로 이루어지는 패턴 결정 과정에서 중복 비교의 발생으로 인한 코드의 트리 구성에 많은 비용이 드는 단점을 보완하기 위한 것이다.

DFA 를 이용한 최적화 시스템은 [그림 1]과 같이 Code Optimization Generator, DFA Pattern Table, 로 구성할 수 있다.

코드 최적화 생성기(Code Optimization Generator)는 중간 코드에 대한 코드 생성 규칙으로 기술한 패턴을 입력 받아 DFA 패턴 테이블로 구성한다.



[그림 1] 최적화 시스템 구성도

DFA 패턴 테이블에 존재하는 패턴 DFA 형태와 매핑을 시도하여 이에 부합하는 패턴에 대하여 sil 를 최적화된 sil DFA 로 재구성한다. 패턴 DFA 의 arc 의 이름(name)은 sil 코드명을 사용하며, sil 코드를 보고 트랜잭션(transaction)이 일어나게 된다.

코드 최적화기(Code Optimizer)는 중간코드인 *.sil 코드를 라인 단위로 입력 받아 코드 최적화 생성기에 의해 생성된 DFA 패턴 테이블 정보를 참조하여 실질적으로 최적화된 *.sil 코드를 생성한다.

각 패턴들의 은 오토마타 형태의 정보를 가지고 있으며 *.sil 코드와 더불어 코드 최적화기(Code Optimizer)에 의해 DFA 를 이용하여 중간코드를 매칭 할 수 있도록 한다.

코드 최적화 생성기는 패턴 기술을 입력받아서 LALR 방법으로 C_0 와 파싱 테이블을 작성할 수 있다. 파싱 테이블의 shift 와 accept 그리고 GOTO 행동은 SLR 과 같고 reduce 행동에서만 lookahead 를 사용한다. 파싱 테이블의 엔트리에서 s 는 shift, r 은 reduce 생성규칙 번호를 나타내며, 테이블의 빈칸은 error 를 의미한다. ACTION 테이블의 shift 와 GOTO 행동은 SLR 방법과 같고, reduce 행동은 각 reduce 아이템의 lookahead 심벌이 만나는 곳에 reduce 될 생성 규칙 번호를 채워 넣는다.

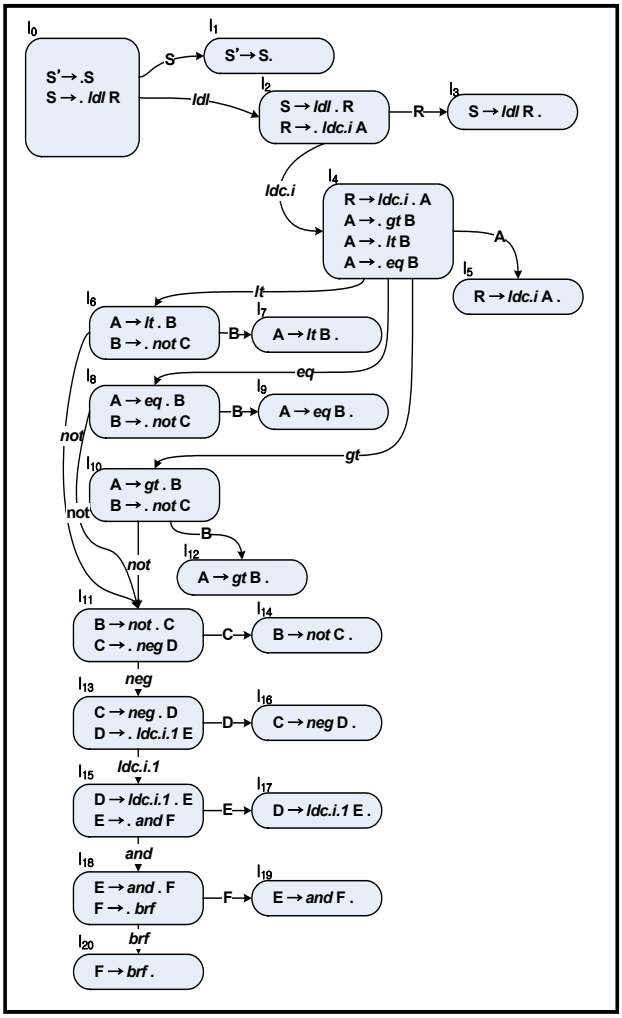
[표 2] 제어 패턴

```
// 패턴부     :     대치부
ldl v1 ldc.i c1 gt not neg ldc.i.1 and brf $$1
: ldl v1 ldc.i c1 gt not neg bne $$1
```

[표 2]는 제어 패턴을 문법을 만든 후, 주어진 문법을 가지고 LALR 파싱 테이블 구성하고 구문 분석하는 과정을 살펴본다. DFA 패턴 테이블을 구성하기 위한 C0 와 GOTO 그래프를 [그림 2]로 도시화 한다. 주어진 문법으로부터 C0 을 구성하고 각 reduce 아이템에 대해 lookahead 를 구하여 [그림 3] 파싱 테이블은 도시화 할 수 있다.

[그림 3] 파싱 테이블의 Action 과 Goto 테이블을 보고 제어 패턴의 구문 분석 과정을 살펴보았다. 상태는 0 - 20 단계, $로 reduce 는 r1 - r10 으로 된다. 주어진 패턴의 명령어를 입력 심벌로 보고 결정적으로 구문 분석할 수 있음을 보여준다.

DFA 패턴 테이블은 패턴부의 원시 SIL 코드 부분의 SIL 명령어에 대한 입력순서가 아니라 구성된 패턴 DFA 의 시작 상태를 기준으로 테이블이 구성된다. 패턴 DFA 의 sil code 의 arc 의 이름(name)이 sil code 를 보고 다음 상태 transaction 이 일어난다. [표 2]와 같은 제어문에 대한 패턴 예제를 가지고 Co 를 구해보면 [그림 2]로 나타낼 수 있다.

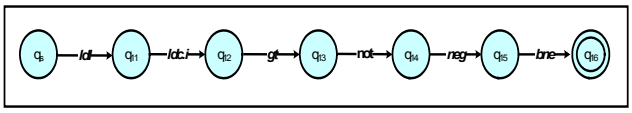


[그림 2] C0 와 GOTO 그래프

상태	ACTION 테이블										GOTO 테이블							
	ldl	ldc.i	gt	lt	eq	not	ldc.i.1	and	brf	$	S	R	A	B	C	D	E	F
0	s2										1							
1										acc								
2		s4										3						
3										r1								
4			s10	s6	s8								5					
5										r2								
6						s11								7				
7										r4								
8						s11								9				
9										r5								
10						s11								12				
11							s13								14			
12										r3								
13								s15								16		
14										r6								
15									s18								17	
16										r7								
17										r8								
18									s20									19
19										r9								
20										r10								

[그림 3] 파싱테이블

DFA 패턴 테이블의 정보를 코드 최적화기가 패턴 DFA 정보를 참조하여 [그림 4]와 같은 결과로 제어 패턴의 대치부 오토마타를 표현할 수 있다.



[그림 4] 제어 패턴의 대치부 오토마타

3.2 패턴 기술

SIL 코드의 특성을 고려하여 작성된 패턴 기술(pattern description)을 통해 DFA Pattern Table 을 생성한다. 패턴 기술은 패턴부(Pattern Part) : 대치부(Replace Part)의 쌍으로 구성되며, 패턴부는 최적화기에 의해 인식될 수 있는 최적화되지 않은 명령어의 열을 나타내며, 대치부는 패턴에 대해 최적화된 명령어로 변환되는 부분을 나타낸다. 기술된 패턴으로부터 COG 를 이용하여 패턴 기술의 내용을 파싱하고 (패턴부 : 대치부) 쌍으로 구성된 입력을 받아들여 DFA Pattern Table 을 출력한다. [표 3]은 최적화에 사용된 SIL 코드 패턴 중에서 상수, 스택, 배정, 제어에 관련된 패턴을 보여주고 있다.

[표 3] 패턴에 반영된 최적화 내용

```
// 상수
ldc.i c1 neg              : ldc.i.m1
                  …
// 스택
ldl v1 ldl v2             : ldl dup v1
                  …
// 배정
ldl v1 ldc.i c1 add        : inc v1 c1
ldl v1 ldc.i c1 sub        : dec v1 c1
                  …
// 제어
lt brf $$l                 : bge $$l
gt brf $$l                 : ble  $$l
                  …
```

3.3 DFA 최적화 알고리즘

SIL 코드 최적화에 적용되는 최적화 알고리즘은 기술된 패턴을 탐색하는 부분과 기존 최적화 알고리즘을 적용하는 부분으로 구성된다.

[표 3]에 기술된 패턴을 COG 에서 입력받아 DFA 패턴 테이블 생성은 [그림 2]와 [그림 3]으로 생성되는 과정을 알 수 있다.

DFA 최적화 알고리즘은 중간 코드의 상태 전이과정과 최적화 패턴과의 연결을 통하여 패턴에 따른 상태 전이과정 계산을 테이블에 반영한다. 오토마타를 이용하여 중간코드 인식을 하고 각 상태 인식과정에서 치환될 최적화 패턴을 결정한다. 마지막으로 최적화 코드로 치환한다.

DFA 최적화 알고리즘의 궁극적인 목표는 중간 코드의 각 명령어 및 인자를 각 상태로 규정이 가능하며, 각 상태간의 전이과정은 결정적으로 확정할 수 있다.

4. 결론 및 향후 연구

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 기존의 최적화 기법들을 분석하고 기법들의 단점을 해결하기 위한 방안으로 결정적 유한 오토마타(Deterministic Finite Automata)를 이용한 코드 최적화를 제안하였다. DFA 를 이용한 최적화는 중간코드를 다른 자료구조로의 변환 불필요하며, 오토마타를 구성하는 비용이 다른 패턴 매칭 기법보다 적으며, 오토마타를 통해 결정적으로 패턴이 확정됨에 따른 패턴 선택 비용이 절감될 것으로 생각된다.

향후 연구 과제로는 DFA 최적화 알고리즘을 보완 및 다양한 패턴이 결정적으로 최적화 시 발생할 수 있는 문제점과 DFA 최적화기 구현에 대한 연구가 필요하다.

참고문헌

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

[2] Christoph M. Hoffmann & Michael J. O'Donnell, "Pattern Matching in Trees," Journal of the Association for Computing Machinery, Vol. 29, No.1, pp. 68 ~ 95, Jan., 1982.

[3] Christopher W. Fraser, Todd A. Proebsting, "Finite-State Code Generation," ACM SIGPLAN, 1999.

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

[5] Susan L. Graham, "Table-Driven Code Generation," IEEE Computer, Vol.13, No.8, pp. 25 ~ 34, Aug., 1980.

[6] Todd A. Proebsting, "BURS automata generation," ACM Transactions on Programming Languages and Systems, 17(3) pp. 461~486, May., 1995.

[7] 김정숙, 트리패턴매칭기법을 이용한 중간코드 최적화 시스템의 설계 및 구현, 동국대학교 박사학위 논문, 1998.

[8] 오세만, 컴파일러 입문 개정판, 정익사, 2004.