EM에서 SPARC 코드로 효율적인 코드 확장

오 세 만[†] · 윤 영 식[††]

요 약

ACK는 가상 스택 기계에 기반을 둔 EM 중간 코드로 부터 레지스터 구조에 기반을 둔 SPARC 기계에 대한 목적 코드를 생성하기 위해서는 코드 확장기(code expander)를 이용하고 있다. 따라서 EM 코드로부터 SPARC 목적 코드를 생성하기 위해 스택 지향 구조로부터 레지스터 지향 구조로 변환하여야 한다. 코드 확장기를 이용한 SPARC 코드 생성 기법은 각 EM 명령어에 대해 SPARC 코드로 확장하는 루틴들로 구성되며 코드 생성기에 비해 코드의 질을 개선하기 위해 푸쉬-팝 최적화 동작을 수행한다. 하지만 코드 확장시에 별도의 자원과 관리를 요구하는 혼합 스택(hybrid stack)을 이용하고 있으며 전단부의 정보 손실로 레지스터 윈도우를 이용한 효율적인 매개변수 전달을 고려하지 않는다.

본 논문에서는 ACK의 전체적인 구조의 변경없이 목적 기계의 스택과 매개변수 전달을 고려한 효율적인 SPARC 코드를 생성하기 위해 EM 트리를 이용한 SPARC 코드 확장기를 설계하고 구현하였다. 이를 위해, 순차적인 EM 코드를 입력으로 받아 스택 속성을 반영한 트리로 구성하며 혼합 스택을 제거하기 위해 지역 변수 정보를 별도로 관리하였다. EM 트리의 순회 및 확장 과정에서 목적 코드를 생성할 수 있는 루틴을 통하여 목적 코드를 출력하며 추출된 정보와 노드의 성격에 출력 시기와 목적 코드를 결정한다.

An Efficient Code Expansion from EM to SPARC Code

Se Man Oh[†] · Young Shick Yun[††]

ABSTRACT

There are two kinds of backends in ACK : code generator(full-fledged backend) and code expander(fast backend). Code generators generate target code using string pattern matching and code expanders generate target code using macro expansion. ACK translates EM to SPARC code using code expander. The corresponding SPARC code sequences for a EM code are generated and then push-pop optimization is performed. but, there is the problem of maintaining hybrid stack. And code expander is not considered to passes parameters of a procedure call through register windows.

The purpose of this paper is to improve SPARC code quality. We suggest a method of SPARC code generation using EM tree. Our method is divided into two phases: EM tree building phase and code expansion phase. The EM tree building phase creates the EM tree and code expansion phase translates it into SPARC code. EM tree is designed to pass parameters of a procedure call through register windows. To remove hybrid stack, we extract an additional information from EM code. We improved many disadvantages that arise from code expander in ACK.

※본 연구는 한국과학재단의 목적기초 연구(과제번호:93-0100-01-01-03) 지원에 의한 것임.

† 종신회원: 동국대학교 컴퓨터공학과 교수
†† 정 회 원: 동국대학교 컴퓨터공학과 석사과정 수료

논문접수: 1997년 2월 10일, 심사완료: 1997년 3월 13일

1. 서 론

컴퓨터 하드웨어의 기술이 급속도로 발전하고 소프트웨어의 규모와 복잡성이 증가함에 따라 신뢰성이 있고 효율적으로 프로그래밍할 수 있는 프로그래밍 언어의 설계 및 이에 따른 컴파일러를 자동적으로 생성할 수 있는 도구에 대한 연구가 진행되고 있다. ACK(Amsterdam Compiler Kit)는 컴파일러의 후단부를 자동화하기 위한 도구로서 이식성과 재목적성이 매우 높은 컴파일러를 만들기 위해 제작되었다[5].

ACK는 가상 스택 기계에 기반을 둔 EM 중간 코드로 부터 CISC 계열의 기계에 대한 목적 코드를 생성하기 위해 코드-생성기 생성기, 코드 생성기로 구성된 코드 생성 시스템(code generation system)을 이용하며 보다 양질의 코드를 생성하기 위해 스트링 패턴 매칭 기법을 적용하고 있다. 하지만 레지스터 집중적 구조(register intensive architecture)를 가진 SPARC 기계에 대한 목적 코드를 생성하기 위해서는 코드 확장기를 이용하고 있다. 따라서 EM 코드로부터 SPARC 목적 코드를 생성하기 위해 스택 지향 구조로부터 레지스터 지향 구조로 변환하는 과정에 기반을 두고 있다[5].

코드 확장기를 이용한 목적 코드 생성 기법은 각 EM 코드에 대해 목적 코드로 확장하는 루틴들로 구성되며 코드 생성기에 비해 코드의 질은 떨어지지만 빠른 시간 내에 효과적으로 코드를 생성할 수 있는 장점을 갖는다[2]. ACK에서 SPARC 기계에 대한 코드 생성 방법으로 코드 확장기를 이용하는 이유는 다음과 같다. 첫째, 코드 생성기에 비해 구현이 용이하다. 둘째, 기계 종속적으로 설계된 컴파일러에 비해 전단부의 정보 손실이 많다. 마지막으로 SPARC과 같은 구조에서는 코드 확장기를 이용하여 생성된 목적 코드의 질이 반드시 떨어진다고 볼 수 없기 때문이다. 코드 확장기는 각 EM 코드에 대해 해당 루틴에서 목적 코드를 출력하며 푸쉬팝 최적화 기법을 이용하여 코드의 질을 개선하고 있다. 그러나, 별도의 자원과 관리를 요구하는 혼합 스택을 이용하고 있으며 전단부의 정보 손실로 레지스터 윈도우를 이용한 효율적인 매개변수 전달을 고려하지 않는다[3].

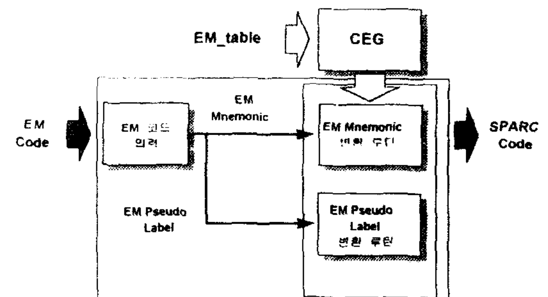
본 논문에서는 ACK의 전체적인 구조의 변경없이 목적 기계의 스택과 매개변수 전달을 고려한 효율적인 SPARC 코드를 생성하기 위해 EM 트리를 이용한 SPARC 코드 확장기를 설계하고 구현하였다. 이를 위해, 순차적인 EM 코드를 입력으로 받아 스택 속성을 반영한 트리로 구성하며 혼합 스택을 제거하기 위해 지역 변수 정보를 별도로 관리하였다. EM 트리는 피연산자와 매개 변수를 자식 노드로 연결하고 EM 명령어의 입력 순서를 유지하고 있다. 각 노드는 코드의 중복을 줄이기 위해 확장 성격을 기준으로 구분하였다. EM 트리의 순회 및 확장 과정에서 목적 코드를 생성할 수 있는 루틴을 통하여 목적 코드를 출력하며 추출된 정보와 노드의 성격에 따라 출력 시기와 목적 코드를 결정한다.

본 논문의 2장에서는 ACK의 SPARC 코드 확장기에 대한 전반적인 사항에 대해 살펴보고 있다. 3장은 EM 트리 변환을 통한 목적 코드의 확장에 대한 사항으로, 시스템의 개요 및 구성, EM 트리의 구성 방법 및 과정, 지역 변수 관리, 목적 코드로의 확장에 대해 알아본다. 4장에서 코드 질에 대한 비교 평가를 하고, 마지막으로 5장에서 결론을 맺는다.

2. ACK의 SPARC 코드 확장

2.1 시스템의 개요

코드 확장기는 빠른 속도로 중질(medium-quality)의 목적 기계 코드를 생성하는 도구로서 EM 중간 코드에 대해 목적 기계 코드를 생성하는 기능과 생성된 목적 코드를 재배치 가능한 형태로 변환한다. 그러나, RISC 계열의 SPARC은 CISC 계열의 기계와는 다른 재배치 정보를 유지하므로 CISC 계열을 기반으로 설



(그림 1) ACK의 SPARC에 대한 코드 확장기의 구성
(Fig. 1) A Configuration of SPARC code expander in ACK

계된 ACK에서는 SPARC에 대해 어셈블리 수준의 코드까지만 생성한다[2][3].

코드 확장기는 (그림 1)과 같이 각 EM 코드에 대한 확장 루틴들로 구성된다. EM 코드의 확장 루틴은 코드 확장기 생성기(Code expander generator)의 출력인 EM 명령어 처리 루틴과 의사 명령어 및 레이블을 처리할 수 있는 루틴으로 구성된다. 코드 확장기 생성기의 입력은 EM_table이며 목적 기계의 어셈블리 코드를 생성할 수 있는 정보를 기술하고 있다.

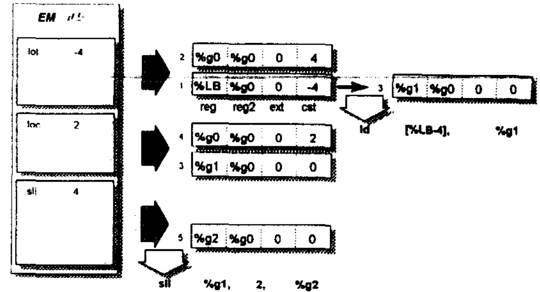
2.2 SPARC 코드 확장

ACK에서는 EM 중간 코드로부터 RISC 프로세서 구조를 갖는 SPARC에 대한 목적 코드를 생성하기 위해 스택 지향 구조로 부터 레지스터 지향 구조로 변환하는 과정에 중점을 두고 있다. 이 과정에서 SPARC 코드 확장기를 이용한 코드 생성 방법의 주된 특징은 EM 스택을 목적 기계의 스택과 접목시킨 혼합 스택과 푸쉬팝 최적화 기법으로 집약할 수 있다.

SPARC의 스택은 자동 변수들을 위한 공간으로 푸쉬와 팝을 이용한 자료의 접근을 제공하지 않는 반면, 스택 포인터와 프레임 포인터를 저장하고 있는 레지스터를 이용하여 값의 적재와 저장이 이루어진다. 그러나, 혼합 스택은 목적 기계의 스택에 EM의 스택 공간을 함께 지원하며 EM 기계의 스택 포인터(%SP)와 프레임의 위치를 가리키는 %LB 레지스터를 별도로 배정하여 운용하고 있다.

SPARC에서는 파이프라인을 통하여 각 명령어가 실행되며 명령어들이 일정한 수의 사이클 이내에 수행됨으로써 효과를 얻을 수 있다. 따라서, 저장과 적재 연산을 제외한 모든 연산은 레지스터에서 행해지고 있다. 각 EM 명령어에 대한 목적 기계 코드로의 코드 확장은 스택의 속성을 반영하여 빈번한 저장과 적재 연산을 수반하게 된다. 따라서, ACK에서는 푸쉬팝 최적화 기법을 적용하여 메모리 참조의 수를 감소시키고 있다. 푸쉬팝 최적화 기법은 푸쉬 연산을 최대한 지연시키고 이를 가능한 다른 EM 명령어의 입력으로 이용하여 메모리 참조의 수를 줄이는 기법으로써 가상 스택(fake stack)과 유사한 캐쉬(cache)라는 임시적 구조를 이용하고 있다. 캐쉬는 다음에 수행될 연산자에 대한 피연산자로서 이용될 레지스터 정보, 상수, 스트링을 저장할 수 있는 구조를 갖는다.

(그림 2)는 푸쉬팝 최적화 기법을 적용한 출력 과정과 결과를 예로 보여주고 있다.

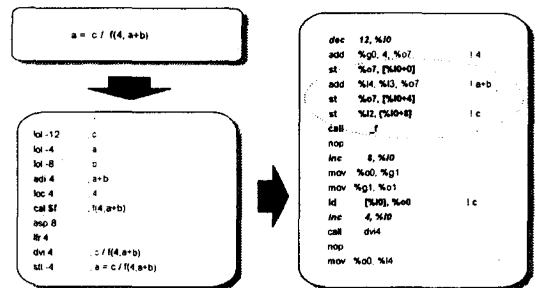


(그림 2) 푸쉬팝 최적화 기법을 이용한 코드 확장
(Fig. 2) Code expansion using push-pop optimization technique

2.3 생성된 목적 코드에 대한 고려 사항

ACK에서는 푸쉬팝 최적화 기법을 이용하여 코드 확장기에 의해 생성된 코드의 질을 개선하고 있다. 그러나, CISC 구조에 적합한 구조를 갖는 EM의 특징으로 인하여 혼합 스택 관리를 위한 부담과 레지스터 윈도우를 통한 매개 변수 전달을 고려하고 않는 단점을 갖는다.

혼합 스택은 별도의 레지스터 배정, 스택 공간 및 이를 관리하기 위한 부가적인 목적 코드를 요구한다. 또한, 매개 변수들을 혼합 스택 공간을 통하여 전달함으로써 기존 라이브러리와의 연동을 억제하고 레지스터 윈도우의 중첩에 따른 효율적인 매개 변수 전달이 목적 코드에 반영되지 않는다. 프로시저 호출 명령어



(그림 3) ACK의 SPARC 코드 출력 예
(Fig. 3) An example of SPARC code in ACK

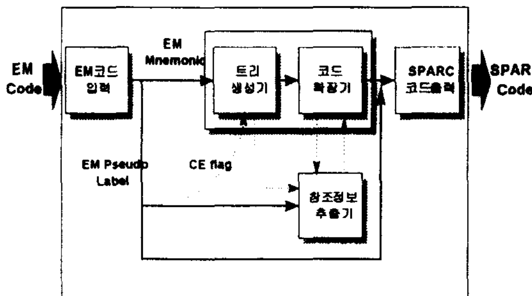
에 대한 확장 시점에서 캐쉬에 남아있는 모든 피연산자는 매개 변수의 대상이 되며 불필요한 저장과 적재를 유발한다. (그림 4)와 같이 매개 변수들은 혼합 스택의 %SP로 배정된 %l0의 상대적인 위치에 저장되고 혼합 스택을 관리하기 위한 inc, dec 명령어를 사용하며 변수 c는 불필요하게 저장 및 적재되고 있다.

3. EM 트리 변환을 통한 SPARC 코드 확장

3.1 시스템의 개요

본 논문에서는 EM 코드로부터 RISC 구조에 보다 적합한 SPARC 코드로의 변환을 목적으로 하고 있다. 순차적인 EM 코드를 스택 기반 구조의 속성을 반영한 EM 트리로 재구성하고 자동 변수에 대한 정보를 추출하였다. 이를 통해 혼합 스택을 제거하고 레지스터 윈도우를 이용한 매개 변수 전달이 이루어 지도록 하였다. 구성된 EM 트리는 순회하면서 목적 코드로 확장되며 추출된 정보와 노드의 확장 성격에 따라 목적 코드를 출력한다. 또한, 확장된 어셈블리 수준의 목적 코드는 기존의 어셈블러와 로더를 통해 실행 가능한 목적 코드로 변환하였다.

EM 트리를 이용한 SPARC 코드 확장기의 구성은 (그림 4)와 같이 트리 생성기와 참조 정보 추출기, 코드 확장기로 구성된다. 트리 생성기는 EM 명령어를 입력으로 받아 EM 트리를 생성하는 단계로서 EM 인터프리터 정보를 이용하여 트리를 생성한다. EM 트리는 연산자에 대한 피연산자 및 매개 변수를 자식 노드로 유지함으로써 매개 변수 정보 추출 및 관계를 유지할 수 있는 구조를 갖는다. 참조 정보 추출기는 코드 확장시 필요한 자동 변수 정보를 추출하는 단계로 이를 통하여 혼합 스택의 요소를 배제하기 위해 설계되었다. 코드 확장기는 EM 트리를 입력으로 받아 목적 코드를 생성하는 단계로서 ACK의 캐쉬 구조와 유사한 코드 확장 스택을 이용하여 푸쉬팝 최적화 기법을 적용하고 자동 변수 정보를 이용하여 목적 코드를 출력한다.



(그림 4) EM 트리를 이용한 SPARC 코드 확장기
(Fig. 4) A configuration of SPARC code expander using EM tree

3.2 EM 트리의 생성

EM 트리는 연산자와 피연산자의 관계 및 매개 변수를 유지할 수 있는 구조로서 EM 인터프리터 정보를 이용하여 구성된다. EM 인터프리터 정보는 각 EM 명령어에 대해 스택 운용 정보와 항목의 크기 정보를 표현하고 있다. 동작을 의미하는 정보로는 스택에 영향을 주지 않는 동작('0'), 푸쉬('+'), 팝('-')이 있고 항목에 대한 정보로 목적 기계의 워드 크기('w'), 주소 크기('p'), EM 명령어의 인자값 크기('a'), 이중 워드 크기('d'), 스택의 top에 위치한 항목에서 명시한 크기('x'), 스택의 top-1에 위치한 항목에서 명시한 크기('y'), 알 수 없는 크기('?')가 있다. 예를 들어 EM 명령어 adi("-a-a +a")는 인자값의 크기를 갖는 두 피연산자를 스택에서 팝하고 그 결과를 다시 인자값의 크기로 푸쉬하는 동작을 수행한다. EM 트리를 생성하는 과정에서 각 EM 명령어는 노드로 구성되고 동작을 모의하기 위해 스택 구조인 트리 생성 스택을 이용하였다. EM 트리의 노드 구조는 다음과 같다.

```
typedef struct AttrEM-Node {
  int        _nodeid;        /* 노드 구분 ID */
  node_t     _nodeflag;      /* 변환, 지연, 모조, 스킵 노드 */
  int        _size;          /* EM 트리 생성시의 항목의 크기 */
  int        _reg, _reg2;    /*매개 변수에 대한 레지스터 사전 정보*/
  struct     e_instr _em;    /* EM 코드의 세부 정보 구조 */
  struct     AttrEM_Node  *son;
  struct     AttrEM_Node  *brother;
} AttrEM_Node;
typedef struct AttrEM_Node *AttrEM_NodePtr;
```

EM 트리 생성기는 각 EM 명령어에 대해 [알고리즘 1]과 같이 EM 인터프리터 정보를 이용하여 트리

를 구성하는 단계와 (그림 5)의 EM 명령어 cal과 lfr에서 처럼 별도의 루틴을 이용하여 구성하는 단계로 나눠진다. cal 명려어는 후위 명령어를 이용하여 매개 변수를 트리 생성 스택으로부터 구할 수 있다. (그림 5)에서는 후위 EM 명령어인 asp 8을 이용하여 트리 생성 스택에서 8 크기의 매개 변수를 팝하여 형제 노드로 연결하고 cal의 자식 노드 포인터에 연결한다.

[알고리즘 1] EM 인터프리터 정보를 이용한 EM 트리 생성

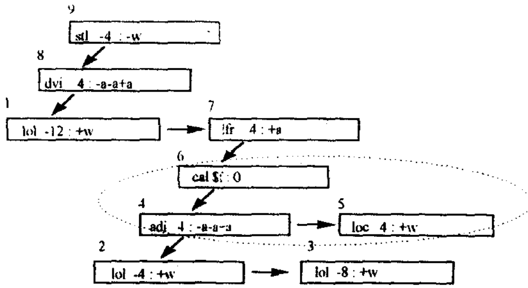
```
BEGIN
    pushflag = false
    for EM 인터프리터 정보 ≠ empty do
        Action과 Item Size 정보를 얻는다.
        case Action of
            '-':{ 자식 노드를 팝한 후 자식 노드
                 리스트에 추가 }
            '+':{ 자식 노드를 형제 노드로 연결
                 한 후 포인터에 연결
                 EM 트리 생성 스택에 푸쉬
                }
            otherwise:;
        end
    end for
    if pushflag ≠ true then {
        자식 노드를 형제 노드로 연결한 후 현 노
        드의 자식 노드 포인터에 연결
        노드의 항목 크기를 0으로 설정하고 노드
        를 모조 노드로 설정
        EM 트리 생성 스택에 푸쉬 }
    fi
END
```

EM 트리의 각 노드는 코드 확장 시기와 목적 코드 선정을 위해 다음과 같이 구분하였다. 상위 노드와의 관계를 고려하여 변환 노드와 지연 노드로 나눠지고 순서적 의미의 모조(dummy) 노드, 확장 여부를 나타내는 스킵(skip) 노드로 구분하였다. 노드의 구분은 주로 EM 트리 생성 단계에서 결정되며 스킵 노드는 코드 확장 단계에서 결정된다. 이 중에서 변환 노드는 상위 노드와 관계없이 목적 코드로 확장되고 결과값을 담고 있는 레지스터를 상위 노드에서 이용한다. 지연 노드는 출력을 지연하고 상위 노드에서 출력 시기를 결정하며 상위 노드에 의존적인 피연산자에 대해 중복을 줄인 목적 코드의 출력을 목적으로 한다. (그림 5)의 EM 트리에서 cal의 모든 자식 노드는 지연 노드로 설정된다. 모조 노드는 EM 트리 구성시 순서적인 의미의 상실을 방지하기 위한 노드로 (그림 5)의 노드 stl은 순서적 의미로만 푸쉬되고 차기 EM 명령에 대해 피연산자의 의미를 갖지 않는 자식 노드로 연결될 수 있다. 따라서, 모조 노드는 상위 노드 및 형제 노드와 무관하게 목적 코드로 출력하게 된다. 스킵 노드는 코드 확장기의 순회 과정에서 이미 목적 코드로 확장된 노드를 의미한다.

3.3 목적 코드 확장을 위한 참조 정보의 구성

SPARC과 같은 RISC 기계에서는 변수에 대해 최대한 레지스터로 유지함으로써 적재와 저장 연산을 줄일 수 있다. 본 논문에서는 ACK의 지역 변수 테이블을 확장하고 이를 코드 확장 과정에서 이용할 수 있도록 하였다. 지역 변수에 대한 정보는 EM 의사 명령어 mes 3에서 제공하고 있다. mes 3은 오프셋, 변수의 크기, 변수의 종류, 가중치의 정보를 제공한다. mes 3에서 제공된 지역 변수의 오프셋 정보는 목적 기계의 스택에서 프레임 포인터에 대한 상대적 위치를 계산하기 위한 정보로 이용된다. 또한, 가중치는 레지스터를 대피(spill)하거나 최대한 변수를 레지스터에 유지하기 위한 지역 변수 선택 정보로 이용된다. 지역 변수 테이블은 EM 의사 명령어인 mes 3에서 제공하는 EM 스택에 대한 오프셋, 크기, 종류, 가중치와 목적 기계 스택에 대한 오프셋과 레지스터 플래그를 갖는다.

각 자동 변수에 대한 참조는 EM 명령어의 인자값에 나타난 오프셋을 키로 하여 지역 변수 테이블을 검색하고 레지스터는 코드 확장 과정에서 적재와 저장 연산시 할당된다. 최대한 지역 변수들을 레지스터로 유지하기 위해 mes 3의 가중치 정보를 이용하여 레지스터를 할당하고 이를 생존 범위 내에서 최대한 유지하였다. 그러나, 변수의 값이 적재된 주소를 통해 변경되는 연산이 발생할 경우 저장과 재적재가 이루어진다. 매개 변수로 이용된 변수에 대한 값을 담고



(그림 5) EM 트리의 구성 예
(Fig. 5) An example of EM tree

있던 레지스터의 값은 실제 변수가 가지고 있는 값과는 일관성을 갖지 못하기 때문에 해당 레지스터로 재적재를 행하는 목적 코드를 출력해야 한다.

3.4 효율적인 SPARC 코드로의 확장

구성된 EM 트리는 레이블과 분할의 의미를 갖는 EM 의사 명령어에 의해 확장 시기가 결정되고 입력 순서에 준하여 순회하면서 목적 코드를 출력한다. 확장 시점에서 트리 생성 스택에는 한 개 이상의 트리가 존재하게 된다. 트리 생성 스택에 존재하는 각 EM 트리의 루트는 저장에 관련된 명령어이거나 분기 명령어, 비교 명령어 등이다. 확장 시점에서 트리 생성 스택의 인덱스 0으로부터 top으로 진행하며 각 EM 트리를 순회한다. 각 EM 트리의 순회 순서는 EM 명령어의 입력 순서와 동일하며 [알고리즘 2]와 같이 순회된다.

[알고리즘 2] EM 트리 순회 알고리즘

```
Trav_EMTree(node)
BEGIN
    if (node)
        Trav_EMTree(node → son);
        CodeExpansion(node);
        Trav_EMTree(node → brother);
    fi
END
```

코드 확장기에서는 EM 트리를 순회하며 목적 코드를 출력하고 차기 EM 코드에서 필요한 피연산자 정보를 저장하고 이용하기 위해 코드 확장 스택을 이용하였다. 코드 확장 스택을 이용하는 주된 이유는 EM 노드의 구조를 단순화시키고 목적 코드 출력에 의존적인 요소를 EM 트리 생성 과정에서 배제시킬 수 있다. 따라서, 구현이 용이하고 EM 명령어간의 상호 연관성을 쉽게 고려할 수 있다는 장점을 갖는다. 코드 확장 스택은 ACK의 캐쉬 구조와 유사한 구조이며 목적 기계 스택에 대한 오프셋 필드를 추가하여 다음과 같은 구조를 갖는다.

```
typedef struct CE_StackElement {
    int           nodeid;
    LoadType type;  /* reg, string, cst, offset etc. */
    reg_t         reg, reg2;
    long          cst;
    char          *ext;
    int           offset;
} CE_StackElement;
```

피연산자는 코드 확장 스택에 푸쉬되고 코드 확장 스택의 top에 위치한 피연산자는 연산자의 요구에 따라 레지스터나 상수 및 스트링 정보를 제공한다. 그러나, 요구의 형태가 top에 있는 피연산자 정보와 다를 경우에는 요구 형태로 변환하는 목적 코드를 출력하고 그 결과를 제공한다. 코드 확장시에 효과적인 레지스터의 할당을 위해 본 연구에서는 ACK의 방법을 확장하여 다음과 같은 구조를 이용하였다.

```
typedef struct regdat_t {
    int       inuse;    // 사용 빈도, 만약 0일 경우 할당 가능
    boolean   localflag; // 지역 변수에 할당된 레지스터 유무
    int       localidx;  // 지역 변수 테이블의 index
};
static struct regdat_t     _reg[TOTAL_REGS]
```

레지스터의 할당은 요구 시점에서 일어나며 순차적으로 검색하여 할당 가능한 레지스터를 선정한다. 만약, 요구 시점에서 가용한 레지스터가 존재하지 않을 경우 코드 확장 스택에 푸쉬되어 있는 레지스터 중에서 선정하여 할당한다. 선정된 레지스터에 대해서는 메모리로 적재되는 목적 코드 출력하고 해당 정

보는 스택에 대한 오프셋으로 변경된다. 메모리로 저장될 주된 레지스터는 자동 변수이며 지역 변수 테이블의 가중치 값과 레지스터 관리 테이블의 사용 빈도에 따라 결정된다. 자동 변수 이외에 대피될 레지스터는 스택의 임의 공간에 저장되고 목적 코드의 스택 크기를 조정한다. 따라서, 목적 코드의 스택 크기는 지역 변수 정보와 목적 코드 확장 과정에서 결정된다.

각 EM 노드는 1:1로 확장 함수와 매핑되고 실질적인 목적 코드 확장은 변환 노드, 지연 노드, 모조 노드에서 이루어진다. 변환 노드는 목적 코드에서 연산의 결과 값을 담고 있는 레지스터를 코드 확장 스택에 푸쉬하여 차기 명령어의 피연산자로 이용될 수 있지만 저장, 분기, 비교 등의 의미를 갖는 모조 노드는 코드 확장 스택에 반영되지 않는다. 지연 노드의 상위 노드에서 확장 시기가 결정되며 상위 노드에서 요구한 레지스터를 이용하여 해당 목적 코드만을 출력한다. EM 명령어 loc에 목적 코드 확장 함수는 다음과 같다.

```
yOut_loc(node)
AttrEM_NodePtr node;
{
 if (node_flag(node) == TransNode) {
      CE_PushCst(node_id(node), node_emcst(node));
      // 코드 확장 스택에 상수를 push
      if (node_reg(node) != reg_g0) {
                    // reg_g0는 %g0로 항상 상수 0.
           따라서, reg_g0를 null로 이용
           CE_Pop(node_reg(node), node_reg2(node));
                    // 매개 변수인 레지스터로 코드
           확장 스택 top의 값을 move한다
      }
      node→ _nodeflag = SkipNode;
 }
}
```

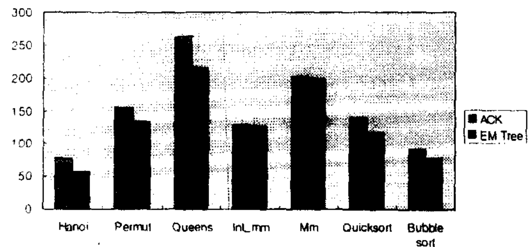
EM 명령어 loc의 확장 함수에서는 변환 노드일 경우에만 목적 코드를 출력한다. 노드 loc가 매개 변수가 아닐 경우에는 코드 확장 스택에 상수값을 푸쉬하고 차기 연산자의 피연산자로 이용된다. 그러나, 지연 노드로 설정된 매개 변수일 경우에는 정해진 출력 레지스터를 이용하여 목적 코드를 출력함으로써 불필요한 코드의 중복을 억제할 수 있다.

4. 코드질에 대한 평가

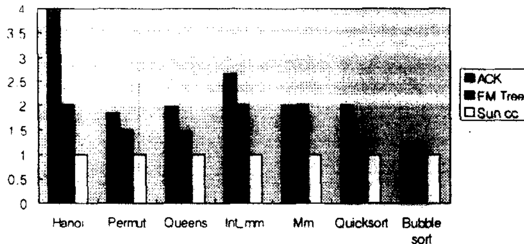
EM 트리를 이용한 SPARC 코드 확장기와 ACK에서 생성된 목적 코드를 평가하기 위해 동일한 EM 코드를 입력으로 ACK에서 출력한 SPARC 코드 간의 목적 코드의 크기와 실행 시간을 비교하였다. 실행 시간을 비교는 본 시스템과 ACK, Sun의 cc를 대상으로 하며 −O2를 동일한 옵션으로 제공하였다. 평가 환경은 Sun SparcStaction 5의 SunOS 4.1.3 환경 하에서 측정하였고 C 언어로 작성된 Stanford 벤치마크 프로그램을 이용하였다.

어셈블리 수준의 목적 코드 비교는 (그림 6)과 같이 혼합 스택을 이용한 ACK가 많은 목적 코드를 수반하는 것으로 나타나고 있다. 실행 속도의 비교에 있어서는 (그림 7) 같이 Sun의 cc를 1로 정하고 이에 대한 상대적인 실행 시간을 측정하였다. 빈번한 매개 변수 전달이 수반되는 함수 호출 시에는 EM 트리를 이용한 SPARC 코드 확장기가 ACK보다 우수하게 나타나고 있으며 Sun의 cc에 가까운 실행 속도를 유지하고 있다. 그러나, 매개 변수를 수반한 함수 호출의 수가 적고 전역 변수, 실수형 연산에 대해서는 ACK와 유사한 결과를 얻었다.

이식성과 재목적성을 목적으로 설계된 ACK의 EM 코드의 특징을 반영한 목적 코드의 질은 기계 종속적으로 설계된 컴파일러와의 실행 시간 면에서 많은 차이를 보이고 있다. 그러나, EM 트리를 이용한 SPARC 코드 확장기는 ACK에 비해 SPARC의 특징을 고려한 목적 코드를 출력하여 코드의 질을 개선하고 있다.



(그림 6) 목적 코드의 크기 비교
(Fig. 6) A comparison of target code size



(그림 7) 목적 코드의 실행 시간 비교
(Fig. 7) A comparison of run-time for target code

5. 결 론

ACK는 가상 스택 기계에 기반을 둔 EM 중간 코드로 부터 CISC 계열의 기계에 대한 목적 코드를 생성하기 위해 코드-생성기 생성기, 코드 생성기로 구성된 코드 생성 시스템(code generation system)을 이용하며 보다 양질의 코드를 생성하기 위해 스트링 패턴 매칭 기법을 적용하고 있다. 하지만 레지스터 구조에 기반을 둔 SPARC 기계에 대한 목적 코드를 생성하기 위해서는 코드 확장기를 이용하고 있다. 따라서 ACK에서는 EM 코드로부터 SPARC 목적 코드를 생성하기 위해 스택 지향 구조로부터 레지스터 지향 구조로 변환하는 과정에 중점을 두고 있다.

ACK에서 SPARC 기계에 대한 목적 코드를 생성하기 위해 사용하고 있는 코드 확장 방법은 빠른 시간 내에 중질의 코드를 얻을 수 있으며 생성된 코드의 질을 개선하기 위해 푸쉬팝 최적화 기법을 이용하여 적재와 저장의 연산을 줄이고 있다. 하지만, ACK 전단부의 정보 손실로 인해 SPARC에서 제공하는 레지스터 윈도우의 장점을 이용하지 못하고 혼합 스택 구조를 이용하여 부가적인 자원과 목적 코드를 요구하는 단점을 갖는다.

본 논문에서는 ACK의 전체적인 구조 변경 없이 EM 코드로부터 SPARC 목적 코드르 생성하는 과정에서 ACK 코드 확장기의 매개 변수 전달 방법을 개선하여 효율적인 SPARC 코드를 생성하기 위해 EM 트리를 이용한 SPARC 코드 확장기를 설계하고 구현하였다. 이를 위해, 순차적인 EM 코드를 입력으로 받아 스택 속성을 반영한 트리로 구성하며 혼합 스택을 제거하기 위해 지역 변수 정보를 별도로 관리하였다. EM 트리는 피연산자와 매개 변수를 자식 노드로 연결하고 EM 명령어의 입력 순서를 유지하고 있다. 각 노드는 코드의 중복을 줄이기 위해 확장 성격을 기준으로 구분하였다. EM 트리의 순회 및 확장 과정에서 SPARC 코드를 생성할 수 있는 루틴을 통하여 목적 코드를 출력하며 추출된 정보와 노드의 성격에 따라 출력 시기와 목적 코드를 결정한다. 이러한 코드 생성 방식은 빈번한 매개 변수 전달이 요구되는 함수 호출이 일어나는 경우와 혼합 스택의 의존도가 높은 경우에 대해서는 기존의 ACK의 코드 확장 방식에 비해 우수하게 나타났지만 함수 호출이 적고 전역 변수, 실수형 연산에 대한 목적 코드의 평가에서는 유사한 결과를 얻었다.

앞으로 레지스터의 효율적인 할당과 목적 코드 선택 기법을 고려한 효율적인 목적 코드 확장기를 구현할 것이며 EM 코드로부터 RISC 프로세서 구조를 가진 목적 코드를 생성하기 위한 보다 효율적인 변환 방법을 개선할 것이다.

참 고 문 헌

[1] David G. Bradlee & susan J. Eggers & Robert R. Henry, "Integrating Register Allocation and Instruction Scheduling for RISCs," Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1991.

[2] Frans Kaashoek, Koen Langendoen, "The Code Expander Generator," Dept. of Mathematics and Computer Science, Vrije Universiteit, 1989.

[3] Philip Homburg, Raymond Michiels, "A Fast Backend for SPARC Processor," report-81, Netherlands Vrije Universiteit, 1989.

[4] Richard P. Paul, "SPARC ARCHITECTURE, ASSEMBLY LANGUAGE PROGRAMMING, AND C", Prentice-Hall, 1994.

[5] Hans van Staveren, "The table driven code generator from ACK 2nd. Revision," report-81, Netherlands Vrije Universiteit, 1989.

[6] Sun Microsystems, The SPARCTM Architecture Manual, Revision 50 of August, 1987.

[7] Sun Microsystems, SPARC Assembly Language reference Manual, Sun Microsystems, 1994. 8.
[8] Andrew S. Tanenbaum, Hans van Staveren and Johan W. Stevenson, "A Practical Tool Kit for Making Portable Compilers," CACM, Vol. 26, No. 9, Sep., 1983.
[9] David W. Wall, "Register Windows vs. Register Allocation", Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, June 1988.
[10] 윤영식, 고광만, 오세만 "속성 EM-트리를 이용한 SPARC 코드 생성기의 설계 및 구현," 96' 봄 학술발표논문집, 23권 1호, pp. 377-380, 한국정보과학회, 1996. 4.
[11] 최광무 외, "컴파일러 개발에 관한 연구(2)", 한국과학기술원, 1989.
[12] 한국자원개발연구소, "ISDN CHILL 컴파일러 연구, 최종연구보고서", 한국전자통신연구소.

오 세 만

1977년 서울대학교 사범대학 수학과(학사)
1979년 한국과학기술원 전산학과(석사)
1985년 한국과학기술원 전산학과(박사)
1988년~1989년 미국 USL 대학 교환교수
1985년~현재 동국대학교 컴퓨터공학과 교수
관심분야:컴파일러, 프로그래밍언어, 병렬 및 분산처리 언어

윤 영 식

1995년 동국대학교 공과대학 컴퓨터공학과 졸업(공학사)
1997년 동국대학교 대학원 컴퓨터공학과 졸업(공학석사)
현재 한국통신기술주식회사 연구원
관심분야:프로그래밍 언어, 컴파일러, 멀티미디어