문법-지시적 변환 기법을 이용한 ARM 코드 생성 시스템

ARM Code Generation System using Syntax-Directed Translation Technique

고광만
상지대학교 컴퓨터정보공학부

Kwang-Man Ko(kkman@sangji.ac.kr)

요약

ARM 프로세서는 다양한 임베디드 시스템에서 활용되고 있다. 또한 대부분의 ARM 프로세서는 C 프로그램을 입력으로 받아 GNU gcc 크로스 컴파일 방식을 사용하여 ARM 어셈블리 코드를 생성한다. 또한 생성된 목적코드의 질을 개선하고 효율적인 목적코드 생성하기 위한 다양한 연구가 진행되고 있다. 본 논문에서는 표준 C 프로그램으로부터 ARM 프로세서에 대한 목적코드를 문법-지시적 변환 기법을 이용하여 생성하며 성능평가 결과를 GNU gcc 크로스 컴파일 방식과 비교하여 제시한다. 본 연구에서 제시한 기법은 생성규칙을 확장하는 방법이 GNU gcc 크로스 컴파일러에 비해 간편하고 편리하다.

■ 중심어 : | ARM 프로세서 | 코드 생성기 | 문법-지시적 변환 | 임베디드 시스템 |

Abstract

ARM processors are being utilized in a variety of embedded systems. It is also that most ARM processor accepts C application, and then generates ARM assembly code using GNU gcc Cross-compiler. For the purpose of improving the quality of code generated and the efficient code generation, the various researches are underway. In this paper, we generates the ARM assembly code from the ANSI C programs using Syntax-directed Translation Techniques, and then the performance evaluation results for our research experimental compare to GNU gcc Cross-compiler are described. The techniques are presented in this research compared to GNU gcc cross-compiler very simple and convenient in extension of the production rules.

■ keyword : | ARM Processor | Code Generator | Syntax-Directed Translation | Embedded System |

I. 서 론

최근 임베디드 시스템 분야는 프로세서 진화 및 개발, 응용 소프트웨어 및 소프트웨어 개발환경 요구, 비약적인 시장 확대, 전문 인력 요구 및 배출 등 모든 관련 부분에서 중요성 및 필요성이 적극적으로 대두되고 있다. 특히, 유비쿼터스 환경에서는 임베디드·모바일 장치 사용이 더욱더 증가될 것이며 이를 위한 소프트웨어 지원의 중요성과 필요성이 크게 부각되고 있다. 임베디드 시스템에 관련된 응용 분야의 비약적인 확장은 다양하고 시간에 민감한 프로세서의 개발 및 소프트웨어 개발을 요구한다. 즉, 프로세서의 개발 주기 및 새로운 프로세서 개발 요구가 빨라지고 이를 지원할 수 있는 소프트웨어 개발 환경인 특정 프로세서를 위한 컴파일러, 어셈블러 등의 지원이 중요한 요소가 되었다[1].

ARM 프로세서는 현재 다양한 임베디드 시스템에 탑재되어 활용되고 있으며, 특히 C 언어로 작성된 어플리케이션을 수행하기 위한 컴파일러, 어셈블러와 같은 소프트웨어 개발 도구도 다양하게 개발되고 있다[2]. 특히, ARM 프로세서에 적합한 목적코드 생성을 위해 대

접수번호 : #080523-001
접수일자 : 2008년 05월 23일
심사완료일 : 2008년 06월 16일
교신저자 : 고광만, e-mail : kkman@sangji.ac.kr

부분 gcc 크로스 컴파일러를 이용하여 전단부와 후단부를 사용하고 있다. 본 논문에서는 ARM 코드 생성을 위해 절대적으로 사용되고 있는 gcc의 의존도를 탈피하고 생성되는 코드 질의 효율성을 높이기 위해 문법-지시적 변환 방법을 적용하여 ARM 코드를 생성한다. 생성된 ARM 코드를 실제로 임베디드 툴 킷에 적용하여 실행결과를 검증하고 gcc를 통해 생성된 ARM 코드와 성능평가를 수행한 후 결과를 제시한다.

II. 관련연구

2.1 문법-지시적 변환 (Syntax-Directed Translation)

문법-지시적 변환[3]은 [그림 1]과 같은 구조로서 각 생성규칙에 있는 문법 심볼을 이용하여 그 생성규칙에 해당하는 의미 수행 코드를 직접 기술하여 필요한 일을 처리하는 방법으로서 파서가 의미 수행 정보를 호출하여 중간 코드를 생성하는 방법이다. CFG 형태로 기술된 생성규칙은 메타언어 또는 범용 프로그래밍 언어로 작성된 의미 수행 코드 또는 의미규칙을 이용하여 중간언어 생성, 심벌 테이블 관리를 수행한다. 이러한 문법-지시적 변환 방법을 이용하여 특정 기계어나 어셈블리어를 직접 생성하거나 소스 프로그램에 대한 추상화 구문 트리(AST)를 생성한 후 AST의 각 노드를 순회하면서 특정기계에 대한 실행 코드를 생성한다[4].

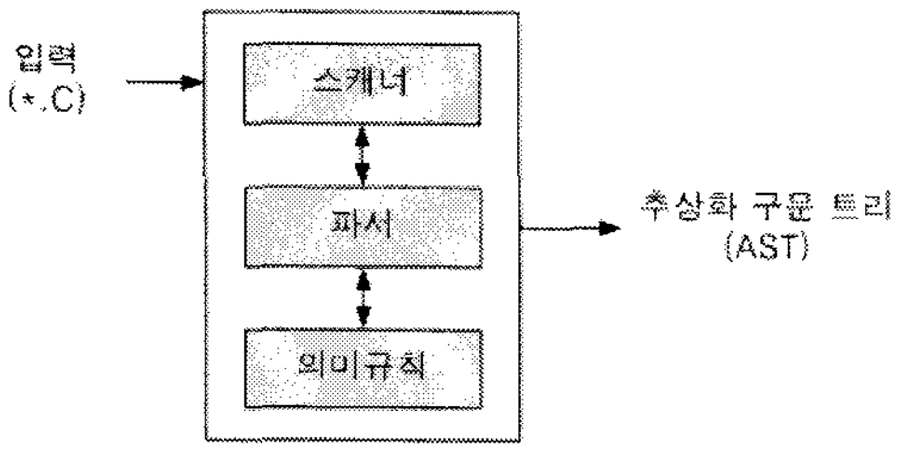

그림 1. 문법-지시적 변환기 구조

문법-지시적 변환기를 구축하기 위해서는 첫째, 입력언어에 대한 구문을 CFG 형태로 고안한 후 생성규칙을 고려하여 입력 문법을 설계한다. 둘째, 설계된 문법에 대한 파서 생성기(PGS)를 이용하여 어휘 정보와 파싱 테이블을 얻는다. 마지막으로 입력 언어의 문법 구조에 따라 생성규칙에 대한 적당한 의미 수행 코드를 문법-지시적 변환기의 목적에 따라 작성한다[5].

2.2 추상화 구문 트리(AST)

AST는 소스 프로그램에 대한 어휘분석, 구문분석을 수행한 후 파스 트리의 중간노드에서 표현되는 의미를 갖지 않는 노드를 제거하고 코드 생성 단계에서 이용되는 의미 정보만을 표현한다. AST는 입력에 대한 문법 구조를 표현하고 속성을 첨가하여 의미분석 단계에서 이용될 수 있고 속성이 첨가된 트리에 대해 기계 독립적인 최적화가 가능하고 코드를 생성하기 위해 이용될 수 있다[5][6].

소스 프로그램에 대해 AST와 같은 중간표현은 중간 코드 생성기에 의해 중간언어로 번역된다. 중간 코드 생성기는 문장의 의미를 분석하고 옳은가를 검사하는 의미분석 과정을 병행한다. 중간 코드를 생성하는 방법은 문법-지시적 변환 방법에 의해 구문분석 과정에서 직접 코드들 생성하는 방법과 파스트리 또는 AST와 같은 중간표현을 생성한 후 중간표현으로부터 중간 코드 또는 목적기계 코드를 생성하는 방법으로 구분된다. 각 방법은 장단점을 가지고 있으며 본 연구에서는 소스 프로그램에 대해 문법-지시적 변환 방법을 적용하여 AST를 생성한 후 AST를 순회하면서 ARM 어셈블리 명령어 및 실행 정보를 생성한다[7][8].

2.3 ARM 아키텍처 및 명령어

ARM은 내부버스, 레지스터, 연산장치 등이 32비트로 구성되어 한 번에 32비트 단위로 데이터를 처리할 수 있는 아키텍처이며 RISC에 기초하고 있다[1]. ARM 프로세서 코어는 ARM 아키텍처의 기본 원리를 이용하여 구현한 프로세서의 핵심 부분을 말하며 ARM 프로세서는 ARM 코어 캐시, MMU, 쓰기버퍼, TCM, BIU와 같은 주변 회로를 포함하는 독립된 형체를 말한다. ARM 프로세서 기반으로 프로그램을 작성하는 경우 프로그래머는 명령어, 메모리 구조, 데이터 구조, 프로세서 동작모드, 내부 레지스터 구성 및 사용법, 예외처리, 인터럽트 처리와 같은 프로세서 내부 구조 및 동

작에 대한 이해가 선행되어야 한다. 명령어는 32비트 ARM 명령어와 16비트 Thumb 명령어로 구분된다. 동작모드는 프로세서가 어떤 권한을 가지고 무슨 일을 처리하는지를 나타내는 동작상태를 나타내며 크게 7가지 동작모드로 구분된다. 레지스터는 32비트 길이의 범용 레지스터 30개, 프로그램 카운터 7개, 상태 레지스터 6개를 지원한다. 메모리 구조는 데이터 또는 프로그램을 8비트, 16비트, 32비트 단위로 읽거나 쓸 수 있으며 빅 엔디안과 리틀엔디안 두가지 방식을 모두 허용하고 있다. 실질적으로 ARM 명령어의 처리는 명령어를 읽고, 해석하고, 처리하고, 메모리에서 데이터를 읽고, 결과를 저장하는 과정이 1사이클 내에 가능하며 명령어 파이프라인을 방식을 이용하여 고속으로 실행될 수 있는 장점을 가지고 있다.

ARM 명령어는 모든 명령이 32비트로 구성되어 있으며 피연산자는 레지스터 또는 명령어 내에 포함되어 있는 Immediate 상수만을 사용할 수 있고 직접 메모리 내에 포함되어 있는 데이터를 사용하거나 그 결과를 메모리에 저장할 수 있다. 따라서 레지스터에 데이터를 읽어오고 레지스터에 저장된 결과를 저장하기 위해서는 별도의 명령어를 필요로 하는 Load/Store 구조를 갖으며 모든 주소 지정 방식은 상대주소 방식을 이용한다. 16비트 Thumb 명령어는 32비트 ARM 명령어와 유사한 형태를 포함하고 있으며 32비트 명령어가 16비트 길이로 재구성되어 있다. 16비트 Thumb 명령어를 지원하는 이유는 32비트가 아닌 16비트, 8비트 길이로도 ADD, SUB와 같은 32비트 ARM 명령이 할 수 있는 기능을 구현할 수 있다. 따라서 동일한 기능을 32비트 길이가 아닌 16비트로 구성한다면 코드 크기는 32비트에 비해서 훨씬 작아질 것이므로 제품의 단가를 낮추고 소형화하기 위해서 사용된다[9].

III. ARM 코드 생성 시스템 설계

3.1 코드 생성 시스템 개요

표준 C 언어로 작성된 어플리케이션을 입력으로 받아 ARM 코드를 생성하는 코드 생성 시스템은 [그림 2]와 같이 구성되어 있다. C 언어로 작성된 기존 연구 결과[4][5]를 이용하여 AST를 생성한다. AST 생성시에 효율적인 ARM 코드의 생성과 올바른 실행을 위해 의미 규칙을 추가하였다. 본 연구에서 구현한 트리 순회기는 생성된 AST를 순회하면서 ARM 코드를 생성한다. 생성 ARM 코드는 실제적인 실행이 가능한 형태이며 실제적으로 ARM 임베디드 킷인 EMPOS II에서 실행하여 그 결과를 제시한다.

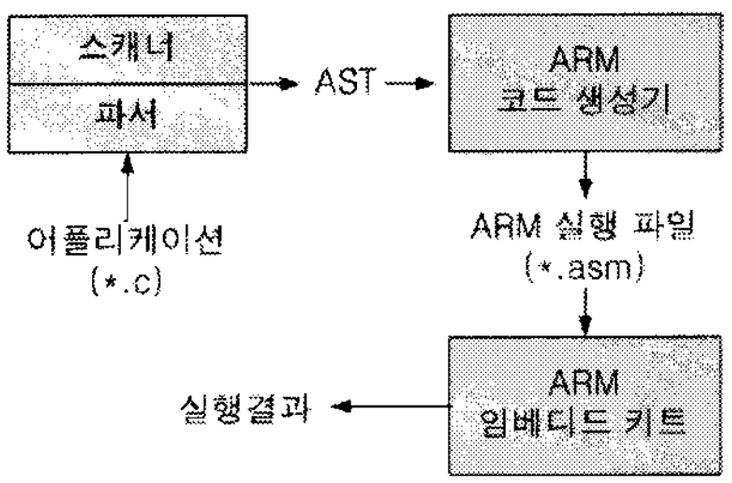

그림 2. ARM 코드 생성 시스템

3.2 ARM 코드 생성

생성된 AST를 순회하면서 ARM 코드의 생성은 선언부, 문장부, 함수부로 나누어 코드 생성 방법을 기술한 후 전체를 통합하여 완성된 코드와 실행 정보를 생성하였다.

선언부는 자료형을 가리키는 지정어 다음에 명칭 또는 명칭 리스트가 나열되는 문법적인 성질을 가지고 있으며 변수를 저장하기 위한 공간을 확보하기 위해 변수 이름과 저장될 위치의 (base, offset) 형태로 상대주소와 크기가 심볼 테이블에 저장된다. 정수형 선언시 4바이트의 기억공간을 기본적으로 할당한다. C 언어의 선언에 대해 실제로 생성되는 ARM 코드를 [표 1]과 같다.

표 1. 선언부에 대한 ARM 코드 생성

int x;	sub sp, sp, #4
int x, y;	sub sp, sp, #8

산술 연산을 위해 덧셈, 뺄셈, 곱셈은 직접적으로 ARM 명령어를 생성하지만 특별히 나눗셈과 나머지 연산을 위해서는 지정된 명령어가 없으므로 다른 명령어의 조합으로 최적의 ARM 코드를 [표 2]와 같이 생성한다.

표 2. 산술 연산에 대한 ARM 코드 생성

x+y	ldr ldr add	r3 , [fp, #-16] r2 , [fp, #-20] r3 ,r3 ,r2
x/y	ldr ldr bl	r1 , [fp, #-20] r0 , [fp, #-16] __divsi3
x%y	ldr ldr bl	r1 , [fp, #-20] r0 , [fp, #-16] __modsi3

배정문은 { 332*N | N>0 }의 공식으로 배정되므로 한번을 배정하고 남은 수를 배정한 수에 더하며 이를 위해 연산 전에 숫자가 332 * N 인지를 검사하는 과정을 거치며 [표 3]과 같은 ARM 코드를 생성한다.

표 3. 배정문에 대한 ARM 코드 생성

a=333;	mov add	r3, #332 r3, r3, #1
b=666;	mov add	r3, #664 r3, r3, #2

제어 명령은 조건, 분기, 반복 명령어를 포함하고 있으며 if-else, while 같은 문장에 대해 ARM 코드를 [표 4]와 같이 생성한다. if-else와 while의 조건을 cmp로 연산하여 그다음 줄의 분기 명령이 조건의 TRUE와 FALSE를 판단하여 분기한다. while문은 루프를 위해 계속 상위 .L3으로 분기한다.

표 4. 제어 명령에 대한 ARM 코드 생성

if-else	cmp r3 , #5 bge .L3 〈source〉 .L3
while	.L3 : ldr r3 ,[fp, #-16] cmp r3 , #5 bge .L4 〈source〉 b .L3 .L4

함수 호출은 실행 후 복귀되어야 하는 복귀주소가 필요하기 때문에 복귀주소를 자동으로 LR에 저장해주는 BL 분기문을 [표 5]와 같이 생성한다. 또한 함수 호출시에 전달되는 실매개 변수를 위해 실매개 변수의 개수를 계산하여 레지스터 r0부터 차례로 저장되고 함수가 호출된 후에 호출된 함수의 몸체가 메모리에 저장된다.

표 5. 함수 호출에 대한 ARM 코드 생성

main()	.section .rodata
	.text
	.align 2
	.global main
	.type main,function
	main:
	mov ip, sp
	stmfd sp!, {fp, ip, lr, pc}
	sub fp, ip, #4
	.....
	〈source〉
	.....
	ldr r0 ,[fp, #-20]
	bl factorial
	mov r3 ,r0
	.....
	〈source〉
	.....
	.L2 :
	ldmea fp, {fp, sp, pc}
	.Lfe1:
	.size main,.Lfe1-main

factorial()	.text
	.align 2
	.global factorial
	.type factorial, function
	factorial:
	mov ip, sp
	stmfd sp!, {fp, ip, lr, pc}
	sub fp, ip, #4
	.....
	〈source〉
	.....
	.L5 :
	ldmea fp, {fp, sp, pc}
	.Lfe2:
	.size factorial,.Lfe2-factorial

배열은 실제 메모리의 상위 주소로부터 배열의 첫 번째 원소가 저장되며 선언시 메모리의 처음 변수가 [fp, #-16] 으로 설정되며 배열의 10번째 원소가 [fp, #-16] 으로 잡힌 것을 볼 수 있으며 [표 6]과 같은 ARM 코드를 생성한다.

표 6. 배열에 대한 ARM 코드 생성

int a[10]	sub	sp, sp, #40
a[0]=8	mov str	r3, #1 r3, [fp, #-52]
a[9]=a[0]	ldr str	r3, [fp, #-52] r3, [fp, #-16]

3.3 ARM 코드 생성 핵심 모듈

표준 C 언어로부터 ARM 코드를 생성하는데 필요한 핵심 모듈은 AST 생성 부분, AST 순회하면서 실질적으로 ARM 코드를 생성하는 부분으로 구성되어 있다. AST 생성은 트리에 대한 노드를 구성하는 buildnode() 함수([표 7]) 와 노드를 연결하여 AST를 생성하는 buildtree() 함수([표 8])에 의해 수행된다.

표 7. AST에 대한 노드 구성 함수

```
Node *buildNode(struct tokenType token) {
Node *ptr;
ptr = (Node *) malloc(sizeof(Node));
if (!ptr) {
    printf("malloc error in buildNode()\n");
exit(1);
}
ptr->token = token;
ptr->noderep = terminal;
ptr->son = ptr->brother = NULL;
return ptr;
}
```

표 8. AST에 대한 구성 함수

```
Node *buildTree(int nodeNumber, int rhsLength) {
  int i, j, start;
  Node *first, *ptr;
  i = sp-rhsLength + 1;
  while (i <= sp && valueStack[i] == NULL) i++;
  if (!nodeNumber && i > sp) return NULL;
     start = i;
  while (i <= sp-1) {
     j = i + 1;
     while (j <= sp && valueStack[j] == NULL) j++;
     if (j <= sp) {
       ptr = valueStack[i];
       while (ptr->brother) ptr = ptr->brother;
       ptr->brother=valueStack[j];
     }
     i = j;
  }
first = (start > sp) ? NULL : valueStack[start];
if (nodeNumber) {
    ptr = (Node *) malloc(sizeof(Node));
    if (!ptr) { printf("malloc error in buildTree()\n");
    exit(1);
}
    ptr->token.number = nodeNumber;
    ptr->noderep = nonterm;
    ptr->son = first;
    ptr->brother = NULL;
    return ptr;
}
else return first;
```

생성된 AST를 순회하면서 실질적으로 ARM 코드를 생성하는 루틴은 [표 9]와 같다.

표 9. ARM 코드 생성 루틴

```
void codeGen(Node *ptr) {
    Node *p;
    p=ptr->son;
    armStart();
      while(p != NULL) {
            if(!p->brother)
                flagWrite+=2;
            processFuncStart(p->son);
                        p=p->brother;
            }
}
```

3.4 성능평가

본 논문에서 제안한 문법-지시적 변환 방법을 통해 생성된 ARM 코드는 Linux(Readhat 9.0) 환경의 GNU gcc(ver. 2.95)에서 지원되는 ARM 크로스 컴파일러와 비교하여 [표 10]과 같은 프로그램을 통해 실행결과 확인 및 성능평가를 수행하였다.

표 10. 실행결과 확인 예제 프로그램

예제 프로그램	실행결과
완전수 프로그램(perfect.c)	정상
버블정렬(bubble.c)	정상
배열 프로그램(array.c)	정상
함수호출 프로그램(func.c)	정상
재귀적호출(factorial.c)	정상
문장 프로그램(stmt.c)	정상
회문수(pal.c)	정상

또한 실제적인 실행결과 확인을 위해 본 연구의 실행결과와 gcc 크로스 컴파일러 통해 생성된 ARM 코드의 실행 결과를 EMPOSII 임베디드 툴킷에서도 확인하였다.

성능평가 방법은 생성된 ARM 코드에 대해 time() 함수를 이용한 실행속도(sec)와 코드의 크기(byte)를 비교하여 측정하였다. 실행속도 측정에서 AST를 순회하면서 선언문, 배열, 배정문, 배열, 함수 호출 등에 대한 부분적인 측정은 아주 미세하거나 거의 동일한 결과를 얻었다.

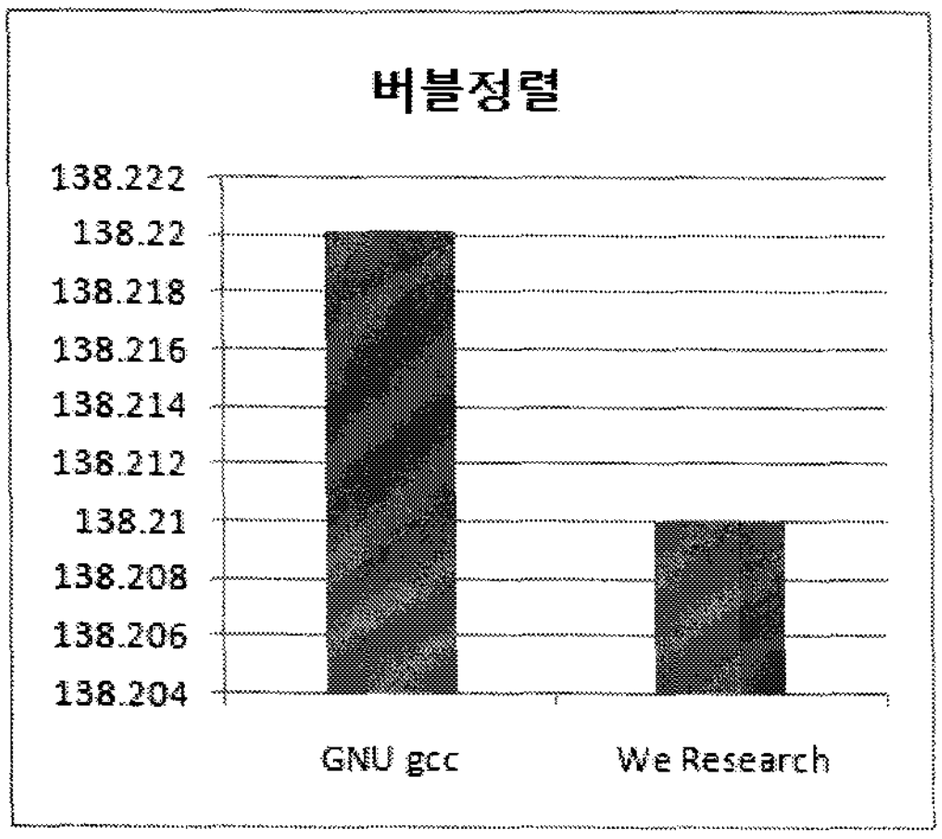

그림 3. ARM 코드 실행속도 비교(버블정렬, 단위 sec)

버블정렬은 배열, 반복문, 선언문 등의 성능을 복합적으로 측정할 수 있는 예제로서 본 논문에서는 기존 GNU gcc에 비해 속도가 감소하는 결과를 [그림 3]과 같이 얻을 수 있었다. 또한 반복적인 함수 호출이 발생되는 경우에 대한 실행속도 측정을 위해 팩토리얼 값을 구하는 프로그램을 통해 [그림 4]와 같은 결과를 측정하였다.

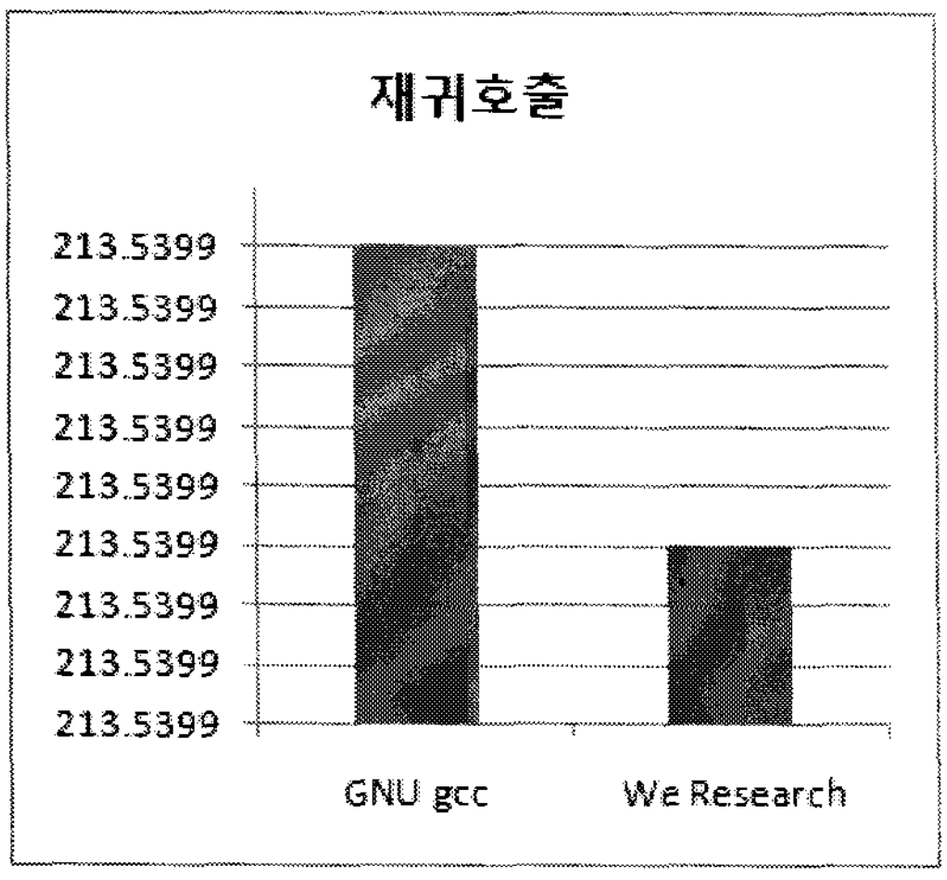

그림 4. ARM 코드 실행속도 비교(팩토리얼, 단위 sec)

생성된 코드의 크기 ARM 코드 생성시에 최적화를 고려하여 생성하여 미세하지만 [표 11]과 같이 감소를 되는 결과를 확인할 수 있다.

표 11. 코드크기 비교(Kbyte)

프로그램	GNU gcc	We Research
perfect.c	14.3	14.1
bubble.c	13.7	13.5
factorial.c	13.5	13.3
pal.c	14.6	14.5

V. 결론 및 향후 연구 방향

ARM 프로세서는 현재 다양한 임베디드 시스템에 탑재되어 활용되고 있으며, 특히 C 언어로 작성된 어플리케이션을 수행하기 위한 컴파일러, 어셈블러와 같은 소프트웨어 개발 도구도 다양하게 개발되고 있다. 특히, ARM 프로세서에 적합한 목적코드 생성을 위해 대부분 gcc 크로스 컴파일러를 이용하여 전단부와 후단부를 사용하고 있다.

본 논문에서는 표준 C 언어로 작성된 어플리케이션을 입력으로 받아 ARM 코드를 생성하는 코드 생성 시스템은 AST를 생성한 후 AST를 순회하면서 적절한 ARM 생성 루틴을 개발하고 생성된 ARM 코드에 대해 GUN gcc와 실행속도 및 코드크기를 비교 측정하였다.

본 연구를 통해 생성된 ARM 코드는 gcc 크로스 컴파일 방식이 비해 미세하지만 속도 향상을 얻을 수 있었다. 속도 향상의 이유는 AST 순회하면서 코드 크기와 실행 속도를 고려하여 ARM 코드를 지정하였기 때문이다. gcc의 경우는 다양한 프로세서를 위한 후단부 지원이 가능하도록 하여 특정 프로세서를 위해 제작되는 컴파일러에 비해서는 양질의 코드 생성을 기대할 수 없지만 다양한 목적기계 의존적 최적화를 통해 성능 향상이 기여하고 있다.

향후에는 포인터 연산, 구조체, 동적 기억장소 할당과 같은 기능을 보완하고 현재는 정수형만 지원하는 단점을 개선하여 보다 완벽한 코드 생성시스템을 구축하고자 한다. 본 연구의 결과는 GNU gcc 기반의 크로스 컴파일러의 의존성을 탈피하고 독자적인 임베디드 프로세서 전용 C 컴파일러를 구축하는데 활용될 수 있으며 MIPS, SPARC 목적코드를 생성하는 분야에도 활용될 수 있다.

참 고 문 헌

[1] R. Leupers, "Compiler Design Issues for Embedded Processors," IEEE Design & Test of Computers, 2002(7).

[2] S. Furber, ARM System-on-Chip Architecture, Addison-Wesley, 2000.

[3] T. Jea-Paul and G. S. Paul, "The Theory and Practice of Compiler Writing," McGraw-Hill International Editions, 1989.

[4] K. G. John, Syntax Analysis and Software Tools, Addison-Wesley Publishing Company, 1988.

[5] 오세만, *컴파일러 입문*, 정익사, 2006.

[6] R. Leupers and M. Peter, "Retargetable Compiler Technology for Embedded Systems; Tools and Applications," Kluwer Academic Publishers, 2001.

[7] F. Chistopher and H. David, "A Retargetable C Compiler: Design and Implementation," The Benjamin/Cummings Publishing Company Inc., 1995.

[8] L. Benini, M. Kandemir, and J. Ramanuijam, "Compilers and Operating Systems for Low Power," Kluwer Academic Publishers, 2003.

[9] N. S. Andrew, S. Dominic, and W. Chris, "ARM System Developer's Guide: Design and Optimization System Software," Elsevier Inc., 2004.

저 자 소 개

고 광 만(Kwang-Man Ko) 정회원

- 1991년 2월 : 원광대학교 컴퓨터공학과(공학사)
- 1993년 2월 : 동국대학교 컴퓨터공학과(공학석사)
- 1998년 2월 : 동국대학교 컴퓨터공학과(공학박사)
- 2001년 8월 : 광주여자대학교 컴퓨터과학과(전임강사)
- 2003년 1월 ~ 2월 : Queensland University of Technology 연구교수
- 2001년 9월 ~ 현재 : 상지대학교 컴퓨터정보공학부 부교수

<관심분야> : 프로그래밍언어론 및 컴파일러