제20회 한국정보처리학회 추계학술발표대회 논문집 제10권 제2호 (2003. 11)

임베디드 시스템 지원을 위한 LCC Retargetable Compiler에 관한 연구

홍일경*, 김기창**
*인하대학교 전자계산공학과
**인하대학교 정보통신공학부
e-mail : whitecode@super.inha.ac.kr

A Study of LCC Retargetable Compiler for Embedded Systems

Il-Kyeong Hong*, Ki-Chang Kim**
*Dept. of Computer Science and Engineering, Inha University
**School Of Information & Communication Engineering

요 약

과거 임베디드 시스템은 산업용 같은 특수한 경우와 여러 전자 제품에서 눈에 뛰지 않는 형태로 사용이 되어 왔지만 최근 들어 정보통신 분야가 발전하고 가격 대 성능비가 우수한 Microcontroller 와 CPU 들이 등장하면서 더욱 더 각광을 받기 시작했다. 이들의 개발환경은 C 언어와 어셈블리어가 주를 차지하고 있는데 개발상의 많은 장점 때문에 C 언어를 선택하여 사용하고 있지만 target machine 의 C 컴파일러가 전부 제공되지 않고 일부 machine 만 제공되고 있기 때문에 C 언어의 사용은 아직 제한적이다. 이에 본 논문에서는 Retargetable Compiler 인 LCC 를 이용하여 임베디드용 machine 에 대해 retargeting 이 가능하도록 LCC 를 개선시키고자 한다.

1. 서론

임베디드 시스템은 마이크로프로세서가 개발된 이후로 대부분의 전자제품과 첨단 제어 시스템에 이용되어 왔으며 현재에 이르러는 정보통신기기를 중심으로 그 수가 급속하게 증가하고 있는 추세이다.

이렇듯 최근 임베디드 시스템의 중요성이 부각되고 있고, 새로운 아키텍쳐를 가지는 제품들의 개발 주기가 점차 짧아져 가는 추세에 있다. 또한 그를 지원할 수 있는 컴파일러 기술의 중요성도 더불어 증대되고 있다.

그러나 전통적인 방식의 컴파일러는 개발 프로세스가 길기 때문에 최근의 빠른 시장변화에 대응하는데 큰 어려움이 있다.

따라서 새로운 타겟 아키텍쳐에 대한 필요한 정보들을 기술하여 전체 컴파일러의 재작성 없이 짧은 시간 안에 최적의 성능을 얻을 수 있는 retargetable compiler 의 개발이 이슈가 되고 있다. 이러한 부분은 고속 프로세서, DSP, 네트워크 프로세서등에서 많이 요구되어 지고 있으며 산업용 제어 시스템에서도 이식성과 재사용 그리고 개발 프로세스 단축의 목적으로 크게 요구되어 지고 있다. 따라서 본 논문에서는 8 비트용 Microcontroller 를 대상으로 리타겟팅이 가능하도록 lcc 컴파일러를 개선하는 방안에 대해 제시하고자 한다.

본 논문의 구성은 다음과 같다. 2 장에서는 관련연구에 대해 기술하고 3 장에서는 임베디드 타입의 프로세서를 리타겟팅 시의 문제점에 대해 설명하고 4 장에서는 임베디드 타입의 프로세서 리타겟팅을 위한 컴파일러의 개선 방안에 대해 기술하고 5 장에서는 결론과 향후 과제를 기술한다.

2. 관련 연구

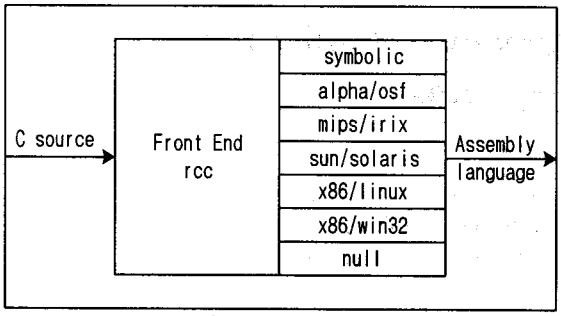
2.1 Retargetable Compiler

일반 컴파일러의 경우 general-purpose processor 와

같이 하나의 프로세서를 타겟으로 하고 있기 때문에 다른 프로세서들에 대해서는 재사용하기가 힘들다. 그러나 retargetable compiler 의 경우는 프로세서 아키텍처가 변화하였을 시에도 전체적인 컴파일러의 재작성 없이 타겟 아키텍처에 대한 필요한 정보들을 기술하여 description 을 작성함으로써 C 언어와 같은 하이 레벨 언어를 그대로 사용하여 리타겟팅을 함으로 개발 프로세스를 단축시킬 수 있다.

2.2 LCC 컴파일러의 특성

lcc 컴파일러는 monolithic compiler 로써 그림[1]과 같은 과정으로 C 소스를 입력 받아 어셈블리 코드를 출력하도록 설계되어 있다.



그림[1] lcc 의 framework

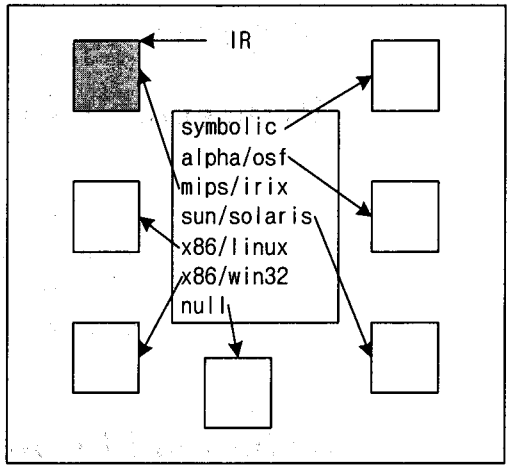
lcc 의 전반부를 처리하는 rcc 는 소스 코드를 입력 받아 DAG(Directed Acyclic Graph)형태의 중간표현을 생성한다. 확장자가 md 인 code generator 는 타겟 머신에 관한 specification 정보가 기술되어 있으며 각 타겟 머신 당 한 개씩 존재한다.

이 code generator 파일을 코드 생성기 자동생성기(code generator-generator)인 lburg 가 읽어 들여 전반부에서 만들어진 DAG 에 대해 처리할 후반부 코드를 출력하게 된다. 출력된 후반부 코드는 Code-Generation Interface 와 Code-generation Function 으로 구성이 되어 있는데 공유를 위한 자료구조와 18 개의 인터페이스 함수, 33 개의 중간표현 연산자, 6 개의 타입 접미사 그리고 중간표현연산자+타입+사이즈의 형태로 표현되는 130 개의 연산자로 구성되어 있다. 이 후 중간 언어가 생성이 되고 생성된 중간 언어를 순차적으로 register allocation 과정을 거치게 되면 최종적인 어셈블리 코드가 생성되게 된다.

다음과 같이 명령어에 처리할 타겟 머신을 옵션으로 지정하게 되면 해당 타겟에 맞는 어셈블리 코드를 출력하게 된다.

```
%lcc -Wf-target=mips/irix -S test.c
```

lcc 는 각 타겟에 대한 정보를 담기 위해 IR 이라고 하는 Interface Record 구조체를 가지고 있는데 명령어 옵션에서 '-target=해당타겟' 을 입력하게 되면 그림[2]와 같이 해당타겟에 바인딩이 일어나고 해당 타겟의 IR 정보를 이용하여 처리가 이루어 진다.



그림[2] Interface Record binding

3. 임베디드 타입의 프로세서 리타겟팅의 문제점

본 논문에서는 리타겟팅의 대상으로 8 비트 코어를 사용하는 RISC 타입의 Microcontroller 를 사용하였다. 그 중 모델이 된 MCU 는 ST 사에서 나오는 ST5 계열을 리타겟팅의 대상으로 정하였다.

현재의 lcc 구조는 32 비트용 범용 프로세서에 기준하여 설계가 되어 있다. 그래서 8 비트용 프로세서의 경우 변수 타입의 제한적 사이즈나 캐스팅의 제한적 사용에 대해 처리를 하지 못하고 오버플로우가 발생하면서 컴파일이 중단되는 현상이 발생하게 된다.

위 계열의 MCU 들은 정수형 사이즈는 1byte, 포인터형 사이즈는 2byte, long size 는 2byte 이고 char 는 unsigned 만 지원이 되고 floating point 와 4byte integer 는 지원되지 않는다. 또한 각 함수의 스택 프레임 사이즈는 64byte 를 넘을 수 없다.

이러한 특성이 반영될 수 있도록 리타겟팅을 하는데 있어 여러 문제점들 중에서 중요하게 살펴 보아야 할 두 가지의 문제점에 대해 살펴보도록 하겠다.

우선 배열의 경우 예를 들어 정수형(int) array[constant]가 선언되어 있다고 한다면 constant 의 값은 128 을 넘을 수 없다. 정수형은 1byte 라는 제약 때문이다. 따라서 128 보다 큰 수를 입력하게 되면 오버플로우가 발생하게 된다.

물론 이 부분은 프로그래머가 사이즈를 초과하지 않도록 코딩을 잘 해주면 오버플로우가 발생하지는 않겠지만 lcc 는 그러한 오류에 대해서 융통성을 발휘하지 못하고 결과적으로 오버플로우를 발생시키며 어셈블리 코드를 출력하지 못한다.

두 번째로 포인터 연산과 관련된 문제이다. 덧셈 연산에서 포인터 연산인 경우는 해당 오브젝트의 사이즈와 관련이 있다. 즉, 다시 말해서 포인터 덧셈 연산의 경우 위 MCU 들은 포인터형 사이즈가 2byte 로 고정이 되어 있기 때문에 long 형의 사이즈로만 계산이 이루어져야 하지만 lcc 에서는 포인터에 선언된 타입의 사이즈로 계산하도록 되어 있기 때문에 오류가 발생할 수 있다.

이러한 타겟 프로세서의 제한된 특성 때문에 리타겟팅을 위해서 lcc 가 수정이 이루어져야 한다.

4. 리타겟팅을 위한 lcc 컴파일러의 개선 방안

본 논문에서 제안하고 있는 개선 방안은 특정 타겟 몇개를 대상으로 하고 있기 때문에 lcc 코드의 수정이 타겟 종속적으로 이루어 졌지만 여러 종류의 타겟에 대한 타겟팅을 위해선 독립적인 코드 구성이 필요함을 알려둔다.

먼저 해당 타겟을 식별하기 위한 플래그를 code generator 파일 선언부에 선언해 두고 초기화를 담당하는 progbeg()함수가 호출될 때 셋을 시켜 둔다.

```
%{
/* 선언부 */
int stmcu;
%}
...
...
static void progbeg(int argc, char *argv[]) {
...
stmcu = 1;
...
}
```

그림[3] code generator

그리고 나서 수정이 필요한 함수에 위 변수를 extern 으로 선언한 후 제어문을 이용하여 처리 루틴을 삽입하면 된다.

위 3 장에서 제기한 문제점을 중심으로 개선되어야 할 부분의 코드에 대해 설명하도록 하겠다.

첫 번째 제기한 변수의 오버플로우 문제점을 해결하기 위해 그림[4]와 같이 코드 수정을 하였다.

```
static Type dclr1(char **id, Symbol **params, int abstract) {
...
case '[': t = gettok(); { int n = 0;
    if (kind[t] == ID) {
        extern int stmcu
        if (stmcu);
            n = longintexpr(']', 1);
        else
            n = intexpr(']', 1);
...
}
```

그림[4] 배열내 선언된 변수의 오버플로우 방지를 위해 수정된 코드(decl.c)

위와 같이 수정된 코드로 배열 내에 선언된 정수형 타입의 식별자가 범위를 초과해도 오버플로우가 발생하는 것을 방지할 수 있다. 또한 longintexpr()함수가 존재하지 않기 때문에 그림[5]의 코드를 새로 추가해 주어야 한다.

```
long int longintexpr(int tok, int n) {
    Tree p = constexpr(tok);

    needconst++;
    if (p->op == CNST+I || p->op == CNST+U)
        n = cast(p, longtype)->u.v.i;
    else
        error("integer expression must be constant\n");
    needconst--;
    return n;
}
```

그림[5] longintexpr() 함수(simp.c)

두 번째로 제기한 포인터의 덧셈 연산시의 계산과 관련된 문제점을 해결하기 위해 그림[6]과 같이 코드를 수정한다.

```
static Tree addtree(int op, Tree l, Tree r) {
...
if (n > 1)
{
    extern int stmcu;
    if (stmcu)
        l = multree(MUL, cnsttree(l->type, n), l);
    else
        l = multree(MUL, cnsttree(signedptr, n), l);
}
...
}
```

그림[6] 포인터 덧셈 연산을 위한 수정 코드 (enode.c)

그림[6]은 덧셈 연산의 경우 처리되는 addtree() 함수 루틴이다. 그 중에서 포인터 연산인 경우에 처리되는 루틴으로 lvalue 의 타입을 long 형으로 크기를 잡고 덧셈 연산이 이루어지면 곱셈 연산을 하는 루틴에 해당된다.

```
Type binary(Type xty, Type yty) {
  extern int stmcu;
...
        if (stmcu)
        {
            xx(inttype);
            xx(unsignedchar);
            xx(chartype);
        }
...
}

Tree cast(Tree p, Type type) {
    Type src, dst;
    extern int stmcu;
...
    if (stmcu)
    {
        if (src->op != FLOAT
        && dst->op != FLOAT
        && src->size == dst->size
        && p->op != CNST+I
        && p->op != CNST+U
        && p->op != CNST+P
```

```
        && !explicitCast)
          return p;
        }
...
}
```

그림[7] 관련 수정 코드 일부

그림[7]의 경우는 오버플로우 문제점과 포인터 덧셈 연산 문제점과 연관되어 수정해 주어야 할 부분에 대한 코드이다.

종합적으로 컴파일러의 전반부에 해당하며 타겟 독립적인 코드 처리 과정에 해당하는 부분에서 수정되어야 할 파일을 정리해 보면 다음과 같다.

```
bind.c  dag.c  decl.c  enode.c  expr.c  gen.c
init.c  simp.c
```

lcc 컴파일러는 execution time 이 빠르지만 global optimization 을 수행하지 않기 때문에 generation 된 code 의 quality 면에서는 다소 성능이 떨어진다고 할 수 있다. 이러한 부분이 lcc 컴파일러가 계속적으로 연구가 되어 져야 할 부분이지만, 위에서 제시한 내용과 같이 적용함으로써 얻을 수 있는 장점은 같은 계열내의 타겟들에 대해서 타겟 description 을 한 번만 작성하고 필요 시 약간의 수정을 통해서 리타겟팅이 가능하다는 것이다.

이는 전반적인 컴파일러의 제작 시간을 상당히 단축 시킬 수 있으며, 기존의 사용하던 프로그램의 재사용의 많은 효과를 가져올 수 있다.

5. 결론 및 향후 과제

본 연구는 특정 임베디드 타입의 프로세서 지원이 가능하도록 lcc 컴파일러를 수정하여 구현해 보았다. 이는 현재 산업용 제어 시스템에서 사용하고 있는 Microcontroller 나 PLC(Programmable Logic Controller)분야에 적용이 가능하도록 참고할 수 있으며 일부 적용 또한 가능하다.

향후 연구 방향으로는 첫째, 다양한 임베디드 타입에 대한 공통적인 특성을 분석하여 lcc 컴파일러가 수정되지 않고도 적용이 가능토록 개선하는 것이고. 둘째는 무엇보다도 임베디드 시스템 분야에서는 코드가 작아야 하고 빠르게 돌아가야 하는 제한적 특성을 만족시켜야 하지만 현재의 lcc 컴파일러는 최적화 부분에서는 아직 미흡한 부분이 많은 것이 사실이므로 코드의 최적화가 동시에 이루어 질 수 있도록 개선하는 연구가 필요하다.

참고문헌

[1] C. W. Fraser and D. R. Hanson, A Retargetable Compiler for C:Design and Implementation, Addison-Wesley, Menlo Park, CA, 1995

[2] Ball, Stuart R. Embedded Microporcessor System: Real World Design. Newton, Mass.:Butterworth-Heinemann, 1966

[3] Aho, A. V., R. Sethi, and J.D.Ullman. 1986. Compilers: Principles, Techniques, and Tools. Reading, MA: Addison Wesley.

[4] Ritchie,D.M. 1993. The development of the C language. Preprints of the Second ACM SIGPLAN History of Programming Languanges Conference (HOPL-II),SIGPLAN Notices 28(3), 201-208

[5] Sethi, R. 1981. Uniform syntax for type expressions and declarators. Software-Practice and Experience 11(6), 623-628