The Journal of the Convergence on Culture Technology (JCCT)
Vol. 8, No. 1, pp.577-582, January 31, 2022. pISSN 2384-0358, eISSN 2384-0366

http://dx.doi.org/10.17703/JCCT.2022.8.1.577

JCCT 2022-1-65

LLVM 기반의 임베디드 시스템 성능 분석기의 연구

A Study of LLVM-based Embedded System Performance Analyzer

조두산*

Doosan Cho*

요 약 새로운 임베디드 시스템을 개발할 때 응용 프로그램과 에뮬레이터, 그리고 컴파일러를 동시에 개발한다. 모든 시스템 구성요소의 성능을 최적으로 개발하기 위해서는 응용 프로그램 개발시 부분 최적화를 동시에 진행하여야 한다. 이를 위하여 소스 레벨 성능 분석기를 개발하면 응용 프로그램 소스 코드를 모듈별로 성능 평가하여 최적화하는 것이 가능하다. 일반적으로 응용 프로그램의 성능은 반복문에서 결정된다. 소스코드를 중간표현 (Intermediate Representation) 코드 생성기를 이용하여 변환하고, 변환된 중간 표현 단계의 명령어들로 실행시간을 분석 할 수 있다. 실행시간 성능 평가 결과를 바탕으로 응용 프로그램 코드를 개선하면 최종적으로 개발된 통합 플랫폼에서 더 나은 결과물을 얻을 수 있다. 본 연구에서는 새로운 임베디드 시스템의 개발중에 응용 프로그램을 동시 개발하는 과정에서 사용 가능한 소스 레벨 성능 분석기에 대하여 기술하고 있다. 성능 분석기를 사용하면 최종 임베디드 시스템의 성능을 보다 빠르게 최적화하는 것이 가능하게 된다.

주요어 : 성능, 시스템 소프트웨어, 코드 최적화, 중간 표현, 임베디드 시스템

Abstract For developing a new embedded system, an application program/an emulator and a compiler are developed simultaneously. In order to provide the optimal performance of all system components, local optimization should be carried out for the developing process. For this purpose, if a source-level performance analyzer is developed, it is possible to optimize the application program's source code by the performance evaluation. In general, the performance of an application program is determined in the loop iterations. The Intermediate Representation (IR) code generator generates IR code from the source code, and evaluates the execution time with the instructions in the intermediate representation code. If the source code is improved based on the evaluated result, better results can be obtained in the final application code. This study describes the source-level performance analyzer that can be used during the simultaneous development of the new embedded system and its application programs. The performance analyzer makes it possible to more quickly optimize the performance of the new embedded system.

Key words : PERFORMANCE, SYSTEM SOFTWARE, CODE OPTIMIZATION, INTERMEDIATE REPRESENTATION, EMBEDDED SYSTEM

Ⅰ. 서 론

일반적으로 구현된 응용 프로그램의 성능은 타겟 프로세서에서 실행되거나 혹은 에뮬레이터에서 실행 결과를 분석하여 최적화 작업을 진행할 수 있다. 새로운 임베디드 시스템 개발은 하드웨어와 별도로 소프트웨어 개발을

*정회원, 순천대학교 전자공학과 교수 (제1저자)
접수일: 2021년 12월 31일, 수정완료일: 2022년 1월 5일
게재확정일: 2022년 1월 8일

Received: December 31, 2021 / Revised: January 5, 2022
Accepted: January 8, 2022
*Corresponding Author: dscho@scnu.ac.kr
Dept. of EE, Sunchon National Univ, Korea

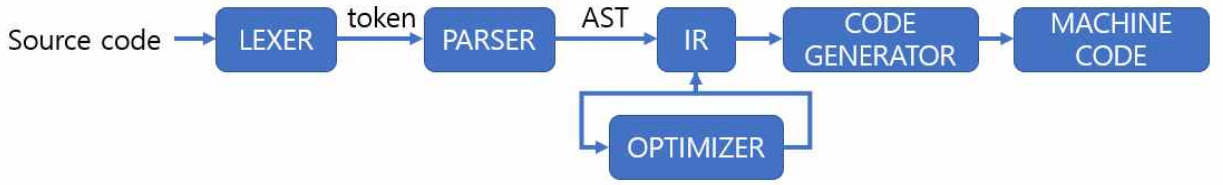

그림 1. LLVM 컴파일러 구성
Figure 1. LLVM compiler components

진행한다. 현재는 소프트웨어 개발 비용이 전체 시스템에서 차지하는 비중이 더 크기 때문에 개발 중에 다양한 최적화 및 검증을 진행하게 된다. 본 연구에서는 이러한 최적화 절차를 프로그램의 모듈 별로 진행 할 수 있도록 지원하는 성능 분석기에 대한 연구 결과를 소개한다.

본 연구에서 사용한 성능 분석기는 LLVM 컴파일러 [1] 라이브러리를 사용한다. LLVM이란 약자가 아닌 컴파일러 프로젝트 이름 그 자체이다. LLVM은 컴파일러 개발 플랫폼으로 고성능의 컴파일러를 빠른 시간안에 개발하도록 지원하고 있다. 일반적으로 요구되는 컴파일러 구성요소가 완벽하게 모듈화되어 있기 때문에 필요한 모듈을 라이브러리 형식으로 호출하여 사용할 수 있다. 본 연구에서 개발한 프로그램 코드 성능 분석기는 먼저 구현된 소스코드 전체 혹은 일부를 입력으로 사용하여 프로그램의 실행 성능을 평가한다.

사용자는 이렇게 평가된 성능 측정 결과를 바탕으로 개발중인 프로그램의 부분 혹은 전체의 성능을 최적화 할 수 있게 되는 것이다. 다음 장에서는 LLVM에 대한 배경지식과 성능 분석기의 구성에 대한 설명을 진행한다. 3장에서는 소스 코드 성능 분석기의 동작에 관한 설명을 진행한다. 4장에서는 시스템 성능 분석기를 사용한 간단한 실험 결과를 제공하고, 마지막으로 5장에서는 결론을 기술한다.

II. 배경 지식

LLVM은 컴파일러를 만들 때 필요한 여러 가지 모듈의 집합체인 라이브러리 플랫폼이다. 이것은 기존의 다양한 연구에 사용되었다. [2]의 연구는 LLVM과 같은 컴파일러 분석기에서 생성된 명령어 디펜던스 그래프를 바탕으로 최적화된 코드 매핑 기법을 소개한다. [3]은 LLVM의 레지스터 할당기에서 생성된 스필 데이터를 제거하기 위한 재계산 기법을 소개한다. [4]는 LLVM 컴파일러로 분석한 메모리 사용 패턴을 이용하여 멀티뱅크 메모리에 최적화된 데이터 할당 기법을 제안한다. [5]는 멀티미디어 응용에 최적화된 데이터 할당 기법을 LLVM 분석기를 이용하여 제안하였다.

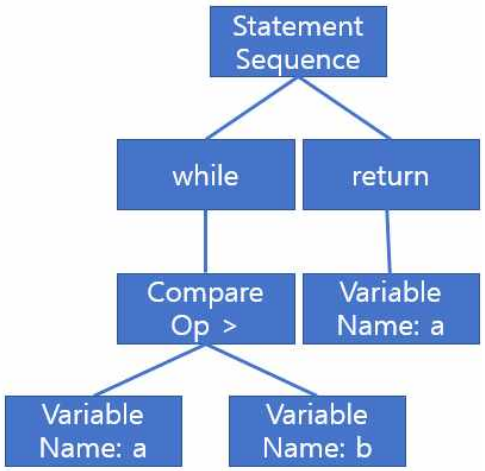

그림 2. 추상 구문 트리
Figure 2. Abstract syntax tree

그림 1을 보면 LLVM으로 구성할 수 있는 분석기가 소스코드에서부터 어휘분석기 (lexer), 구문분석기 (parser), 중간표현 (IR), 최적화기 (optimizer), 코드 제너레이터 (code generator), 머신 코드 (machine code)로 이어지는 일련의 절차에 따르는 모듈로 구성되어 있음을 보이고 있다. 어휘분석기는 소스코드에 있는 문자들을 스트림으로 읽어들여 토큰을 만들고 토큰들을 그룹으로 구성하여 추상 구문 트리 (abstract syntax tree)로 만든다. 추상 구문 트리란 프로그램의 소스코드를 구성하는 키워드와 피연산자, 식별자 (identifier)들을 트리로 구성한 것으로 원래의 소스코드에서 각 노드들에 대한 상세 정보를 포함하지 않기 때문에 추상이라는 단어를 사용한다. 그림 2에 추상 구문 트리를 나타내었다.

그림 2에 나타난 추상 구문 트리를 보면 비교문과 피연산자 변수 a, b를 확인할 수 있다. 이것을 통하여

변수의 이름, 데이터 타입 등을 확인할 수 있다. 이와 같이 소스 코드가 추상 구문 트리로 변환되고, 각 연산자가 중간 표현(IR)을 구성하는 명령어로 변환이 완료되면 LLVM 중간 표현 코드 생성이 종료된다. LLVM IR을 구성하는 모듈들은 다음과 같다.

- Target Information : data layout
- Symbol table
  |-Global variables
  |-Type declarations : custom struct type
  |-Function declarations : linked functions
  |-Function definitions : defined functions

그림 3. LLVM 중간표현 구성 모듈
Figure 3. LLVM IR module

그림 3의 타겟 정보를 보면 데이터 레이아웃 정보를 포함하고 있다. 심볼 테이블은 중간 표현 (IR)에 포함되어 있는 모든 식별자를 포함하여 전역 변수, 함수 이름, 구조체 선언 등을 검색할 수 있다. IR을 구성하는 모듈은 추상 구문 트리를 만드는 클래스로 구성되어 있는데, global variable, type declaration, function declaration, function definitions, binary operation, unary operation, funcation call, for loop, if statement 등으로 구현되어 있다.

본 연구에서 제안하는 소스 레벨 성능 분석기는 LLVM IR 기반으로 동작한다. LLVM 플랫폼에서 제공하는 LLC 툴 혹은 CLANG 컴파일러의 프론트엔드를 사용하여 소스코드를 입력으로 LLVM IR 파일을 생성할 수 있다. 또한 LLVM 라이브러리 혹은 기존 LLVM 프론트엔드의 어휘분석기와 파서를 사용하면 LLVM IR을 생성하고 입력 프로그램 코드에서 for loop 같은 프로그램 핵심 요소를 추출하여 분석을 진행할 수 있게 된다. 본 연구에서 제안된 성능 분석기는 어휘분석기, 구문 분석기, 최적화기로 구성되어 있다. LLVM 라이브러리 사용 설명은 LLVM [1]에 잘 설명되어 있다. 생성된 LLVM IR을 이용한 성능 분석 방식은 3장에 기술되어 있다.

III. 소스 코드 성능 분석기

본 연구에서 사용한 성능 분석기는 소스 코드를 어휘 분석기로 토큰화하여 구문 분석기에 전달하고, 구문 분석기는 토큰 스트림을 추상 구문 트리로 변환하여, LLVM IR을 생성한다. 정의된 토큰의 일부는 아래와 같다.

EOF_TOKEN : 파일의 끝을 지정
NUMERIC_TOKEN : 숫자 타입의 토큰
IDENTIFIER_TOKEN : 식별자 토큰
PARAN_TOKEN : 괄호 토큰
DEFINE_TOKEN : 함수 정의 지정
IF_TOKEN, ELSE_TOKEN : 조건문 토큰
FOR_TOKEN : 반복문 토큰
BINARY_TOKEN : 이항연산자 토큰
UNARY_TOKEN : 단항 연산자 토큰

이외에도 while문 switch/case문, do/while문 등 C언어를 파싱하는데 필요한 각종 TOKEN들이 정의되어사용되었다. 이러한 토큰들은 어휘 분석기를 통하여 구문 분석기에 전달되어 LLVM IR 코드를 생성하게 된다. 그림 4에 LLVM IR 코드 예제를 나타내었다.

```
void test(int n) {
  for (int i = 0; i < n; i += 1)
    // Loop body
}
```
(a) source code

```
define void @test(i32 %n) {
entry:
  br label %for.header

for.header:
  %i = phi i32 [ 0, %entry ], [ %i.next, %latch ]
  %cond = icmp slt i32 %i, %n
  br i1 %cond, label %body, label %exit

body:
  ; Loop body
  br label %latch

latch:
  %i.next = add nsw i32 %i, 1
```

```
  br label %for.header

exit:
  ret void
}
```
(b) IR code

그림 4. LLVM IR과 소스코드
Figure 4. LLVM IR and source code

그림 4의 (a)가 소스코드를 나타내고, (b)가 소스코드에서 생성된 LLVM IR 코드를 나타낸다. 소스코드는 for 루프가 i 변수 값을 1씩 증가하면서 n보다 작을 때까지 반복하는 함수 test를 나타낸다. 이 소스 코드에서 생성된 IR 코드는 define test함수와 함수 인자 정수 i32 n을 시작으로 entry, for.header, body, latch, exit 총 5개로 레이블된 IR 코드 블록으로 구성된다. IR 코드에서 %기호는 레지스터를 나타내는 것으로 LLVM IR에서 기본적으로 사용된다. IR 코드는 정적 단일 할당(Static Single Assignment, SSA) 방식으로 생성되기 때문에 %기호 뒤의 레지스터 번호는 무한대 레지스터 집합에서 할당된다. 번호 이외의 레지스터 명칭은 특별한 규칙에 의하여 생성된다. SSA를 사용하면 IR 코드를 최적화기에서 최적화 할 때 그 결과가 더 개선된다.

그림 5에서 생성된 그림 4에 나타난 IR 코드를 베이직 블록 단위를 노드로 하여 제어 흐름 그래프를 나타내고 있다. 제어 흐름 그래프는 entry 노드, for.header 노드, body 노드, exit 노드, latch 노드로 구성되어 있다. C코드에 나타난 for 루프 반복문의 실행 흐름은 entry -> for.header -> body -> latch ->for.header -> exit 노드로 이어지게 된다. 각 노드에서 실행되는 명령어들을 살펴보면 br (branch), icmp (compare), add (addition), ret (return)들로 구성되어 있다. 각 명령어가 실행되는 실행횟수와 명령어당 실행 시간이 분석되면 해당 반복문의 성능 측정 결과를 평가할 수 있게 된다. 예를 들면, ARM 명령어 집합의 경우 branch 명령어는 3cycle, add 명령어는 1cycle이 소요된다. 그림 5의 반복문이 100회 실행되는 경우 add 명령어는 100cycle, branch 명령어는 4개*3cycle*100회 1200cycle이 소요된다. body에서 실행되는 명령어들의 실행시간을 고려하여 branch 명령어들의 오버헤드를 경감하도록 반복문의 반복 횟수를 최적화하는 것이 가능하게 된다.

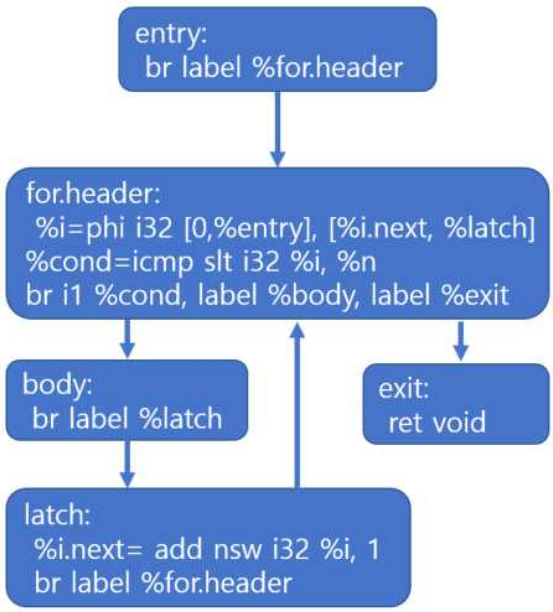

그림 5. LLVM IR의 제어 흐름 그래프
Figure 5. Control flow graph of LLVM IR

IV. 실험 및 결과

실험은 LLVM 버전 10.0과 clang, llc 등을 사용하여 진행하였다. 그림 6에 실험용 소스코드와 생성된 LLVM IR코드를 나타내었다. n은 10으로 10회 반복 실행한다.

```
void experiment(int n) {
  for (int i = 1; i < n; i += 1)
    x = x+1
}
```
(a) source code

```
define void @experiment(i32 %n) {
entry:
  br label %for.header

for.header:
  %i = phi i32 [ 1, %entry ], [ %i.next, %latch ]
  %cond = icmp slt i32 %i, %n
  br i1 %cond, label %body, label %exit

body:
  %x = add i32 %x, 1
  br label %latch
```

```
latch:
  %i.next = add nsw i32 %i, 1
  br label %for.header

exit:
  ret void
}
```
(b) IR code

그림 6. LLVM IR과 실험 소스코드
Figure 6. LLVM IR and experimental source code

반복문과 합산을 계산하는 소스코드를 입력으로 하고, 생성된 IR 코드를 출력으로 나타내었다. loop body에서 합산을 계산하는 add 명령어가 실행시간 10cycle 소요한다. branch 명령어가 4개 * 3cycle * 10회 실행시간 120cycle 소요한다. 이러한 경우 branch로 인한 실행시간 오버헤드가 크기 때문에 loop unrolling 최적화를 통하여 loop을 제거하고, add 명령어 9개를 추가하여 실행하는 것이 실행시간 개선에 유리하다. loop unrolling을 적용하면 branch 명령어 등 loop 오버헤드를 모두 제거하여 add명령어 실행시간 10cycle에 기존 반복문을 실행하는 것이 가능하다. 이와 같은 방식으로 이 예제에서는 총 130cycle을 제거하여 루프 성능이 86% 개선되었다.

본 연구에서는 LLVM IR을 이용하여 명령어 레벨에서의 성능 평가 기법을 제안하였다. 명령어의 종류에는 일반적인 덧셈/뺄셈/곱셈/나눗셈과 같은 연산 명령어 이외에 데이터를 읽고/쓰기 위한 로드/스토어 명령어가 포함된다. 이러한 메모리 액세스 관련 명령어의 성능 평가는 런타임에 정확히 결정되는 경우가 많기 때문에 다음과 같은 연구들과 함께 적용하면 보다 개선된 결과를 얻을 수 있을 것이다. [1][2][3][4]와 같은 메모리 전력 소모를 줄이는 컴파일러 기술을 동시에 적용할 수 있다. 또한, [5][6][7][8]과 같은 추가적인 성능 향상을 얻을 수 있는 컴파일러 기술도 함께 적용된다면 더욱 개선된 결과를 얻을 수 있을 것으로 기대된다. [9][10]의 연구에서는 인공 지능이나 머신 러닝 기술을 사용하여 메모리 시스템 성능 향상을 유도하는 기술을 소개한다. 앞으로는 AI 관련 기술을 본 연구에 적용하는 연구가 계속 진행될 것이다.

V. 결 론

새로운 임베디드 시스템 개발 프로세스에서 응용 프로그램 개발은 컴파일러가 없는 상태에서 명령어 레벨 성능 측정이 어렵다. 응용 프로그램 개발중에 컴파일러와 에뮬레이터가 없는 상황에서 명령어 레벨 성능 측정을 이용한 소스 코드 최적화를 지원할 수 있는 성능 측정기의 필요성이 제기된다. 본 연구는 LLVM 라이브러리를 이용하여 응용 프로그램의 소스코드를 중간 표현 명령어로 변환하고, 타겟 독립적인 명령어 레벨에서 최적화를 지원할 수 있도록 성능 측정 결과를 제공하는 분석기의 개발과 이용에 대하여 기술하고 있다. 이것을 이용하면 보다 빠르게 최적화된 응용 프로그램의 개발 진행 할 수 있을 것으로 예상한다.

References

[1] Online : https://llvm.org/
[2] D. Cho, “A Study on Improvement of Low-power Memory Architecture in IoT/edge Computing,” *Journal of the Korean Society of Industry Convergence*, Vol. 24, No. 1, pp. 69-77, Feb. 2021. DOI: https://doi.org/10.21289/KSIC.2021.24.1.69
[3] J. Yoon, D. Cho, “A spill data aware memory assignment technique for improving power consumption of multimedia memory systems,” Multimedia Tools and Applications, Vol. 78, No. 5, pp. 5463-5478, Mar. 2019. DOI: https://doi.org/10.1007/s11042-018-6783-x
[4] J. Cho, J. Yoon and D. Cho, “Improvement Energy Efficiency for a Hybrid Multibank Memory in Energy Critical Applications,” Technical gazette, vol.27, no. 6, pp. 1946-1955, 2020.
[5] J. Yoon, D. Cho, “Improving memory system performance for multimedia applications,” Multimedia Tools and Applications, Vol. 76, No. 4, pp. 5951-5963, 2017. DOI: https://doi.org/10.1007/s11042-015-2807-y
[6] J. Cho, J. Lee, D. Cho, “Efficient memory design for medical database,” Basic & Clinical Pharmacology & Toxicology, Vol. 125, pp. 198, July. 2019.
[7] J. Cho, J. Lee, D. Cho, “Low-End Memory Subsystem Optimization Process,” International Journal of Smart Home, Vol. 13, No. 2, pp. 11-16, Oct. 2019. DOI: http://dx.doi.org/10.21742/ijsh.2019.13.2.02

[8] J. Cho, D. Cho,, Y. Kim “Study on LLVM application in Parallel Computing System,” The Journal of the Convergence on Culture Technology, Vol. 5, No. 1, pp. 395-399, Feb. 2019.
[9] J. Cho, D. Cho, “Development of a Prototyping Tool for New Memory Subsystem,” International Journal of Internet, Broadcasting and Communication, Vol. 11, No. 1, pp. 69-74, Jan. 2019.
[10]D. Cho, “Study on Memory Performance Improvement based on Machine Learning,” The Journal of the Convergence on Culture Technology, Vol. 7, No. 1, pp. 615-619, Feb. 2021.
[11]D. Cho, “Memory Design for Artificial Intelligence,” International Journal of Internet, Broadcasting and Communication, Vol. 12, No. 1, pp. 90-94, Dec. 2020. DOI: https://doi.org/10.7236/IJIBC.2020.12.1.90