프로파일링 데이터를 이용한 가상기계 코드 최적화

신 양 훈[†] · 이 창 환[††] · 오 세 만[†††]

요 약

가상기계(Virtual Machine)는 소프트웨어로 제작되어 논리적인 시스템 구성을 갖는 컴퓨터이기 때문에 그 수행 속도와 필요 저장 공간 측면에서 성능이 떨어질 수밖에 없다. 이러한 환경에서의 가상기계 코드 최적화는 실행 성능을 향상시킬 수 있기에 중요하다. 특별히 임베디드 장치(Embedded Device)에서 작동하는 가상기계 환경에서의 최적화는 기존의 최적화에 비해 수행 비용 대비 효과에서 높은 효율을 요구한다. 이에 따라 프로파일링을 통하여 성능에 크게 영향을 주는 함수 또는 기본 블록(Basic Block)을 찾아 최적화하는 것이 효과적이다.

본 논문에서는 프로파일링을 이용한 가상기계 코드 최적화기를 설계하고 구현하였다. 먼저, 가상기계 코드 최적화를 위해 코드를 실행하여 얻을 수 있는 동적 정보인 프로파일링 데이터(Profiling Data)를 정의하였고, 프로파일링 정보를 이용한 가상기계 코드 최적기를 구현하였다. 또한, 구현과 실험에 있어서 가상기계 코드는 EVM(Embedded Virtual Machine)의 중간 언어인 SIL(Standard Intermediate Language)를 사용하였고, 구현된 최적화기에 대한 실험을 통해 최적화기의 효과를 확인하였다.

키워드 : 프로파일, 프로파일링 데이터, 코드 최적화, 가상기계 코드, 가상기계

Virtual Machine Code Optimization using Profiling Data

Yang-Hoon Shin[†] · Changhwan Yi[††] · Se-Man Oh[†††]

ABSTRACT

VM(Virtual Machine) can be considered as a software processor which interprets the abstract machine code. Also, it is considered as a conceptional computer that consists of logical system configuration. But, the execution speed of VM system is much slower than that of a real processor system. So, it is very important to optimize the code for virtual machine to enhance the execution time. Especially the optimizer for a virtual machine code on embedded devices requires the highly efficient performance to the ordinary optimizer in the respect to the optimized ratio about cost. Fundamentally, functions and basic blocks which influence the execution time of virtual machine is found, and then an optimization for them may get the high efficiency.

In this paper, we designed and implemented the optimizer for the virtual(or abstract) machine code(VMC) using profiling. Firstly, we defined the profiling information which is necessary to the optimization of VMC. The information can be obtained from dynamically executing the abstract machine code. And we implemented VMC optimizer using the profiling information. In our implementation, the VMC is SIL(Standard Intermediate Language) that is an intermediate code of EVM(Embedded Virtual Machine). Also, we tried a benchmark test for the VMC optimizer and obtained reasonable results.

Key Words : Profile, Profiling Data, Code Optimization, Virtual Machine Code, Virtual Machine

1. 서 론

가상기계란 하드웨어로 이루어진 물리적 시스템과는 달리 소프트웨어로 제작되어 논리적인 시스템 구성을 갖는 개념적인 컴퓨터이다. 가상기계 기술을 이용하면 실행 환경인 프로세서나 운영 체제가 바뀌더라도 응용 프로그램을 수정하지 않고 사용할 수 있기 때문에 프로그램의 이식성을 높일 수 있다. 하지만, 가상기계는 소프트웨어적인 컴퓨터이기 때문에 기존의 물리적인 시스템 위에서 직접 수행되는 것보다 그 수행 속도와 필요 저장 공간 측면에서의 성능이 떨어질 수밖에 없다. 이러한 가상기계 환경에서의 최적화는 가상기계의 소프트웨어적인 측면을 보완하여 그 성능을 향상시키는 면에서 중요하다.

최근 임베디드 장치와 같은 소규모 장치의 확산에 따라 가상기계 또한 그 크기가 소형화 되고 있다. 소형화된 가상기계 환경에서 가상기계 코드에 대한 최적화는 최적화 수행 비용 대비 최적화 효과 면에서 높은 효율을 요구한다. 높은 효율을 위해서는 먼저, 프로그램을 실행하여 프로그램의 동적 정보인 프로파일링 데이터를 얻고, 프로파일링 데이터를 이용하여 프로그램의 성능에 크게 영향을 주는 함수 또는

†준 회 원 : 벨록스소프트 연구원
††종신회원 : 동국대학교 산업기술연구원 겸임교수
†††종신회원 : 동국대학교 컴퓨터공학과 교수(교신저자)
논문접수 : 2006년 9월 8일, 심사완료 : 2007년 3월 8일

기본 블록을 찾아 그 부분을 최적화해야 한다.

본 논문에서는 가상기계 코드 최적화를 위해 코드를 실행하여 얻을 수 있는 동적 정보인 프로파일링 데이터를 정의하고, 프로파일링을 이용한 가상기계 코드 최적화 시스템을 설계하였다. 이를 이용하여 프로파일 정보를 이용한 가상기계 코드 최적화를 수행할 수 있는 기반을 마련하였고, 가상기계 코드의 성능 개선을 도모하였다. 또한 EVM(Embedded Virtual Machine)에서 실행되는 SIL(Standard Intermediate Language) 코드를 대상으로 프로파일링 시스템과 최적화 시스템을 구현하여 실제 가상기계 코드에 대하여 프로파일링 데이터를 추출하고, 추출된 프로파일링 데이터를 이용한 최적화를 수행하여 최적화 효과를 확인하였다.

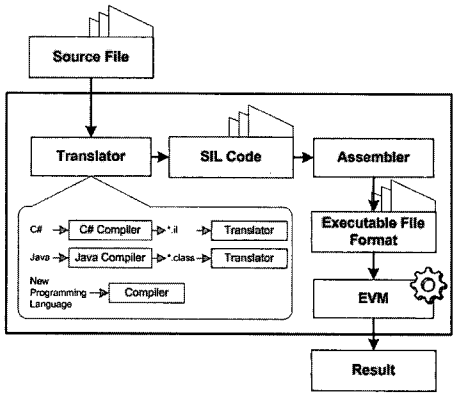
본 논문의 2장에서는 가상기계 코드에 대한 소개와 특징에 대하여 언급하고, 최적화와 최적화를 위한 프로파일링에 대하여 설명한다. 그리고 본 논문에서 사용한 가상기계인 EVM에 대하여 간략하게 소개한다. 본 논문의 핵심이 되는 3장에서는 가상기계 코드 최적화 시스템에 대해 소개하고 각 모듈에 대한 설명과 본 시스템에서 정의하고 사용할 데이터 형식을 기술한다. 4장 실험 결과에서는 구현된 최적화 시스템을 통하여 실제 가상기계 코드에 대한 프로파일링 데이터를 추출하고, 추출된 프로파일링 데이터를 이용하여 성능에 크게 영향을 주는 함수 또는 기본 블록을 찾아 최적화를 수행한다. 또한 실험 결과를 분석하여 가상기계 코드 최적화 비용 대비 성능 개선을 확인한다. 마지막 5장 결론에서는 본 논문에서 제시한 프로파일링 데이터를 이용한 최적화의 의미에 대해 간략하게 요약하고 향후 연구 방향에 대하여 언급한다.

2. 관련연구

2.1. 가상기계 코드

가상기계 코드란 가상기계의 어셈블리 코드로서, 가상기계에서 즉시 실행될 수 있는 이진 코드와 1:1로 대응되는 명령어의 집합을 말하며, 가상기계의 어셈블리 코드라 할 수 있다.

가상기계 코드는 연산 방식에 따라서 스택 기반 코드와 레지스터 기반 코드로 구분할 수 있으며, 일반적으로 복잡한 레지스터 할당 구현의 불필요, 코드 생성 용이, 인터프리터 구현 용이 등의 이유로 스택 기반 코드로 설계된다. 대표적인 스택 기반 가상기계 코드는 Java Bytecode[1], .NET IL[2], 그리고 EVM SIL[3]이 있으며, 레지스터 기반 가상기계 코드는 PASM(Parrot ASseMbly language)[4]이 있다. 가상기계 코드의 구조는 크게 데이터부(Data Section)와 코드부(Code Section)로 구분할 수 있다. 데이터부에는 연산에 이용될 데이터 정보가 지시어로서 표현되어 있으며, 코드부에는 가상기계 코드가 실행되는 연산 정보가 명령어로서 표현되어 있다. 이러한 데이터부와 코드부는 가상기계 코드가 수용하는 고급 언어의 기능, 특성, 그리고 수에 따라서 의사 코드와 연산 코드가 규칙적으로 나타난다[5].



(그림 1) EVM 시스템 구성도

2.2 EVM

EVM은 모바일 디바이스(Mobile Device), 셋톱 박스(Set-top Box), 디지털 TV(Digital TV)등에 탑재되어 동적 응용 프로그램을 다운로드하여 실행하는 가상기계 솔루션이다.

EVM은 크게 C#이나 자바 등의 고급 프로그래밍 언어로 작성된 프로그램을 형태로 번역하는 부분, SIL 코드를 가상기계에서 실행 가능한 형태인 *.evm 포맷으로 변환하는 어셈블러 부분, 실제 하드웨어에 탑재되어 *.evm 파일을 실행하는 가상기계 부분으로 구성된다. EVM 시스템 구성도는 (그림 1)과 같다[4].

SIL은 EVM의 어셈블리 언어로서 EVM에서 실행될 수 있는 이진 코드와 대응되는 명령어 집합이다. 이는 어셈블러에 의해 *.evm 형태로 변환되고 시스템의 운영 체제나 구조에 상관없이 EVM에 의해 실행된다[3].

SIL은 임베디드 시스템을 위한 가상기계의 표준 중간 언어로 설계되었다. 따라서 다양한 프로그래밍 언어를 수용하기 위해서 Bytecode[1], .NET IL[2]등 기존에 널리 사용되고 있는 가상기계 어셈블리 언어들의 분석을 토대로 정의하였다. SIL은 클래스 선언 등 특정 작업의 수행을 의미하는 의사 코드와 가상기계에서 실행되는 실제 명령어에 대응되는 연산 코드로 이루어져 있다. 연산 코드는 스택 기반의 명령어 집합이며 특정 프로그래밍 언어에 종속되지 않는 언어 독립성과 하드웨어 및 플랫폼 독립성을 갖고 있다. 따라서 연산 코드의 니모닉은 특정 하드웨어나 소스 언어에 종속되지 않는 추상적인 형태를 지닌다.

2.3 코드 최적화

코드 최적화란 입력 프로그램과 의미적으로 동등하면서 좀 더 효율적인 코드로 변환하는 것을 의미한다. 따라서 코드 최적화기는 코드의 계산 횟수를 줄이고, 보다 빠른 명령을 사용하여 실행 시간이 짧은 코드로 변환해야 하며, 기억 장소의 요구량을 최소화해야 한다.

코드 최적화 과정은 최적화되는 관점에 따라 여러 가지로 분류된다. 먼저, 프로그램의 부분적인 관점에서 비효율적인 코드를 구분해내어 좀 더 효율적인 코드로 만드는 지역 최

적화와 전체적인 관점에서 최적화하는 전역 최적화로 나눌 수 있다. 또한, 코드를 정적으로 분석하여 최적화하는 정적 최적화와 코드를 실행하여 얻어진 동적인 정보를 분석하여 최적화하는 동적 최적화로 구분할 수 있다[6][7].

효과적인 코드 최적화를 위해서는 성능에 크게 영향을 주는 함수 또는 기본 블록을 최적화하는 것이 중요하다. 잘 알려진 90-90법칙에 따르면 프로그램 코드의 10%가 시스템 자원의 90%정도를 사용하는 경향이 있다. 이것은 복잡하고 방대한 양을 계산하는 대부분의 알고리즘들이 루프를 포함하기 때문이다. 이러한 루프는 종종 순차적으로 처리되는 대신 여러 단계의 중첩을 갖는 계층을 형성한다. 즉, 가장 깊은 단계의 중첩을 갖는 루프가 프로그램의 실행 시간에서 가장 큰 몫을 차지한다. 코드의 양이 많고 느리지만, 거의 호출되지 않는 함수의 최적화는 사실상 의미가 없다. 결과적으로 최적화 효율을 극대화하는 방법은 이와 같은 부분을 최적화 하는 것이다[8][9][10].

2.4 최적화를 위한 프로파일링

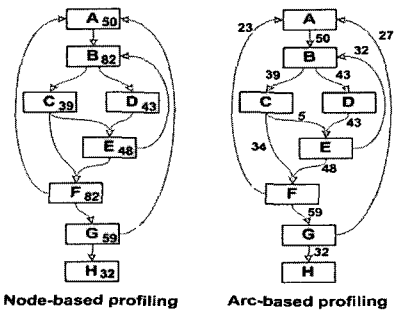
(1) 프로파일링

최적화에 있어서 프로파일링은 프로그램의 전체 또는 특정 부분의 성능을 분석하는 과정을 의미한다. 프로파일링은 대개 프로그램의 성능에 있어 크게 영향을 주는 함수 또는 기본 블록을 파악하기 위해 사용된다[6].

또한, 프로파일링은 프로파일을 하는 방법에 따라 크게 노드 기반(Node-based) 프로파일링과 아크 기반(Arc-based) 프로파일링으로 구분할 수 있다. 노드 기반 프로파일링은 소스 코드를 기본 블록으로 나누어 각 기본 블록의 정보를 수집하는 프로파일링으로서 기본 블록의 특성을 분석하는데 중점을 둔 기법이다. 또한, 아크 기반 프로파일링은 소스 코드를 기본 블록으로 나누고, 기본 블록들 사이의 정보를 수집하는 프로파일링으로서 전체 프로그램의 특성과 기본 블록 간의 관계를 분석하는데 중점을 둔 것이다. (그림 2)는 노드 기반 프로파일링과 아크 기반 프로파일링의 예를 보여준다[11][12].

(2) 가상기계코드 최적화를 위한 프로파일링 데이터

가상기계 코드의 최적화를 위한 프로파일링 데이터가 최적화에서 효과적으로 사용되기 위해서는 가상기계 코드의 특성을 연구하여 그에 알맞은 프로파일링 데이터를 정의하는 것이 중요하다.



(그림 2) 프로파일링의 종류

가상기계 코드는 프로그램 작성의 편의보다는 프로그램의 하드웨어의 독립성을 확보 할 수 있도록 설계된 코드로서 단계적 컴파일을 통한 컴파일러 제작의 용이성을 확보 할 수 있다. 이로 인하여 가상기계 코드는 직접 하드웨어를 접근하여 수행되기보다 가상기계 내의 논리적인 메모리영역인 레지스터(Register), 스택(Stack)과 힙(Heap)영역에서 간접적으로 처리하게 된다. 따라서 프로파일링 데이터는 기본적인 기본 블록 실행 수 (basic block count), 기본 블록 크기 (size of basic block), 함수 호출 수 (function call count)와 더불어 논리적인 메모리 영역을 접근하는 스택 읽기 수 (stack read count), 스택 쓰기 수 (stack write count), 메모리 읽기 수 (memory [heap] read count), 메모리 쓰기 수 (memory[heap] write count), 레지스터 읽기 수 (register read count), 레지스터 쓰기 수 (register write count), 등으로 표현할 수 있다[8][13][14].

3. 코드 최적화 시스템

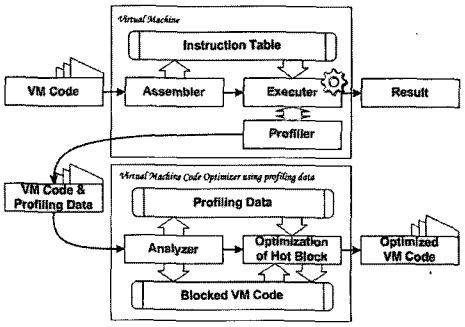
본 논문에서 설계한 프로파일링을 이용한 가상기계 코드 최적화기는 (그림 3)에서 볼 수 있듯이 크게 가상기계 부분과 최적화기 부분으로 나눌 수 있다. 시스템 우선 가상기계가 프로그램의 소스를 컴파일한 결과인 가상기계 코드를 입력으로 받아 실행하여 가상기계 내부의 프로파일러를 통해 가상기계 코드와 가상기계 코드에 대한 프로파일링 데이터를 출력하게 된다. 이 출력은 곧 최적화기의 입력이 되고 최적화기는 프로파일링 데이터를 분석하여 가상기계 코드를 최적화하고 최적화된 가상기계 코드를 출력한다.

3.1 가상기계

가상기계 부분은 (그림 3)에서 볼 수 있듯이 내부적으로 어셈블러 (Assembler), 실행 엔진 (Execution Engine), 그리고 프로파일러(Profiler)로 구성된다. 그 중 최적화를 위해 핵심이 되는 부분은 프로파일러이다. 프로파일러는 기본 블록의 특성과 함수의 호출 및 함수 자체의 특성을 파악하는 역할을 한다. 이는 실행 엔진의 스택 영역과 힙 영역의 정보를 수집하는 방법으로 동작한다.

3.2 프로파일링 데이터

본 논문에서 정의한 가상기계 코드를 최적화하기 위한 프



(그림 3) 최적화 시스템 구성도

〈표 1〉 프로파일링 데이터

이름	Bytes	설명
Function Name	128	Function name
Block ID	8	Basic block ID
Start Line	8	Starting point of function or block
Size	8	Size of function or block
Count	8	Execution count of function or block
Stack Read	8	Stack read count
Stack Write	8	Stack write count
Heap Read	8	Memory read count
Heap Write	8	Memory write count

Profiling Data File Format

```
<file> ::= '%File:' <file_name> { <function_def> }
          '%End-File:'<file_name>

<function_def> ::= <function_info> { <block_info> }

<function_info> ::= '%Function:'<function_name>','
<start_line>','<size>','<count>','<stack_read>','
<stack_write>',' <heap_read>',' <heap_write>

<block_info> ::= <block_ID>','<start_line>','
<size>','<count>','<stack_read>','<stack_write>','
<heap_read>','<heap_write>

<file_name> ::= '$identifier'
<function_name> ::= '$identifier'
<block_ID> ::= '$identifier'
<start_line> ::= '$integer'
<size> ::= '$integer'
<count> ::= '$integer'
<stack_read> ::= '$integer'
<stack_write> ::= '$integer'
<heap_read> ::= '$integer'
<heap_write> ::= '$integer'
```

(그림 4) 프로파일링 데이터 파일 형식 (EBNF)

Algorithm of Profiling

```
Divided according to basic block
{
  Check starting point
  Check finishing point
  Computing size of basic block
}
[Executing source code]
Gathering profiling information of each basic block
{
  Computing execution count
  Stack and heap monitoring
}
Printing profiling information
```

(그림 5) 프로파일링 알고리즘

로파일링 데이터는 <표 1>과 같고, 프로파일링 데이터 파일 형식을 EBNF로 표현한 것은 (그림 4)와 같다. 이와 같은 정보를 통하여 프로그램을 구성하는 함수와 기본 블록에 대한 특성을 파악할 수 있고, 이를 이용하여 성능에 크게 영향을 주는 함수 또는 기본 블록에 대해 최적화를 수행할 수 있다. 본 논문에서 수행한 프로파일링 과정은 먼저 기본블록을 나누고 소스코드를 실행하면서 각각의 기본블록에 대해 프로파일링 정보를 수집 하였고 그 알고리즘은 (그림 5)와 같다.

3.3 코드 최적화기

본 논문의 중심이 되는 코드 최적화기는 (그림 3)에서와 같이 내부적으로 분석기 (Analyzer) 부분과 핫 블록 최적화 (Optimization of Hot Block) 부분으로 구성된다. 분석기 부분은 프로파일러를 통해 산출된 프로파일링 데이터를 분석하여 프로그램의 성능에 크게 영향을 주는 함수나 기본 블록을 찾아 최적화할 부분을 결정하는 역할과 SIL 코드를 함수와 기본 블록으로 블록화하는 역할을 한다. 핫 블록 최적화 부분은 분석기에 의해 얻어진 정보를 토대로 대상 SIL 코드 블록을 최적화하는 역할을 한다. 프로파일링을 이용한 최적화방법론은 다양한 방법론이 사용될 수 있으나 여기서는 패턴 매칭을 이용한 코드 최적화 방법론을 사용하였다.

최적화는 프로파일링 정보에 의해 최적화 대상으로 결정된 함수나 기본 블록을 지역적으로 관찰하면서 불필요한 명령어는 제거하고, 개선될 수 있는 코드는 동등한 의미의 효과적인 코드, 또는 가상기계에서 제공하는 동등한 의미의 복합 코드 (Complex Instruction)로 변환한다. 이러한 코드 변환은 코드의 크기를 줄이고 그에 따라 처리 속도를 향상 시킨다.

프로파일링을 이용한 최적화기의 구현 측면에서 더욱 효과적인 최적화를 위하여 최적화 대상 기본 블록이나 함수에 대한 접근 테이블(Access Table)을 두어 최적화 대상인 함수와 코드에 빠르게 접근할 수 있도록 구성하였다.

4. 실험 결과

실험을 위해서 우선 소스 프로그램을 컴파일러를 통해 EVM의 중간코드인 SIL 코드로 출력하고, EVM을 통해 SIL 코드를 실행하여 프로파일링 데이터를 산출한다. 최적화기는 산출된 프로파일링 데이터와 SIL 코드를 입력으로 받아 최적화를 수행하여 최적화된 SIL 코드를 출력한다.

실험은 최적화 이전의 SIL 코드와 코드의 변화가 없을 때까지 최적화하여 얻어진 SIL 코드 그리고 본 논문에서 제시한 프로파일링을 이용한 최적화기를 통하여 얻어진 SIL 코드의 실행 시간, 코드의 크기 그리고 최적화에 필요한 시간을 비교 분석하여 최적화 효율을 측정한다. 본 논문에서 제시하는 최적화기는 ANSI-C 호환 컴파일러인 Visual C++ 6.0 컴파일러를 사용하여 구현하였고, 실행환경으로 EVM을 사용하여 실험하였다.

4.1 최적화 예제

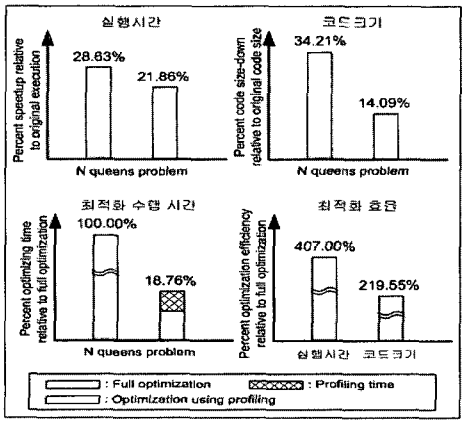
실험 결과를 내기에 앞서 한 프로그램에 대해 최적화를 수행하여 최적화 코드의 실행 시간 및 코드의 크기, 최적화 수행에 필요한 시간 그리고 최적화 효과에 대해 살펴본다. 테스트 소스는 C 언어로 구현된 "The N-queen problem"을 사용하였다. EVM을 통하여 산출된 프로그램의 프로파일링 데이터 결과는 (그림 6)와 같다. 최적화 이전과 충분히 최적화한 것과, 본 논문에서 제시한 최적화기로 최적화한 것의 실행 시간, 코드 크기 그리고 최적화 효율에 대해 나타낸 그래프는 (그림 7)과 같다. 본 논문에서 언급하고 있는 '충분히 최적화'하였다는 것의 의미는 가상기계코드를 최적화하여 더 이상 변화가 없는 상태 즉, 최적화를 지속적으로 수행하여 더 이상 가상기계코드가 최적화되지 않는 상태까지 최적화한 것을 의미한다.

실험을 통해 실행 시간은 21.86% 개선되었고 코드 크기는 14.09% 개선되었다. 이는 충분히 최적화한 것에 비해

Profiling Data - N queen problem

```
%File: N_queens.c.sil
%       Function: main
%                       $$0,20,4,1,0,5,0,1
%                       $$1,24,4,9,18,18,9,0
%                       $$2,28,4,8,16,16,8,8
%                       $$3,32,3,8,8,0,8,8
%                       $$4,35,3,1,0,0,0,1
%                       $$5,38,4,16,32,32,16,0
%                       $$6,42,8,15,75,75,60,30
%                       $$7,50,3,15,15,0,15,15
%                       $$8,53,8,1,0,2,2,2
%       End-Function: main,20,41,1,13,18,11,9
%       Function: rn_queens
%                       $$9,61,10,1965,7860,9825,1965,5895
%                       $$10,71,5,17685,53055,53055,53055,0
%                       $$11,76,6,15720,15720,31440,31440,31440
%                       $$12,82,44,2056,74016,74016,22616,6168
%                       $$13,126,2,92,0,0,0,92
%                       $$14,128,4,828,1656,1656,1656,0
%                       $$15,132,21,736,736,4416,2208,7360
%                       $$16,153,3,736,736,0,736,736
%                       $$17,156,6,92,92,92,0,184
%                       $$18,162,8,1964,3928,7856,3928,3928
%                       $$19,170,40,2056,67848,67848,18504,6168
%                       $$20,210,1,15720,0,0,0,0
%                       $$21,211,8,15720,94320,78600,47160,15720
%                       $$22,219,3,1965,0,0,3930,0
%       End-Function:
%               rn_queens,61,161,1965,176850,190605,76635,55020
%       Function: position_ok
%                       $$23,222,16,15720,172920,188640,62880,31440
%                       $$24,238,12,5508,60588,60588,27540,0
%                       $$25,250,3,15720,15720,15720,15720,0
%                       $$26,253,14,3420,44460,44460,13680,0
%                       $$27,267,4,15720,31440,31440,31440,0
%       End-Function:
%               position_ok,222,49,15720,597360,613080,251520,31440
%End-File: N_queens.c.sil
```

(그림 6) 프로파일링 데이터



(그림 7) 최적화 결과

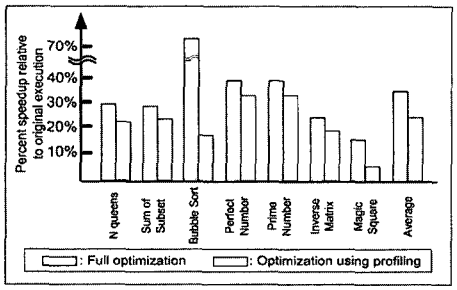
실행 시간은 91.34%, 코드 크기는 76.58% 정도의 성능을 보이는 것이다. 최적화 수행 시간은 충분히 최적화한 것의 18.76%정도 소비되는 것으로 나타났다. 따라서 최적화 효율은 실행 시간에 있어서 충분히 최적화한 것에 비해 407.00% 정도 나타났고, 코드 크기에 대해서는 219.55%정도 나타났다.

4.2 최적화 실험 결과

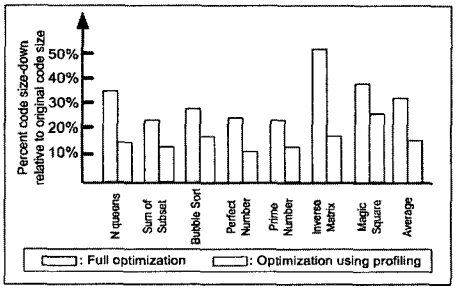
본 장에서는 다양한 프로그램을 위와 같은 방법으로 최적화하여 최적화 이전과 충분히 최적화한 것 그리고 본 논문에서 제시한 최적화기로 최적화한 것의 실행 시간, 코드 크기 그리고 최적화 효율에 대해 실험하여 최적화기의 성능에 대해 설명한다.

실험 결과는 다양한 프로그램에 대하여 다음의 네 가지로 표현하였다. 먼저 최적화하지 않은 SIL 코드를 기준으로 충분히 최적화한 것과 본 논문에서 제시한 최적화기로 최적화한 것의 실행 시간의 차이와 코드 크기의 차이를 비율로 표현하였다. 그 결과는 각각 (그림 8)과 (그림 9)와 같다. 또한 최적화 수행 시간을 충분히 최적화한 것을 기준으로 본 논문에서 제시한 최적화기로 최적화한 것의 최적화 수행 시간을 비율로 표현하였다. 그 결과는 (그림 10)와 같다. 마지막으로 최적화 수행 시간 대비 최적화 효과를 충분히 최적화한 것의 효율을 기준으로 본 논문에서 제시한 최적화기로 최적화한 것의 효율을 비율로 표현하였다. 그 결과는 (그림 11)과 같다.

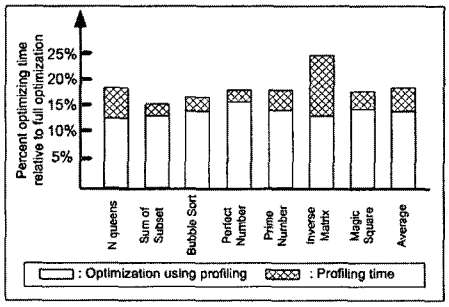
(그림 8)과 (그림 9)에서 볼 수 있듯이 최적화 효과는 실행 시간 측면에서 22.44%, 코드 크기 측면에서 14.77% 정도의 효과가 나타났고, 또한, 이는 충분히 최적화한 것에 비하여 실행 시간 측면에서 63.88%정도, 코드 크기 측면에서는 47.05%정도의 효과임을 확인하였다. 최적화 수행 시간과 최



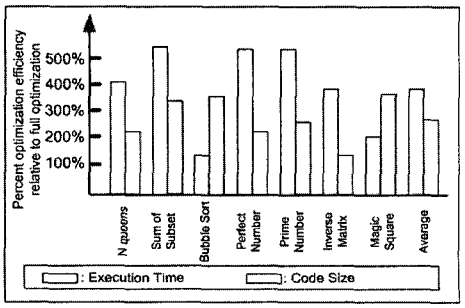
(그림 8) 실행 시간



(그림 9) 코드 크기



(그림 10) 최적화 수행 시간



(그림 11) 최적화 효율

적화 효율 측면에서는 (그림 9)와 (그림 11)에서 볼 수 있듯이, 최적화 수행 시간은 충분히 최적화한 것의 18.42%정도임을 확인하였고, 이에 따른 최적화 효율은 충분히 최적화한 것을 기준으로 볼 때 실행 시간 측면에서 378.10%정도, 코드 크기 측면에서 271.33%정도의 효율을 나타냄을 확인하였다.

5. 결론 및 향후 연구

최근 임베디드 장치와 같은 소규모 장치의 확산에 따라 가상기계 또한 그 크기가 소형화 되고 있다. 이러한 가상기계 환경에서의 최적화는 기존의 최적화에 비해 최적화 수행 비용 대비 최적화 효과 면에서 높은 효율을 요구한다. 이에 따라 프로파일링을 통하여 성능에 크게 영향을 주는 함수 또는 기본 블록을 찾아 최적화하는 것이 중요하다.

본 논문에서는 가상기계 코드의 특징을 분석하여 그 특징에 맞는 프로파일링 데이터를 정의하였고, 프로파일링 데이터를 이용한 가상기계 코드의 최적화 시스템 설계하여 성능에 크게 영향을 주는 함수 또는 기본 블록을 선택적으로 최적화 할 수 있도록 하였다. 나아가 EVM의 가상기계 코드인 SIL을 통하여 실험하여 프로파일링 데이터를 산출 및 최적화를 수행하여 성능 향상을 확인하였다. 실험 결과 프로파일링을 이용한 가상기계 코드 최적화를 통해 충분히 최적화한 것의 평균 18.42% 정도의 비용으로 수행 시간에 있어서 충분히 최적화 한 것의 평균 63.88% 정도, 코드 크기에 있어서는 평균 47.05%에 이르는 효과를 얻을 수 있음을 확인 할 수 있었다. 이를 통해 프로파일링을 이용하여 프로그램의 성능에 영향을 주는 부분을 찾아 최적화하는 것이 최적화 효율에 있어서 충분히 최적화한 것에 비해 수행 시간은 평균 378.10%, 코드 크기는 평균 271.33%정도 효과적임을 확인하였다.

향후에는 최적화 알고리즘을 보다 효율적으로 보완하여, 보다 양질의 최적화된 코드를 생성할 것이다. 또한, 프로파일링 오버헤드를 경감하여 가상기계 코드를 가상기계에서 실행하는 중에 최적화 할 수 있도록 최적화기를 가상기계 안에 탑재 할 계획이다.

참 고 문 헌

[1] Tim Lindholm, Frank Yellin, "The Java Virtual Machine Specification", 2nd Ed., Addison Wesley, 1999.

[2] "MSIL Instruction Set Specification-Version 1.9 Final", Microsoft Corporation, 2000.10

[3] 남동근, 윤성림, 오세만, "가상기계를 위한 어셈블리 언어", 정보처리학회 학술 발표 논문집, 제11권, 제9호, pp.519-522, 2004.3.

[4] D. Sugalsk, Perl.org sites, www.parrotcode.org.

[5] 오세만, 이양선, 고광만, "임베디드 시스템을 위한 가상기계의 설계 및 구현", 멀티미디어학회 논문지, 제8권, 9호, pp.1282-1291, 2005.8.

[6] Kris Kaspersky , "Code Optimization: Effective Memory Usage", A-List Publishing, 2003.9.

[7] 오세만, 컴파일러 입문 개정판, 정익사, 2005.

[8] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," in PLDI'00, pp.1 - 12, ACM Press, 2000.

[9] Thomas Bell, "The Concept of Dynamic Analysis," ACM SIGSOFT international symposium on Foundations of software engineering table of contents, pp.216-234, 1999.

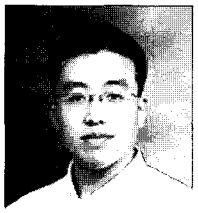
[10] Jon Louis Bentley, "Bumper-Sticker Computer Science," Communications of the ACM, Volume 28, Issue 9, pp.896-901, 1985.9.

[11] Thomas Ball, Peter Mataga, Mooly Sagiv, "Edge Profiling versus Path Profiling," ACM Symposium on Principles of Programming Languages, pp.134-148, 1998.

[12] Michael D. Bond, Kathryn S. McKinley, "Continuous Path and Edge Profiling," IEEE/ACM International Symposium on Micro-architecture table of contents, pp.130-140, 2005.

[13] Pohua P. Chang, Scott A. Mahlke, Wen-mei W. Hwu, "Using Profile Information to Assist Classic Code Optimizations," IEEE Software Practice and Experience, Vol.21, No.12, pp.1301-1321, 1991.

[14] James E. Smith, Ravi Nair, "Virtual Machines - Versatile Platforms for Systems and Processes", Morgan Kaufmann, 2005.



신 양 훈

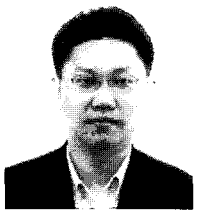
e-mail: salz23@dongguk.edu

2005년 동국대학교 컴퓨터공학과 졸업

2007년 동국대학교 대학원 컴퓨터공학과 공학석사

2007년~현 재 벨록스소프트 연구원

관심분야 : 프로그래밍 언어, 가상기계, 실시간 운영체제



이 창 환

e-mail: yich@dongguk.edu

1998년 동국대학교 컴퓨터공학과 졸업

2000년 동국대학교 대학원 컴퓨터공학과 공학석사

2003년 동국대학교 대학원 컴퓨터공학과 공학박사

2006년~현 재 (주) 링크젠 책임연구원

2007년~현 재 동국대학교 산업기술연구원 겸임교수

관심분야 : 프로그래밍 언어, 컴파일러, 임베디드 시스템

오 세 만

e-mail: smoh@dongguk.edu

1993년~1999년 동국대학교 컴퓨터공학과 대학원 학과장

2001년~2003년 한국정보과학회 프로그래밍언어연구회 위원장

2004년~2005년 한국정보처리학회 게임연구회 위원장

1985년~현 재 동국대학교 컴퓨터공학과 교수

관심분야 : 프로그래밍 언어, 컴파일러, 모바일 컴퓨팅