특집

임베디드 프로세서를 위한 최적화 컴파일러 기술

이제형, 이준표, 이승일, 문수묵 (서울대학교 전기컴퓨터공학부)

I. 개요

내장형 시스템은 다른 기기 내에 탑재되어 주어진 기능을 수행하기 위한 소형 컴퓨터 시스템이다. 예전엔 주로 셋톱 박스, 휴대용 단말장치 등의 예를 들어 설명했으나 이젠 우리 주변의 거의 모든 전자제품에 탑재되어 있다고 봐도 무리가 없다. 내장형 시스템은 일반 범용 컴퓨터와 비교해서 뚜렷하게 차별되는 특징들을 가지고 있는데, 사용자가 임의로 시스템을 확장하기 힘들고, 최소한의 사양으로 기능을 만족시켜야 하며, 휴대용 기기들의 경우 지속적인 전원을 공급받을 수 없기 때문에 전력소모에 매우 민감하다는 것이 그것이다.[1]

내장형 시스템은 기기가 요구하는 성능을 발휘할 수 있는 최소 사양의 프로세서, 최소한의 저장장치 등으로 구성되어 있기 때문에 탑재될 소프트웨어의 개발에는 많은 제약이 따르며, 이러한 저 사양을 극복하기 위한 소프트웨어 최적화의 필요성이 존재한다. 또한 같은 기능을 수행하면서도 최적화 되지 못한 소프트웨어는 더 긴 수행시간을 필요로 하므로 더 많은 전력소모를 야기하게 되어 저전력 설계에 대한 요구를 만족하지 못할 수 있다. 특히 최근에는 서로 다른 영역의 제품들이 한 제품에 통합되는 퓨전형 가전 모델들이 시장을 선도 함에 따라 다양한 기능을 수행하는 복잡한 소프트웨어 시스템이 등장하고 있기에 소프트웨어 최적화의 필요성은 더욱 부각되고 있다.

소프트웨어를 최적화하는 데 있어서 필요한 기술중의 하나는 작성된 프로그램을 보다 작고 빠르게 동작하는 코드로 생성해 내는 최적화 컴파일러 기술이다. 최적화 컴파일러의 목표는 똑같은 동작을 수행하면서도 보다 작은 저장공간을 차지하고, 보다 빠른 실행이 가능한 코드를 만들어 내는 것이다. 이 두 가지 큰 목표는 동시에 달성할 수 있기도 하지만 때로는 서로 상충하기도 한다. 특히 내장형 시스템은 그 공간 제약적인 요소 때문에 성능 지향적인 최적화보다는 공간 지향적인 최적화가 더 중요한 경우가 종종 발생한다. 이 경우 성능을 다소 희생하더라도 어떻게든 작은 코드를 생성하여 소프트웨어가 탑재되어 동작할 메모리를 최소화하는 것이 관건이다. 이것은 메모리의 단가가 가격결정에 중대한 요소로 작용하기 때문이며, 한번 시스템이 제작되어 출시되면 대개 더 이상 메모리를 포함한

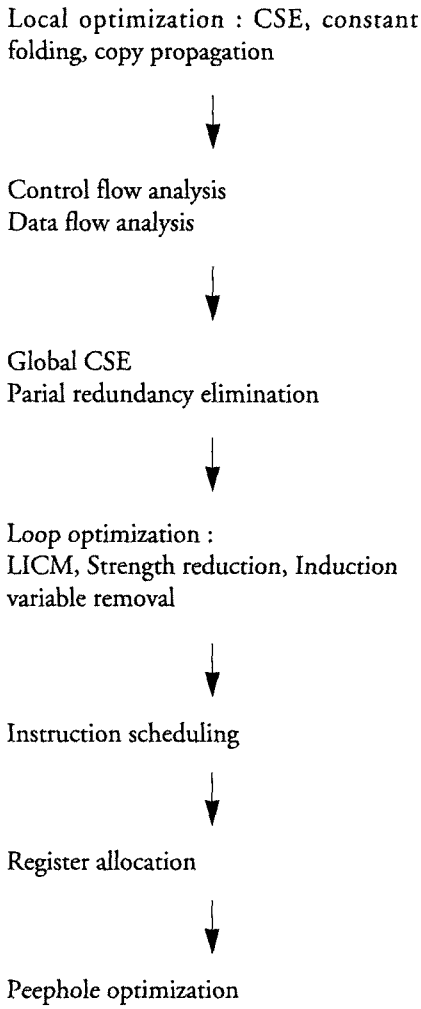
시스템의 하드웨어적인 확장이 불가능한 것도 중요한 이유이다. 특히 소프트웨어의 업그레이드가 가능하거나, 여러 개의 소프트웨어가 동시에 동작해야 하는 시스템의 경우에는 공간 최적화가 더욱 중요하다.

본 고에서는 내장형 시스템을 위한 최적화 컴파일러 기술을 소개한다. 우선 Ⅱ절에서 일반적인 최적화 컴파일러 기술에 대하여 간단히 리뷰하고 Ⅲ절에서는 내장형 시스템을 위하여 기존의 최적화 기술이 어떻게 보완되고 수정되어야 하는지를 설명한다. Ⅳ절에서는 최근 내장형 시스템에서 많이 채택하고 있는 자바 플랫폼에 대하여, 수행 속도를 높이기 위해 바이트코드를 기계어 코드로 번역하는 방식에서 사용하는 최적화 기술을 소개한다. 마지막으로 Ⅴ절에서 최적화 컴파일러 기술에 대하여 요약한다.

Ⅱ. 일반적인 컴파일러 최적화

일반적으로 컴파일러의 구조는 프로그래밍 언어의 문법에 따라 어휘, 구문, 의미 분석을 담당하고 중간 코드를 생성하는 전단부(front-end)와 중간 코드에서 기계어 코드를 생성하고 최적화를 수행하는 후단부(back-end)로 나누어진다. 이 중 후단부는 프로그램의 성능에 직접적인 영향을 주기 때문에 많은 연구 개발이 수행되고 있다.

후단부의 코드 최적화는 다양한 최적화와 분석 기법들을 단계별로 구분하여 순서를 가지고 적용하도록 되어있는 데 <그림 1>은 최적화 단계 순서의 한 예를 보여주고 있다. 입력으로 기호(symbolic) 레지스터를 가진 유사(pseudo) 기계어 코드를 받아 (이러한 기호 레지스터를 앞으로는 변수라고 지칭하겠음) 각 단계를 거치며 더 좋은 코드로 변환하여 최종적으로 하드웨어 레지스터를 할당한 실제 기계어 코드를 생성한다.



〈그림 1〉 최적화 단계 적용 순서의 예

최적화 기법은 그 적용되는 범위에 따라 국부(local) 최적화 기법과 광역(global) 최적화 기법으로 구분된다. 국부 기법은 코드내의 기본 블록(basic block) 이나 확장된 기본 블록(extended basic block) 내에서만 이루어지는 최적화이고 광역 기법은 코드 전체에 대하여 이루어지는 최적

화이다. 많은 최적화 기법은 국부용 기법과 광역용 기법이 따로 존재하고 있다. (예: 국부 레지스터 할당 기법과 광역 레지스터 할당 기법)

1. 국부 최적화 기법

위에서 언급한 기본 블록이란 연속된 명령어들의 집합으로 그 최초 명령어에서 시작하여 마지막 명령어까지 순차적으로 수행되며 도중에 프로그램의 흐름이 끼어들거나 분기되지 않는, 최적화의 최소 단위이다. 따라서 기본 블록의 첫 번째 명령어는 함수의 시작점이거나 분기문의 목적 명령어, 혹은 분기문의 바로 다음에 오는 명령어이다.[2] 확장된 기본 블록은 그 블록 안으로 프로그램의 흐름이 끼어들지는 못하지만 밖으로 분기되는 것은 허락되는 연속된 명령어의 집합이며 하나 이상의 기본 블록들로 이루어 진다. 국부 최적화는 블록 안에 복잡한 제어 흐름(control flow)이 존재하지 않기 때문에 간단한 데이터 흐름(data flow) 분석만을 통하여 쉽게 이루어질 수 있으므로 보통 최적화의 첫 단계에서 수행된다. 또한 컴파일러의 가장 기본적인 최적화 플래그를 켰을 때(예 : -O1) 컴파일러는 국부 최적화만 수행하고 국부 레지스터 할당으로 코드를 생성한다. 국부 최적화에 사용되는 최적화 기법들은 다음과 같다.

CSE(Common Subexpression Elimination) : 어떤 연산 결과가 아직 유효한 영역에서 동일한 결과 값을 가지는 연산이 있을 때 이 연산을 common subexpression 이라고 하며 이것을 제거하는 기법이다. 예를 $x = y * z$ 라는 명령어 이후에 $w = y * z$ 라는 명령어가 나타나는 경우 그 사이에 y 나 z 를 정의하는 명령어가 없다면 $y * z$ 는 common subexpression 이며 이 명령어를 $w = x$ 로 바꿈으로써 불필요하게 다시 $y * z$를 연산해야 하는 낭비를 피할 수 있다. 만약 뒤의 명령어가 $x = y * z$였다면 이것은 $x = x$ 로 변환되며, 불필요한 명령어로 인식되어 완전히 제거할 수 있다.

Dead-code elimination : 해당 명령어에서 정의된 값을 어떤 명령어도 사용하지 않는 경우, 그 명령어는 dead-code가 되며 이것은 안전하게 제거할 대상이 된다.

Copy propagation : $x = y$ 라는 복사 명령어가 존재할 때, 이후 x 값이 변하지 않는 범위 내에서 나타나는 x 를 모두 y 로 치환하는 방법이다. 즉 $x = y$ 다음에 $w = x + y$ 가 있으면 이는 $w = y + y$로 변환할 수 있다. 그리고 더 이상 $x = y$의 결과를 사용하는 명령어가 없다면 $x = y$는 dead-code가 되어 제거할 수 있다.

Constant folding : 연산 결과값이 상수라는 것이 컴파일시에 파악 가능한 명령어에 대해 직접 그 상수로 대체하는 기법이다. 예를 들어 $x = 3 * 4$는 $x = 12$로 바꿀 수 있다. 이것은 다음의 constant propagation이라는 최적화가 적용될 기회를 가져다 준다.

Constant propagation : copy propagation 과 마찬가지로 $x = c$ (여기서 x 는 변수, c 는 상수)라는 복사 명령어가 존재할 때, 이후 x 값이 변하지 않는 범위 내에서 나타나는 x 를 모두 c 로 치환하는 방법이다.

위에서 설명한 최적화 기법들은 프로그래머가 서투르게 프로그래밍을 했기 때문에 필요한 것

이 아니라 기계어로 번역하는 과정에서, 혹은 이후 최적화가 진행되는 과정에서 부득이 비효율적인 코드가 생성되기 때문이다. 예를 들어 a[i] = a[j]+1; 이라는 문장의 경우 a[i]와 a[j]의 번역을 위해 배열 a[ ]의 시작 주소를 구하는 명령어를 각각 생성해야 하는 데(예 : x = &a와 y = &a) 이 경우 CSE를 필요로 하게 되며 (y = &a를 y = x로 치환) 이후 copy propagation과 dead-code elimination도 가능할 수 있다.

2. 광역 최적화 기법

코드 전체를 대상으로 최적화가 이루어지므로 코드의 복잡한 프로그램 흐름을 고려하여야 한다. 이를 위해 우선 제어 흐름 분석을 통해 루프(loop)등의 제어 구조를 찾고 데이터 흐름 분석을 통해 각 명령어가 정의(define)한 변수가 코드의 어디까지 도달(reach)하게 되는 지, 그리고 코드의 각 부분에서 그 이후에 사용(use)되고 있는 변수는 무엇인지를 계산하게 된다. 이러한 분석 결과를 바탕으로 다음의 최적화 기법들이 코드 전체에 대하여 적용된다.

Global CSE : 광역 코드에 대하여 CSE를 적용하는 기법이다. 광역 분석을 통해 각 블록의 시작점에서 유효한 연산(available expressions)을 모두 찾고 블록 안에 같은 연산을 하는 명령어가 있으면 CSE를 적용한다. 때로는 프로그램의 제어 흐름에 따라 특정 경로에 대해서만 유효한 common subexpression이 있을 수 있는 데 이를 부분적으로 제거하는 기법을 PRE(Partial Redundancy Elimination) 라고 부른다.[2] 예를 들어 if( ) {w=y*z}; w=y*z; 에서 y*z는 if( )가 참인 경로에서 중복 수행되는 데 이를 if( ) {w=y*z} else {w=y*z}; 로 바꾸면 중복 수행을 피할 수 있다.

Loop optimization: 루프는 프로그램 수행의 많은 시간을 차지하는 부분이므로 여러 기법을 사용하여 최적화한다. LICM(Loop Invariant Code Motion)은 루프 내부에 존재하면서도 연산 결과가 항상 상수 값을 유지하는 명령어들을 루프 밖으로 끄집어 냄으로써 루프 내부에서 수행되는 명령어의 개수를 줄여 성능향상을 꾀하는 기법이다. strength reduction은 루프의 매 반복마다 일정한 크기로 변화하는(예 : x = x + 1) induction 변수를 이용한 곱셈이나 나눗셈 연산을, 이보다는 더 연산비용이 적은 덧셈이나 뺄셈으로 대체하는 것을 말한다. 이러한 과정을 통해 기존의 induction 변수를 이용한 루프 출구 테스트도 덧셈이나 뺄셈 결과에 대한 테스트로 바꿀 수 있고 결국은 induction 변수가 더 이상 유용하게 사용되지 않아 제거할 수 있는 데 이를 induction variable elimination이라고 한다.

Instruction scheduling : 명령어의 순서를 바꿈으로써 여러 싸이클 동안 수행되는 명령어(예: load, floating-point computation)의 수행 시간을 감추는 효과를 얻거나, 수퍼스칼라(superscalar)나 EPIC(Explicitly Parallel Instruction Computing)처럼 명령어 수준의 병렬처리를 위한 마이크로프로세서에서 코드의 병렬도를 높이는 기법이다.

Register allocation : 레지스터 기반의 연산이 메모리 기반의 연산보다 빠르므로 가능한 많은 live range(변수의 값이 유효한 구간)를 레지스터에 할당하는 것이 좋다. 흔히 사용하는 기법은 각각의 live range를 그래프의 노드(node)로 나타내고 같은 레지스터로 할당될 수 없는 live range 노드들 사이에는 edge를 그린 다음, 주어진 레지

스터들을 노드에 할당하되 edge로 연결된 노드에는 같은 레지스터를 할당하지 않는, 그래프 컬러링(Graph Coloring)에 바탕을 두고 있다.[2] 이 과정에서 복사 명령어로 복사되는 노드들에 동일한 레지스터를 할당하면 복사 명령어를 제거하는 효과를 얻을 수 있다. 주어진 레지스터들로 그래프의 모든 노드를 할당할 수 없으면 적당한 노드들을 메모리에 방출(spill)해야 한다. 선택된 노드들의 방출을 위해 load와 store명령을 삽입한 새로운 코드에 대하여 그래프 컬러링을 반복한다. 이러한 이유 때문에 레지스터 할당 단계는 최적화 단계들 중에서 가장 오랜 시간이 걸리며 어려운 문제들을 많이 포함하고 있다.

Peephole optimization : 모든 최적화가 끝나면 마지막 단계에서 적은 노력으로 할 수 있는 마무리 최적화를 수행하는데 여기에는 분기문들을 재배치하거나 조건문을 달리하여 불필요한 분기문을 제거하는 작업등 좁은 영역(peephole)을 대상으로 하는 많은 국부 최적화가 포함된다.

III. 내장형 컴파일러 최적화

내장형 시스템에서 적용되는 컴파일러 최적화 기술은 성능도 추구하지만, 최종 코드의 크기를 줄이는 방법이 우선시되는 경우가 종종 발생한다. 위에서 설명한 일반적인 최적화 기법들이 내장형 시스템을 위해 어떻게 맞추어 사용되는지 살펴본다.

Redundancy elimination : 코드 생성단계에서 불필요하게 생성되었거나, 실행흐름의 분석을 통해 굳이 남겨두지 않아도 되는 명령어들을 색출하여 제거함으로써 코드의 크기를 줄임과 동시에 성능에도 좋은 효과를 발휘하는, 내장형 시스템에 가장 적합한 최적화라고 할 수 있다. CSE, dead-code elimination, PRE 등이 이 부류에 속하며 이 기법들은 비교적 구현하기도 쉽기 때문에 대부분의 최적화 컴파일러에서 이 기법들을 채택하고 있다.

Loop optimization : 대부분의 루프 최적화는 코드 크기의 감소 효과는 작을지 몰라도 성능 향상에는 크게 도움이 된다. 특히 행렬연산 등의 반복연산을 주로 하는 DSP의 경우에 루프 최적화는 매우 효과적이다.

Register allocation : 내장형 시스템은 대개 극소수의 레지스터를 가지는 프로세서를 사용하는 경우가 대부분이다. 따라서 효율적이지 못한 레지스터 할당은 많은 메모리 방출 명령어를 양산하게 되므로 최대한 메모리 방출을 줄이거나, 필요하더라도 수행 빈도가 적은 곳에 위치하도록(예: 루프 외부)최적화해야 한다. 특히 내장형 시스템의 마이크로프로세서는 뱅크 구조(banked) 레지스터 등 특이 구조의 레지스터 파일이 사용되거나 메모리 접근성을 용이하게 하기 위한 특이 명령어 등이 제공되는 경우가 많으므로 이런 특성들을 잘 이용해야 한다.[3]

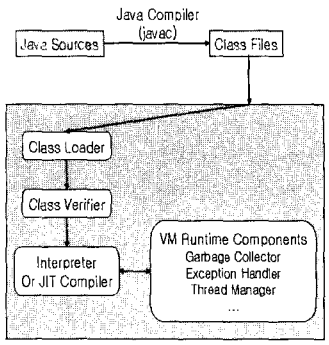
Instruction scheduling : 내장형 시스템에서는 스케쥴링 기법이 적극적으로 이용되지는 않지만 코드 전체를 대상으로 하는 것이 아닌 일부 명령어에 한해 프로세서의 파이프라인을 효과적으로 운용할 수 있도록 제한적인 스케쥴링이 이용되기도 한다.

Peephole optimization : 코드 품질 향상을 위해 실제로는 수행되지 않는 불필요한 코드 영역을 삭제하거나, 연속 혹은 종속적으로 나타나는 분기문을 최적화하고, 프로세서가 가지고 있는 특수한 명령어를 적용할 수 있는 기회를 좁은 영역에서 찾는다.

Code compression : 성능을 희생하면서라도 코드의 크기를 줄이는 가장 적극적인 방법으로 코드를 압축하는 방법이 있다. 비슷한 형태의 코드를 검색하고 인출하여 마치 하나의 서브루틴처럼 만들어주는 procedural abstraction[4]과 같이 따로 다른 프로그램이나 하드웨어의 도움이 없이 적용이 가능한 비교적 소극적인 압축기법이 있는 반면, 잘 알려진 데이터 압축 방법들을 동원하여 실행 코드 자체를 압축하여 가지고 있다가 실제로 수행하는 단계에서는 이 압축을 풀어서 코드를 실행하는 적극적인 방법도 이용되고 있다. 다만 후자의 경우에는 항상 수행 중에 압축을 풀어주는 하드웨어나 소프트웨어의 도움을 받아야 한다. 일반적인 데이터 압축 기법들은 압축을 시작한 곳에서 끝나는 곳까지 순차적으로 모두 풀어낸 후에야 실행에 옮길 수 있는, 시간, 공간적인 단점이 있기 때문에 압축률은 떨어지지만 임의의 위치에서부터 쉽고 빠르게 압축해제가 가능한 간결한 압축법을 개발하여 이용하기도 한다.[5]

IV. 번역 기반의 자바 성능 향상 최적화

최근 많은 내장형 시스템들이 자바를 채택하고 있다. 예를 들면 디지털 TV, 휴대폰, 홈 네트워크, 텔레매틱스 등에서 자바를 내장형 소프트웨어 플랫폼으로 사용하고 있다. 이러한 시스템에서 자바가 채택되는 이유로는 우선 가상 머신



〈그림 2〉일반적 자바 가상 머신 수행 방식

을 사용함으로써 프로세서, OS, 하드웨어 기반이 다양한 내장형 플랫폼에서 일관된 실행 환경을 제공할 수 있고 보안에 있어서도 바이러스나 해킹 코드로 인한 전체 시스템 붕괴의 가능성이 낮으며, 충분히 성숙된 풍부한 API와 프로그램의 안정성을 높여주는 언어 기능(쓰레기 수집기, 예외 처리)으로 인해 소프트웨어 콘텐츠 개발이 용이하기 때문이다.

이러한 장점에도 불구하고 자바는 "write once, run everywhere"로 대변되는 강력한 이식성을 보장하다 보니 성능에는 큰 문제점을 안고 있다. 즉 자바는 기계어 코드로 컴파일되는 것이 아니라 바이트코드라는 중간 코드로 컴파일되기 때문에 하드웨어가 아니라 가상 머신이라는 소프트웨어에 의해 해석기(interpreter)를 통하여 수행되고, 따라서 느릴 수 밖에 없다. <그림 2>는 해석기를 이용한 일반적인 자바 수행 방식을 보여주고 있다. 자바 컴파일러를 통해 컴파일된 프로그램은 클래스 파일로 저장된다. 이 클래스 파일 안에는 실제 코드에 해당하는 바이트코드와 그 바이트코드를 실행하기 위한 여러 가지 정보들이 함께 들어있다. 자바 가상 머신에서는 이 클래스 파일을 읽어 들인 후 이를 가상 머신에서

이해할 수 있는 형태로 변환한 다음에 바이트코드를 해석하여 실행하게 된다.

자바의 성능 단점을 보완하기 위한 방안으로 바이트코드를 기계어 코드로 번역(translation)하여 수행하는 방법이 사용되고 있다. 적시 컴파일러 (Just-in-Time Compiler, JITC)에서는 수행시간 중에 동적으로 바이트코드를 머신 코드로 번역한 뒤 수행하며 선행 컴파일러 (Ahead-of-Time Compiler, AOTC)에서는 수행시간 전에 미리 번역해 두고 수행시간에 바로 실행한다. JITC는 번역하는 데 필요한 오버헤드(번역시간, 메모리, 프로세서 파워)가 실행시간의 오버헤드에 포함되기 때문에 번역자체가 효율적이어야 한다. AOTC는 이러한 번역 오버헤드 문제는 없지만 자바 언어의 특성상 dynamic class loading에 의해 실행시간 중에 외부에서 동적으로 바이트코드를 다운로드할 수 있는데 이런 바이트코드에는 AOTC를 적용할 수 없으므로 별도의 해석기나 JITC와 함께 수행되어야 한다.[8][9]

어떤 방식을 사용하든 단순하게 바이트코드를 기계어 코드로 번역하게 되면 비효율적인 코드를 얻게 되어 충분한 성능 향상 효과를 볼 수 없다. 고성능 코드로 번역하기 위해서는 역시 컴파일러 최적화 기술을 이용하여야 하며, 또한 번역된 코드 크기도 최적화 되어야 한다. 본 절에서는 JITC와 AOTC를 간략히 소개하고 이들을 위한 컴파일러 최적화 기술을 설명한다.

1. JITC

내장형 시스템은 사용되는 환경이나 하드웨어 성능이 다양하기 때문에, 그에 따라 JITC에서 사용하는 최적화의 종류도 역시 달라야 한다. 예를 들어 디지털 TV와 휴대폰의 경우에 있어서 전원 공급 방식, 프로세서 속도, 메모리 크기 등이 매우 다르며, 이로 인해 디지털 TV에서는 거의 PC와 유사한 최적화가 적용될 수 있는 반면 휴대폰은 매우 단순한 최적화만이 적용될 수 있다. 따라서, 본 절에서는 일반적인 내장형 시스템을 대상으로 하면서 고 사양과 저 사양에 대한 고려사항을 병기하도록 한다.

Register allocation : 바이트코드는 스택 머신(stack machine)을 기반으로 하는 코드로서 중간 연산 결과를 저장하는 연산 스택(operand stack)과 국부 변수들(local variables), 그리고 생성된 객체를 저장하는 힙(heap)을 이용하여 연산이 이루어 진다. 이를 레지스터 기반의 마이크로프로세서를 위해 번역할 때 가능한 많은 연산 스택 항목들과 국부 변수들을 레지스터에 할당해야 고성능을 얻을 수 있다. 이 경우 국부 변수와 연산 스택 간의 push와 pop 바이트코드들은 레지스터간의 복사 명령어로 번역되므로 레지스터를 적절히 할당하여 이러한 복사 명령어를 제거하는 것도 중요하다. 그래프 컬러링 기법은 매우 효과적이지만 번역 오버헤드가 너무 크므로 빠르면서도 효율적인 레지스터 할당 기법이 필요하다.[7]

일반적인 최적화 : 레지스터 기반으로 번역된 코드에서 JITC는 앞서 제시한 일반적인 컴파일러 최적화 기법을 대부분 적용할 수 있다. 보통 오버헤드가 적은 국부 최적화 기법이 사용되지만, 일부 고 사양 내장형 시스템에서는 알고리즘을 간소화해서 광역 최적화 기법을 사용할 수도 있다. 저 사양의 내장형 시스템에서는 중간 코드 생성 없이 바이트코드에서 바로 기계어 코드를 생성하기도 하는데, 이 경우에도 간단한 peephole optimization 이 사용된다.[6]

Inlining : 자바와 같은 객체 지향 언어에서는 프로그래머들이 작은 크기의 메소드(method)를 많이 사용해서 프로그램을 작성하는 경향이 있다. 따라서, 메소드를 호출하는 오버헤드를 감소시키기 위해서는 호출되는 메소드의 내용을 호출하는 메소드에 포함시키는 inlining기법이 매우 유효하다. 메소드 내용이 매우 짧은 경우는 inlining 이후에 오히려 머신 코드의 크기가 감소하기 때문에 일반적으로 적용이 가능하지만, 아닌 경우는 기계어 코드 크기가 커지기 때문에 조심스럽게 사용되어야 한다.

Exception check elimination : 자바에서는 잘못된 객체 접근 시에 예외가 발생하여야 한다. 대표적인 것인 객체가 null 일 때 발생하는 null pointer exception과, array 객체에서 객체 경계를 넘어 접근할 때 발생하는 array index out-of-bounds exception 이다. 객체에 접근할 때마다 이와 같은 예외를 검사하는 코드를 JITC가 생성해야 하는데 중복된 검사 코드를 제거하는 최적화가 exception check elimination이다.

Adaptive compilation : JITC가 생성하는 기계어 코드는 바이트코드에 비해 일반적으로 3~6배 정도의 크기를 가진다. 생성하는 기계어 코드의 양을 줄이기 위해 많은 JITC에서는 수행이 매우 빈번한 부분(hot spot)만을 번역하고 나머지 부분은 해석기를 이용해서 실행하는 방법을 사용하는데, 이를 적응 컴파일(adaptive compilation) 기법이라고 한다. 적응 컴파일을 위해서는 hot spot을 인식하는 작업이 중요한 데, 이를 위해 메소드의 수행 횟수나 루프 반복 횟수를 해석기 수행 시 측정한다. 상용화된 내장형 JITC로는 CDC/CLDC HOTSPOT (http://java.sun.com/j2me/), PERC(http://www.newmonics. com/), JBed (http://www.esmertec.com/) 등이 있으며 이들은 모두 적응 컴파일 기법을 사용한다. 또한 적응 컴파일 기법에서는 선택된 메소드에 대해 번역 오버헤드에 대한 걱정 없이 보다 강력한 최적화 기법을 사용할 수 있다.

2. AOTC

AOTC에서는 일반적인 컴파일러 최적화 기법을 그대로 사용할 수 있다. 그러나 많은 AOTC는 이러한 최적화 기법을 직접 구현하여 바이트코드에서 최적화된 기계어 코드로(bytecode-to-native) 번역하는 대신, 우선 바이트코드에서 C 코드로(bytecode-to-C) 번역하고 기존의 최적화 C 컴파일러를 이용하여 C 코드로부터 기계어 코드를 얻는 방식을 택하고 있다.[9] 이런 방식을 통해 AOTC의 복잡도를 줄이고 기존 컴파일러의 강력한 최적화 기술을 이용할 수 있다. 한가지 문제점은 기존의 C 컴파일러는 이 C 코드가 바이트코드로부터 번역된 C 코드라는 것을 인식하지 못하기 때문에 자바의 특성을 이용한 최적화는 할 수 없다는 것이다. 따라서 C 코드로 번역할 때 다음에 설명하는 추가의 최적화를 해주는 것이 고성능 코드를 얻기 위해 필요하다.

우선 JITC가 행하는 inlining 과 exception check elimination 등도 바이트코드에서 C 코드로의 번역시 AOTC가 수행한다.

또한 클래스 상속 관계를 이용한 최적화를 수행한다. 즉, AOTC가 적용되는 클래스의 범위는 AOTC를 할 때 정해지기 때문에 각 클래스들 간의 상속 관계를 분석하는 것이 가능하다. 이러한 상속 관계 분석을 통해서 virtual call de-virtualization이나 virtual method inlining 등과

같은 최적화를 할 수 있다. virtual call de-virtualization 은 간접 호출 방식으로 구현되는 virtual call에 대해서, 상속 관계 분석을 통해 호출할 메소드를 알수있는 경우에 직접 호출로 변환하는 최적화 방법이다. virtual call inlining 은 이러한 간접 호출 메소드에 대해 메소드 호출 대신 호출하는 부분에 직접 메소드 내용을 삽입하는 최적화를 말한다.

AOTC 에서는 JITC 에서 시도하기 힘든, 시간이 오래 걸리는 분석 방법도 시도가 가능하다. 대표적인 것이 escape analysis 이다. escape analysis 는 어떤 객체가 사용되는 범위가 어디까지인지를 분석한다. 어떤 객체가 메소드 안에서만 생성되고 사용되며 외부에서는 접근할 수 없는 것이 확인되면 이 객체는 힙이 아니라 C 스택에 생성하도록 코드를 만들 수 있다. 이러한 최적화 방법을 stack allocation of objects라고 한다. 이렇게 C 스택에 할당된 객체에 대해서 synchronized 메소드를 수행할 때, 이 객체는 다른 쓰레드(thread)에서 접근할 수 없다는 것이 보장되기 때문에 다른 쓰레드와 synchronization을 위한 경쟁 관계에 놓이지 않는다. 따라서 synchronization은 불필요하며 이를 위한 코드도 삭제할 수 있다. 이런 최적화를 synchronization elimination이라고 한다.[10]

V. 결 론

내장형 시스템에서 소프트웨어의 최적화를 위해 컴파일러 최적화 기술의 중요성은 점차 커지고 있다. 예를 들어 국내에서 DSP, MCU 같은 내장형 마이크로프로세서를 새로 개발하는 경우 그 구조에 특화된 최적화 컴파일러가 필요하다. 또한 자바를 채택하는 내장형 시스템을 구축하는 경우 자바 성능 향상을 위해서는 JITC나 AOTC와 같은 번역 기반의 자바 가속이 필요한데, 여기에도 컴파일러 최적화 기술이 요구된다.

컴파일러 최적화 기술은 오랜 기간 동안 축적된 강력한 성능 향상 도구이지만 비교적 폐쇄된 기술로서(최적화 컴파일러만을 전문적으로 다룬 저서는 1997년에 처음 출간되었음[2]) 국내 기업에서도 확실한 기술 확보가 이루어지지 못하고 있다. 또한 최적화 컴파일러는 프로그램 복잡도가 가장 높은 소프트웨어 중 하나로서 컴파일러 자체가 프로그램을 생성해 내는 프로그램이기 때문에 디버깅 작업이 까다롭고 정교한 프로그래밍 기술을 필요로 한다. 따라서 국내 내장형 소프트웨어 최적화를 위해서는 컴파일러 최적화 기술에 대한 보다 적극적인 투자와 연구 개발 및 인력 양성이 요구된다.

참고문헌

[1] Embedded System Definition, http://en.wikipedia.org/wiki/Embedded_system

[2] Steven S. Muchnick, "Advanced Compiler Design and Implementation" , Morgan Kaufmann Publishers, 1997.

[3] Jinpyo Park, Je-Hyung Lee, Soo-Mook Moon, "Register Allocation for Banked Register File", ACM Workshop on Languages, Compilers, Tools for Embedded Systems (LCTES), June 2001.

[4] Saumya K. Debray and Willian Evans, "Compiler Techniques for Code Compaction" , ACM Transactions on Programming Languages and Systems (TOPLAS), 22(2):378-415, 2000.

[5] Haris Lekatsas, Jörg Henkel and Venkata Jakkula, "Design of an One-cycle Decompression Hardware for Performance Increase in Embedded Systems", In Proc. ACM/IEEE Design Automation Conference,

pages 34-39, June 2002.

[6] G. Majunath, V. Krishnan, "A Small Hybrid JIT for Embedded Systems", ACM SIGPLAN Notices, 35(4), April 2000.

[7] Byung-Sun Yang et al., "LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation", In Proc. 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT' 99), October 1999. (LaTTe 소스코드: http://latte.snu.ac.kr)

[8] M. J. Serrano, R. Bordawekar, S. P. Midkiff, and M. Gupta, "Quicksilver: a quasi-static compiler for Java", In Proc. 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications, October 2000.

[9] WEISS, M. et al., "TurboJ: A Java Bytecode-to-Native Compiler", In Proceedings of Workshop on Languages, Compilers and Tools for Embedded Systems, 1998

[10] B. Blanchet, "Escape analysis for Java: Theory and Practice", ACM Transactions on Programming Languages and Systems, 25(6):713-775, Nov. 2003.

저자소개

이 제 형

1999년 경북대학교 전자공학과 학사
2001년 서울대학교 전기공학부 석사
2001년 – 현재 서울대학교 전기컴퓨터공학부 박사과정

저자소개

이 준 표

1998년 서울대학교 전기공학부 학사
2000년 서울대학교 전기공학부 석사
2000년 – 현재 서울대학교 전기컴퓨터공학부 박사과정

이 승 일

1998년 서울대학교 전기공학부 학사
2000년 서울대학교 전기공학부 석사
2000년 – 현재 서울대학교 전기컴퓨터공학부 박사과정

문 수 묵

1987년 서울대학교 컴퓨터공학과 학사
1990년 University of Maryland Computer Science 석사
1993년 University of Maryland Computer Science 박사
1993년 – 1994년 Hewlett-Packard Co., 소프트웨어엔지니어
1997년 IBM T. J. Watson 연구소 방문교수
2002년 – 2003년 Sun Microsystems 방문교수
1994년 – 현 재 서울대학교 전기컴퓨터공학부 부교수
주관심분야 컴파일러 최적화, 자바 가속, 명령어 수준의 병렬처리