자바 적시 컴파일에서의 조건 수행을 이용한 비어있는 포인터의 조기검사

(Early Null Pointer Check using Predication in Java™ Just-In-Time Compilation)

이 상 규 † 최 형 규 †† 문 수 묵 †††
(Sanggyu Lee) (Hyug-Kyu Choi) (Soo-Mook Moon)

요 약 자바에서는 어떤 객체에 접근하는 작업을 할 때마다 항상 해당 객체에 대한 레퍼런스가 널(Null)인지 여부를 먼저 검사하도록 규정하고 있다. 자바 언어는 객체 중심 언어이기 때문에 객체 접근이 빈번하며 이러한 널 포인터 검사는 자바 프로그램의 성능을 크게 저하시킬 수 있다. 이러한 성능 저하를 줄이기 위한 방법으로 불필요하게 반복되는 널 포인터 검사를 제거하는 기법이 사용되고 있다. 즉, 적시(Just-in-Time, JIT) 컴파일러가 사용되는 자바 수행 환경에서 코드 분석을 통해 불필요한 널 포인터 검사 코드를 제거하는 최적화를 한다.

본 논문은 JIT 컴파일러 수행 환경에서 조건 수행(predication)이라는 최근 마이크로프로세서의 특징을 이용하여, 기존의 최적화로는 제거할 수 없는 널 포인터 검사 코드를 추가로 없애는 방법을 제시한다. 일반적으로 널 포인터 검사 코드는 비교와 분기 두 명령어로 이루어져 있는데, 그 중에 비교 명령어를 객체를 사용하기 전에 수행하는 것이 아니라 객체가 정의된 직후에 미리 수행하도록 함으로써 널 포인터 검사를 위해 수행되는 총 비교 명령어 수를 줄이는 것이 이 방법의 주된 내용이다. 실험 결과 기존의 널 포인터 제거로 최적화된 코드에 비해 수행된 비교 명령어의 수는 SPECjvm98에서 평균 3.21% 줄었고, 생성된 비교 명령어 수는 1.98% 줄었다. 이는 인텔 IA-64 아이테니엄(Itanium) 컴퓨터에서 평균 0.32%의 성능 향상을 가져왔다.

키워드 : 자바 가상 머신, JIT 컴파일러, 널 포인터 검사, 조건 수행

Abstract Java specification states that all accesses to an object must be checked at runtime if object refers to null. Since Java is an object-oriented language, object accesses are frequent enough to make null pointer checks affect the performance significantly. In order to reduce the performance degradation, there have been attempts to remove redundant null pointer checks. For example, in a Java environment where a just-in-time (JIT) compiler is used, the JIT compiler removes redundant null pointer check code via code analysis.

This paper proposes a technique to remove additional null pointer check code that could not be removed by previous JIT compilation technqiues, via early null pointer check using an architectural feature called predication. Generally, null pointe check code consists of two instructions: a compare and a branch. Our idea is moving the compare instruction that is usually located just before an use of an object, to the point right after the object is defined so that the total number of compare instructions is reduced. This results in reduction of dynamic and static compare instructions by 3.21% and 1.98%, respectively, in SPECjvm98 bechmarks, compared to the code that has already been optimized by previous null pointer check elimination techniques. Its performance impact on an Itanium machine is an improvement of 0.32%.

Key words : Java virtual machine, Just-in-Time compiler, null pointer check elimination, predication

· 본 연구는 Intel Corporation과의 연구과제와 HP-Intel의 IPF University Grant Program에 의해 지원되었음

† 비 회 원 : 삼성전자 S/W센터
christmaswish@hanmail.net
†† 비 회 원 : 서울대학교 전기공학부
hectoct@altair.snu.ac.kr
††† 종신회원 : 서울대학교 전기공학부 교수
smoon@altair.snu.ac.kr

논문접수 : 2005년 2월 7일
심사완료 : 2005년 5월 30일

1. 서 론

자바의 특징 중 하나는 프로그램의 신뢰성을 높인 것이다[1]. 프로그램이 문제없이 수행되게 하기 위해서, 일차적으로는 컴파일 시간에 검사한다. 예를 들어 초기화되지 않은 변수를 사용하는 프로그램은 컴파일 되지 않는다. 이차적으로는 수행 중에 검사를 한다. 그 중 하나가 본 논문의 주제인 널 포인터 검사이다. 자바 언어는 객체 변수 접근, 메소드 호출, 배열 접근 등 객체를 이용하는 연산을 수행하기 전에 해당 객체가 유효한지를 항상 검사하도록 한다. 만약 해당 객체가 널[1]일 경우에는 NullPointerException이라는 예외를 발생시킨다[2].

객체 중심 언어인 자바의 특성상 널 포인터 검사 코드는 빈번하게 나오고, 그 결과 성능을 저하시킨다. 따라서 널 포인터 검사로 인한 성능 저하를 최소화하기 위한 방법이 연구되어 왔다.

본 논문에서는 자바 적시(Just-In-Time, JIT) 컴파일 환경에서 조건 수행(Predication)이라는 최근 프로세서의 특징을 이용하여, 널 포인터 검사의 부하를 줄이는 방법을 제시한다. JIT 컴파일러는 널 포인터 검사가 필요한 부분에 객체가 널인지 비교하는 명령과 널일 경우에 예외를 발생하는 코드로 분기하는 명령을 삽입한다. 삽입된 명령어는 다음과 같은 방식으로 나쁜 영향을 준다. 첫째, 동적으로 봤을 때는 수행될 때마다 연산 처리기(ALU) 자원을 낭비하게 된다. 둘째, 정적으로 봤을 때는 컴파일된 코드의 크기를 크게 하며 명령어 캐쉬에 영향을 주게 된다. 본 논문은 널 포인터 검사에 필요한 비교 명령과 분기 명령 중에 비교 명령을 객체 사용 직전이 아닌 객체 정의 직후에 삽입하는 조기 널 포인터 검사를 제안한다. 이는 객체가 사용되는 횟수가 정의되는 횟수보다 많다는 일반적인 사실에 기반을 둔다. 조기 널 포인터 검사는 조건 수행이라는 최근 프로세서의 특징을 이용한다.

본 논문의 구성은 다음과 같다. 2장에서는 두 가지 배경 지식을 제공한다. 첫째는 자바 언어가 명세하고 있는 널 포인터 검사이고, 둘째는 조건 수행이다. 3장에서는 널 포인터 검사 중 불필요한 것들을 분석해서 없애는 기존의 방법을 설명한다. 4장에서는 기존 널 포인터 검사 제거 방법의 한계를 제시하며, 이 한계를 극복하기 위한 조기 널 포인터 검사라는 방법을 제시한다. 5장에서는 실험 결과를 보이고, 마지막 6장에서 결론을 도출한다.

2. 배경 지식

2.1 널 포인터 검사

자바 가상 머신[2,3]은 객체에 접근 하는 모든 명령들(예를 들어 객체 변수 접근, 메소드 호출, 배열 접근, …)을 수행하기 전에 해당 객체가 널인지 여부를 항상 검사하여야 한다고 규정하고 있다.

(a) 자바 바이트코드	(b) 의사코드

getfield (foo, x)	if (foo == null)
	  throw NullPointerException
	else
	  read foo.x
	end if

그림 1 널 포인터 검사가 일어나는 예

그림 1(a) getfield(foo, x)는 foo라는 객체의 x라는 필드 값을 읽어 오는 명령이다. 자바 가상 머신은 위 작업을 수행하기 위해 foo 객체에 접근하기 전에 foo가 널인지 여부를 검사한다. 만약 foo가 널일 경우에는 널 포인터 예외(NullPointerException)를 발생시키고, 널이 아니라는 것이 확인된 후에야 원래 의도했던 작업인 foo 객체의 x 필드를 읽어 온다. 그림 1(b)는 이런 작업을 의사코드로 나타낸 것이다. 인터프리터 방식으로 처리되는 가상 머신에서는 인터프리터가 getfield를 만날 때마다 객체의 널 여부를 검사하며, JIT 방식으로 처리되는 가상 머신에서는 JIT 컴파일러가 객체의 널 여부를 검사하는 기계어 코드를 삽입함으로써 위의 의사 코드를 처리한다.

2.2 조건 수행(Predication)

조건 수행은 원래 필요로 하는 입력들 외에 서술 레지스터(predicate register)를 추가적으로 갖는다. 서술 레지스터가 참을 의미하는 1을 가지면 그 명령은 수행이 되고, 거짓을 의미하는 0을 가지면 그 명령은 NOP (no operation)으로 해석이 되어 아무런 일도 하지 않는다.

조건 수행은 아키텍처마다 구현 정도가 다르다. SPARC III와 같은 경우는 조건 수행 가능한 명령어가 분기 명령어와 mov 명령어뿐이다[4]. PA-RISC의 경우는 모든 분기 명령어와 대부분의 계산 명령어가 조건 수행 가능하다[5]. ARM 프로세서의 경우는 모든 명령어가 조건 수행 가능하다[6]. 조건 수행을 가장 자유롭게 사용할 수 있는 프로세서는 IA-64 아이테니엄이다. 몇몇 특수한 명령어를 제외하고 모든 명령어가 조건 수행 가능하며, 서술 레지스터를 64개를 지원한다[7]. 아이테니엄은 조건 수행을 보다 적극적으로 이용할 수 있는 환경을 제공한다.

조건 수행은 프로그램 중에 있는 분기문을 없애고, 흐름 그래프를 단순화시키는 최적화에 주로 이용된다. 분

1) 유효하지 않은 주소를 나타낼 때 사용되는 값으로 전통적으로 0을 이용한다.

기문을 갖고 있는 일련의 명령어들을 조건 수행 명령어들로 바꾸어서 분기문을 없애는 변환을 if-변환[8]이라고 한다. 분기문이 있을 경우에는 분기 명령어 뒤의 명령어들은 그 앞의 분기문의 결과에 따라서 수행될 수도 있고 수행되지 않을 수도 있기 때문에 명령 수행 중에 미리 명령어를 가져오려는 프로세서나, 미리 코드를 스케줄 하려는 스케줄러 모두에게 악영향을 준다. 조건 수행을 이용하여 분기문을 없애는 예가 그림 2에 있다. 그림 2(a)와 같이 if 문을 포함하고 있는 코드는 일반적으로 그림 2(b)와 같이 번역된다. r1이 0일 경우에 아래의 두 명령을 건너뛰기 위한 분기 명령어가 있어야 한다. 하지만 조건 수행을 이용하면 그림 2(c)와 같은 분기문이 없는 코드를 생성할 수 있다. 제일 앞의 비교문은 r1이 0과 같은지 비교한 후 그 결과를 서술 레지스터 p1에 저장한다. p1 결과는 뒤의 add와 load 명령어의 서술 레지스터로 이용된다. 밑의 두 명령어는 앞 비교 결과가 참이었을 때에만 수행되고, 앞 비교 결과가 거짓이었을 때는 실행 도중에 NOP으로 해석되어 아무 일도 하지 않는다. NOP 자체도 실행 시간을 차지하기 때문에 이러한 if-변환은 분기문 제거에 의한 효과가 NOP으로 지나치게 되면서 잃게 되는 것보다 높을 때만 적용돼야 하며, 그래서 단순한 if 문을 가지고 있을 때에만 적용하는 것이 일반적이다.

```
if (r1) {
  add r2 = r3, r4
  load r6 = r[5]
}
```
(a) 의사 코드

```
cmp.ne p1 = r1, 0
(p1) br end_if
r2 = r3, r4
load r6 = [r5]
end_if:
```
(b) if-변환 전

```
cmp.eq p1 = r1, 0
(p1) add r2 = r3, r4
(p1) load r6 = [r5]
```
(c) if-변환 후

그림 2 조건 수행을 이용하여 분기문 없애기

3. 불필요한 널 포인터 검사 없애기

널 포인터 체크로 인한 성능이 나빠지는 것을 피하기 위해 기존에 사용되던 방법 중 하나가 불필요한 널 포인터 체크 제거이다.

3.1 불필요한 널 포인터 검사 없애는 기존의 방법

널 포인터 검사로 인한 부하를 극복하기 위해서 사용되는 방법 중에 하나가 불필요한 널 포인터 검사 제거이다. JIT방식으로 동작하는 가상 머신이 코드를 생성할 때에 무조건 널 포인터 검사 코드를 만드는 것이 아니고, 아래와 같이 널이 아니라는 것을 보장받는 경우에는 널 포인터 검사 코드를 삽입하지 않는다[9].

- 이미 널 포인터 검사를 통과한 변수가 다시 널 포인터 검사되는 시점까지 값이 변하지 않았으면 계속 널이 아니다. 해당 변수가 널이면 앞선 널 포인터 검사에서 NullPointerException 예외가 발생하면서 예외처리 코드로 분기되기 때문에 널 포인터 검사 코드 밑으로 제어가 왔다는 것은 해당 변수가 널이 아니라는 것을 보장하기 때문이다.
- 바이트 코드의 특성상 메모리를 할당하는 바이트 코드 명령어(new, anewarray, multianewarray, …)의 리턴 값은 항상 널이 아니다. 메모리 할당에 실패했다면 자바 가상 머신은 OutOfMemoryException이라는 예외를 던지도록 되어 있기 때문이다.
- 객체가 널인지 여부를 비교한 후 분기하는 ifnull / ifnonnull 바이트 코드 뒤의 2개의 흐름 중에 한쪽에서는 위의 바이트 코드로 검사된 변수가 널이 아님을 알 수 있다. 예를 들어 ifnonnull의 경우는 비교 결과가 참일 경우에 택하는 경로에서는 비교되었던 변수가 널이 아니라는 보장을 받는다.
- 객체 메소드의 시작에서 this 객체는 널이 아니다. 해당 객체가 널이었으면 메소드 호출 전의 널 포인터 검사에서 NullPointerException 예외가 이미 발생된다.

위와 같은 경우들에 대해서 불필요한 널 포인터 검사 코드를 생성하지 않으면 많은 수의 널 포인터 검사가 사라진다. 뒤의 5장에서 설명하고 있는 동일한 실험 환경에서 SPECjvm98[10] 벤치마크에 불필요한 널 포인터 검사를 적용한 결과가 표 1에 있다. 생성된 널 포인터 검사 수는 JIT 컴파일 되면서 생성된 코드 안에 있는 널 포인터 검사 수를 의미하고, 수행된 널 포인터 검사

표 1 기존의 널 포인터 검사 제거 기법을 통해 제거된 널 검사 수

	생성된 널 포인터 검사 수			수행된 널 포인터 검사 수		
	제거 전	제거 후	제거된 비율 (%)	제거 전	제거 후	제거된 비율 (%)
201_compress	3,646	631	82.69	3,208,133,589	1,195,263,793	62.74
202_jess	6,578	1,651	74.90	500,586,042	185,017,019	63.04
209_db	3,803	713	81.25	899,006,829	429,828,448	52.19
213_javac	12,084	3,067	74.62	677,451,932	163,497,341	75.87
222_mpegaudio	10,244	1,468	85.67	2,689,247,482	1,136,075,675	57.75
227_mtrt	5,286	1,198	77.34	646,274,426	205,887,260	68.14
228_jack	8,332	2,046	75.44	265,464,655	73,965,026	72.14
기하평균			78.74			64.10

수는 SPECjvm98이 수행되면서 실제로 수행된 널 포인터 검사 수를 의미한다. 생성된 널 포인터 검사 수는 78.74%, 수행된 널 포인터 검사 수는 64.1% 줄어든다.

3.2 기존의 불필요한 널 포인터 검사 제거 기법의 한계

앞에서 설명한 방법으로 불필요한 널 포인터 검사가 많이 제거되지만, 그 방법으로는 해결되지 않는 한계가 몇 가지 있다[11].

첫 번째 예는 루프(loop) 안에서 처음으로 객체 이용이 있는 경우이다. 그림 3(a)의 자바 코드를 중간 단계 코드로 바꾸게 될 경우 그림 3(b)와 같이 루프 안에 널 포인터 검사가 남게 된다. 왜냐하면 객체 x가 루프 안에서 이용되기 전에 한 번도 널 포인터 검사가 된 적이 없기 때문이다. 루프 안에 남게 된 널 포인터 검사는 한 번이 아니라 루프의 반복횟수만큼 수행되기 때문에 성능에 나쁜 영향을 주게 된다.

```
void foo(X x)
{  do {
     x.func();
   } while (some condition);
}
```

(a) 자바 코드

```
void foo(X x)
{  do {
     nullcheck x
     x.func();
   } while (some condition);
}
```

(b) 중간 단계 코드

그림 3 기존 포인터 검사의 한계의 예 1

기존의 불필요한 널 포인터 검사 제거가 갖는 한계의 두 번째 예는 부분적으로 불필요한 널 포인터 검사이다. 그림 4와 같은 흐름 그래프가 있다고 하자.

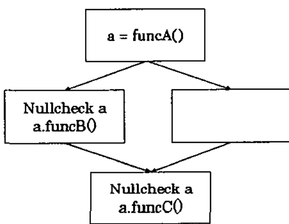


그림 4 기존 널 포인터 검사의 한계의 예 2

왼편 경로를 통해 왔을 경우에는 밑의 접합점(join point)에 있는 널 포인터 검사는 불필요한 것이다. 왼편 경로로 올 경우에는 널 포인터 검사를 한 번 더 하게 된다. 그렇다고 밑 접합점에 있는 널 포인터 검사를 제거할 수는 없다. 왜냐하면 오른쪽 경로로 왔을 경우에는 널 포인터 검사가 되지 않은 상태로 오기 때문이다. 접합점에 있는 널 포인터 검사는 왼편 경로에서 봤을 때는 불필요한 것이지만, 오른편 경로에서 봤을 때 필요하기 때문에 제거되지 않는다.

위와 같이 남아 있는 널 포인터 검사의 부하를 줄이기 위한 최적화 방법들은 있지만 최적화 부담이 클 수 있다. 루프 불변 코드 이동(Loop Invariant Code Motion)[12,13] 최적화 기법을 적용하면 첫째 예의 문제가 해결 가능하고, 부분적으로 불필요한 코드 없애기(PRE: Partial Redundancy Elimination)[13,14]를 이용한다면 첫째, 둘째 문제 모두 해결 가능하다. 하지만 첫째로 위의 두 방법은 컴파일 시간이 실행 시간에 포함되는 Just-In-Time 컴파일러의 특성상 자주 수행되지 않은 메소드들에서는 최적화에 들인 시간 만큼의 이득을 못 볼 수도 있다. 그리고 둘째로 자바의 정확한 예외(Precise exception)[3] 처리를 지키기 위해서 코드 이동이 불가능한 경우도 있다. 정확한 예외 처리란 예외가 발생하기 전에 있는 명령어의 모든 효과는 눈에 보여야 하며, 예외가 발생한 후에 있는 명령어의 모든 효과는 눈에 보이지 말아야 한다는 것이다. 따라서 최적화 시간이 얼마 들지 않는 가벼운 방식으로 위의 문제를 해결하기 위한 새로운 방법이 필요하다.

4. 조기 널 포인터 검사

4.1 조기 널 포인터 검사

3장에서 예를 들었던 한계를 극복하기 위해서 본 논문에서는 조기 널 포인터 검사라는 방법을 제시한다. 조기 널 포인터 검사란 변수의 사용 직전에서 수행하던 널 포인터 검사를 변수의 정의 직후로 옮겨서 미리 계산하는 기법을 말한다.

조건 수행(Predication)을 이용해서 널 포인터 검사를 다음과 같이 구현할 수 있다.

```
p1 = cmp.eq x, 0
  --------------------------- (1) (비교 부분)
(p1) call throw_null_pointer_exception
                  -------------- (2) (분기 부분)
```

조기 널 포인터 검사는 널 포인터 검사 중 비교 부분을 변수의 사용 직전으로부터 변수의 정의 직후로 옮겨 놓는다. 분기 부분은 변수의 사용 직전 위치에 그대로 남게 된다. 이것은 일반적으로 변수의 사용이 변수의 정의보다 더 빈번하게 일어난다는 사실에 기반을 두고 있다. 매번 변수가 이용될 때마다 검사가 되던 것을 변수가 정의되자마자 미리 비교를 해 놓음으로써 실제로 수행되는 비교 명령어의 수를 줄일 수 있게 된다. 미리 수행한 비교 명령의 결과를 저장하기 위해 각각의 지역변수마다 1개씩 미리 할당한 서술 레지스터를 이용한다. 조건 수행을 가장 잘 지원하는 아이테니엄은 총 64개의 서술 레지스터를 갖고 있기 때문에 각각의 지역 변수마다 전용으로 서술 레지스터를 할당할 수 있다.

4.2 조기 널 포인터 검사 알고리즘

본 절에서는 조기 널 포인터 검사 알고리즘을 설명한다. 조기 널 포인터 검사를 적용한 JIT 컴파일러는 그림 5와 같은 단계를 거치며, 조기 널 포인터 검사는 상위 중간 코드 최적화 단계에서 이루어진다.

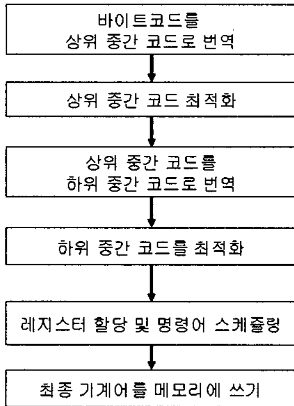


그림 5 JIT 컴파일 단계

상위 중간 코드(HIR: high-level instruction representation)는 자바 바이트 코드와 유사한 명령어로 표현되며, 하위 중간 코드(LIR: low-level instruction presentation)는 기계어 명령어와 1:1로 매칭이 되도록 표현된다. 상위 중간 코드 명령어 세트는 자바 바이트 코드에서는 보이지 않던 명령어들도 포함한다. 예를 들어 정수 배열 x의 i번째 원소를 읽어 오는 iaload (x, i)라는 바이트코드가 상위 중간 코드로 번역된 결과는 그림 6과 같다. 조기 널 포인터 검사는 이러한 상위 중간 코드를 입력으로 받는다.

그림 6 바이트 코드의 상위 중간 코드로의 번역 예

조기 널 포인터 검사는 기존의 널 포인터 검사를 기반으로 추가적인 일을 한다. 기존의 널 포인터 검사는 데이타 흐름 방정식을 반복 알고리즘(iterative algorithm).[12,13]을 푸는 방식으로 수행한다. 그 다음 조기 널 포인터 검사를 위해 각 지역 변수에 대해 정의된 횟수(numDefine)과 널 포인터 검사 횟수(numNC) 등의 부가정보를 추가적으로 모은다. 수집된 정보를 이용해서 정의된 회수가 1이고 널 검사 회수가 1이상인 널 포인터 검사를 조기 널 포인터 검사로 바꾼다. 조기 널 포인터 검사를 위해 추가된 부분만 간략하게 알고리즘을 서술하면 다음과 같다.

```
procedure fg_earlyNullCheck (fg, lvInfo)
    // fg: flowgraph
    // lvInfo: array for local variable information

    for i = 0 while (i < fg's max local) do
        lvInfo[i].defineNum = 0
            // i번째 지역변수 정의 횟수
        lvInfo[i].numNC     = 0
            // i번째 지역변수 널 포인터 검사 횟수
        lvInfo[i].defineInstr = NULL
            // i번째 지역변수 정의 명령어 위치
        lvInfo[i].predicate = not assigned
            // i번째 지역변수 조기 널 검사 결과 저장할
              서술레지스터
        i = i + 1
    end for

    for each basic block bb in fg do
        bb_earlyNullCheck(bb, lvInfo)
    end for

    // 지역변수 정의 뒤에 비교명령어 삽입하기
    for i = 0 while (i < fg's max local) do
        if (lvInfo[i].numNC > 0) and (lvInfo[i].numDefine
           = 1) then
            allocate a predicate register pr for local
            variable #i
            lvInfo[i].predicate = pr
            insert compare instruction after
            lvInfo[i].defineInstr
            i = i + 1
        end if
    end for

    // 위에서 조기 널 포인터 검사 하기로 결정된 지역
       변수에 대한
    // 널 포인터 검사를 조기 널 포인터 검사라고 기록
       하기
    for each basic block bb in fg do
        for (each instr from the first to the last
             instruction in bb)
            if (instr is nullcheck) then
                if (src is local) and (lvInfo[localNo].
                    predicate is assigned) then
                     mark nullcheck as early nullcheck
                     instr.predicate =
                     lvInfo[localNo].predicate
                     // 조기 널 포인터 검사로 마크된 명령은
                     // 이 후 코드 생성(emit) 단계에서
                        비교 명령어 없이 분기 명령어만
                        생성한다.
                end if
            end if
        end for
    end for

end of fg_earlyNullCheck;

procedure bb_earlyNullCheck (bb, lvInfo)
    // bb: basic block
    // lvInfo: array for local variable information
    for (each instr from the first to the last instruction
```

```
      in bb)
      if (instr is copy and operand type is reference)
        then
          if (dst is local variable) then
              localNo = dst's local variable number
              if (lvInfo[localNo].defineInstr = NULL)
                  then // first definition
                    lvInfo[localNo].numDefine = 1
                    lvInfo[localNo].defineInstr = instr
              else     // 이미 정의 명령어가 있다.
                       // 즉, 이 지역 변수에 2개 이상
                          의 정의가 있다.
                       // 정의 횟수만 증가시킨다.
                    lvInfo[localNo].numDefine++
              end if
          end if
      else if (instr is nullcheck) then
          if (src is local variable) then
              localNo = src's local variable number
              lvInfo[localNo].numNC++
          end if
  end for
end of bb_earlyNullCheck;
```

4.3 조기 널 포인터 검사 적용 예

조기 널 포인터 검사 기법으로 앞 장에서 설명한 기존 널 포인터 검사 제거 기법이 갖고 있던 한계가 어떻게 해결이 되는지 예를 보면서 살펴보도록 한다.

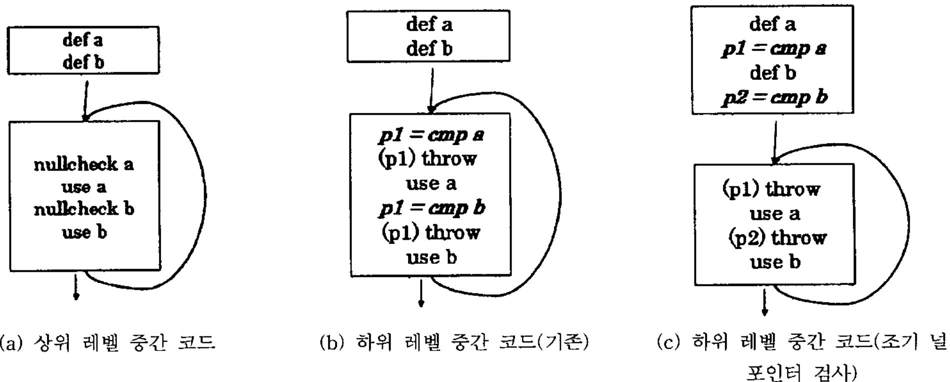
앞 장에서 예로 들었던 루프 안에 처음으로 객체 이용이 있는 경우에 조기 널 포인터 검사를 적용한 예가 그림 7에 있다. 루프 안에 널 포인터 검사가 남는 경우는 실험 벤치 마크였던 SPECjvm98에서 실제로 여러 번 일어난다. 예를 들어 _202_jess에서 빈번히 수행되는 메소드 중 하나인 Token.data_equals()에서는 함수의 인자로 넘어온 객체가 루프 안에서 처음으로 사용되기 때문에 루프 안에 널 포인터 검사가 남게 된다.

기존의 널 포인터 검사를 이용할 경우에는 그림 7 (a)와 같이 루프 안에 남아 있는 널 포인터 검사가 그림 7(b)와 같이 컴파일된다. 그 결과 루프 안에 비교 명령어와 분기 명령어가 남아 있게 되고, 루프가 수행될 때마다 비교 명령이 계속 반복된다.

조기 널 포인터 검사를 적용할 경우에는 그림 7(c)와 같이 객체 사용 직전에 있던 널 포인터 검사 코드 중 비교 명령어들이 객체 정의 직후로 옮겨지면서 자연스럽게 비교 명령어가 루프 바깥으로 옮겨 나오게 된다. 그림 7(c)에 있는 서술 레지스터 p1과 p2는 변수 a와 변수 b를 위해 전용으로 할당되었다. 서술 레지스터 p1과 p2는 변수 a와 변수 b가 널인지 여부를 알려주는 지시자 역할을 하게 된다. 첫 번째 예에 조기 널 포인터 검사를 적용하면 컴파일 후 생성된 비교 명령어 수에는 변화가 없지만, 수행되는 비교 명령어 수는 기존에는 루프 반복 횟수만큼에서 한 번으로 줄어들게 된다.

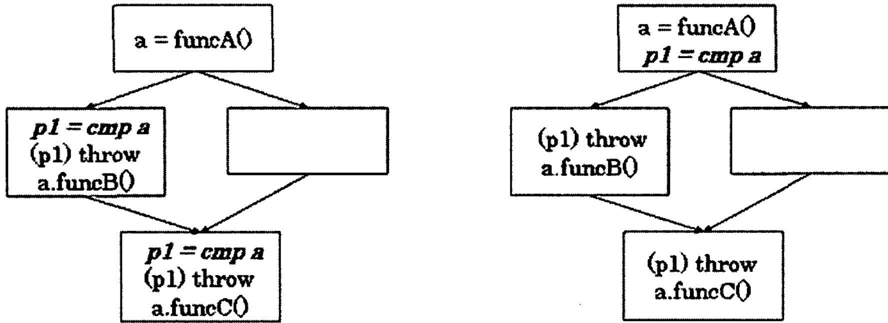
앞 장에서 기존 널 포인터 검사의 한계의 두 번째 예인 부분적으로 불필요한 널 포인터 검사에 조기 널 포인터 검사를 적용한 예가 그림 8에 있다. 그림 8(a)는 기존의 널 포인터 검사의 결과이고, 그림 8(b)는 조기 널 포인터 검사를 적용한 결과이다. 예전에는 왼편 경로를 통해서 수행될 경우에 두 번의 비교가 수행되었지만 이제는 왼편 경로를 통해서 수행되어도 한 번만 비교가 수행된다. 오른편 경로를 통해서 수행될 때에 수행되는 횟수는 한 번으로 양쪽에 차이가 없다. 이 경우에는 수행되는 비교 명령어의 숫자 뿐 아니라 생성된 비교 명령어 수도 줄어든다는 것이다. 그림 8(b)는 그림 8(a)에 비해서 생성된 비교 명령어도 하나 줄었다.

하지만 조기 널 포인터 검사가 항상 좋은 것은 아니다. 정의가 사용보다 많은 예가 그림 9에 있다. 이런 경우는 SPECjvm98 벤치마크의 213_javac의 메소드 Parser.parseExpression()과 같은 경우에서 발생한다. 위의 함수는 switch-case 문을 포함하고 있고, 그림 9 (a)와 같이 각각의 case 문들에 객체 정의가 있다. 이런 경우에 조기 널 포인터 검사 기법을 적용하면 각각의



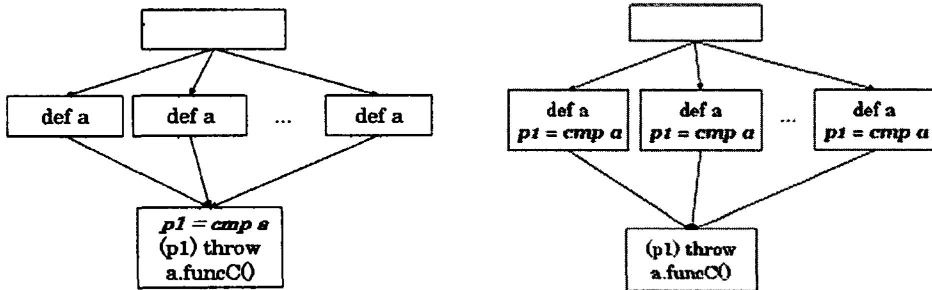
(a) 상위 레벨 중간 코드 (b) 하위 레벨 중간 코드(기존) (c) 하위 레벨 중간 코드(조기 널 포인터 검사)

그림 7 조기 널 포인터 검사 예 1 - 루프 안에서 남은 널 포인터 검사



(a) 하위 레벨 중간 코드 (기존) (b) 하위 레벨 중간 코드 (조기 널 포인터 검사)

그림 8 조기 널 포인터 검사 예 2 - 부분적으로 불필요한 널 포인터 검사



(a) 하위 레벨 중간 코드 (기존) (b) 하위 레벨 중간 코드 (조기 널 포인터 검사)

그림 9 조기 널 포인터 검사 예 3 - 적용하면 손해 보는 예

case 문의 객체 정의 문 뒤에 비교 명령어를 삽입하기 때문에 그림 9(b)와 같이 생성된 비교 명령어 수가 늘어나게 된다. 따라서 이런 경우는 조기 널 포인터 검사를 적용하지 않는다.

이와 같이 정의가 한 번만 있는 경우에만 조기 널 포인터 검사를 적용하게 되면, 정의는 루프 바깥에서 여러 곳에 흩어져 있고, 사용은 루프 안에 있는 그림 10과 같은 경우에 대해서는 조기 널 포인터 검사를 적용했으면

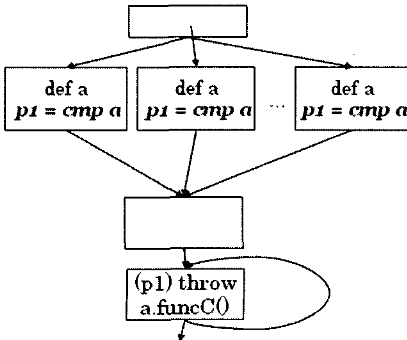


그림 10 조기 널 포인터 검사 예 - 정의 여러 번, 사용 1번

얻을 수 있었을 성능 향상을 얻지 못하게 된다. 하지만 이런 경우를 고려하기 위해서는 컴파일 부하가 있는 루프 분석이 필요로 하기 때문에 그런 경우는 포기를 한다. 조기 널 포인터 검사는 작은 컴파일 부하로 모든 메소드에 적용 가능한 방법이기 때문이다. 빈번히 수행되는 메소드는 프로파일링(Profiling) 후에 루프 불변 코드 이동, 부분적으로 불필요한 코드 이동과 같은 최적화 방법으로 해결할 수 있을 것이다.

5. 실험 결과

5.1. 실험 환경

본 논문의 실험은 IA-64 아이테니엄[7]에서 동작하는 Intel의 Open Runtime Platform 자바 가상 머신[15]에 서울대학교 마이크로 프로세서 구조 및 시스템 소프트웨어 연구실(MASS Lab)에서 개발한 VLaTTe라는 JIT 컴파일러 상에서 수행되었다[16]. VLaTTe는 나무모양 영역(tree region)[2)]이라고 불리는 영역을 단위로 스케줄링 하는 JIT 컴파일러이다. 나무모양 영역은 하나의 입구만이 존재하고 제어 접합점(join point)이 없

2) 확장 기본 블럭(Extended basic block)이라고 부르는 경우도 있다.

는 서로 연결된 기본 블럭(basic block)의 집합을 말한다. 자바 클래스 라이브러리로는 GNU의 Classpath[16]을 이용하였으며, 벤치마크로는 SPECjvm98을 이용하였다.

5.2 생성된 비교 명령어 수 변화

조기 널 포인터 검사는 기존의 널 포인터 검사가 없애지 못해서 생성되는 비교 명령어 중 기하 평균으로 1.98% 를 없앴다. 정의가 1번만 있는 지역 변수에 대해서만 조기 널 포인터 검사를 적용했기 때문에 무조건 1개만 생성된다. 따라서 사용되는 곳이 두 곳 이상이었을 경우에는 생성되는 비교 명령어 수가 줄어들게 된다. 이에 따라 불필요한 코드 제거의 효과가 나타나게 된다. 그림 11은 각각의 벤치마크에 대해서 조기 널 포인터 검사 적용 후에 생성된 비교 명령어 수의 줄어든 비율을 보여준다.

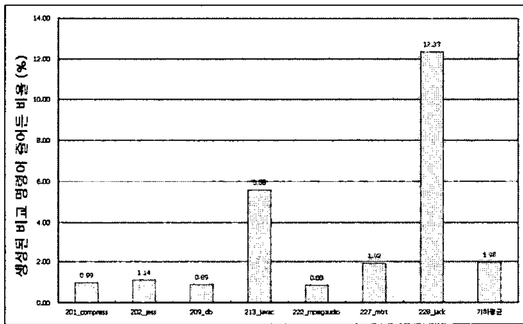


그림 11 생성된 비교 명령어 줄어든 비율

213_javac, 228_jack는 다른 벤치마크에 비해서 명령어가 많이 줄었다. 그 이유는 213_javac, 228_jack는 정의는 1번 있지만 사용은 여러 곳에 흩어져 있는 경우가 많았기 때문이다.

예를 들어 213_javac의 메소드 Instruction.write (DataOutputStream, ConstantPool)는 switch-case 문을 포함하고 있는데, case 문마다 널 포인터 검사 명령어를 갖고 있었다. 조기 널 포인터 검사 적용 후에는 비교 명령어가 해당 변수 정의 후 한 곳에서만 생성되기 때문에 20개의 비교 명령어가 1개로 줄어들었다.

5.3 수행된 비교 명령어 수 변화

널 포인터 검사를 위해 수행되는 비교 명령어 수는 기존의 널 포인터 검사만 적용했을 때에 수행되는 비교 명령어 수에 비해서 기하평균으로 3.21% 줄어들었다. 벤치마크 각각의 수행된 비교 명령어 수 변화는 그림 12에 나타나 있다. 209_db와 213_javac를 제외한 나머지 벤치마크들에 대해서 조금씩 수행된 비교 명령어 수가 줄어들었다.

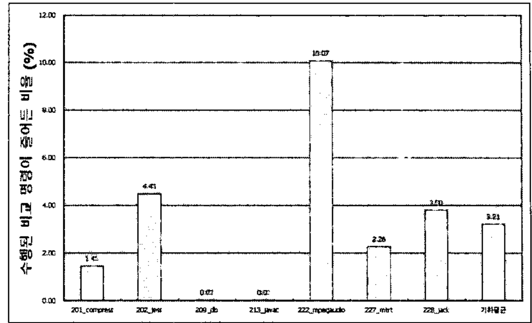


그림 12 수행된 비교 명령어 줄어든 비율

209_jess, 222_mpegaudio 등은 비교적 많이 줄었다. 그 중에 222_mpegaudio가 제일 높은 비율로 줄어들었다. 그 이유는 222_mpegaudio에서 제일 많은 시간동안 수행되는 메소드 q.l (short[], int)에서 루프 안에 있던 비교 명령어가 루프 바깥으로 빠져 나왔기 때문이다. 202_jess도 많이 수행되는 메소드들인 runTests(Token, Token, Token), data_equals(Token)에서 루프 바깥으로 비교 명령어가 빠져 나왔다.

5.3 JIT 컴파일 시간

조기 널 포인터 검사를 적용하고 난 후에 JIT 컴파일 시간의 변화는 평균 0.16% 증가했다. 각 벤치마크 별 JIT 컴파일시간 변화는 그림 13에 있다. 조기 널 포인터 검사 기법은 기존의 널 포인터 검사 기법 단계에 부가적인 정보를 조금 더 모으는 정도에 불과하기 때문에 위 기법으로 인한 부하가 거의 없다. 오히려 _213_javac는 JIT 컴파일 시간이 줄어든 것으로 나왔는데, 이는 조기 널 포인터 검사를 적용한 후에 생성된 비교 명령어 수가 줄어들어서 널 포인터 검사 제거 이후 단계인 명령어 스케줄 및 레지스터 할당 단계의 시간이 줄어들었기 때문이다. JIT 컴파일 환경에서는 JIT 컴파일 시간도 수행시간 중에 포함이 되기 때문에 JIT 컴파일 시간이 적게 걸리는 것이 중요하다. 현재 JIT 컴파일 시간은 전체 수행 시간의 7% 가량을 차지하고 있기 때문

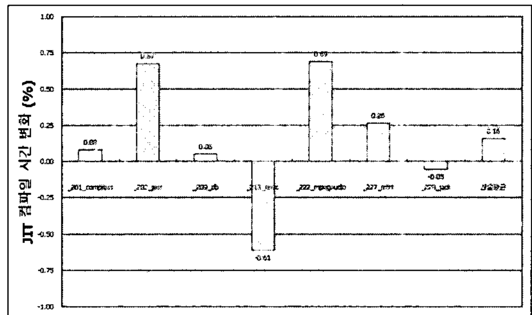


그림 13 JIT 컴파일 시간 변화

에 전체 수행 시간에 조기 널 포인터 검사 기법으로 인해 든 부하는 0.01%에 불과하다.

5.4 성능 향상

조기 널 포인터 검사 기법은 JIT 컴파일시간까지 포함한 총 수행시간을 기하평균으로 0.32% 향상시켰다. 그 각각의 성능 향상 비율은 그림 14에 나타나 있다. 성능 변화가 거의 없다고 할 수 있는 222_mpegaudio를 제외하고는 모든 벤치마크에서 고루 성능 향상이 있었다. JIT 컴파일 시간의 변화가 거의 없었기 때문에 성능 향상에서의 차이는 JIT된 코드의 실행 시간의 차이다. 이런 차이는 생성된 비교 명령어 수가 줄어든 효과와 수행된 비교 명령어가 줄어든 긍정적인 효과가 복합적으로 작용한 결과이다. 사라진 비교 명령어의 자리를 스케줄러가 얼마나 유용한 명령으로 채웠느냐에 따라서 그 성능 차이가 다르게 나타났다.

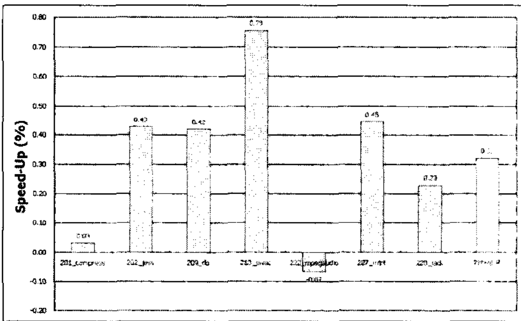


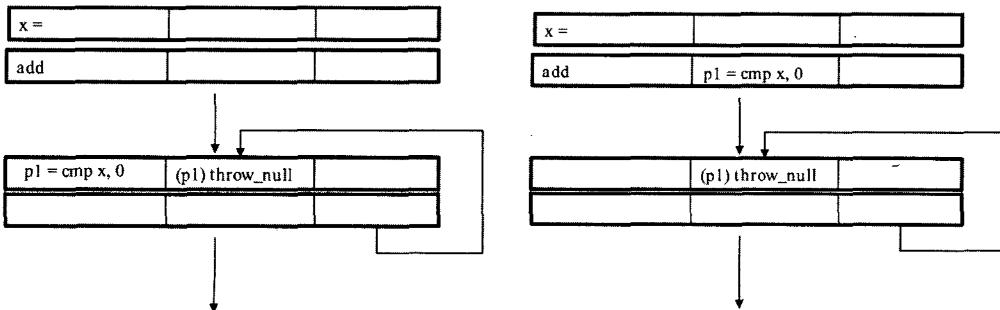
그림 14 성능 향상

222_mpegaudio는 수행된 비교 명령어 수는 많이 줄었지만 성능 향상이 없었다. 그 이유는 수행된 비교 명령어 수의 대부분을 줄여 준 메소드인 mpegaudio의 제일 빈번히 수행되는 메소드 q.l(short[], int)에서 성능 이득을 보지 못했기 때문이다. 이 메소드에 조기 널 포인터 검사를 적용하기 전의 코드는 그림 15(a)와 같다. 그림에서 각 줄은 동시에 수행 가능한 명령어들을 나타낸다. 그림 15(a)의 세 번째 줄을 보면 객체 x를 위한 널 포인터 검사의 비교 명령어와 분기 명령어가 동시에 수행되고 있다. 조기 널 포인터 검사를 적용하고 난 후의 코드인 그림 15(b)를 보면 세 번째 줄에 있던 비교 명령어는 객체 x가 정의된 첫 번째 줄 직후 즉 두 번째 줄로 옮겨졌다. 하지만 세 번째 줄에 비교 명령어가 빠진 자리는 여전히 빈 칸으로 남아 있다. 이와 같은 경우는 수행된 비교 명령어 수는 줄지만 성능상 이득은 없다. 만약 조기 널 포인터 검사 적용 후에 비교 명령어가 옮겨 가야 하는 줄이 존재하지 않는다면 한 줄을 더 증가시키는 악영향을 줄 수도 있다.

6. 결 론

자바 프로그램을 수행하면서 빈번히 발생하는 널 포인터 검사를 위해 JIT 컴파일러는 비교 명령어, 분기 명령어를 생성하게 된다. 조기 널 포인터 검사는 비교 명령어를 정의 다음으로 옮김으로 이전의 널 포인터 검사 제거 기법으로 없앨 수 없었던 경우에 대해서 비교 명령어의 생성된 명령어 수와 수행된 명령어 수를 줄인다. 실험 대상이었던 아이테니엄에서 널 포인터 검사를 위해 수행된 비교 명령어의 수를 3.21%, 기존 널 포인터 검사가 없애지 못한 비교 명령어 중 1.98%를 더 줄였다. 이로 인한 성능 향상은 0.32% 있었다.

위의 실험은 아이테니엄에서 행해졌으나, 모든 명령어에 조건부 수행을 지원하며 서술 레지스터(predicate register)를 충분히 제공하는 아키텍처라면 일반적으로 적용 가능한 기법이다. 그리고 기존의 널 포인터 검사 기법에서 추가적으로 정보만 조금 더 모으면 되기 때문에 전체 JIT 컴파일시간에 거의 변화가 없다. 따라서 어떤 메소드가 빈번히 수행되는지 모르는 상황에서도 적용할 수 있는 기법이다.



(a) 조기 널 포인터 검사 적용 전 (b) 조기 널 포인터 검사 적용 후

그림 15 수행된 비교 명령어 수가 줄어들어도 성능 이득을 보지 못하는 예

참 고 문 헌

[1] James Gosling, and Henry McGilton, The Java Language Environment. A White Paper, http://java.sun.com/docs/white/langenv/, May 1996.
[2] Tim Lindholm, and Frank Yellin, The Java Virtual Machine Specification, Second Edition, http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html, April 1999.
[3] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, The Java Language Specification, Second Edition, http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html , June 2000.
[4] Sun Microsystems, UltraSPARC III Cu User's Manual version 2.0, February 2003.
[5] PA-RISC 2.0 Instruction set architecture, Hewlett-Packard, August 1995.
[6] ARM, ARM Developer guide version 1.0, October 1999.
[7] Intel Itanium Architecture Software Developer's Manual Vol. 1: Itanium Application Architecture. http://developer.intel.com/design/itanium/manuals.htm
[8] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," Proc. 10th ACM Symp. Principles of Programming Languages, pp. 177-189, January 1983.
[9] J. Whaley, Dynamic optimization through the use of automatic runtime specialization, Massachusetts Institute of Technology, May 1999.
[10] Standard Performance Evaluation Corp, "SPEC JVM98 Benchmarks," http://www.spec.org/osg/jvm98/
[11] Motohiro Kawahito, Hideaki Komatsu Toshio, and Nakatani, "Effective Null Pointer Check Elimination Utilizing Hardware Trap," November 2000.
[12] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, Principles, Techniques and Tools," Addison-Wesley, Reading, MA, January 1986.
[13] S. S. Muchnick, "Advanced Compiler Design and Implementation," Morgan Kaufmann Publishers, San Francisco, CA, 1997.
[14] E. Morel and C. Renvoise, "Global optimization by suppression of partial redundancies," Communications of the ACM, Vol. 22, No. 2, pp. 96-103, February 1979.
[15] Intel Corporation. "Open Runtime Platform (ORP)," http://orp.sourceforge.net
[16] Suhyun Kim, Soo-Mook Moon, Kemal Ebcioglu and Erik Altman, "VLaTTe: A Java Just-in-Time Compiler for VLIW with Fast Scheduling and Register Allocation," IEICE Transactions on Information and Systems, Vol. E87-D, No. 7, pp. 1712-1720, July, 2004.
[17] GNU Classpath, http://www.gnu.org/software/classpath/classpath.html

이 상 규

1999년 2월 서울대학교 컴퓨터 공학과 학사. 2004년 2월 서울대학교 전기컴퓨터공학부 석사. 1999년 3월~2001년 7월 인포뱅크 근무. 2004년 3월~삼성전자 기술총괄 소프트웨어센터.

최 형 규

정보과학회논문지 : 소프트웨어 및 응용 제 32 권 제 3 호 참조

문 수 묵

정보과학회논문지 : 소프트웨어 및 응용 제 32 권 제 3 호 참조