임베디드 시스템을 위한 공간 효율적인 메모리 관리 기법†

삼성종합기술원 임근수

1. 서 론

임베디드 컴퓨터 시스템의 생산 단가 및 실행 시간의 소모 전력은 탑재한 메모리의 크기에 비례한다[1]. 임베디드 시스템은 프로세서, 메모리, 입력 장치, 그리고 출력 장치의 4가지 하드웨어 컴포넌트로 구성된다. 이 중에서는 프로세서와 입출력 장치는 대치할 수 없는 고유한 기능을 가지고 있다. 반면에 메모리는 간단한 저장 기능을 수행하는 셀의 조합으로 구성되어 있어서 최적화의 여지가 높다. 특히 메모리는 비교적 넓은 실리콘 면적을 차지하며 상대적으로 많은 양의 전력을 소모하기 때문에 대량 생산되는 임베디드 시스템에서는 메모리 크기 최소화가 핵심 설계 요소이다. 메모리 최적화를 위한 기법은 코드 메모리와 데이터 메모리 측면에서 이루어질 수 있다. 본 논문에서는 코드 메모리에 대해서만 설명하고 데이터 메모리의 최적화 기법에 대해서는 참고문헌 [13]로부터 확인할 수 있다.

코드 메모리의 경우 코드 컴팩션(Compaction) 기법을 활용해 프로그램 코드의 크기를 실행 가능한 형태로 줄일 수 있다[2]. 대표적으로 함수로 요약하기(Procedural Abstraction)의 경우 반복되는 명령어 조합(Sequence)을 하나의 함수로 구성하고 원 조합을 새롭게 구성한 함수에 대한 호출 명령어로 대치한다[3,4]. 공간 효율성을 보다 개선하기 위하여 유사한 명령어 조합을 명령어 재배치(instruction reordering), 레지스터 재명명(register renaming), 그리고 상대 주소의 활용 등의 기법을 통하여 동일한 명령어 조합으로 변경한 후에 이 함수로 요약하기를 적용할 수 있다. 그림 1(a)은 이의 한 예제 코드로 동일한 알파벳은 동일한 명령어 조합을 의미한다. 명령어 조합 B가 두 번 반복되기 때문에 이는 그림 1(b)에 제시된 것과 같이 함수로 요약될 수 있다. 비록 함수로 요약하기 위해서 두 개의 호출 명령어(Call)와 하나의 리턴 명령어(Ret)을 사용하지만 전체적으로는 2개의 명령어를 줄이는 효과를 얻을 수 있다.

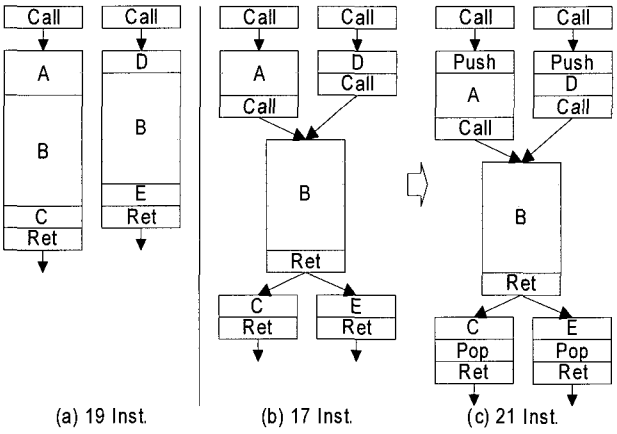

그림 1 함수로 요약하기 기법의 예제

그림 1에서 동일한 알파벳은 동일한 명령어 조합을 의미한다. 'Call'은 링크 레지스터를 스택에 저장하고 지정된 위치로 분기하는 함수 호출 명령어를 의미하고, 'Ret'는 함수로부터의 리턴 명령어를 의미하며, 'Push'는 링크 레지스터를 스택에 저장하는 명령어를 의미하며, 'Pop' 명령어는 스택에서 하나의 레지스터 크기의 값을 빼내서 링크 레지스터에 저장하는 명령어를 의미한다.

그러나 이 기법은 일반적으로 리턴 주소를 저장하고 있는 링크(link) 레지스터를 시스템 스택을 통하여 관리하기 위하여 부가적인 명령어를 활용하게 된다. 예를 들어, 그림 1(c)는 명령어 조합 ABC와 DBE가 다른 함수에 대한 호출을 하지 않는 최하단 노드인 경우이다. 최하단 노드이기 때문에 이 명령어 조합은 링크 레지스터를 스택에 저장하고 스택으로부터 복구하지 않는다. 그러나 함수로 요약하기 기법을 적용하면 B에 대한 함수 호출이 이뤄지기 때문에 ABC와 DBE는 최하단 노드가 되지 않는다. 따라서 함수로 요약하기 기법을 적용한 이후에는 ABC와 DBE의 시작과 끝 지점에서 링크 레지스터를 스택에 저장하기 위한 추가적인

† 본 논문은 저자가 RTCSA 2006에서 발표한 논문[12]의 내용을 편집 및 재구성한 것이다.

'Push'와 'Pop' 명령어를 사용해야 한다. 이로 인하여 함수로 요약하기 기법이 적용된 코드는 원 코드의 크기보다 커지게 되고 이 기법이 적용될 수 있는 범위를 줄여 궁극적으로 이의 공간 효율성을 저하하는 결과를 초래한다.

본 논문에서는 운영체제의 차원에서 소프트웨어 기법을 통하여 함수로 요약하기 기법의 공간 효율성을 개선하는 방법을 제안한다. 제안하는 코드 컴팩션 기법에서는 상술한 추가 명령어를 명시적으로 사용하지 않으면서 운영체제에 의해서 이 명령어와 동일한 기능을 자동으로 수행한다. 제안하는 기법은 세 가지 수행 방법을 제공하며 이들은 각각 서로 다른 성능상의 특성이 있다. 특히 이 중 한 기법은 ROM 기반 코드 메모리 시스템에도 적용될 수 있다. 실험 결과 프로그램의 개별 기본 블록(basic block)의 수행 빈도수를 프로파일링을 통하여 측정하여, 자주 사용되지 않는 기본 블록에 제안하는 기법을 선택적으로 적용함으로써 수행 시간에 영향을 주지 않으면서도 프로그램 코드의 크기를 크게 줄일 수 있음을 보인다.

본 논문의 구성은 다음과 같다. 2절에서는 관련 연구에 대해서 조사한다. 3절에서는 제안하는 코드 최적화 기법을 설명한다. 4절에서는 제안하는 기법의 성능을 구현과 실측 그리고 시뮬레이션을 통하여 평가한다. 그리고 5절에서는 결론을 맺는다.

2. 관련 연구

프로그램 코드의 크기를 줄이는 대표적인 방법은 압축과 컴팩션이다[2,5]. 압축은 공간 효율성은 높은 반면에 복원 시간으로 의해 수행 속도를 저하시키는 단점이 있다. 복원 시간을 단축하기 위하여 하드웨어 복원기를 사용할 수 있지만, 이는 생산 단가가 낮은 임베디드 시스템에는 적합하지 않다. 반면에 컴팩션은 코드의 크기를 실행 가능한 형태로 줄이기 때문에 복원 과정이 존재하지 않으며 이로 인한 성능 저하를 야기하지 않는다. 컴팩션은 컴파일러의 기본적인 최적화 기법인 중복되는 코드와 호출되지 않는 코드를 제거하는 기법을 포함한다[6]. 고급 컴팩션 기법으로는 로컬 팩터링(Local Factoring)과 함수로 요약하기 기법이 있다. 로컬 팩터링은 입구 또는 출구가 동일한 기본 블록들에 존재하는 이동 가능한 동일한 명령어를 이 기본 블록들의 입구 또는 출구로 이동시키는 방법이다.

함수로 요약하기 기법은 반복되는 명령어들을 하나의 함수로 구성하고 원 명령어들을 새롭게 구성한 함수에 대한 호출 명령어로 대치한다. 이의 공간 효율성을 보다 개선하기 위하여 유사한 명령어들을 명령어 재배치, 레지스터 재명명, 그리고 상대 주소의 활용 등을 통하여 동일한 명령어들로 변경한 후에 함수로 요약하기 기법을 적용할 수 있다. 또한 하나의 명령어 조합이 다른 하나의 명령어 조합의 부분 집합이 되는 경우 이의 차집합에 해당하는 명령어를 조건에 따라 실행하지 않도록 함수를 구성하는 방법을 사용하여 함수로 요약하기 기법의 공간 효율성을 개선할 수 있다[7].

하지만 이 기법은 그림 1에서 예시한 것과 같이 상당한 수의 부가 명령어를 요구할 수 있다. 그림 2(a)는 반복되는 명령어가 최하단 노드가 아닌 경우의 예제이다. 명령어 조합 BCD가 2번 반복되기 때문에 BCD는 그림 2(b)와 같이 함수로 구성될 수 있다. 그러나 명령어 조합 BD는 명령어 조합 C를 호출하기 때문에 BD는 Push와 Pop 명령어를 사용하여 링크 레지스터의 값을 보호해야 한다. 그렇지 않으면 C를 호출하는 시점에 링크 레지스터의 값이 수정되어 D를 수행한 이후에 잘못된 위치로 리턴하게 된다. 따라서 그림 2(c)와 같이 Push와 Pop 명령어를 사용하게 되면 총 5개의 명령어가 추가적으로 사용된다.

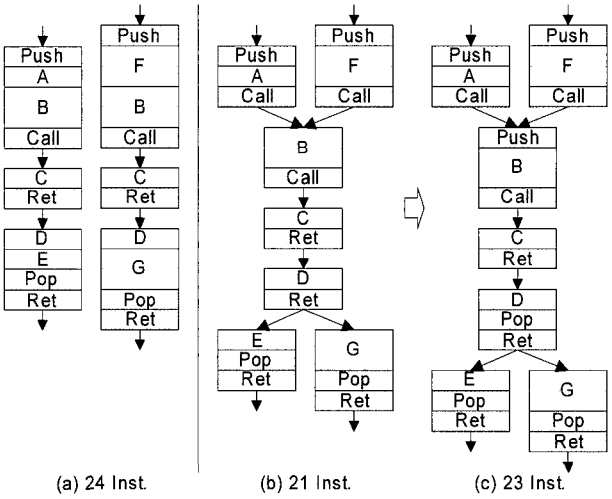

그림 2 함수로 요약하기 기법의 단점

이러한 함수로 요약하기 기법의 단점을 보완하기 위하여 2개의 새로운 명령어가 고안되었다. 이는 순차 Echo 명령어인 'echo.s offset, length'과 비트마스크(Bitmask) Echo 명령어인 'echo.b offset, mask'이다[8, 9]. 순차 Echo 명령어는 'offset'을 통해서 명시된 위치로 분기하여 'length' 개수 만큼의 명령어를 수행하고 'echo.s' 명령어 다음 명령어로 리턴한다. 비트마스크 Echo 명령어는 'offset'을 통해 명시된 위치의 명령어를 선택적으로 수행한다. 명시된 위치의 N번째 명령어는 mask & (1<<(N-1))의 값이 참인 경우에만 수행한다. Echo 명령어는 마이크로 명령어를 조합하여 프로세서 명령어로 구현될 수 있으며 이 경우 실행

시간의 추가 비용을 감출 수 있다. Echo 명령어는 사전 기반 압축 방식과 유사한 원리를 가지고 있기 때문에[5], 종래의 컴팩션 기법에 비하여 높은 공간 효율성을 보이는 특성이 있다.

하지만 현존하는 어떠한 상용 프로세서도 Echo 명령어를 지원하지 않는다. 따라서 프로세서의 재설계(Redesign and Fabrication)가 요구되며 이는 개발 비용을 증가시키는데 증가된 개발 비용은 제품의 생산 수량에 따라서 Echo 명령어를 사용해 줄이는 메모리 하드웨어의 단가보다 클 수 있다. 하드웨어의 수정이 필요하지 않은 자바 가상 기계에서 Echo 명령어를 활용한 예가 있다[9]. 비록 본 논문에서 제안하는 코드 컴팩션 기법의 개념은 Echo 명령어와 유사하나 제안하는 컴팩션 기법은 프로세서의 수정을 요구하지 않는다는 장점이 있다.

3. 코드 컴팩션 기반 메모리 관리기법

그림 3은 제안하는 코드 컴팩션 기법의 전체 구조이다. 유사한 명령어 조합은 동일한 명령어 조합으로 치환되며, 동일한 명령어 조합은 컴팩션의 대상이 된다. 컴팩션 대상의 컴팩션 방법은 해당 영역의 수행 빈도를 바탕으로 결정한다. 만약 수행 빈도가 높으면 종래의 함수로 요약하기 기법이 사용되는데 그 이유는 이 기법의 수행 속도가 제안하는 기법의 그것보다 빠르기 때문이다. 반면에 수행 빈도가 낮거나 0이면 제안하는 기법이 적용되는데 그 이유는 제안하는 기법이 보다 높은 공간 효율성을 갖기 때문이다. 세부적으로 제안하는 기법은 세 가지 수행 방법을 갖는다.

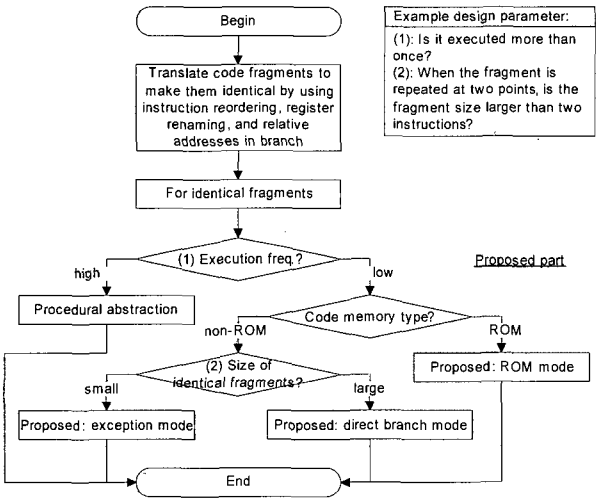

그림 3 제안하는 코드 컴팩션 기법의 전체 구조

3.1 예외 상황을 활용한 수행 방법

제안하는 기법은 기본 반복(Basic Repeat, Repeat.B)과 조건부 반복(Conditional Repeat, Repeat.C) 명령어를 사용한다. 그림 4는 Repeat.B 명령어를 그림 1(a)의 예제에 적용한 결과이다. 예외 상황을 활용한 수행 방법에서는 Repeat.B 명령어는 프로세서의 정의되지 않은 명령어(Undefined Instruction)이기 때문에, 이 명령이 수행되는 경우 예외 상황이 발생하고 프로세서는 운영체제상의 해당 예외 상황의 처리기로 분기하게 된다. 그림 4에서 I은 반복 대상 명령어 조합의 다음 명령어를 Return.B 명령어로 치환하는 부분이며, II는 치환된 명령어를 복원하는 부분이다.

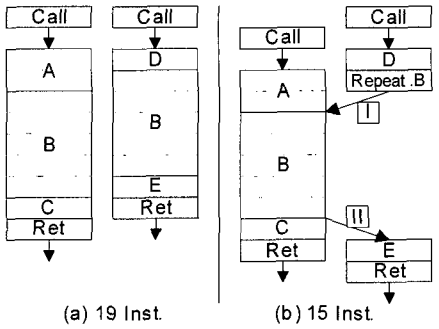

그림 4 기본 반복(Repeat.B) 명령어

예외 상황 처리기는 처리기에서 사용되는 레지스터를 저장하고 Repeat.B 명령어의 피연산자의 값을 읽어 'offset'과 'length'의 값을 계산한다. 다음으로 반복 대상 명령어 조합의 다음 명령어를 임의의 위치에 저장해 두고 이 명령어의 위치에는 정의되지 않은 명령어인 Repeat.B 명령어를 저장한다. 마지막으로 저장한 레지스터를 복구하고 반복 대상 명령어 조합으로 분기한다.

이후 프로세서는 대상 명령어 조합을 수행하고 중국에는 Return.B 명령어를 수행하게 된다. Return.B 명령어는 다른 예외 상황을 발생시키고 이 예외 상황의 처리기는 이 명령어로 치환하기 이전의 명령어를 복원하고 Repeat.B 다음 명령어로 리턴한다. 예외 상황 처리기는 프로그램 카운터와 스택 포인터 그리고 링크 레지스터를 효율적으로 관리해야 한다.

Repeat.C 명령어도 Repeat.B 명령어와 유사하게 동작하는데 다음의 면에서 차이가 있다. 그림 5는 Repeat.C 명령어를 사용해 반복 대상 명령어 조합에서 일부 명령어를 수행하지 않는 예이다. 조건 비트마스크 $11011_{(2)}$는 세 번째 명령어를 수행하지 않아야 함을 의미한다. 비트 마스트의 최하위 비트는 대상 명령어 조합의 첫 번째 명령어를 의미한다. 이 예에서 Repeat.C 명령어의 처리기는 대상 명령어 조합의 세 번째의 명령어를 'No Operation' 명령어로 치환하고, 치환한 명령어는 Return.C 명령어의 처리기에서 복원한다.

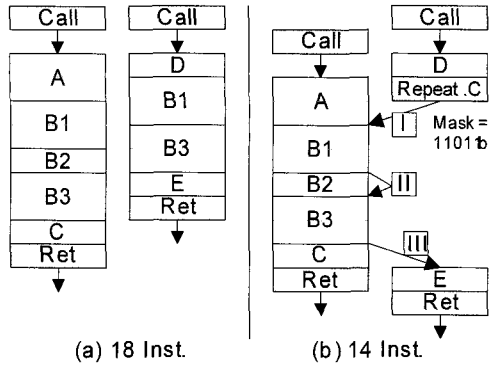

그림 5 조건부 반복(Repeat.C) 명령어

그림 5에서 흐름 I은 B2를 'No Operation'으로 변경하고 반복 대상 명령어 조합의 다음 명령어를 Return.C 명령어로 치환하며, 흐름 II는 'No Operation'으로 치환되었기 때문에 원 명령어가 수행되지 않는다. 그리고 흐름 III은 치환된 명령어를 복원한다.

3.2 분기 명령어를 사용한 수행 방법

예외 상황은 간접 분기와 파이프라인 초기화(Flushing)로 인하여 수행 시간을 지연시킨다. 이러한 단점을 극복하기 위하여, 분기 명령어를 직접 활용하는 방법을 제안한다. 이 명령어는 스택 Push 명령어와 분기 명령어('Jump')를 사용한다. Push 명령어를 대상 명령어 조합의 위치와 크기인 'offset'과 'length' 값을 저장하고 Jump 명령어를 사용해 Repeat.B 또는 Repeat.C 명령어의 처리기로 직접 분기한다. 이 수행 방법의 Repeat.B와 Repeat.C 명령어의 처리기는 'offset'과 'length' 값을 스택에서 읽어온다는 점을 제외하고는 예외 상황을 활용한 수행 방법의 그것과 동일한 방법으로 동작한다.

이 방법은 예외 상황으로 인하여 야기된 수행 시간 지연을 완화하는 장점이 있지만, 추가적으로 Push 명령어를 사용해 공간 효율성을 예외 상황을 활용한 방법에 비하여 저하시키는 단점이 있다. 따라서 예외 상황을 활용한 방법은 반복되는 명령어 조합의 크기가 작거나 수행 빈도수가 낮은 경우에 적용되는 것이 보다 적합하다.

3.3 ROM을 위한 수행 방법

이상의 두 수행 방법은 코드를 실행 시점에 수정한다. 이는 ROM 기반 시스템에서는 불가능하다. 따라서 ROM 기반 시스템을 위하여 코드를 직접 수정하지 않고 원 코드를 RAM상의 버퍼로 복사한 후에 수정한 후 RAM상에 코드를 직접 수행하는 방법을 활용한다.

그림 6은 이 수행 방법을 활용한 Repeat.C 명령어의 동작 예제이다. 조건 비트마스크가 $10001_{(2)}$인 경우 대상 명령어 조합의 첫 번째(C1)와 다섯 번째(C4) 명령어만 실행되면 된다. 따라서 Repeat.C 명령어의 처리기에서는 C1과 C4를 점선으로 표시된 RAM 버퍼로 복사하고 복사한 명령어의 다음 명령어로 Return.C 명령어를 저장한다. 이를 통하여 RAM 버퍼에 복사된 명령이 모두 수행된 이후에 Return.C 명령어를 통하여 프로세서의 제어 흐름을 Repeat.C 명령어의 다음 명령어로 넘길 수 있다.

조건부 반복 명령어의 경우 ROM 기반 시스템이 아닌 경우에도 이 기법을 활용하여 분기 명령어를 사용한 방법에 비하여 빠른 수행 성능을 제공할 수 있다. 이러한 경우는 예를 들면 분기 명령어를 사용하는 방법에서 수행되지 않아야 하는 명령어를 'No Operation'으로 변환하는데 걸리는 시간이 ROM을 위한 방법에서 수행되어야 하는 명령어를 RAM으로 복사하는 시간보다 긴 경우이다. 그림 6에서 I은 대상 명령어 조합 중에서 수행해야 하는 명령어인 C1과 C4를 RAM 버퍼로 복사하고, II는 프로세서의 제어 흐름을 Repeat.C 다음 명령어로 넘긴다.

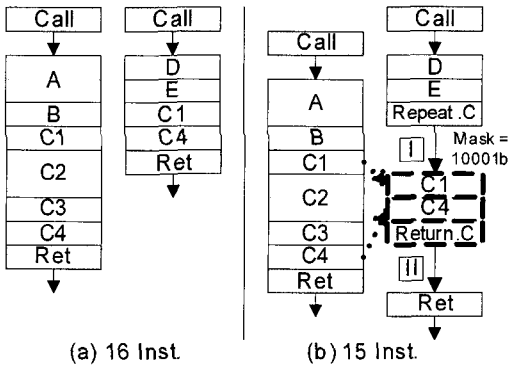

그림 6 ROM을 위한 수행 방법을 활용한 Repeat.C 명령어

4. 성능 평가

제한하는 코드 컴팩션 기법을 실시간 운영체제를 기반으로 구현하였다. 이때 하드웨어 플랫폼은 명령어와 데이터 캐시의 일관성 프로토콜을 지원하는 경우를 가정한다. Repeat와 Return 명령어의 경우 정의되지 않은 명령어를 사용하는데 ARM의 경우 이는 표 1과 같다.

표 2는 예외 상황을 활용한 수행 방법의 처리기의 의사 코드(Pseudo Code)이다. (1-7줄) 예외 상황을 발생시킨 명령어의 타입을 인식한다. (8줄) Repeat.B 명령어인 경우 저장된 레지스터를 복원한다. (9-11줄) 예외 상황을 발생시킨 명령어에서 'offset' 값을 읽으며

표 1 제안하는 명령어의 인코딩 방법 예제

	3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
0	Cond |0 1 1|x x x x x x x x x x x x x x x x x x x x x x x x|1|x x x x
1	Cond |0 1 1|0 0| Offset of target fragment |Length1|1|Length2
2	Cond |0 1 1|0 1| Unused |1| Unused
3	Cond |0 1 1|1 0| Offset of target fragment | Mask1 |1| Mask2
4	Cond |0 1 1|1 1| Unused |1| Unused

* 0: Undefined instruction space, 1: Repeat.B 명령어, 2: Return.B 명령어, 3: Repeat.C 명령어, 4: Return.C 명령어.

표 2 제안하는 반복 명령어의 처리기 의사 코드

01		.extern buffer	; Index of a custom stack
02	handler:	Push R0	; Push to stack
03		R0 <- [LR - 4]	
04		R0 <- (R0 & 0x01800000) >> 23	
05		if (R0 == 0) Jump repeat_b;	
06		else if (R0 == 1) Jump return_b;	
07		...	
08	repeat_b:	Push R1, R2	; Push to stack
09		R0 <- [LR - 4]	; Repeat.B inst.
10		R1 <- (R0 << 5) >>> 9	; Offset operand
11		R1 <- R1 x 4	; In ARM mode
12		R2 <- (R0 & 0x1e0) >> 1	
13		R2 <- R2 + (R0 & 0xf)	; Length operand
14		R0 <- (LR - 4) + R1 + R2 + 4	; Next inst.
15		buffer <- buffer + 4	; Increase the index
16		[buffer] <- LR	; Store LR
17		buffer <- buffer + 4	; Increase the index
18		[buffer] <- [R0]	; Store the original
19		[R0] <- #Return.B	; Set a Return inst.
20		R1 <- LR - 4 + R1	
21		Push R1	
22		Pop R0, R1, R2, PC	; Pop from stack
23	return_b:	R0 <- [buffer]	
24		buffer <- buffer - 4	
25		[LR - 4] <- R0	; Restore the inst.
26		R0 <- [buffer]	
27		buffer <- buffer - 4	
28		Push R0	
29		Pop R0, PC	; Return

* PC: Program counter, LR: Link register, [R]: Memory value where address = R, #: Constant, >>>: Arithmetic shift right.

이때는 음수인 경우에 대비하여 산술 쉬프트 (>>>) 명령어를 사용한다. (12-13줄) 유사하게 'length' 값을 읽는다. (14, 19줄) 이 두 값을 바탕으로 대상 명령어 조합의 다음 명령어를 수정한다. (15-18줄) 이 명령어를 수정 전에 원 명령어를 'buffer' 변수가 가리키는 스택 자료 구조에 저장한다. 이때 스택을 활용하기 때문에 제안하는 처리는 재진입 가능하다. 또한 이 스택은 태스크 의존적인 자료 구조로 문맥 교환시에 다른 문맥과 함께 교환된다. (20-22줄) 마지막으로 저장된 레지스터를 복원하고 대상 명령어 조합으로 분기한다.

제안하는 코드 컴팩션 기법의 성능평가 지표로 코드 크기와 수행 시간을 사용한다. 제안하는 기법의 코드 크기는 Echo 명령의 그것과 동일하다. 종래 실험 결과를 통하여 제안하는 기법의 성능을 분석할 수 있다 [8-11]. 분석 결과 제안하는 기법은 SPEC2000 벤치마크에 대해서 코드 크기를 30-40% 가량 줄이며 Media 벤치마크에 대해서는 10-20% 가량 줄인다. 이는 함수로 요약하기 기법에 비해서 10% 이상 큰 값이다.

반면에 제안하는 기법은 함수로 요약하기 기법에 비하여 처리기에서 소모하는 시간으로 인하여 수행 속도가 느리다는 단점이 있다. 실험 결과 코드의 수행 빈도

수를 활용하여 이러한 수행 속도 측면의 단점을 크게 완화할 수 있음을 확인하였다. 세부적으로 그림 7에 제시된 것과 같이 임베디드 시스템에서 높은 비율의 코드가 거의 수행되지 않거나 1번도 수행되지 않음을 확인하였다. 59%의 실시간 운영체제 코드가 424개의 다양한 테스트 케이스를 수행하였음에도 불구하고 1번도 수행되지 않았다. 수행되지 않은 코드는 테스크 간 통신 관리자의 소멸자 함수와 부팅 및 초기화 루틴과 같이 1회만 수행되는 코드 내부의 선택되지 않는 분기 명령(untaken-branch)의 대상 코드 등이었다. 따라서 제안하는 기법을 이와 같이 수행이 되지 않는 코드에 적용한다면 수행 시간의 지연을 경험하지 않으면서도 코드의 크기를 크게 줄일 수 있다.

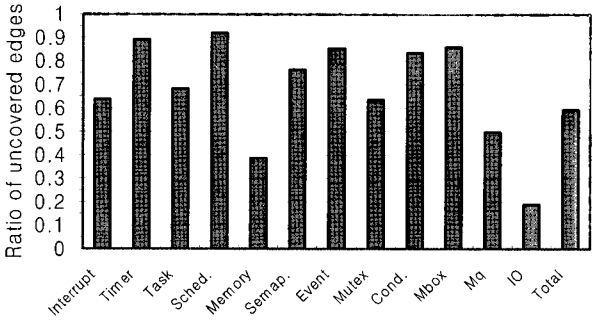

그림 7 400여개의 테스트 케이스를 수행한 경우에 운영체제 각 컴포넌트별 내부의 수행되지 않은 코드 영역의 비율

5. 결 론

본 논문에서는 임베디드 시스템의 단가를 낮추고 소모 전력을 줄이기 위하여 하드웨어의 지원 없이 코드 메모리의 크기를 최적화시키는 기법을 제안하였다. 제안하는 코드 컴팩션 기법은 종래의 하드웨어 컴팩션 기법인 Echo 명령어를 운영체제를 통해 소프트웨어적인 방법으로 지원함으로써 Echo 명령어와 동일하게 프로그램 코드의 크기를 줄이는 장점을 얻었다. 또한 임베디드 소프트웨어 중에서 높은 비율의 코드가 수행되지 않거나 거의 수행되지 않는 특성을 활용하여 이 영역에 선택적으로 제안하는 컴팩션 기법을 적용해 수행 시간상의 성능 저하를 예방할 수 있었다.

참고문헌

[1] D. Chanet, B.D. Sutter, B.D. et al., "System-wide Compaction and Specialization of the Linux Kernel," In Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pp. 95-104, 2005.

[2] A. Beszedes, R. Ferenc, and T. Gyimothy, "Survey of Code-Size Reduction Method," ACM Computing Surveys, Vol. 5, No. 3, pp. 223-267, 2003.

[3] S.K. Derray, W. Evans, R. Muth, and B.D. Sutter, "Compiler Techniques for Code Compaction," ACM Transactions on Programming Languages and Systems, Vol.22, No.2, pp.378-415, 2000.

[4] R. Muth, Alto: A Platform for Object Code Modification, University of Arizona, Ph.D. Dissertation, 1999.

[5] K.S. Yim, J.-S. Lee, J. Kim, S.-D. Kim, and K. Koh, "A Space-Efficient On-Chip Compressed Cache Organization for High Performance Computing," Lecture Notes in Computer Science (LNCS), Vol. 3358, pp. 952-964, 2004.

[6] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, pp.660-720, 1986.

[7] W. Cheung, W. Evans, and J. Moses, "Predicated Instructions for Code Compaction," LNCS, Vol.2826, pp.17-32, 2003.

[8] C.W. Fraser, "Method and System for Compressing Program Code and Interpreting Compressed Program Code," United States Patent, No.2003/0229709 A1, 2002.

[9] J. Lau, S. Schoenmackers, T. Sherwood, and B. Calder, "Reducing Code Size with Echo Instructions," In Proceedings of CASES, pp.84-94, 2003.

[10] K.D. Cooper and N. McIntosh, "Enhanced Code Compression for Embedded RISC Processors," ACM SIGPLAN Notices, Vol. 5, pp.139-149, 1999.

[11] B.D. Sutter, H. Vandierendonck, B.D. Bus, and K.D. Bosschere, "On the Side-Effects of Code Abstraction," In Proceedings of LCTES, pp. 244-253, 2002.

[12] K.S. Yim et al., "Operating System Support for Procedural Abstraction in Embedded Systems," In Proceedings of the

12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2006.
[13] K.S. Yim et al., "A Software Reproduction of Virtual Memory for Deeply Embedded Systems," Lecture Notes in Computer Science(LNCS), Vol.3980, pp.1000-1009, 2006.

임 근 수

2003 연세대학교 컴퓨터산업공학(학사)
2003 연세대학교 전기전자공학(학사)
2005 서울대학교 전기컴퓨터공학(석사)
2005 일본 정보통신연구기관(NICT) 방문연구원
현 재 삼성종합기술원 연구원
관심분야 : 운영체제, 소프트웨어 개발도구, 미들웨어, 컴퓨터 구조, 그리고 컴퓨터 네트워크임.
E-mail : keunsoo.yim@samsung.com