# DDMB 구조에서의 런타임 메모리 최적화 알고리즘

조 정 훈[†] · 백 윤 흥[††] · 권 수 현[†††]


## 요 약

대부분의 디지털 신호 처리기 (Digital Signal Processor)는 두 개 이상의 메모리 뱅크를 가지는 하버드 아키텍처 (Harvard architecture)를 지원한다. 다중 메모리 뱅크 중에서 하나는 프로그램용으로 나머지는 데이터용으로 사용하여 프로세서가 한 명령어 사이클에 메모리의 여러 데이터에 동시 접근을 가능하게 한다. 이전 연구에서 우리는 다중 메모리 뱅크에 효율적으로 데이터를 할당하는 방법에 대하여 논하였다. 본 논문에서는 이전 연구의 확장으로 런타임 메모리의 최적화에 대한 우리의 최근 연구에 대하여 소개한다. 듀얼 데이터 메모리 뱅크(Dual Data Memory Bank)를 효율적으로 이용하기 위해 각 메모리 뱅크에 할당된 변수를 관리하기 위한 독립적인 두 개의 런타임 스택이 필요하다. 프로시저에 대한 두 메모리 뱅크의 활성화 레코드(Activation Record)의 크기는 각 메모리 뱅크에 할당된 변수의 개수가 일정하지 않기 때문에 다를 수 있다. 따라서 여러 개의 프로시저가 연속으로 호출될 때 두 개의 런타임 스택의 크기가 크게 달라질 수 있다. 이러한 두 메모리 뱅크 사이의 불균형은 하나의 메모리에 여유 공간이 있음에도 불구하고 다른 하나의 메모리 뱅크의 사용량이 온칩 메모리(on-chip memory)범위를 초과하는 원인이 될 수 있다. 본 논문에서는 온칩 메모리를 효율적으로 사용하기 위해 두 런타임 스택의 균형 맞추기를 시도했다. 본 논문에서 제안하는 알고리즘은 상대적으로 단순하지만 효율적으로 런타임 메모리를 사용할 수 있다는 것을 실험결과를 통해 보여주고 있다.


**키워드 : 실행 시간 환경, DSP, 듀얼 데이터 메모리 뱅크, 컴파일러, 온칩 메모리**

# Run-time Memory Optimization Algorithm for the DDMB Architecture

Jeonghun Cho[†] · Yunheung Paek[††] · Soohyun Kwon[†††]


## ABSTRACT

Most vendors of digital signal processors (DSPs) support a Harvard architecture, which has two or more memory buses, one for program and one or more for data and allow the processor to access multiple words of data from memory in a single instruction cycle. We already addressed how to efficiently assign data to multi-memory banks in our previous work. This paper reports on our recent attempt to optimize run-time memory. The run-time environment for dual data memory banks (DDMBs) requires two run-time stacks to control activation records located in two memory banks corresponding to calling procedures. However, activation records of two memory banks for a procedure are able to have different size. As a consequence, dual run-time stacks can be unbalanced whenever a procedure is called. This unbalance between two memory banks causes that usage of one memory bank can exceed the extent of on chip memory area although there is free area in the other memory bank. We attempt balancing dual run time stacks to enhance efficiently utilization of on-chip memory in this paper. The experimental results have revealed that although our algorithm is relatively quite simple, it still can utilize run-time memories efficiently; thus enabling our compiler to run extremely fast, yet minimizing the usage of run-time memory in the target code.


**Key Words : Run-time environment, DSP, dual data memory banks, compiler, on-chip memory**

## 1. 서 론

일반적으로 DSP는 프로세서와 메모리 간의 속도 차이를 줄이기 위해 온칩 메모리를 가지고 있다. 이 온칩 메모리는 프로그램 메모리와 데이터 메모리로 분류되어 있어 여러 메모리에 동시 접근이 가능한 하버드 아키텍처로 구성되어있다[6]. Analog Device ADSP2100, DSP Group PineDSPCore, Motorola DSP5600, NEC uPD77016와 같은 최근 DSP 제품들은 이러한 다중 데이터 뱅크를 사용하여 메모리 액세스 성능을 향상시켰다. 이 성능 향상으로 인해 식 (1)의 FIR 필터와 같은 범용 DSP 알고리즘은 a 배열과 b 배열을 메모리로부터 동시에 읽어 연산을 실행할 수 있기 때문에 생성되는 코드의 크기 측면이나 실행 속도 측면에서 향상을 기대할 수 있다.

※ 이 논문은 2005년도 경북대학교학술진흥연구비에 의하여 연구되었음
† 정 회 원 : 경북대학교 전자전기컴퓨터학부 전임강사
†† 정 회 원 : 서울대학교 전기컴퓨터공학부 부교수, 교신저자
††† 준 회 원 : 경북대학교 전자공학과 석사과정
논문접수 : 2006년 5월 13일, 심사완료 : 2006년 8월 21일

$$c(n)=\sum_{i=0}^{N-1}(a(i)\times b(n-i)) \quad (1)$$

DSP에 내장되는 온칩 메모리는 크기가 작기 때문에 DSP 소프트웨어 개발 과정에서 효율적인 메모리 사용이 중요한 이슈가 되고 있다. 이전 연구에서 우리는 이러한 이슈에 초점을 맞추어 다중 데이터 메모리 뱅크에 변수를 할당하여 전반적인 시스템 성능을 향상시킬 수 있는 MBA (Memory bank assignment) 알고리즘을 제안하였다[2]. 그러나 더 효율적인 듀얼 데이터 메모리 뱅크(Dual Data Memory Bank, DDMB) 지원을 위해 우리가 개발한 컴파일러 기술에 적합한 실행 환경을 제안하려 한다. DDMB 안에 저장된 변수를 위한 실행 환경을 관리해주는 컴파일 기술의 효율성에 따라 DSP 시스템의 전반적인 성능이 크게 영향 받는 것을 알 수 있었다.

본 논문에서는 이러한 문제를 해결하기 위해 상용 DSP의 DDMB에서 각 활성화 레코드 (activation record, AR)에 저장되는 여러 변수들을 효율적으로 관리하는 방법을 제시하고 있다. 이 방법은 컴파일러가 DDMB안에 저장된 AR의 전체 메모리 공간의 균형을 맞춰 타깃 DDMB 구조에서 메모리 사용의 효율성을 최대로 향상시킨다. 이 방법이 DSP의 온칩 SRAM을 조금 더 효율적으로 사용할 수 있는 것을 실험을 통해 보여준다. 이번 연구를 위해 우선 X와 Y 메모리 뱅크에 각각 저장되는 두 개의 AR을 프로시저마다 생성한다. 이것은 우리의 이전 연구인 DDMB 할당 테크닉[2]을 사용한다. 이렇게 생성된 두 개의 AR은 그 크기에 따라 두 메모리 뱅크의 사용량이 같도록 적절하게 메모리 뱅크에 할당하여 낭비되는 메모리 공간을 최소화시키는 방법을 적용한다.

2 장에서는 MBA 문제에 관한 이전 연구들에 대해 간략하게 설명한다. 또한 다중 메모리 뱅크에 AR을 저장하는 방법들을 검토하고 끝부분에는 런타임 메모리 관리에 대한 여러 가지 방법들에 대해 논하겠다. 3장과 4장에서는 DDMB 구조에 AR을 최적으로 저장하는 문제를 정의하고 이 문제에 대처하는 접근 방법을 제시할 것이다. 5장에서는 DDMB구조를 가진 상업용 DSP에 DSP 벤치마크들을 이용한 실험 결과를 제시하고 다른 결과들과 우리가 제안하는 알고리즘의 성능을 비교할 것이다. 마지막으로 6장에서 결론을 맺는다.

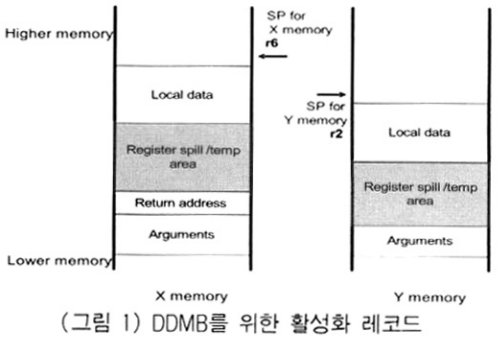
## 2. 이전 연구

MBA 문제에서 각 프로시저에서 참조되는 모든 지역 변수와 인수들은 두 개의 그룹으로 나누어져 각각 데이터 메모리 뱅크에 할당되어야 한다. 첫 번째로 Leupers 와 Kotte가 이 문제에 대해 접근한 방법은[5], 각 어셈블리 코드 함수에 대허 데이터 의존 그래프(Data Dependency Graph, DDG)를 만드는 것이다. 이 DDG를 가지고 그래프의 edge가 가능한 병행(potential parallelism)을 뜻하는 간섭 그래프(Interference Graph, IG)를 만든다. 이 IG의 크기를 줄인 다음, IG를 분할하기 위해 integer linear programming(ILP)를 적용했다. 이 할당 문제를 풀기 위해서 에 기반한 방식은 응용프로그램(application)에 적용될 경우 컴파일 시간이 상당히 증가한다는 단점이 있다. 우리는 컴파일 시간을 단축시키기 위해 maximum spanning tree알고리즘을 바탕으로 polynomial time에서 다중 메모리 뱅크 할당을 해결하기 위한 접근법을 제시한 바 있다[2].

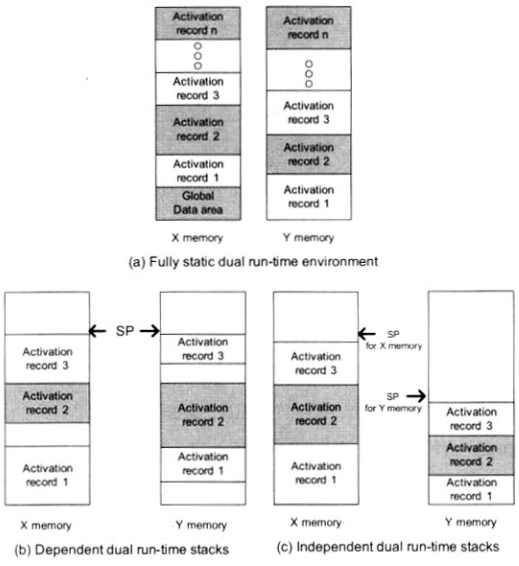
효율적으로 할당된 메모리 뱅크를 사용하기 위해서는 잘 디자인된 런타임 환경이 필수적이다. 대표적으로 C 프로그램 언어용 런타임 환경은 타깃 머신(target machine)에 의존하지 않는 스택을 기반으로 하여 구현되어 있다. Motorola DSP56300과 같이 두 개의 데이터 메모리 뱅크를 가지고 있는 타깃 머신의 경우에는 두 개의 런타임 스택을 고려해야 한다. 그러나 여러 개의 프레임 포인터, 스택 포인터와 함께 다수의 런타임 스택을 동시에 관리하는 것은 하나의 런타임 스택을 관리하는 기존의 방법보다 여러 가지 복잡한 최적화 문제가 생긴다. 이러한 문제점들을 줄이기 위해 SPAM 컴파일러는 스택 기반 환경 대신 완전 정적 런타임 환경(fully static run-time environment)을 사용한다[8](그림 2(a)). 정적 환경에서는 모든 AR들이 프로그램이 실제 실행되기 전에 메모리 안의 고정된 장소에 저장되어야 하므로 실행시키기 쉽고, 실행 시간 동안 실행 경로를 따라가기 위한 다수의 포인터를 유지 할 필요가 없다. 그러나 포트란과 같이 재귀호출(recursive calls)이 없는 프로그래밍 언어에서만 쓰일 수 있다는 치명적인 단점이 있다. 그래서 그들이 제시한 방법은 재귀반복이 없는 C코드에만 제한적으로 적용되며 런타임에 모든 AR들이 할당되어 있어야 하므로 메모리의 낭비가 크다.

우리의 이전 연구에서는 두 개의 런타임 스택을 유지함으로써 이러한 단점의 극복하였다 (그림 2(b))[2]. 이전 연구에서 스택 조작(manipulation) 레지스터의 사용을 줄이기 위해 하나의 스택 포인터로 X와 Y 메모리에 저장된 AR세트를 관리하는 방식을 사용하였다. 이 방식의 가장 뚜렷한 장점은 단 하나의 프레임 포인터(스택 포인터)만 쓰기 때문에 두 개의 스택을 유지하는 방법이 하나의 스택을 유지하는 것만큼 간단하다. 그러나 두 스택은 항상 동기화 되어 있어야 하므로 낭비되는 메모리는 여전히 존재한다(그림 2(b)). 이러한 단점을 완화하기 위해 본 논문에서는 두 스택을 개별적으로 관리하는 방법을 제시한다(그림 2 (c)). 이들 두 가지의 스택 실행 방법을 구분하기 위해서 전자를 dependent stack 실행, 후자를 independent stack 실행이라고 명명한다. 비록 AR을 independent 스택에 할당하는 것이 대부분의 경우 메모리 사용면에서 더 나은 결과를 얻을 수 있지만, 두 스택 크기의 균형을 맞추기 위해서는 복잡한 방법이 필요하다.

한편 두 메모리 뱅크 중 한 스택이 온칩(on-chip) SRAM의 용량보다 커지면 오프칩(off-chip) 메모리를 사용해야 하는 일




(그림 1) DDMB를 위한 활성화 레코드




(그림 2) DDMB 런타임 환경

이 발생하게 되는데 이로 인해 성능이 감소된다. 다행히도 컴파일러가 AR의 크기와 영역(field)를 결정하기 때문에 AR의 크기는 컴파일 시간(compile time)에 결정된다(그림 1). 우리는 이러한 성질을 이용해서 컴파일 시간 동안 전체적으로 사용되는 메모리 요구량을 미리 계산하여 두 개의 스택의 균형을 맞출 수 있었다. 이때 여분의 스택 포인터가 필요하게 된다(그림 2(c)). 그림 1에서 주소 레지스터 r2, r6는 각각 X와 Y 런타임 스택들을 관리하는 스택 포인터로 X와 Y 메모리 뱅크에서 다음 사용 가능한 데이터 메모리 위치를 가리킨다.

## 3. 밸런스 된 런타임 환경

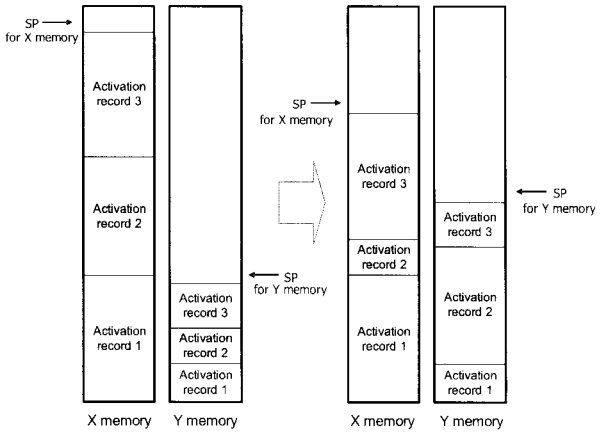
본 장에서는 DDMB 구조에서 런타임 메모리를 최적화하기 위한 방안과 그로 인해 얻을 수 있는 장점에 대하여 언급한다. 우선 런타임 메모리의 사용을 최적화 시키게 된 동기를 설명하고 최적화를 위한 우리의 효율적인 알고리즘을 설명하겠다.

### 3.1 동기

듀얼 런타임 스택의 밸런싱에 대한 정의를 먼저 한다.

**[정의 1]** $A = \{(A_{j0}, A_{j0}), (A_{j1}, A_{j1}), ..., (A_{jn-1}, A_{jn-1})\}$를 어떤 순간에 런타임 스택의 AR(activation record) 집합이라고 하자. $A_{jk}$ 와 $A_{jk}$ 는 함수 k $(0 \le k \le n-1)$의 두 메모리 뱅크를 위한 AR을 의미한다. 그리고 $A_X = \{A_{X0}, A_{X1}, ....A_{Xn-1}\}$ 와 $A_Y = \{A_{Y0}, A_{Y1}, ..., A_{Yn-1}\}$ 는 $A_{jk}$ 와 $A_{jk}$에서 함수 k $(0 \le k \le n-1)$를 위한 각각 X와 Y 메모리 뱅크에 위치한 런타임 스택에 있는 AR의 집합이라 하자. X와 Y 메모리에 $A_X$ 와 $A_Y$ 크기 차이가 최소화되도록 AR이 할당될 때 그 시점에서 런타임 스택이 **밸런스(balanced)되었다**고 한다.

만약 온칩 메모리의 크기가 전체 프로그램을 실행시킬 수 있을 만큼 충분히 크다면 두 런타임 스택의 균형을 맞추는 일은

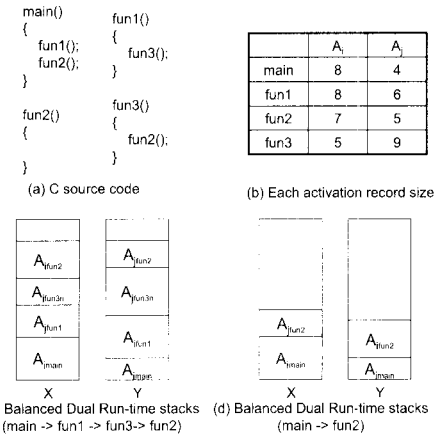



(그림 3) Balanced 듀얼 런타임 환경

의미가 없어질지도 모른다. 그러나 일반적으로 온칩 메모리의 크기는 한정되어 있기 때문에(Motorola DS56301은 프로그램 메모리로 3k words, X, Y 메모리로 각각 2k words씩 가지고 있다[3]), 온칩 메모리의 공간을 모두 사용하면 오프 칩 메모리를 사용할 수 밖에 없다. 그러나 오프 칩 메모리를 사용하게 되면 명령어를 실행하기 위한 긴 메모리 액세스 타임으로 인해 하나의 명령어를 한 머신 사이클에 실행할 수 없게 된다. 또한 이것은 DDMB 아키텍처의 장점인 다중 메모리 뱅크의 동시 접근을 불가능하게 만들어 성능의 저하를 가져오게 된다. 다음의 그림에서는 동일한 프로그램에 대하여 두 개의 런타임 스택이 균형이 맞았을 때와 그렇지 않을 때의 차이점을 보여준다.

(그림 3)의 왼쪽은 두 개의 런타임 스택의 균형이 맞지 않는 경우이고 오른쪽은 두 스택 사이의 균형이 잡힌 경우이다. 새로운 AR4가 스택에 할당될 때 왼쪽 스택의 경우 X메모리의 런타임 스택이 온칩 X메모리에 충분한 공간이 없기 때문에 온칩 Y메모리에 여유공간이 있는데도 새로운 AR4는 오프칩 메모리에 위치해야 한다. 그러나 그림 3의 오른쪽처럼 AR의 재배치를 통해 균형 잡힌 두 개의 런타임 스택에서는 새로운 AR4를 온칩 메모리에 위치할 수 있게 되어 성능의 저하가 발생하지 않는다. 따라서 X와 Y메모리 사이에 AR을 균형 있게 배정하였을 때 온칩 메모리의 사용 능력을 향상시키고 실행 속도의 성능을 증가시킬 수 있는 것을 알 수 있다.

본 장에서 두 개의 런타임 스택의 균형을 유지하기 위한 새로운 접근 방법을 설명한다. 그림 4를 예로 들어보면 main, fun1, fun2 와 fun3 이렇게 4개의 함수가 있고 fun2를 호출하는 경로가 2가지가 있다. 하나는 main → fun1 → fun3 → fun2, 다른 하나는 main → fun2 이다(그림 4(a)). X, Y메모리에 할당되는 두 경로의 각 함수에 관한 AR을 보자. 두 스택의 균형을 맞추기 위한 AR의 배정은 다음과 같다. 첫 번째 경로의 AR은 $AX = \{A_{jmain}, A_{jfun1}, A_{jfun3}, A_{jfun2}\}$, $AY = \{A_{jmain}, A_{jfun1}, A_{jfun3}, A_{jfun2}\}$(그림 4(c))이고 두 번째 경로의 AR은 $AX = \{A_{jmain}, A_{jfun2}\}$, $AY = \{A_{jmain}, A_{jfun2}\}$(그림 4(d))이다. 위의 예에서 런타임 스택의 균형을 맞추기 위해서 fun2 함수는 호출 순서(calling sequence)에 따라 각각 다른 메모리(X 또는 Y메모리)에 할당되어야 함을 알 수 있다. 이러한 문제는 함수 복제(function cloning)를 사용하여 해결할 수 있지만 코드의 크기가 커지는 단점이 있다.




| | $A_i$ | $A_j$ |
|---|---|---|
| main | 8 | 4 |
| fun1 | 8 | 6 |
| fun2 | 7 | 5 |
| fun3 | 5 | 9 |

(그림 4) Balanced 듀얼 런타임 스택의 예

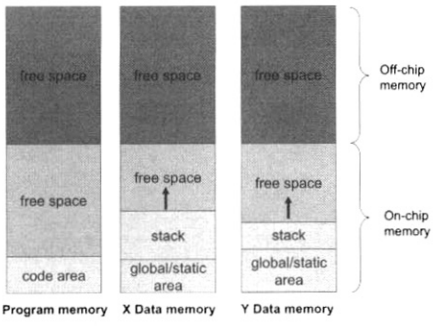
그러므로 여러 개의 호출 순서들 중에서 어떤 것으로 균형을 맞출 것인지를 결정해야 한다. 앞에서 언급한 바와 같이 우리의 주 목적은 X와 Y메모리 뱅크에서 각각의 AR 합의 차가 최대가 되어 가장 균형이 맞지 않을 때 두 런타임 스택의 균형을 잡아주어 온칩 메모리의 사용을 최대화하는 것이다. 그러므로 X, Y 런타임 메모리 사용량의 차이가 가장 큰 콜 체인에 밸런싱 알고리즘을 적용하고 만약 AR이 X 나 Y메모리에 이미 할당되어있다면 할당을 건너뛴다. 더 이상 변동이 일어나지 않거나 모든 AR이 고정 될 때까지 밸런싱 알고리즘은 반복된다. 다음 절에서 알고리즘에 대하여 좀더 자세하게 설명한다.

### 3.2 DDMB를 위한 밸런싱 알고리즘

DDMB 구조에서 실행 환경은 프로그램 메모리, X 메모리, Y 메모리 이렇게 세가지 메모리 뱅크를 사용한다. 프로그램 메모리에는 컴파일러에 의해 생성된 코드가 적재되고 X와 Y 메모리에는 모든 정적, 동적 데이터가 적재된다. 전역 변수와 정적 변수들은 X, Y 메모리의 전역/정적 영역에 할당되고 지역 변수와 프로시저의 activation을 관리하는 정보들은 스택 영역에 할당된다. 우리가 사용할 컴파일러는 동적으로 할당되는 변수(예. C 프로그래밍 언어의 malloc 함수)는 지원하지 않기 때문에 런타임 환경에서 힙(heap) 영역은 제외한다.

본 논문의 핵심은 X와 Y메모리 사이의 AR 크기의 균형을 유지하는 것이고 두 AR의 내용에 관한 자세한 정보는 본 논문에서 언급하지 않았다. 프로그램이 실행됨에 따라 스택 공간이 상위주소로 자랐다가 하위주소로 줄어들기를 반복하기 때문에 온칩 메모리의 범위를 초과할 수 있다. 또 DDMB 구조는 두 개의 데이터 메모리 뱅크를 가지고 있기 때문에 하나의 메모리에 온칩 메모리의 빈 공간이 있다고 해도 다른 메모리의 온칩 메모리는 다 써버릴 수 있다. 그러므로 두 메모리 뱅크 사이의 균형을 유지하는 것이 온칩 메모리를 효율적으로 사용하는 것이 된다.

우리의 밸런싱 알고리즘을 좀더 쉽게 설명하기 위해 프로그램의 종류를 재귀호출(recursive calls)이 있는 경우와 없는 경우로 구분하였다. 이 절에서는 재귀호출이 없는 프로그램을 다룬다. 런타임 메모리의 최대 사용량을 알기 위해 우선 확장 activation 트리(Extended activation tree, EAT)를 정의한다. DDMB 구조를 지원하기 위해 [1]의 activation 트리를 확장하였다.




(그림 5) DDMB의 메모리 배치

**[정의 2]** T = (V, E, W)를 weighted tree라 하자. V는 프로시저의 activation 집합을 나타내고, E는 프로시저 사이의 호출 관계를 나타내는 edge를, W는 자식 프로시저가 호출될 때 X와 Y 메모리 AR 크기의 증가를 뜻하는 weight 집합을 의미한다. edge $(V_i, V_j)$는 프로시저 $V_i$ 가 $V_j$를 호출한다는 것을 뜻하고 $W_k = (W_{xk}, W_{yk})$는 프로시저가 호출될 때 X 메모리가 $W_{xk}$ 만큼 증가하고 Y 메모리가 $W_{yk}$ 만큼 증가함을 뜻한다. 우리는 이 weighted tree T를 extended activation tree(EAT)라고 부른다. 위에서 정의된 EAT는 전역/정적 영역을 뜻하는 pre-defined node인 root node를 가지고 있다.

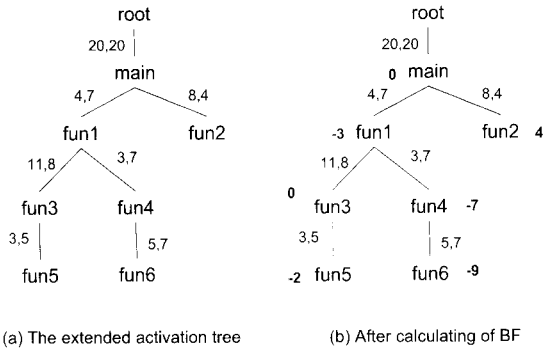
정의 2의 EAT 정의로부터,

1. 각각의 노드는 유일한 activation이다.
2. 프로시저는 호출 순서에 의존하는 다수의 activation들로 나타내어진다
3. 콘트롤이 activation $V_i$ 에서 $V_j$ 로 이어지면 $V_i$ 는 $V_j$ 의 부모이다.
4. 프로그램의 콘트롤은 두 번째 child를 처리하기 전에 루트부터 첫 번째 child의 가장 멀리 있는 descendent까지 먼저 처리하는 EAT의 depth-first traversal 경로를 따라 흐른다.

두 스택의 시작 주소는 X와 Y메모리 뱅크에 저장되는 전역, 정적 변수에 따라 달라지기 때문에 서로 다르다. 이 서로 다른 시작 주소를 고려하기 위해서 초기 설정 루트 노드(default root node)를 사용한다. 이 루트 노드가 X, Y메모리에서 전역, 정적 변수 저장에 쓰인 메모리 사용량을 뜻하는 weight pair와 함께 main함수를 호출한다고 가정한다. DDMB를 위한 우리의 알고리즘을 설명하기 위해 Balancing Factor (BF)를 정의한다.

**[정의 3]** Balancing Factor(BF)는 ux(X메모리의 사용량) - *uy(Y메모리의 사용량)로 정의한다.*

정의 3에서 BF가 양의 값이면 X메모리의 사용량이 Y메모리의 사용량보다 많다는 것을 뜻하고, '0'이면 두 메모리의 사용량이 같음을 나타내므로 두 개의 런타임 스택이 균형이 잡혀 있음을 뜻하며, 음의 값이면 Y메모리의 사용량이 X메모리의 사용량보다 크다는 것을 뜻한다. BF의 절대값이 클수록 두 스택의 균형이 맞지 않음을 뜻하기 때문에 우리의 알고리즘은 |BF|가 가장 큰 경로를 찾아서 |BF|가 작아 지도록 X와 Y메모리에 있는




(a) The extended activation tree (b) After calculating of BF

(그림 6) EAT의 구성과 BF 계산

AR을 서로 바꾸는 방법(swap)을 통해 경로의 균형을 맞출 것이다. 그림 7은 좀더 자세한 밸런싱 알고리즘이다. 이해를 돕기 위해 간단한 예제를 통해 우리의 알고리즘을 설명한다. 그림 6 (a)에서 main 함수가 fun1 함수와 fun2 함수를 호출하고 fun1함수는 fun3 함수와 fun4 함수를 호출한다. fun3 함수는 fun5 함수를 호출하고 fun4 함수는 fun6 함수를 호출한다. Edge 'root → main' 의 weight(20,20)는 X, Y 메모리 뱅크 안에 각각 할당되는 main함수의 정적 변수와 전역 변수의 합을 뜻한다. main함수가 fun1을 호출할 때 (4,7)의 의미는 X메모리에 4 워드만큼 할당하고 Y메모리에는 7 워드만큼 할당한다는 것을 의미한다.

EAT의 모든 단말(leaf) 노드들을 찾는 그림 7 두 번째 줄의 findLeafNodes함수를 시작으로 알고리즘이 실행된다. 단말 노드란 마지막으로 호출되는 프로시저로 단말 노드 밑으로는 더 이상 activation 노드가 없다는 뜻이기 때문에 런타임 스택이 마지막 단말 노드에서 지역적 최대값(local maximum)을 가진다. 지역적 최대값이란 전체적인 스택 사용량의 최대값을 항상 만족하지 못할 수도 있지만 그 호출 순서 내에서 런타임 스택의 최대 사용량을 의미하는 것이다. 그림 7에서 L의 값은 두 개의 런타임 스택에서 메모리 사용이 지역 최대치(local maxima)인 {fun2, fun5, fun6}이 되고 그림 6 (b)에서처럼 EAT의 각 activation 노드마다 BF를 계산한다. fun2 activation의 BF는 X 메모리 뱅크의 런타임 스택의 크기가 Y 메모리 뱅크 런타임 스택보다 4 워드 더 큼을 뜻하는 4 이고, fun5 activation의 BF는 Y 메모리 뱅크가 X 메모리 뱅크보다 2 워드 더 큼을 뜻하는 -2 이다.

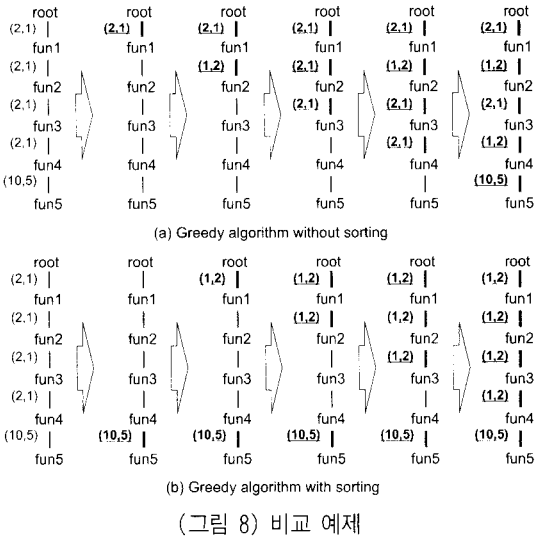
이제 핵심 알고리즘을 적용하기 위한 준비 과정이 끝났다. 알고리즘은 단말 노드 집합(L)이 공집합이 될 때까지 반복해서 적용된다. 우선 그림 7의 5번째 줄의 L에서 BF가 가장 큰 activation을 선택한다. 루트에서부터 선택된 이 activation까지의 경로가 전체 호출 순서들 중에서 local maximum를 가진 가장 균형이 잡히지 않은 호출 순서이다. BF가 클수록 두 개의 런타임 스택의 균형이 잡혀있지 않다는 뜻이기 때문에 이 경로를 가지고 균형을 맞추기 시작한다. 첫 번째 activation으로 fun6이 선택되고 그에 따른 호출 순서는 root → main → fun1 → fun4 → fun6 이다(그림 6). 첫 번째 계산이기 때문에 이 경로의 모든 activation들은 pre-assigned 메모리 뱅크를 가지고 있다. 다시 말해 모든 activation이 아직 특정 메모리 뱅크에 고정되지 않았다는 뜻이다.

(그림 7)에서 13번째 줄부터 32번째 줄까지가 밸런싱 알고리즘

```
Input:  EAT
Output: Balanced EAT
algorithm: MakeBalancing
1  E = ∅; // A set of assigned edges in EAT.
2  L = findLeafNodes(EAT); // A set of leaf nodes in EAT.
3  calculate_BF(EAT);
4  while (L != ∅)
5      l = find a node with maximum BF in L;
6      p — path from root node to l;
7      q = sort(p);   // descending sort for each activation record
8                     // of the found path
9      // Balancing the found path
10     A — ∅;         // Set of assigned node
11     recal_node — NULL;      // The node to recalcuate BF
12     current_bf = 0;  // Current BF
13     for all activation records q_i in q do
14         if (q_i is not an element of A)
15             if ((current_bf > 0) and (size(X_{q_i}) > size(Y_{q_i})))
16                 swap(X_{q_i}, Y_{q_i});
17                 if ((recal_node equals to NULL)
18                     or (q_i is not descendent of recal_node))
19                     recal_node — q_i;
20                 end if
21             end if
22             if ((current_bf < 0) and (size(X_{q_i}) < size(Y_{q_i})))
23                 swap(X_{q_i}, Y_{q_i});
24                 if ((recal_node equals to NULL)
25                     or (q_i is not descendent of recal_node))
26                     recal_node — q_i;
27                 end if
28             end if
29             A — A ∪ {q_i};          // To fix assignment
30         end if
31         current_bf = current_bf + bf_of(X_{q_i}, Y_{q_i});
32     end do
33     // Recalculate BF using S
34     recalc_BF_of_subtree(S);
35     L — L - {l}
36 end while
end algorithm
```

(그림 7) 밸런싱 알고리즘

의 핵심 부분이다. 7줄의 sort함수는 다음 문단에서 설명하겠다. 우리 알고리즘의 기본 개념은 두 메모리 뱅크의 크기를 비교해서 사용량이 많은 메모리 뱅크에 작은 activation을 할당하는 greedy algorithm이다. 초기 단계에서 X와 Y 메모리 뱅크에 할당된 activation이 없기 때문에 첫 번째 activation은 어느 메모리에 배정되어도 상관없다. 따라서 선택된 콜 체인의 첫 번째 activation들은 전 단계의 MST 알고리즘으로부터 배정받은 메모리로 할당 된다. 콜 체인의 두 번째 activation부터 greedy algorithm이 적용되는데 X 메모리에 할당된 워드의 수가 Y 메모리에 할당된 워드 수보다 크다면 Y 메모리 뱅크에 두 번째 activation중에서 더 큰 activation을 할당한다. 이러한 과정을 경로의 모든 activation 노드마다 반복적으로 수행한다.




(a) Greedy algorithm without sorting

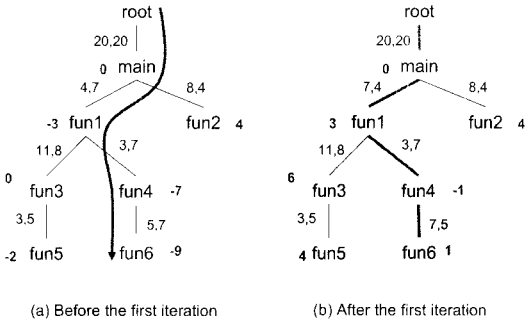
(b) Greedy algorithm with sorting

(그림 8) 비교 예제

우리는 호출 순서의 균형을 맞추기 위해 내림차순 정렬(descending sort)을 포함하는 greedy 접근 방식을 사용하였다. 그림 7의 pseudo 코드 7번 라인에 더 큰 activation을 먼저 처리하기 위한 내림차순 정렬을 적용한다. 만약 정렬을 하지 않으면 주어진 콜 체인에 대하여 런타임 메모리를 최소한으로 사용하기 위한 메모리 뱅크 할당은 최적의 결과를 얻을 수 없다. 그림 8은 정렬의 필요성에 대한 예를 보이고 있다. 그림 8 (a)에서 듀얼 런타임 스택의 균형을 잡는데 greedy 접근 방식만 적용된다면 두 번째와 네 번째 X, Y activation은 서로 교환되어야 하고 할당된 결과는 BF가 5가 되는 (2, 1), (1, 2), (2, 1), (1, 2), (10, 5)이다. 직관적으로 이것은 최적의 답이 아님을 알 수 있다. 만약 greedy 알고리즘에 정렬을 적용한다면 edge 'fun4 → fun5' 의 weight(10,5)가 맨 처음 할당될 것이고 그 다음 edge 'root → fun1' 의 weight(2,1)가 (1,2)로 교환되어 할당될 것이다. 이 반복 과정이 끝나면 BF가 1인 (1, 2), (1, 2), (1, 2), (1, 2), (10, 5)로 최적의 값이 할당 되는 것을 알 수 있다.

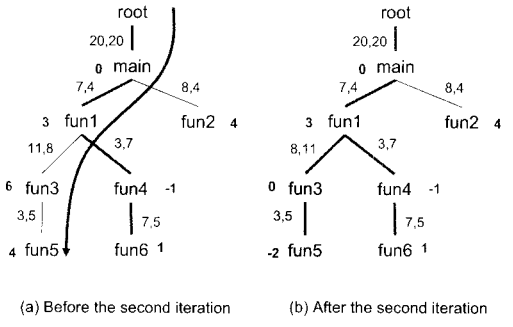
제시한 예(그림 6)의 첫 번째 계산은 그림 9에서 설명하고 있다. 위에서 찾은 root → main → fun1 → fun4 → fun6 경로는 메모리를 사용하는 크기에 따라 균형이 맞추어 진다.

1. fun1 → fun4의 BF는 4이다. 이 activation의 현재 BF(*current_bf, 알고리즘에 있는 변수*)는 0으로 처음으로 균형이 잡히는 것이므로(*current_bf*는 0이다) 미리 배정된 뱅크 (3,7)를 할당 받으면 *current_bf*는 -4가 된다.
2. main → fun1의 BF는 3이다. 전 단계로부터 받은 *current_bf*가 음수(-4)로 Y메모리의 activation크기가 더 크기 때문에 여기서 할당 받을 activation들은 (4,7)에서 (7,4)로 교환되어야 한다. 그렇게 되면 *current_bf*는 이전 BF -4와 (7,4)의 BF 3을 합하여 -1이 된다. 그리고 *recal* 노드는(다시 BF를 계산해야 하는) fun1이 된다.
3. fun4 → fun6의 BF는 2이다. *Current_bf*가 음수이고 Y메모리의 activation크기가 더 크기 때문에 이들 또한 (7,5)로 교환되어야 하고 그렇게 되면 *current_bf*는 1이 된다. 그리고 *recal* 노드는 fun6이 단계 2의 *recal* 노드 fun1의 자손이기 때문에 여전히 fun1이다.
4. root → main의 BF는 0이다. X, Y메모리 뱅크의 두 activation의 크기가 같기 때문에pre-assigned 뱅크를 할당 받는다. 결과적으로 마지막 BF는 1이 된다.




(a) Before the first iteration (b) After the first iteration

(그림 9) 첫 번째 반복




(a) Before the second iteration (b) After the second iteration

(그림 10) 두 번째 반복

선택된 경로의 균형을 잡으면서 X, Y메모리 사이의 activation들을 교환하였기 때문에 이전에 계산된 BF는 올바른 값이 아니다. 따라서 밸런싱 알고리즘을 한번 실행 시킨 후에 BF를 다시 계산해야 한다. 그러나 EAT의 모든 노드에 대하여 BF를 다시 계산할 필요는 없고 변화된 가장 높은 레벨의 노드를 찾아 그 노드의 자손에 대하여만 BF를 다시 계산하면 된다. 즉, 위의 예에서 main → fun1 과 fun4 → fun6 의 activation이 교환되었기 때문에 그림 9 (b)의 회색 부분 안의 fun1과 fun6의 자손들의 BF를 다시 계산해야 한다. 그러나 이때 fun6은 fun1의 자손이기 때문에 fun1의 BF가 계산되고 fun6의 BF를 재계산해야 한다. 위의 알고리즘 설명에서 보이는 *recal* node가 재계산을 필요로 하는 서브트리의 루트노드가 된다. 모든 계산이 끝나고 BF는 1이 되었다(그림 9 (b)).

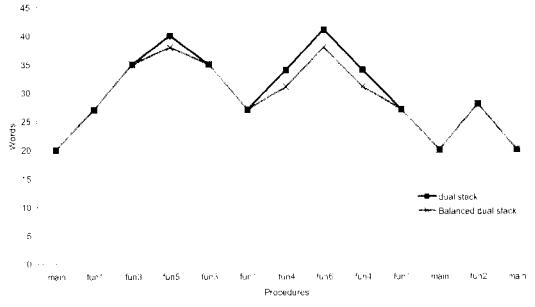
두 번째 경로에 알고리즘을 적용할 때 root→ main → fun1 → fun3 → fun5 는 그림 10처럼 사용되는 메모리의 크기에 따라 균형이 맞춰진다.

1. root→ main 은 이미 할당 되어 있다. 그러므로 *current_bf*는 0이다.
2. main → fun1 은 이미 할당 되어 있다. 그러므로 *current_bf*는 3이다.
3. fun1 → fun3 의 BF는 3이다. 그전 단계에서 *current_bf*는 양의 값이고 X메모리의 activation 크기가 더 크기 때문에 이 activation은 (8,11)로 교환되어야 한다. 그러면 current bf 는 0이 되고 *recal* node는 fun3이 된다.
4. fun3 → fun5의 BF는 2이다. *current_bf*의 값이 0으로 듀얼타임 스택의 균형이 정확하게 잡혔기 때문에 pre-assigned 뱅크가 할당된다.

위에서 언급했듯이 교환된 activation이 있으면 그 아래 노드들의 BF를 다시 계산 해야 한다. fun1 → fun3에서 교환이 일어났으므로 fun3 activation의 모든 자손들의 BF를 다시 계산해야 한다. 그 결과는 그림 10에 나타나있다.

세 번째 계산에서 경로 root → main → fun2 의 균형을 잡는다. 그러나 root → main 이미 할당되어있고 BF가 0이기 때문에 main → fun2 activations은 pre-assigned 메모리 뱅크에 할당된다. 세 번째 계산을 끝으로 루트부터 단말 node까지의 모든 경로를 처리하여 X나 Y메모리 뱅크에 모든 activation을 할당함으로써 우리의 알고리즘은 종료된다.

그림 11의 그래프는 그림 6의 프로시저 호출에 따른 런타임




(그림 11) 알고리즘 실행 후 결과 비교

메모리의 사용을 나타내고 있다. Y축의 words는 프로시저가 호출되고 종료됨에 따라 변하는 두 메모리 뱅크 중 더 큰 activation 크기를 나타낸다. 이 그래프는 두 메모리 뱅크의 activation의 균형을 맞추는 것이 런타임 메모리의 사용을 감소시킨다는 것을 증명하고 있다.

## 4. 실험 결과

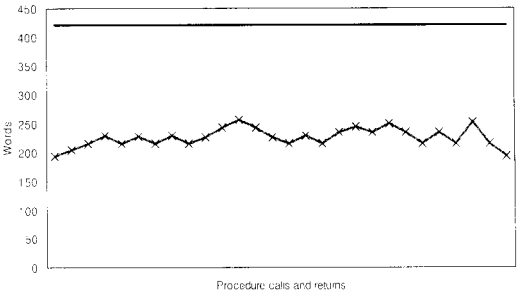
런타임 메모리에 관한 우리 알고리즘의 성능을 측정하기 위해 컴파일러에 알고리즘을 적용한 다음 상용 Motorola DSP56300[3]를 가지고 MediaBench benchmark의 IDCT, G721 encoder, G721 decoder[4]를 실험했다. 성능은 프로시저 호출에 따른 런타임 메모리의 사용량으로 측정하였다. 이번 장에서는 실험으로 얻어진 성능을 제시하고 다른 작업들과 실험 결과를 비교하였다.

### 4.1 메모리 사용량 측정

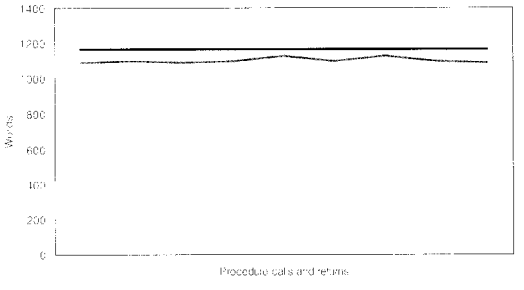
우리의 알고리즘의 효율성을 증명하기 위해 SPAM컴파일러[9]의 완전 정적 듀얼 런타임 환경 (fully static dual run-time environment)과 우리의 런타임 환경을 비교하였다. SPAM을 가지고 비교함으로써 완전 정적 듀얼 런타임 환경과 대립되는 우리의 알고리즘의 장단점을 설명할 수 있었다. 특히 메모리 뱅크 할당에 대한 영향을 알아보기 위해 SPAM 컴파일러로부터 완전 정적 듀얼 런타임 환경만을 발췌하였다. 즉 우리 알고리즘을 적용했을 때 메모리 뱅크 할당 결과에서 X, Y 런타임 스택의 정적 크기만을 계산하였다.

그림 12는 IDCT 벤치마크 프로그램에 대한 결과 그래프이다. X축은 새로운 함수의 호출 또는 귀환을 의미하며 Y축은 X와 Y 런타임 메모리 사용량의 합을 나타낸다. 완전 정적 런타임 스택은 1165 words를 사용하였고 우리의 알고리즘을 적용했을 때는 1125 words를 사용하였다. 이 벤치마크 프로그램은 전역/정적 공간으로 1025 words를 사용하기 때문에 결과적으로 스택 영역은 27.1% 절약되었고 전체 메모리는 3.3% 절약되었다.

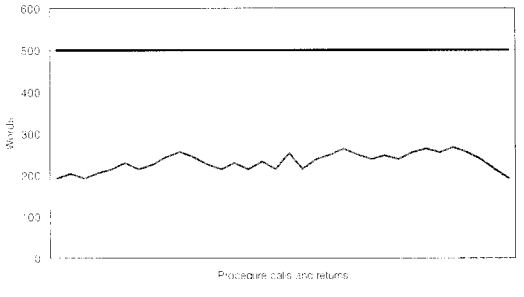
IDCT 벤치마크 프로그램의 경우 1024의 short 타입 배열이 전역 변수로 존재한다. 현재 하나의 배열은 메모리 뱅크 할당 시에 한 메모리 뱅크에 할당이 되기 때문에 시작부터 불균형이 심화되어 나머지 코드로 균형을 맞추기에 힘들다. 따라서 완전정적 듀얼 런타임 환경과 비교하여 큰 이득이 없게 된다. 그러나 다음의 두 벤치마크 프로그램의 경우에는 위와 다른 결과를 보인다.




(그림 12) IDCT 벤치마크 프로그램




(그림 13) G721 encoder 벤치마크 프로그램



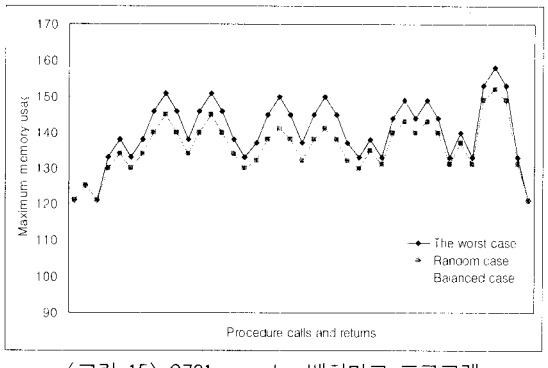

(그림 14) G721 decoder 벤치마크 프로그램

두 번째 벤치마크 프로그램은 g721 encoder이다. 완전 정적 런타임 스택은 421 words가 필요했고 우리의 실행결과는 257 words가 필요했다. 이 벤치마크 프로그램은 전역/정적 공간으로 80 words를 사용하므로 스택 영역은 48% 절약되었고 전체 메모리는 39% 절약되었다.

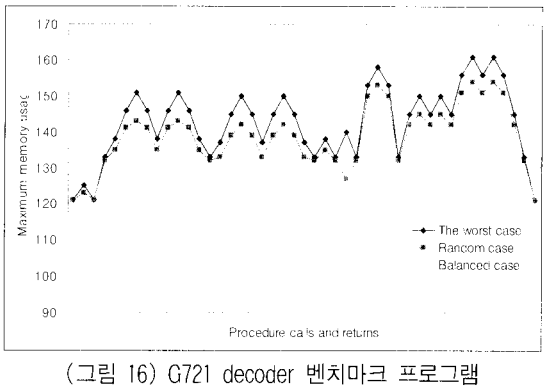
세 번째 벤치마크 프로그램은 g721 decoder이다. 완전 정적 런타임 스택은 500 words, 우리의 실행결과는 272 words를 필요로 했다. 이 벤치마크 프로그램의 경우 전역/정적 공간으로 80 words를 사용하기 때문에 스택 영역은 54% 절약되었고 전체 메모리는 45.6% 절약되었다.

### 4.2 런타임 메모리 최적화

이 절에서는 메모리 뱅크를 할당하였을 때 최악의 경우, 랜덤으로 생성된 경우, 그리고 우리의 밸런싱 알고리즘을 사용한 경우에 대하여 런타임 메모리의 최대 사용량을 비교한 결과를 제시한다. 최악의 경우는 큰 activation은 X 메모리 뱅크에 모두 할당을 하고 작은 activation은 모두 Y 메모리 뱅크에 할당한 경우를 의미한다. IDCT 벤치마크 프로그램은 전역 변수의 크기가 크기 때문에 알고리즘의 효과가 상대적으로 작아서 g721 encoder와 decoder 프로그램을 가지고 실험하였다. 이 실험에서




(그림 15) G721 encoder 벤치마크 프로그램




(그림 16) G721 decoder 벤치마크 프로그램

프로시저가 호출되고 소멸될 때 즉 런타임 스택이 변할 때 마다 X 또는 Y 런타임 메모리의 최대 사용량을 계산하였다. 그림 15와 그림 16은 실험 결과 그래프이다. 우리의 알고리즘을 적용하여 런타임의 순간 최대 메모리 사용량이 9%(g721 encoder)에서 10%(g721 decoder)까지 감소된 것을 알 수 있다.

## 5. 결 론

많은 DSP 공급업자들은 어플리케이션이 두 메모리 뱅크에 동시에 접근할 수 있도록 두 개의 데이터 메모리 뱅크를 제공한다. 이전 논문에서 우리는 다중 메모리 뱅크 구조를 이용하기 위한 분리된 접근(decoupled approach)을 주장하였고 본 논문에서는 온칩 메모리를 효율적으로 사용하기 위해 런타임 메모리 최적화 알고리즘을 제안하였다. EAT에서 균형을 맞추기 위한 콜 체인을 선택해주고, 선택된 콜 체인 안에서 AR간의 균형을 맞추어준다. X와 Y 메모리 사용간의 균형을 맞춘다는 것은 X와 Y 메모리 뱅크의 최대 메모리 사용량을 줄이는 것을 의미하기 때문에 결국 온칩 메모리 사용 기회를 늘리는 것과 같은 뜻이 된다. 실험을 통해 우리의 런타임 메모리 최적화 알고리즘이 SPAM 프로젝트에서 수행된 완전 정적 런타임 환경에 비해 적은 런타임 메모리를 사용하여 온칩 메모리를 더 효율적으로 사용할 수 있음을 증명하였다.

## 참 고 문 헌

[1] Aho, A. V., Sethi, R., and Ullman, J. D. 1986. Compilers -Principles, Techniques, and Tools. Addison-Wesley Publishing Company.

[2] Cho, J., Paek, Y., AND Whalley, D. 2004. Fast Memory Bank Assignment for Fixed-Point Digital Processors. and Transactions on Design Automation of Electronic Systems, Vol.9, No.1, pp.52 - 74, Jan.

[3] Motorola Inc. 1999. http://www.motorola-dsp.com. DSP56301 User's Manual.

[4] Lee, C., Potkonjak, M., and Mangione-Smith, W. 1997. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In Proceedings of the 30th Annaul IEEE/ACM Internation Symposium on Microarchitecture, pp.330 - 335, Nov.

[5] Leupers, R. and Kotte, D. 2001. Variable partitioning for dual memory bank DSPs. In Proceedings of the IEEE International Conference on Acoustics Speech and Signal Processing. 1121 - 1124.

[6] Liem, C. 1997. Retargetable Compilers for Embedded Core Processors. Kluwer Academic Publishers.

[7] Saghir, M. A. R., Chow, P., and Lee, C. G. 1996. Exploiting Dual Data-Memory Banks in Digital Signal Processors. ACM SIGOPS Operating Systems, pp.234 - 243.

[8] Sudarsanam, A. and Malik, S. 2000. Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs. ACM Transactions on Design Automation of Electronic Systems, Vol.5, No.2, pp.242 - 264, April.

**조 정 훈**

e-mail : jcho@ee.knu.ac.kr
1996년 KAIST 전기및전자공학과(학사)
1998년 KAIST 전자전산학과(석사)
2003년 KAIST 전자전산학과(박사)
2003년~2005년 하이닉스 반도체 선임연구원
2005년~현재 경북대학교 전자전기컴퓨터학부 전임강사
관심분야 : 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, reconfigurable architecture

**백 윤 흥**

e-mail : ypaek@snu.ac.kr
1988년 서울대학교 컴퓨터공학과(학사)
1990년 서울대학교 컴퓨터공학과(석사)
1997년 UIUC 전산학과(박사)
1997년~1999년 NJIT 조교수
1999년~2003년 KAIST 전자전산학과 부교수
2003년~현재 서울대학교 전기컴퓨터공학부 부교수
관심분야 : 임베디드 소프트웨어, 임베디드 시스템 개발도구, 컴파일러, MPSoC

**권 수 현**

e-mail : yukino4u@ee.knu.ac.kr
2006년 경북대학교 전자전기컴퓨터학부(학사)
2006년~현재 경북대학교 전자공학과 석사과정
관심분야 : 임베디드 소프트웨어, 컴파일러, reconfigurable architecture