논문 2013-08-32

직접 메모리 접근 장치에서 버스트 데이터 전송 기능의 효과적인 활용

(Efficient Utilization of Burst Data Transfers of DMA)

이 종 원, 조 두 산*, 백 윤 흥
(Jongwon Lee, Doosan Cho, Yunheung Paek)

Abstract : Resolving of memory access latency is one of the most important problems in modern embedded system design. Recently, tons of studies are presented to reduce and hide the access latency. Burst/page data transfer modes are representative hardware techniques for achieving such purpose. The burst data transfer capability offers an average access time reduction of more than 65 percent for an eight-word sequential transfer. However, solution of utilizing such burst data transfer to improve memory performance has not been accomplished at commercial level. Therefore, this paper presents a new technique that provides the maximum utilization of burst transfer for memory accesses with local variables in code by reorganizing variables placement.

Keywords : Memory system, Code optimization, Embedded system, Optimizing compiler

Ⅰ. 서론

내장형 시스템에 있어서, 메모리는 실행시간과 전력소모량에 가장 큰 영향을 미치는 요소 중 하나이다. 메모리 시스템의 성능이 메모리 접근 지연(memory access latency)에 큰 영향을 주기 때문이다. 따라서 메모리 시스템은 최소의 접근 지연을 제공할 수 있도록 최적화 되어야 한다. 최적화된 접근 지연 성능을 제공하기 위하여 많은 상용 내장형 시스템들이 직접 메모리 접근 장치 (DMA)를 채택하고 있다[1-3]. 오프칩 메모리 (off-chip memory)에서 온칩메모리 (on-chip memory)로 혹은 온칩메모리에서 레지스터로 많은 데이터가 이동할 때 CPU 부담이 커지는데, DMA가 지원되면 데이터를 블록단위로 이동시키기 때문에 CPU의 개입이 필요없게 된다. CPU에서는 데이터 이동이 완료되었다는 한번의 인터럽트만 발생한다. 따라서 데이터가 전송되는 동안 CPU는 다른 작업을 수행할 수 있게 되어 효율성이 높아진다. 그림 1에 본 연구에서 고려하고 있는 타겟 아키텍처를 나타내었다.

일반적으로 DMA의 블록 데이터 전송을 버스트 모드 (burst mode)라 한다 [4-6]. 버스트 모드 전송은 오프칩메모리에서 온칩메모리(혹은 레지스터)로 데이터 전송시 첫 번째 데이터 주소를 사용하여 연속된 데이터를 고속으로 전송한다. 따라서 두 번째 데이터부터는 별도의 셋업시간 (setup time= RAS+ CAS)이 필요 없기 때문에 순차적인 메모리 접근 시간 대비 버스트 전송이 약 2.7배 빠르다.

예를 들면, IBM의 Cu-11 [4] 내장형 DRAM의 일반 접근 시간은 10ns, 버스트 모드에서 5ns 정도가 소요된다. 소모되는 활성전류는 싸이클 당 각각 60mA/Mb, 13mA/Mb이 소요된다. 따라서 버스트 모드를 데이터 전송에 적극적으로 활용하는 기술의

*Corresponding Author (dscho@scnu.ac.kr)
Received: 28 Mar. 2013, Revised: 25 Mar. 2013, 12 June 2013, Accepted: 12 June 2013.
D.S. Cho: EE, Sunchon National University
J.W. Lee, Y.H. Paek: ECE, Seoul National University
※ 본 연구는 교육과학기술부/한국과학재단 우수연구센터 육성사업(No. 2012-0000470), 2012년도 정부(교육과학기술부)의 재원으로 한국과학재단의 국가지정연구실사업(No. 0421-2012-0047), (재) 스마트 IT 융합 시스템 연구단 (글로벌프론티어사업) ((재) 스마트 IT융합 시스템 연구단 0543-20110012), 한국연구재단의 기초연구사업 (No. 2010-0024529) 및 IDEC의 지원을 받아 수행된 것임

ISSN : 1975-5066
http://dx.doi.org/10.14372/IEMEK.2013.8.5.255

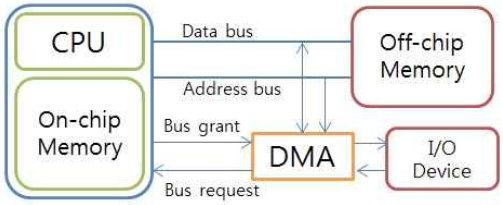

그림 1. 타겟 아키텍처 구조
Fig. 1 A structure of target architecture

개발은 시스템의 성능 및 전력 소모 최적화 측면에서 필수적이다.

불행하게도, 기존의 메모리 시스템은 DMA의 버스트 모드를 제대로 활용하지 못하고 있다. 버스트 모드로 데이터를 전송하기 위해서는 전송 대상이 되는 데이터들이 메모리 주소공간에 순차적으로 배열되어 있어야 한다. 기존 시스템은 프로그램 실행 흐름을 따라 전송 대상이 되는 데이터를 분석하고 이를 순차적으로 배치하는 효율적인 알고리즘이 없었다.

데이터의 배치는 프로그램 소스코드를 컴파일할 때 컴파일러에 의하여 결정된다. 기존의 상용 컴파일러들 (ARM application development suite, TI code composer, GCC ARM compiler)은 데이터 배치를 다음 두 가지 방식으로 결정한다.

1. 변수들의 이름순 (alphabetical order)
2. 선언된 순서순 (defined order)

결과적으로 이러한 방식으로는 버스트 모드 전송이 최적으로 생성되지 못한다. 유사한 시간에 접근되는 데이터들이 순차적으로 메모리에 놓이지 못할 확률이 높기 때문이다. 본 연구에서는 컴파일러가 블록 전송의 대상이 될 수 있는 데이터를 분석하고, 이들이 메모리 공간에 순차적으로 배치되도록 보장하는 알고리즘을 제안하였다.

II. 관련 연구

변수들의 메모리 위치(Memory Offset)을 조정하여 코드 크기를 줄이는 문제는 Bartley에 의해 처음으로 논의되었다. 그는 DSP 프로세서가 제한된 종류의 주소 연산 모드를 제공하며, 특히 Pre-increment/decrement 혹은 Post-increment/decrement 주소 연산은 다른 ALU 연산과 병렬로 수행할 수 있음에 주목하였다. 그리고 이들 병렬 주소 연산 명령을 효과적으로 사용함으로써, 코드에 포함된 주소 연산 코드를 줄이는 기법을 제안하였다 [7]. S. Liao등은 Bartley가 제기한 문제를 SOA(Simple Offset Assignment), GOA(General Offset Assignment) 문제로 정형화하였으며, 이 문제가 최대 가중 경로 검색 (Maximum Weight Path Cover) 문제로 변환될 수 있음을 보임으로써 NP-Complete 문제임을 증명하였다. 그리고 그들은 이 문제에 대한 해결책으로 Kruskal의 최대 스패닝 트리 (Maximum Spanning Tree)에 기반한 휴리스틱 알고리즘을 제안하였다 [8, 9]. 이들 연구는 모두 SOA/GOA를 확장한 연구로서, 메모리 위치 재할당을 통해 코드 내에 존재하는 주소 연산 명령어들을 감소하는데 초점을 맞추고 있다. 반면에, 본 연구에서 논의할 문제는 메모리 재배치를 수행한다는 점에서는 비슷하지만, 그 목적이 버스트 모드 활용의 최적화에 있다는 점에서 차이가 있다.

메모리상의 변수 재배치에 관련된 가장 최근의 연구는 Purdue 대학의 Nandivada 등에 의해 수행된 연구이다 [10]. 그들은 레지스터 할당 과정에서 발생하는 Spill Code를 SDRAM에 최적으로 배치하는 문제에 대해 논의하였다. 핵심 내용은 레지스터 Spill에 의해 발생하는 변수들을 SDRAM의 주소 영역에 적절하게 배열함으로써 읽고/쓰는 작업이 블록 전송을 통해 이루어질 수 있도록 최적화하는 것이다. 이때 SDRAM의 블록전송 모드를 사용하게 되므로 메모리 접근 시간이 줄어들게 된다. 하지만 이들의 기법은 우리의 문제와 몇 가지 차이가 있다. 첫째 우리의 기법은 실행시간과 전력 소모량을 동시에 줄이는 것에 초점을 맞춘 반면 Nandivada 등의 연구는 실행 시간을 단축하는데 초점을 맞추었다[10]. 뿐만 아니라 우리의 기법이 일반적인 블록 전송을 생성하는데 비해 Nandivada 등의 기법은 두 개의 워드로 구성된 블록 전송만을 생성한다. 따라서, 그들의 기법은 본 연구에서 다루는 문제의 부분 집합이라고 할 수 있다. 둘째, 그들은 문제를 ILP(Integer linear programming)로 변환함으로써 해결하였다. 이 방법을 사용하면 최적의 해를 찾는 것이 가능하지만, 시간이 매우 오래 걸린다는 단점을 가지고 있다. 반면, 우리는 2단계로 이루어진 근사(Heuristic) 알고리즘을 고안함으로써 수행 시간의 문제를 해결하고자 하였다. 다음 3장에서 알고리즘을 자세히 논하도록 하겠다.

III. 버스트 전송의 생성

이번 장에서는 버스트 문제를 정의하고, 이를 해

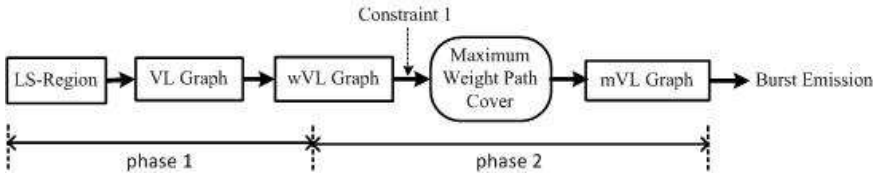

그림 2. 버스트 생성 기법의 흐름도
Fig. 2 A flow chart of the burst generation technique

결하기 위한 휴리스틱 알고리즘을 소개한다. 알고리즘은 2단계로 이루어져 있으며, 각 단계에서 요구하는 제한 조건들을 차례로 만족시킴으로써 문제를 해결한다.

1. 문제 정의

논의의 진행을 위해 먼저 병렬 데이터 로드(parallel data load)와 병렬 데이터 스토어(parallel data store)를 다음과 같이 정의한다.

정의 1. 병렬 데이터 로드/ 병렬 데이터 스토어

병렬 로드: 프로그램 코드에서 동시에 실행이 가능한 로드 (load operation) 연산들의 집합

병렬 스토어: 프로그램 코드에서 동시에 실행이 가능한 스토어 (store operation) 연산들의 집합

버스트 전송을 생성하는 문제는 프로그램에 존재하는 병렬 로드와 병렬 스토어를 찾아내어 이것들을 블록전송으로 병합하는 문제로 생각할 수 있다. 하지만 이 병합을 생성하기 위해서는 다음과 같은 제한조건을 만족해야만 한다.

제한 조건 1(Memory sequence constraint).

버스트 전송에 포함되는 n개의 변수는 메모리 주소 공간 상에서 순차적으로 배치되어야 한다. 즉, 첫 번째 변수가 Var[offset]이라면 다음 변수는 Var[offset + type*1], Var[offset + type*2], ... , Var[offset + type*(n-1)]의 순서로 메모리 주소공간에 위치한다. offset은 첫 변수의 주소, type은 개별 데이터의 크기를 나타낸다.

위의 제한 조건을 이용하여, 우리는 버스트 전송문제를 다음과 같이 정의할 수 있다.

정의 2. 버스트 전송 생성 문제

버스트 생성문제는 프로그램에 존재하는 병렬 로드와 병렬 스토어를 찾아내어, 제한 조건 1을 만족시킴으로써 이들을 최소 개수의 버스트 전송으로 변환하는 문제이다.

이 문제는 불행하게도 NP-Hard 문제이다. Nandivada 등은 버스트 문제의 부분집합인 더블워드 버스트 생성(burst transfer in double words) 문제를 연구하였는데, 이를 위한 메모리 위치 할당(memory offset assignment) 문제가 그래프상에서 해밀턴 패스 (Hamilton path) 찾기 문제와 동일하다는 것을 보임으로써 NP-Complete임을 증명하였다 [8]. 따라서 버스트 문제에 대해 최적의 해를 구하는 것은 불가능하므로 우리는 2단계로 이루어진 휴리스틱 알고리즘을 고안하였다. 그림 2은 버스트 생성 알고리즘의 전체 구조이다.

Phase 1에서는 각 로드/스토어 연산에 대해, 그 명령이 코드 상에서 움직일 수 있는 영역을 계산한다. 그리고 이 정보를 이용하여 병렬 로드와 병렬 스토어를 찾아내고, 이를 변수 위치 그래프 (VLG, Variable Location Graph)라는 자료 구조로 나타낸다. phase 2 에서는 메모리 옵셋 할당(memory offset assignment)을 수행하여 제한조건 1을 만족시키고, 마지막으로 버스트 모드 전송이 생성된다.

1.1 phase 1 - 로드/스토어 영역 및 그룹화

phase 1에서는 먼저 프로그램에 흩어져 있는 로드/스토어 연산을 분석하여, 서로 중첩하여 수행할 수 있는 로드/스토어 연산들을 병렬 로드/스토어로 모으는 작업을 수행한다.

이를 위해, phase 1에서는 제일 먼저 각 로드/스토어 연산에 대해 프로그램의 데이터 흐름을 해치지 않으면서 각 로드/스토어 연산이 코드상에서 움직일 수 있는 영역을 계산한다. 이 영역을 각 로드 가능 영역 (loadable region, L영역) 스토어 가능 영역 (storable region, S영역)이라고 명명하였다. L영역과 S영역은 다음과 같이 정의된다.

```
Algorithm 1 Build_GVL(Program P)
GVL ← empty set;
For each basic block B in P do
  RL ← compute_L_regions(B);
  RS ← compute_S_regions(B);
  R ← RL U RS
  While R is not empty do
    //arbitrary variables u, v in B, u≠v
    // intU : lifetime interval of the variable u
    //intU.ub (upper bound of the interval), intU.lb (lower bound of the interval)
    Select intV in R such that for all intU in R, intV.ub <= intU.ub;
    OP ← { intV }
    if intv is in RL then // Loadable Region
      Add to OP all the L-Regions overlapping with intV
    else // Storable region.
      Add to OP all the S-Regions overlapping with intV
    C ← build_complete_graph(OP);
    Add C to GVL
    R ← R − OP;
  od
od
```

그림 3. 변수 위치 그래프 구성 알고리즘

Fig. 3 An algorithm of variable placement graph construction

정의 3. L영역

프로그램의 한 베이직블록 B가 N개의 연산으로 이루어져 있다고 하자. B에서 변수 v에 대한 load 연산 "r=v"이 I번째 위치를 차지한다고 하면, 이 로드 연산에 대한 L-region int_V는 정수쌍 [lb, ub]로서 다음과 같이 정의된다. 여기서 r은 온칩메모리를 의미한다.

1) 만약 B에서 로드 연산 이전에 v에 값을 저장하는 스토어 연산이 존재하고, 이들 중 가장 나중에 수행되는 스토어 연산의 위치가 j라고 하면 int_V.lb = j+1이다. 만약 v에 스토어하는 연산이 없을 경우에는 int_V.lb = 0이다.

2) 로드 연산 이후, r을 처음으로 사용하는 연산의 위치를 k라고 하면 int_V.ub = k-1이 된다. 만약 B에서 r이 사용되지 않으면, int_V.ub = N-1 이다.

정의 4. S영역

프로그램의 한 베이직블록 B가 N개의 연산으로 이루어져 있다고 하자. B에서 변수 v에 대한 스토어 연산 "v = r"이 i번째 위치를 차지한다고 하면, 이 스토어 연산에 대한 S영역 int_V는 정수쌍 [lb, ub]로서 다음과 같이 정의된다. 여기서 r은 온칩메모리를 의미한다.

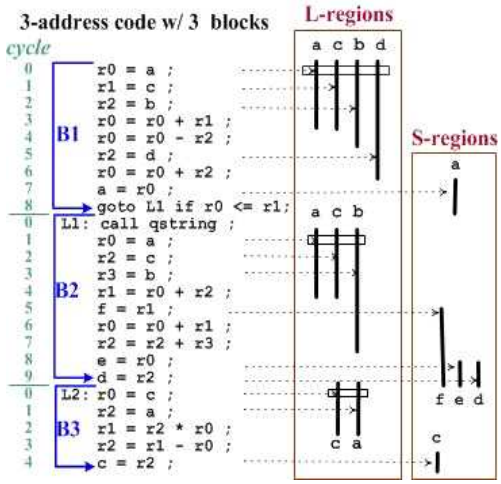

그림 4. L/S 영역 및 생성된 병렬 로드/스토어

Fig. 4 L/S region and generated parallel loads/stores

1) 스토어 연산이 사용하는 온칩메모리 r의 값이 마지막으로 정의된 명령의 위치를 j라고 하면 int_V.lb = j+1이다. 만약 r의 값을 정의하는 명령이 B에 없는 경우, int_V.lb = 0 이다.

2) 스토어 연산 이후, v를 처음으로 읽는 로드 연산의 위치를 k라고 하면 int_V.ub = k-1이 된다. 만약 이러한 로드 연산이 B에 존재하지 않으면, int_V.ub = N-1 이 된다.

Build_GVL() 알고리즘에서 L영역과 S영역은 각각 compute_L_regions()와 compute_S_regions() 루틴에 의해 계산된다. 그림 3은 임의의 한 예제 코드에 대해서 로드 가능 영역과 스토어 가능영역을 찾은 결과를 나타낸다.

논의의 진행을 편하게 하기 위해, 어셈블리 명령어를 3-주소 (address) 코드로 변환하여 표시하였다. 그림 4에서 베이직블록 B1을 살펴보자. B1의 코드를 보면, 변수 a의 값은 0번 위치에서 로드되어, 3번 위치에서 처음으로 사용된다. 따라서 변수 a는 3번 위치 이전에 로드되어야 하므로, 해당 L영역은 0-3이 된다. 변수 d의 경우, 5번 위치에서 로드되고, 6번에서 처음으로 사용된다. 하지만, 0~5번 위치 사이에 변수 d에 값을 기록하는 스토어 연산이 존재하지 않으므로, 변수 d는 0~6위치 어디에서 로드되어도 상관없다. 계산된 L영역과 S영역들을 살펴보면, 서로 영역이 겹치는 L/S영역들을 볼 수 있다. 서로 겹치는 영역들은 서로 간의 의존성이

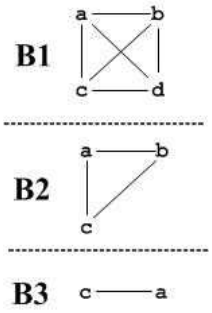

그림 5. 생성된 변수 위치 그래프 (VLG)
Fig. 5 Generated variable location graph

없기 때문에 동시에 수행되어도 프로그램의 수행 결과에 영향을 주지 않는다. 따라서 서로 영역이 중첩되는 L/S영역들은 병렬 로드와 병렬 스토어 로 묶을 수 있다. 묶인 병렬 로드/스토어들은 VL-Graph라 불리는 Complete Graph로 표현할 수 있다. VL-Graph에서 각 노드는 해당 변수에 대한 로드/스토어 연산을 나타내고, 에지는 명령들이 서로 동시에 수행될 수 있음을 의미한다. 그림 5는 위에서 찾아낸 병렬 로드들을 VL-Graph로 표현한 것이다.

위의 예제에서 생성된 병렬 로드는 총 3개이다. 이는 버스트 모드로 데이터 블록전송이 최소한 3회 이상 생성될 수 있음을 의미한다.

1.2 phase 2 - 메모리 위치 결정

Phase 1에서는 프로그램에 존재하는 병렬 로드/스토어를 찾아내어 이들을 VLG로 나타내었다. 하지만 병렬 로드/스토어를 버스트 전송으로 변환하기 위해서는 제한조건 1을 만족시켜야한다. Phase 2에서는 제한조건 1 (memory sequence constraint)를 만족시키기 위해 메모리 옵셋 할당 (Memory Offset Assignment) 작업을 수행한다.
제한조건 1은 버스트 전송의 대상이 되는 변수들이 메모리상에서 연속으로 배열될 것을 요구한다. 만약 병렬 로드/스토어를 구성하는 변수들이 이 조건을 만족시키지 못하면, 둘 혹은 그 이상의 버스트 전송으로 분할되어야 한다. 따라서 Phase 2에서 해결해야 하는 문제는 Phase 1에서 생성한 병렬 로드/스토어의 분할을 최소화 하면서 제한조건 1을 만족하도록 변수의 순서를 정하는 것이 된다.

정의 5. 메모리 옵셋 할당 문제

Phase 1에서 찾아낸 병렬 로드/스토어들의 분할을 최소화하도록 프로그램 변수들의 순서를 정하는 문제.

메모리 옵셋 할당 문제는 NP-Complete 문제이다. 이 문제의 특수한 경우로, 주어진 병렬 로드/스토어들이 모두 2개의 변수만으로 이루어졌다고 가정하면, 이 문제는 더블 워드 버스트 생성을 위한 메모리 순서 결정 문제와 같아진다. Nandivada 등은 이미 이 문제가 해밀턴 문제와 동일하다는 것을 보임으로써 NP-Complete임을 증명하였다[8]. 메모리 옵셋 할당 문제에 대한 최적의 해를 찾는 것은 불가능하므로, 우리는 휴리스틱 방법으로 이 문제를 해결하였다. 이를 위해 이 문제를 또 다른 NP-Complete 문제인 최대 가중 패스 찾기 (Maximum Weight Path Cover, MWPC) 문제로 변환하여 변수들의 순서를 결정하였다. 우리의 메모리 옵셋 할당 알고리즘은 다음과 같다.

Algorithm 2. Solver(Graph G_{VL}, Program P)

```
G_wVL <- Build_wVLGraph(G_VL);
mwp <- MWPCGen(G_wVL);
 // See Fig. 8. for MWPCGen()
E_NP <- Get_NonPathEdges(mwp, G_wVL);
 // Construct mVL Graph.
For all complete graphs C=(V_C, E_C)in G_VL do
  if (u, v) in E_NP is also in C then
    Remove (int_U, int_V) from E_C
od
  For every variable v in P do
    v.offset <- assign_offsets_in_memory(v,
mwp);
  od
return G_VL          // Return mVL Graph.
```

문제를 MWPC로 변환하기 위해 우리는 VLG를 통합하여 하나의 weighted Graph로 나타내었다. 이 그래프를 wVL Graph라고 하며, 다음과 같이 정의된다.

정의 6. Weighted VL(wVL) Graph

wVL Graph W = <N, E>는 다음과 같이 정의된다. N은 노드의 집합을, E는 에지의 집합을 나타낸다.

1) 집합 N은 프로그램 변수들의 집합이다.
2) E에 속하는 임의의 에지 e = (u, v)에 대해, 변수 u와 변수 어떤 VL Graph M =

<N', E'>이 존재하여, E'이 (intU, intV)를 포함한다.

3) E에 속하는 임의의 에지 e = (u, v)에 대해, e의 가중치는 (intU, intV)를 포함하는 VL Graph의 수를 나타낸다.

그림 6는 그림 5의 VLGraph에 대해 생성한 wVL Graph이다. wVL Graph에서 각 노드는 프로그램에서 사용되는 메모리 변수들을 나타내고, 각 에지 (u, v)의 가중치는 변수 u와 변수 v가 동시에 사용되는 회수를 나타낸다. 예를 들면, 변수 a, c는 B1, B2, B3의 병렬 로드에 의해 동시에 사용되므로, 3의 가중치를 가진다. Solver에서 이 작업은 Build_wVLGraph() 루틴에 의해 수행된다. 이 루틴은 VL_Graph(G_{VL})를 입력으로 받아 wVL Graph(G_{wVL})를 생성한다.

wVL Graph에서 우리는 직관적으로 동시에 사용되는 빈도가 높은 변수들을 연속된 메모리 주소 공간에 배열하는 것이 버스트 모드 전송을 좀 더 효율적으로 생성하는데 유리함을 짐작할 수 있다. 즉, wVL Graph에서 각 노드의 순서를 정하되 노드들을 연결하는 에지들의 가중치의 합이 최대가 되도록 순서를 정하는 것이다. 앞서 언급했듯이, 이 문제는 Maximum Weight Path Cover (MWPC) 문제로 알려져 있으며, S. Liao 등이 이미 NP-Complete임을 증명하였다. 그들은 또한 이 문제를 해결하기 위해 최대 스패닝 트리 (Maximum Spanning Tree)에 기반한 휴리스틱 알고리즘을 제안하였다[6, 7]. 제안된 Solver() 함수는 이 MST 기반 휴리스틱기반 알고리즘을 사용하여 변수들의 순서를 결정한다. 이 작업은 그림 8의 MWPCGen 함수에 의해 다음과 같이 수행된다.

- G_{wVL}를 입력으로 MWPCGen함수 호출.
- G_{wVL}의 모든 에지를 가중치를 기준으로 내림차순 정렬하여 리스트 F에 저장.
- 최대 가중 경로를 결정할 때 경로에 싸이클이 발생하지 않도록 검사하는 그래프 (Graph_cycle_check)를 초기화, V'는 G_{wVL}와 동일하며, E'는 공집합.
- F에 더 이상 남아 있는 에지가 없을 때까지 가중치가 높은 순서로 에지를 하나씩 꺼내어 E'에 추가, 이때 싸이클 (cycle)이 발생하면 e는 제거.
- 완성된 E'을 최대 가중 경로 솔루션으로 MWPC_record에 저장하여 리턴함.

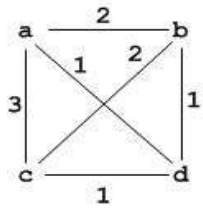

그림 6. 그림 5의 VLG에서 생성된 wVLG
Fig. 6 wVLG generated from VLG of Fig. 5

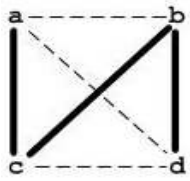

그림 7. 최대 가중 경로 (MWPC)
Fig. 7 Maximum weighted path cover

```
MWPCGen(GwVL)
{
  F = sorted list of edges of GwVL in
  descending order of weight;
  MWPC_record = 0;
  Graph_cycle_check(V',E') : V'=V of GwVL,
  E'={ };
  while (F not empty) {
     choose e = first edge in F
     F = F - {e}
     if(e does not cause a cycle in
     Graph_cycle_check)
        add e to E'
     else
        discard e;
  }
  /*construct an assignment from E'*/
  return MWPC_record = E';
}
```

그림 8. MWPC생성 알고리즘
Fig. 8 Algorithm of MWPC generation

그림 6의 G_{wVL}를 예제로 살펴보면, 가중치가 가장 높은 a-c가 꺼내어져 Path={a-c}로 저장된다. 다음은 a-b와 c-b가 가중치 2로 다음 경로의 대상

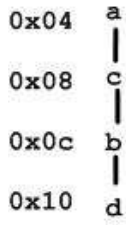

그림 9. 재할당된 메모리 옵셋
Fig. 9 Reassigned memory offset

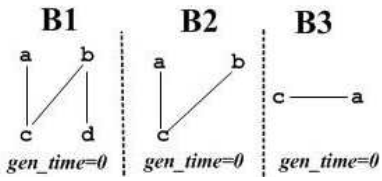

그림 10. 생성된 mVLG
Fig. 10 Generated mVLG

이 된다. 이때 a-c순서로 먼저 솔루션이 저장되어 있어 a-b는 솔루션 Path에 연결되지 못하여 제거되고, c-b가 Path={a-c-b}로 저장된다. 마지막으로 남은 d 변수로의 연결이 Path의 b변수에 직접 연결되어, 즉, b-d로 선택되어 Path={a-c-b-d}로 최종 솔루션이 완성된다.

그림 7에서 굵은 색으로 표시된 에지는 그림 6의 wVL Graph에 대해서 계산한 Maximum Weight Path(MWP)를 나타낸다. MWP에 속한 에지들을 mwp-edge, 그렇지 않은 edge들을 non-mwp edge라 부르기로 하자.

프로그램 변수들은 MWP에 의해 결정된 순서대로 메모리 위치를 할당 받는다. 그림 9은 예제 프로그램에 대해 MWP의 순서대로 메모리 주소를 할당한 결과를 나타낸다. MWP는 또한 메모리 위치 할당 작업을 수행한 결과 생성된 버스트 모드 전송을 계산하는데도 사용된다. mwp-edge에 의해 연결된 변수들은 서로 연속된 메모리 주소를 가진다. 따라서 우리는 mwp-edge들을 제외한 나머지 non-path edge들을 G_{wVL}로부터 제거함으로써 실제로 버스트 전송으로 변환될 수 있는 로드/스토어 그룹들을 찾아낼 수 있다. 이렇게 계산된 그래프를 mVL Graph라 부른다. 생성된 mVL 그래프의 Connected Component들은 서로 연속된 메모리 주소를 참조하는 병렬 로드/스토어를 나타내므로, 버스트 전송으로 변환이 가능하다.

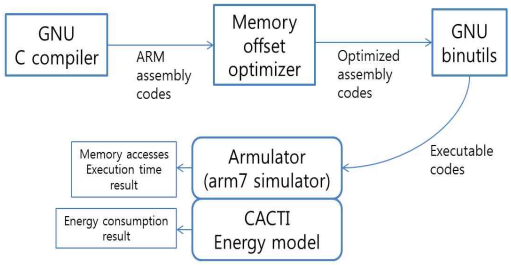

그림 11. 구현 상세
Fig. 11 Detailed Implementation

표 1. 벤치마크 특성
Table 1. Benchmark characteristics

program	acronym	total memory accesses
N_REAL_UPDATES	NRE	98
LMS	LMS	128
N_COMPLEX_UPDATES	NCO	201
STARTUP	STA	113
BIQUAD_N_SECTION	BIQ	167
FIR2DIM	FIR	261
MATRIX1	MAT	81
DOT_PRODUCT	DOT	33
CONVOLUTION	CON	49
FFT_BIT_REDUCT	FFT	299

그림 10에서 gen_time은 각 버스트 전송이 실행될 베이직블록에서의 위치를 나타낸다. 원칙적으로 각 mVLG에 포함되는 메모리 접근들의 공통 영역 상의 어느 위치에도 실행될 수 있다. 제안된 알고리즘은 파이프라인 해저드를 최소화하기 위해 버스트 로드는 공통 영역의 제일 앞부분에, 그리고 버스트 스토어는 공통 영역의 제일 마지막 부분에서 실행된다.

앞의 예제에서 계산된 MWP는 a-c-b-d이다. mVLG Graph는 3개의 Connected Component로 구성되어 있으며, 따라서 acbd, acb, ca 3개의 버스트 전송으로 9번의 로드를 마무리 할 수 있다.

Ⅳ. 실 험

제안하는 버스트 생성 기법이 실제의 경우에도 유용함을 평가하기 위해, ARM7 프로세서를 대상으로 실험을 수행하였다. ARM7 프로세서를

선택하는 이유는 내장형 시스템에서 가장 많이 사용되는 프로세서이고, 버스트 모드를 지원하기 때문이다 [11]. 버스트 생성 기법은 ARM으로 포팅된 GNU C 컴파일러의 포스트최적화 (Post Optimization) 단계로 추가하였으며, 독립적인 모듈로 구현하였다. 그림 11에 제안한 기법을 구현한 툴 (memory offset optimizer)과 함께 실험에 사용한 컴파일러/시뮬레이터의 연동 구성을 나타내었다. 구현된 버스트 생성기는 GNU C 컴파일러가 생성하는 ARM 7 어셈블리 코드를 입력으로 받아 지역 스칼라 변수에 대해 최적화를 수행하고 동일한 포맷으로 출력한다. 최적화된 어셈블리 코드는 GNU binutils (assembler, loader)를 통하여 실행가능 파일로 변환되고 arm7 시뮬레이터에서 실행하여 성능 및 메모리 접근 빈도 등의 정보를 추출하였다. 에너지 소모량 측정을 위해 CACTI [12] 에너지 모델에서 130nm 공정 파라미터를 사용하였다. 실험은 DSPStone [13]과 MediaBench [14] 벤치마크에서 선택한 개별 함수들에 대해 수행하였으며, 표 1에 선별된 프로그램과 메모리 접근 연산의 총 개수를 정리하였다. 시뮬레이션은 1GB 메모리를 가진 Pentium4 3.4GHz PC에서 진행하였다.

본 실험에서는 컴파일러가 생성한 최적화 전의 실행결과 (before)와 그 코드에 제안된 기법이 적용된 결과 (after)를 서로 비교하였다. 다른 최적화 기법과 독립적으로 제안된 기법의 효과를 측정하기 위해 컴파일러 최적화 옵션을 끄고 생성한 어셈블리 코드를 사용하였다.

그림 12와 13에 관련 실험 결과를 정리하였다. 그림 12는 제안하는 기법을 적용하여 생성된 버스트 개수를 나타낸다. 표의 하단에 나열된 각 프로그램 명칭에 대응하는 바그래프 중 왼쪽 (before)이 기존 컴파일러가 callee-caller 세이브를 위하여 생성한 기본 버스트의 개수이며, 오른쪽 (after)이 제안된 기법을 통하여 최적화된 버스트의 개수를 나타낸다. 각 바그래프 상단에 개수가 기록되어 있다.

기존 컴파일러에서는 버스트를 고려하여 메모리 옵셋을 할당하지 않기 때문에 제안된 기법의 결과가 탁월한 최적화 성능을 보이고 있다. 작은 코드 (DOT)에서는 두배, 큰 코드 (FFT)에서는 약 6배 많은 버스트가 생성되었다. 이러한 버스트 증가율은 로드/스토어 연산의 개수에 비례하는데, 메모리 연산 빈도가 높을수록 버스트 생성율이 높은 이유는 각 프로그램에서 사용하는 변수들의 접근들

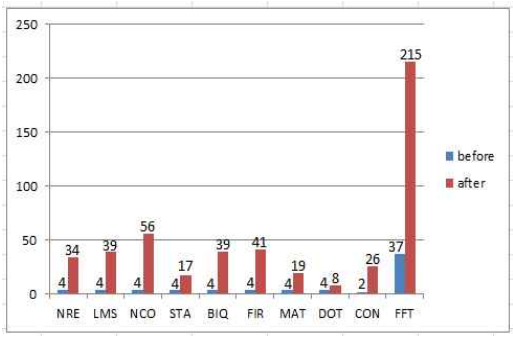

그림 12. 버스트 증감 결과
Fig. 12 Result of burst variation

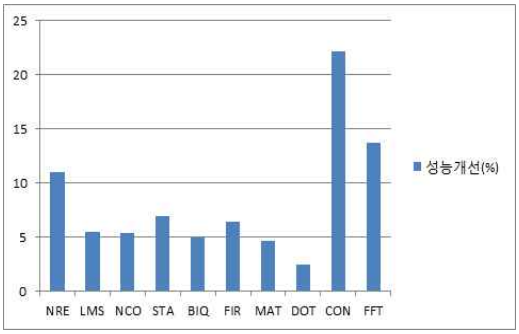

그림 13. 실행 시간 개선 결과
Fig. 13 Result of execution time improvement

(accesses) 사이에 지역성(locality)이 있기 때문으로 분석되었다.

또한 버스트 전송 증가율이 평균 7배에 달하는 원인은 전술하였듯이 기존 상용 컴파일러 (ARM compiler, TI's code composer)가 지역 변수의 스택 배치를 의미 없는 구성, 예를 들면 알파벳 순서 혹은 정의된 순서에 따라 결정하기 때문이다. 그리고 이러한 상용 컴파일러에서 블록 데이터 전송의 생성을 프로시져의 경계에서 변수들의 값을 일괄 전송하는 경우 (callee-caller save)에 제한되어 이용하는 것도 큰 원인이 되고 있다.

그림 13은 증가된 버스트를 통하여 감소한 메모리 접근 지연 시간이 전체 실행시간에 포함된 비중을 %로 나타내고 있다. 메모리 접근시간이 전체 실행시간에서 높은 비중을 차지하는 CON, FFT, NRE 등이 10% 이상 개선됨을 확인할 수 있다. 전체 실행시간의 평균 8.4%가 개선되었다. 하지만 버스트 전송 증가폭은 평균 7배인 반면 실행시간 개선은 평균 8.4%로 비례하여 크지 않다. 이러한 주

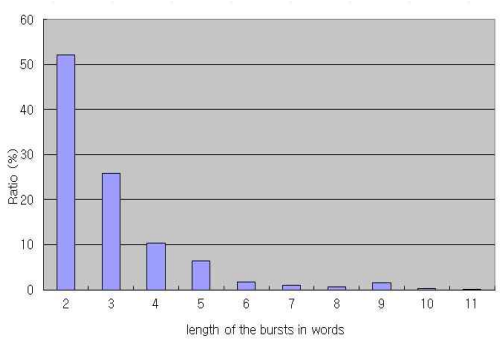

그림 14. 생성된 버스트의 크기 통계
Fig. 14 Stats of generated burst sizes

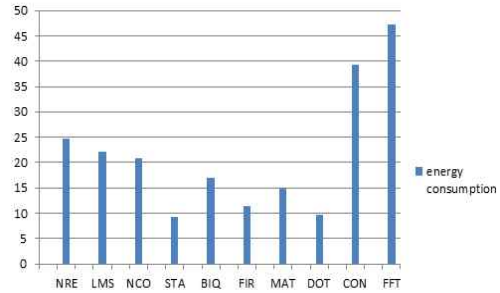

그림 15. 에너지 소모량 개선 결과
Fig. 15 Result of energy reduction

요 원인 중 하나는 제안된 기법을 통하여 생성된 버스트 전송이 주요 루프 코드 안에서 생성되는 경우가 적다는데 있다. 그리고 그림 14에 나타내었듯이 생성된 버스트 전송의 크기가 2~3워드로 작게 구성되는 경우가 약 80%를 차지할 정도로 높아서 실행시간 이득이 상대적으로 작게 나타난 것으로 확인되었다.

버스트 전송은 메모리 주소공간에 순차적으로 배치된 (공통된 행주소에 순차적인 열주소를 가진) 데이터를 행주소와 열주소 시그널 (RAS & CAS, Row Address Strobe & Column Address Strobe)을 생략하고 첫 변수 주소를 시작으로 블록으로 동시에 데이터를 전송한다. 본문에서 살펴본 예제의 경우 abcd순으로 배치되었을 때는 B1: (a, c, b, d)를 전송할 때 RAS+CAS 지연 시간이 4번의 로드 실행에 모두 포함된다. acbd순으로 배치되었다면 단 한번의 주소 시그널(RAS+CAS)로 버스트 전송이 활성화되어 4개 변수 전송이 완료된다.

따라서 2워드 버스트 전송이 50%이상 차지하고 있기 때문에 개선 가능한 RAS+CAS 지연은 첫주소를 제외한 1워드에만 해당되는 경우가 50% 이었다. 3워드 크기 이상의 버스트를 많이 포함하고 있는 CON, FFT가 상대적으로 큰 성능 개선을 나타내었다.

그림 15에 에너지 소모의 개선 정도를 나타내었다. before에서 생성된 코드를 실행하여 측정된 에너지 소모량을 기준으로 after 코드의 실행결과에서 개선 정도를 %값으로 하여 바(bar) 그래프로 나타내고 있다. 에너지 소모량 개선 정도는 비버스트 전송 (non-burst transfer) 대비 제안된 기법을 통하여 생성된 버스트 전송의 증가 폭에 비례한다. 이러한 이유는 버스트 전송을 사용하면 RAS (Raw Address Strobe)를 생략할 수 있다. 또한, 2워드 버스트의 경우 2번의 오프칩 메모리 접근을 1회로 줄일 수 있기 때문에 FFT의 경우 에너지 소모량을 최대 47%까지 절약하는 것이 가능함을 확인하였다. 최소 9%에서 최대47%까지 평균 21% 에너지 소모 절감을 얻을 수 있었다.

V. 결 론

본 연구에서는 내장형 시스템의 DRAM 성능을 개선하도록 지역 변수들을 대상으로 메모리 배치를 최적으로 재구성하는 기법에 대하여 논의하고 있다. 우리는 기존의 컴파일러들이 버스트 전송을 효과적으로 이용하지 못하고 있는 문제를 파악하여, 이를 최적화하는 기법을 제시하였다. 실험을 통하여 제안된 기법이 버스트 전송의 횟수는 평균 720%, 에너지 소모는 평균 21% 개선시키고, 실행시간은 평균 8.4% 개선시키는 것을 확인하였다.

References

[1] ARM, “CoreLink DMA Controllers,” Technical reference manual, 2009.

[2] J. Mangino, "Using DMA with high performance peripherals to maximize system performance," Texas Instrument report, 2007.

[3] Samsung, "Exynos 4 quad," technical document, 2012.

[4] J. Barth, J. Dreibelbis, E. Nelson, "Embedded DRAM design and architecture for the IBM 0.11um ASIC offering," IBM Journal of Research and Development, Vol. 46, No. 6, pp.675-680, 2002.

[5] Fujitsu, “FR80S/T series DMA access speed,” hardware manual AN07-00156-1E, 2008.
[6] STMicroelectronics, "STM DMA API," technical manual, 2011.
[7] D. Bartely, “Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes,” Software Practice & Experience Vol. 22, No. 2, pp.101-110, 1992.
[8] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, "Storage Assignment to Decrease Code Size," Proceedings on SIGPLAN Conference of PLDI, pp.186-195, 1995.
[9] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang, "Storage Assignment to Decrease Code Size," ACM TOPLAS, Vol. 18, No. 3, pp.235-253, 1996.
[10] V.K. Nandivada, J. Palsberg, "SARA: combining stack allocation and register allocation," Proceedings on International Conference on Compiler Construction, pp.232-246, 2006.
[11] ARM, "ARM architecture reference manual," 2007.
[12] P. Shivakumar, N.P. Jouppi, "CACTI 3.0: an integrated cache timing, power and area model," HP Labs, Palo Alto, CA, Technical Report, 2001.
[13] V. Zivojnovic, J.M. Velarde, C. Schager, H. Meyr, "DSPStone- A DSP oriented Benchmarking Methodology," Proceedings on International Conference of Signal Processing Applications and Technology, 1994.
[14] C. Lee, M. Potkonjak, W Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," Proceedings on IEEE International Symposium of Microarchitecture, pp.330-335, 1997.

저 자 소 개

이 종 원

2007년 서울대학교 전기공학부 학사.
2007~현재 서울대학교 전기공학부 박사과정.
관심분야: 컴파일러, 임베디드 시스템, VLIW, MPSoC, CGRA, soft error
Email: jwlee@sor.snu.ac.kr

조 두 산

2001년 한국외국어대학교 전자정보공학과 학사.
2003년 고려대학교 전기전자공학과 석사.
2009년 서울대학교 전기컴퓨터공학과 박사.
현재, 순천대학교 전자공학과 조교수.
관심분야: 임베디드 시스템, 최적화 컴파일러, 매니코어 시스템 설계.
Email: dscho@scnu.ac.kr

백 윤 흥

1988년 서울대학교 컴퓨터공학과 학사.
1990년 서울대학교 컴퓨터공학과 석사.
1997년 UIUC 전산과학 박사.
현재, 서울대학교 전기정보공학부 교수.
관심분야: 임베디드 소프트웨어, 컴파일러, MPSoC, CGRA, 클라우드 컴퓨팅, 보안
Email: ypaek@snu.ac.kr