다중 메모리 뱅크 구조를 위한 고속의 자료 할당 기법

조정훈
하이닉스 반도체
jhcho@soar.kaist.ac.kr

백윤흥 최준식
서울대학교 전기.컴퓨터학부
ypaek@ee.snu.ac.kr, jschoi@compiler.snu.ac.kr

Rapid Data Allocation Technique for Multiple Memory Bank Architectures

JEONGHUN CHO
Hynix Semiconductor

YUNHEUNG PAEK, JUNSIK CHOI
School of Electrical Engineering,
Seoul National University.

요 약

Virtually every digital signal processors(DSPs) support on-chip multi- memory banks that allow the processor to access multiple words of data from memory in a single instruction cycle. Also, all existing fixed-point DSPs have irregular architecture of heterogeneous register which contains multiple register files that are distributed and dedicated to different sets of instructions. Although there have been several studies conducted to efficiently assign data to multi-memory banks, most of them assumed processors with relatively simple, homogeneous general-purpose registers. Therefore, several vendor-provided compilers for DSPs were unable to efficiently assign data to multiple data memory banks, thereby often failing to generate highly optimized code for their machines. This paper presents an algorithm that helps the compiler to efficiently assign data to multi- memory banks. Our algorithm differs from previous work in that it assigns variables to memory banks in separate, decoupled code generation phases, instead of a single, tightly-coupled phase. The experimental results have revealed that our decoupled algorithm greatly simplifies our code generation process; thus our compiler runs extremely fast, yet generates target code that is comparable in quality to the code generated by a coupled approach.

Keywords:Compiler, dual memory, non-orthogonal architecture, memory assignment, graph coloring, and maximum spanning tree

1. 서론

최근의 내장형 프로세서들은 대부분 이종(heterogeneous) 레지스터와 다중 메모리 뱅크를 포함하는 비규칙적인 데이터 패스를 가진다. 한 예로, 그림 1은 Motorola DSP56000의 구조이다. 그림에서 보면 레지스터는 각 구성단위에 분산되어 해당 구성단위에서만 사용된다. 또한 다중 메모리 뱅크 구조에 의해 한 명령어 사이클에 두개의 데이터 워드에 접근할 수 있다. 그러나 불행히도 몇몇 컴파일러들은 ASIP의 주요 특징 중 하나인 다중 메모리 뱅크 구조를 충분히 지원하지 못하고 있다.

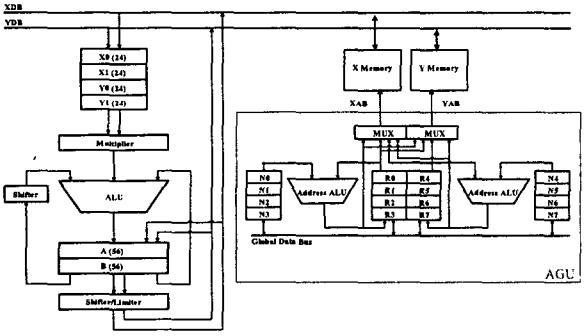


그림 1 듀얼 데이터 메모리 뱅크 X, Y를 가지는 모토롤라 DSP56000 데이터 패스

본 논문에서는 이 문제의 핵심인 레지스터 할당과 메모리 뱅크 할당에 대한 우리의 접근 방식을 기술한다. 이전의 연구[11,12]와 달리 우리의 알고리즘은 이종 레지스터 할당과 다중 메모리 뱅크 할당을 별개의 작업으로 나누어 수행한다. 그 결과, 우리는 컴파일 시간을 10^4의 비율로 줄이면서도 동일한 성능의 코드를 얻을 수 있었다. 2장에서는 먼저 연구에서 사용한 ASIP 구조를 살펴보고, 3장에서 우리의 알고리즘을 설명한 후, 4장에서 초기의 연구[4] 이후 가장 최근에 얻어진 실험결과를 제시하고 5장에서 결론을 맺도록 하겠다.

2. 타겟 머신 모델

이종 레지스터와 다중 메모리 뱅크 구조를 가지는 ASIP의 예제로 우리는 그림 1에서 보인 DSP56000을 사용했다. DSP56000은 다음과 같은 제한 조건을 가진다.

1. 두 워드는 다른 메모리 뱅크에 존재해야 한다.
2. 주소는 어드레스 레지스터를 사용하는 메모리 인디렉트 어드레싱 모드여야 한다.
3. 병행 이동에서 사용되는 각 어드레스 레지스터는 AGU의 두 레지스터 파일에서 서로 다른 집합에 있어야만 한다.

3. 레지스터 할당과 메모리 뱅크 할당

메모리 뱅크의 할당은 레지스터 클래스 할당과 압축이 끝난 후에 이루어진다. 코드 압축 단계 후의 코드는 그림 3과 같은 형태이다.

```
MOVE              a,r0  b,r1
MOVE              c,r2  d,r3
MAC   r0,r1,r2    e,r4  f,r5
MAC   r3,r4,r5    low(r2),low(v)
MOVE              high(r2),high(v)   low(r5),low(w)
MOVE              high(r5),high(w)
```

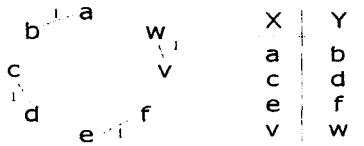
그림 3 그림 2로부터 압축 후의 코드

3.2 메모리 뱅크 배정

레지스터 클래스 할당과 코드 압축 단계를 거친 코드에서 변수들은 하나의 메모리 뱅크로만 배정되어 있다. 본 절에서는 두개의 잘 알려진 알고리즘을 사용하여 변수들을 다중 메모리 뱅크로 배정하는 기법을 제시한다.

3.2.1 MST 알고리즘 사용

먼저 어떤 변수들이 동시에 메모리 참조를 시도하는지를 나타내는 SRG(simultaneous reference graph)를 생성한다. SRG에서 각 노드들은 변수를 나타내고, 에지들은 하나의 명령에서 동시에 참조되는 변수들을 연결한다. 에지의 가중치는 변수들이 동시에 참조된 회수를 나타낸다. 병행 이동의 회수를 최대화하기 위해서는 동시에 참조되는 변수들을 가능한 한 다른 메모리 뱅크에 할당해야 한다. 이 때, 두 쌍의 변수들 사이에 충돌이 발생하면 빈도가 높은 변수 쌍이 더 높은 우선순위를 갖도록 하였다.



(a) SRG (b) 변수에 할당된 메모리 뱅크

```
MOVE              X: a, r0          Y: b, r1
MOVE              X: c, r2          Y: d, r3
MAC   r0, r1, r2  X: e, r4          Y: f, r5
MAC   r3, r4, r5  low(r2), X: low(v)
MOVE              high(r2), X: high(v)  low(r5), Y: low(w)
MOVE                                    high(r5), Y: high(w)
```

(c) 메모리 뱅크 배정

그림 4 그림 3의 코드를 위해 생성된 SRG와 이로부터 결정된 메모리 뱅크 배정 후의 결과

그림 4(a)는 그림 3의 코드를 위한 SRG이고, 그림 4(b)는 이 코드에 대해 최적화된 메모리 뱅크 할당을 보여준다. 그림 4(c)는 최종 결과 코드이다. 좀 더 현실적이고 복잡한 경우를 보이기 위해 5개의 변수들

을 가지고 있는 그림 5의 SRG를 생각하자. 메모리 뱅크 배정 문제는 SRG를 n개의 하위 그래프로 분할하는 문제로 바꾸어 생각할 수 있다. 이를 위해 우리는 정의 3과 같이 비용을 정의하였다.

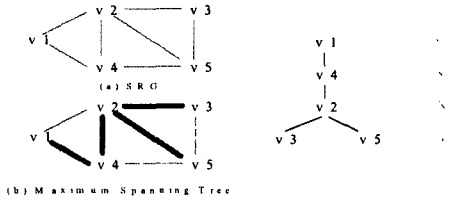


그림 5 SRG의 복잡한 예제와 그로부터 행해진 MST

정의 3: V와 E가 각각 노드와 에지들의 집합이라 할 때, 가중치를 가지고 연결되어 있는 그래프 G를 $G=(V,E)$라 표기하자. 또한 w_e를 에지 $e \in E$의 가중치라고 하자. 이 때, 그래프 G의 분리 방법 $P = <G_1, G_2, ...G_n>$을 찾는 문제는 G를 아래의 조건을 만족하는 n개의 공통부분을 가지지 않는 하위그래프 $G_i = (V_i, E_i)$, $1 \leq i \leq n$ 로 나누는 문제이다.

$$v_j, v_k \in V_i \text{ 이고 } (v_j, v_k) \in E \text{ 이면 } (v_j, v_k) \in E_i$$

이때 분리 방법 P의 비용은 다음과 같이 정의한다.

$$\sum_{i=1}^{n} \sum_{e \in E_i} w_e$$

최소 비용을 갖는 최적의 분할을 찾는 것은 NP-complete 문제이므로, 우리는 학습적인(heurisric) 방법으로 Prim의 MST (maximum spanning tree) 알고리듬[9] 이용하여 MST를 찾은 후 MST를 짝수 노드와 홀수 노드로 이분하여 메모리 뱅크를 할당하였다. MST는 SRG의 스패닝 트리 중 모든 에지의 가중치의 합이 최대인 것을 말한다. 우리의 알고리즘은 그림 6과 같으며 $O(E+VlgV)$의 시간 복잡성을 갖는다. 우리의 문제에서 E와 V는 비슷한 수를 가지기 때문에 알고리즘은 $O(V\lg V)$로 동작한다. 거의 모든 ASIP이 듀얼 메모리 뱅크를 가지므로 우리는 n의 값을 2로 가정하였으나 n이 2보다 큰 경우에도 쉽게 확장 가능하다. 불행하게도 MST로부터 만들어진 분리 방법이 항상 최적의 해를 생성하는 것은 아니다. 그렇지만 우리의 초기 작업[4]에 따르면 MST 개념이 적은 비용을 가지고 최적에 가까운 분리 방법을 생성한다는 것을 알 수 있다. 예로, 그림 5의 SRG에 대해 우리의 알고리즘은 최적의 분리방법을 찾아낸다.

Input: a simultaneous reference graph $G_{SR} = (V_{SR}, E_{SR})$
two memory banks M_X and M_Y
Output: a set V_{SR} whose nodes are all colored either with M_X or M_Y
a set E_{SR} whose edges are unselected in MST
Algorithm: DH A

```
S_T ← Q ← ∅;                          S_T is a set of MSTs and Q is a priority queue
for all nodes v in V_SR do unmark v;
u ← select_unmarked_node_in(V_SR);    Return ⊥ if every node in V_SR is marked
i ← 1; create a new MST T_i;
while u ≠ ⊥ do   // Find all MSTs for connected subgraphs of G_SR
   mark u;
   E_u ← the set of all edges incident on u;
   sort the elements of E_u in increasing order by wights, and add them to Q;
   while Q ≠ ∅ do
      remove an edge e = (u, z) with highest priority from Q;
      if z is unmarked then T_i ← T_i ∪ {e}; u ← z; break;
      if w is unmarked then T_i ← T_i ∪ {e}; u ← w; break;
   od
   if u is marked then   // All nodes in a connected subgraph of G_SR have been visited
      u ← select_unmarked_node_in(V_SR); Select a node in another subgraph, if any, of G_SR
      add T_i to S_T; i ← i+1; create a new MST T_i;
   fi
od
E_T ← Q;   // For generalized memory bank assignment
for all nodes v in V_SR do uncolor v;
for every MST T_i ∈ S_T do   // Assign variables in T_i's to memory banks M_X and M_Y
   next_visitors_Q ← ∅;
   m ← # of nodes in V_SR of M_X-color − # of nodes in V_SR of M_Y-color;
   select an arbitrary node v in T_i;
   if m > 0 then   // More nodes have been M_X-colored
      color v with M_Y-color;
   else   // More nodes have been M_Y-colored
      color v with M_X-color;
   repeat
      for every node u adjacent to v do
         if u is not colored then
            color u with a color different from the color of v;
            append u to next_visitors_Q;
         fi
      v ← extract one node from next_visitors_Q;
   until all nodes in T_i are colored;
od
m ← # of nodes in V_SR of M_X-color − # of nodes in V_SR of M_Y-color;
while m > 0 do   // While there are more M_X-colored nodes than M_Y-colored ones
   if ∃ uncolored node v ∈ V_SR then
      color v with M_Y-color; m−−;
   else break;
while m < 0 do   // While there are more M_Y-colored nodes than M_X-colored ones
   if ∃ uncolored node v ∈ V_SR then
      color v with M_X-color; m++;
   else break;
if m = 0 then
   for any uncolored node v in V_SR do
      color v alternately with M_X and M_Y colors;
return V_SR and E_T;
```

그림 6 듀얼 메모리 뱅크를 위한 메모리 뱅크 배정 알고리즘

3.2.2 이름 분할 및 통합.

우리는 메모리 할당 단위를 변수의 사용 범위로 세분화하면 MST 알고리듬을 향상할 수 있음을 발견하였다. 이를 위해 우리는 메모리 뱅크 간섭 그래프(memory bank interference graph)를 만들고, 이름 분할과 통합(name splitting and merging)이라 불리는 새로운 두 가지 기법을 도입하였다. 그림 7은 3.2.1의 방법으로 메모리 할당을 수행한 예이다.

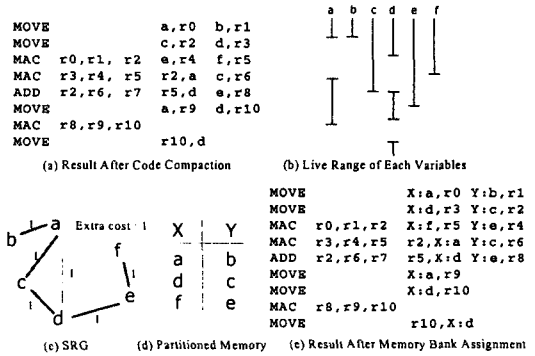


그림 7 이름 분할과 통합을 보이기 위한 코드 예제와 데이터 구조

그림 7(b)에서 변수 a와 d가 여러 사용 범위를 가지고 있음에 주목하자. 이 경우, a와 d는 같은 메모리 뱅크에 배정되어 있으므로 병행 이동으로 사용될 수 없음을 알 수 있다. 그림 8은 이름 분할 후 3.2.1의 알고리듬을 적용한 것이다.

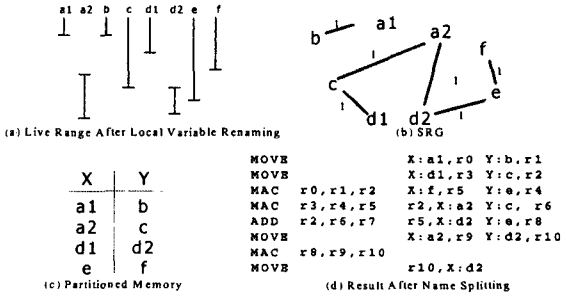


그림 8 로컬 변수를 위한 이름 분할

이 경우 기존 변수 d에서 분할 된 두 개의 변수 d1, d2가 다른 메모리 뱅크에 배정되는 것을 볼 수 있다. 이로 인해 그림 7에서 나타났던 변수 a와 d 사이의 충돌이 없어져 그림 8(d)에서 보듯이 두개의 move 명령어 대신에 병행 이동이 사용되었다. 즉, 병행 이동의 가능성이 높아졌다는 것을 확인 할 수 있다.

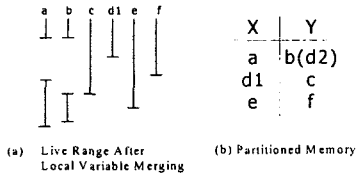


그림 9 로컬 변수를 이름 통합

이름 분할 방법은 코드 크기는 줄여주지만 더 많은 메모리를 사용한다. 이 문제를 해결하기 위해 이름 분할 후에 이름 통합 과정을 실행하였다. 그림 9는 그림 8의 예제에 이름 통합 과정을 적용한 결과이다.

3.3 레지스터 배정

변수에 대한 메모리 뱅크 할당이 끝난 후, 그래프 컬러링 알고리즘을 사용하여 물리 레지스터 할당 작업을 수행하였다.

4. 실험결과

최근에 이 문제는 Princeton과 MIT [1,12]에 있는 연구원들에 의해 행해진 SPAM 프로젝트에서 다루어졌다. 그래서 우리는 SPAM의 실험 결과와 우리의 실험 결과를 비교하였다.

그림 11의 결과는 우리의 메모리 뱅크 배정 알고리즘이 대부분의 경우에 있어서 SPAM의 알고리즘만큼 효율적이라는 것을 보여준다. 또한 그림 12로부터 우리의 컴파일 시간이 SPAM에 비해 대략 10^3에서 10^4정도 빠름을 알 수 있다. 실행 시간에서 코드 크기 감소의 영향을 측정하기 위해 우리는 **최적화되지 않은 코드, 컴파일러에 의해 최적화된 코드, 사람에 의해서 최적화된 코드**를 생성하였다. 그림 13은 실행시간에 대한 비교이다. 그림 13으로부터 우리는 최적화되지 않은 코드에 대해 컴파일러와 사람에 의해 최적화 된 코드가 각각 7.9%와 16.7%의 평균적인 속도 향상을 내는 것을 알 수 있다.

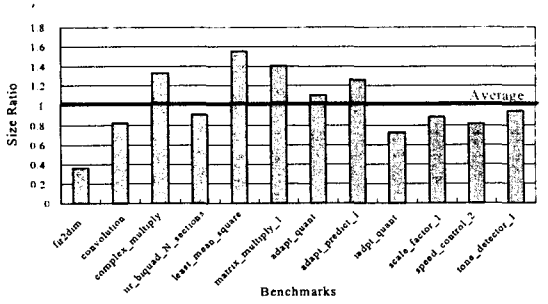


그림 11 SPAM 코드 크기와 우리의 코드 크기 비율

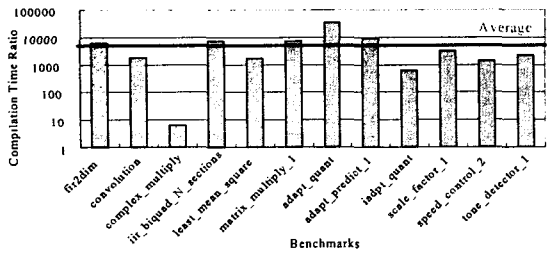


그림 12 SPAM 컴파일러와 우리 컴파일러의 컴파일 시간의 비율 (로그 스케일)

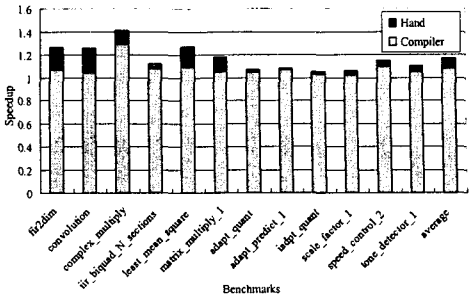


그림 13 최적화되지 않은 코드에 대한 컴파일러와 사람에 의해서 최적화 된 코드의 실행시간 속도 증가

5. 결론과 앞으로의 작업

본 논문에서 우리는 듀얼 메모리 구조를 지원하기 위해 여러 단계로 분리된 접근 방식을 사용하였다. 또한 부가적인 기법으로서 이름 분리와 통합이라는 두 가지 방법을 제시하였다. 그리고 실험 결과로부터 우리의 컴파일러가 코드 크기에 대하여 기존 연구와 필적할 만한 결과를 가지면서도 상당히 빠른 컴파일 속도를 보임을 확인하였다.

그러나 현재의 기법이 가지는 몇 가지 제한 사항 역시 발견되었다. 이 중 하나는 우리의 방식이 스칼라 변수에 제한되어 있어서 배열에 대한 연산에서는 큰 성능 향상을 기대할 수 없다는 것이다. 다른 제한점 중 하나는 함수 호출과 관련이 있다. 함수 호출시 메모리를 통해서 매개 변수를 전달할 때 호출된 함수에 맞게 메모리 뱅크들을 배정해야 한다. 그러나 이 기능은 호출자가 호출되는 함수의 메모리 접근 방식을 알아야 하기 때문에 함수간의 분석이 필요하다. 이 작업은 아직 구현되지 않은 것으로 추후 추가될 예정이다.

참고 문헌

[1] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sudarsanam, S. Tjiang, and A. Wang. *Challenges in Code Generation for Embedded Processors*, pages 48-64. In Marwedel and Goossens [9], 1995.

[2] G. Araujo and S. Malik. Code Generation for Fixed-point DSPs. *ACM Transactions on Design Automation of Electronic Systems*, 3(2):136-161, April 1998.

[3] G. Chaitan. Register Allocation and Spilling via Graph Coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 201-207, June 1982.

[4] J. Cho, J. Kim, and Y. Paek. Efficient and Fast Allocation of On-chip Dual Memory Banks. In *6th Workshop on Interaction between Compilers and Computer Architectures*, Feb. 2002.

[5] S. Jung and Y. Paek. The Very Portable Optimizer for Digital Signal Processors. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 84-92, Nov. 2001.

[6] R. Leupers and P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. In *Internaltional Conference on Computer-Aided Design*, 1996.

[7] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[8] Motorola Inc., Austin, TX. *DSP56000 24-Bit Digital signal Processor Family Manual*, 1995.

[9] R. Prim. Shortest Connection Networks and Some Generalizations. *Bell Systems Technical Journal*, 36(6):1389-1401, 1957.

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

[11] A. Sudarsanam. *Code Optimization Libraries For Retargetable Compilation For Embedded Digital Signal Processors*. PhD thesis, Princeton University Department of EE, May 15, 1998.

[12] A. Sudarsanam and S. Malik. Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):242-264, April 2000.

[13] V.Zivoljnovic, J.M. Velarde, C.Schager, and H. Meyr. DSPStone- A DSP oriented Benchmarking Methodology. In *Proceedings of International Conference on Signal Processing Applications and Technology*, 1994.