상위 단계 합성에서의 스케줄링 효과를 이용한 메모리 탐색

서재원 김태환
한국과학기술원 전산학과
jwseo@vlsisyn.kaist.ac.kr tkim@cs.kaist.ac.kr

Memory Exploration utilizing Scheduling Effects in High-level Synthesis

Jaewon Seo Taewhan Kim
Dept. of EECS, Korea Advanced Institute of Science and Technology

요 약

본 논문에서는 *상위 단계 합성(high-level synthesis)*에서의 메모리 탐색(exploration) 문제를 푸는 데 있어, 현존하는 메모리 합성 시스템들이 간과했던 한 가지 중요한 성질인 *메모리 탐색에서의 스케줄링 효과(scheduling effect)*를 말하고자 한다. 그리고 이 성질을 충분히 활용할 수 있는 새로운 형태의 통합된 알고리즘을 제안한다. 이 알고리즘은 메모리 구성(configuration)과 스케줄을 동시에 고려한다는 것을 가장 큰 특징으로 하는데, 몇 개의 벤치마크 필터 회로에 대한 실험을 통해 제안된 탐색 기법이 빠른 시간 안에 최적에 가까운 메모리 구성을 찾는다는 것을 보일 수 있었다.

1 서론

알고리즘적인 기술(algorithmic description)에서 출발하여 자동으로 RTL 설계를 생성해 내는 과정인 행동적 합성(behavioral synthesis)은 설계 자동화(design automation)에 있어서 가장 중요한 연구 과제 중의 하나이다. 많은 양의 데이터 연산을 포함하는 행동적 기술(behavioral description)에서는 데이터의 저장을 위해 주로 배열을 사용하게 되는데, 이러한 배열을 행동적 합성 단계를 거쳐 최종적으로 메모리 모듈에 의해 구현이 된다. 그러나 대부분의 합성 시스템에서 이 과정은 전적으로 무시되거나 디자이너의 *취향*에 따라 특정 메모리 모듈이 선택되도록 하고 있다. 따라서 메모리를 자주 사용하는 응용에서는, 전체 메모리 비용과 성능의 최적화를 위한 메모리 구성을 찾는 과정을 자동화해야 할 필요가 있다.

지금까지 행동적 합성 단계에 있어서 메모리 탐색 문제에 관한 많은 연구가 있어왔다. [1]에서는 전력소모 제약 하에서 최소의 면적을 갖는 혹은 면적 제약 하에서 최소의 전력소모를 갖는 메모리 구성을 찾기 위한 정수 선형 프로그래밍(integer linear programming) 모델과 휴리스틱(*heuristic*) 알고리즘이 제안 되었다. 그러나 이 방법에서는 고정된 스케줄을 입력으로 하기 때문에, 설계 공간(design space) 탐색에 있어서 피할 수 없는 제약을 갖게 된다. [2]에서는 메모리 구성에 대한 고정된 계층 모델에서의 탐색 기법을 제시하였다. 그러나 이 방법에서는 계층 모델의 종류에 따라 사용 가능한 메모리의 특성이 제약되기 때문에, 일반적인 메모리 라이브러리에 대한 탐색은 불가능하게 된다. [3]에서는 전체 성능과 메모리 비용 간의 타협 점을 고려하여 배열 변수의 구현을 위한 메모리 모듈을 할당하는 상향 클러스터링(bottom-up clustering) 기법에 기초한 메모리 합성 알고리즘을 제시하였다. 이 방법의 한 가지 단점은 서로 다른 액세스 속도와 포트의 개수를 갖는 메모리의 사용을 고려하지 않았다는 점이다. [4]에서는 메모리 액세스 순서를 이용하여 뱅크 안에서의 페이지 미스(page miss)를 최소화하는 알고리즘이 제안되었으나, 역시 다중포트(multi-port) 메모리에 대한 고려가 간과되었다. [5]에서는 실제적인 오프-칩(off-chip) 메모리 액세스 모드를 모델링 하고 이러한 액세스 모드에서의 메모리 액세스를 최적화하는 CDFG 변환 알고리즘이 제안되었다. 이 밖에 상위 단계 합성에서의 메모리와 배열 최적화를 위해 제시된 기존의 여러 방법들은 [6]에 잘 요약되어 있다.

본 논문에서는 *실제적이고 효율적인* 메모리 탐색 기법을 제시하였는데, 우리의 방법이 기존의 것들에 비해 갖는 가장 큰 특징은 메모리 탐색에 있어서의 스케줄링 효과를 처음으로 연계 시켰다는 점이다.

그림 1(a)는 각 16 (bits) × 1024 (words)의 크기를 갖는 네 개의 배열을 사용하는 행동적 기술의 일부이며, 그림 1(b)는 우리가 사용할 수 있는 메모리 모듈 라이브러리라고 가정하자. 그림 1(b)에서는 각 메모리 모듈에 따른 포트(read-only, write-only, read/write)의 개수와 차지하는 면적(메모리 비용)이 나타나 있다. (여기에서 각 메모리 모듈의 면적은 [1]에서 제시된 방법에 따라 계산되었다.) 잠정적으로 이 라이브러리에서 모든 메모리 액세스 연산은 1 클락 싸이클(clock-cycle)이 걸리며, DFG의 모든 연산은 3 클락 싸이클 안에 수행되어야 한다고 가정하자. 그림 2(a)는 그림 1(a)의 메모리 액세스 연산에 대한 하나의 스케줄 *schedule-1*과 그림 1(b)의 라이브러리를 사용했을 경우의 가능한 몇몇 메모리 구성을 보여준다. 여기서 우리는 배열 A, B, D가 하나의 메모리 모듈에 배정될 수 없다는 것을 쉽게 알 수 있다. 왜냐하면 이 세 배열을 하나의 메모리 모듈로 구현했을 경우 세 개의 읽기 액세스 연산이 클락 스텝(clock step) 1에서 동시에 수행되어야 하지만 라이브러리의 어떠한 메모리 모듈도 이러한 동시 수행을 가능케 하지 못하기 때문이다. 그림 2(a)의 표에서 굵은 글씨로 쓰여진 구성은 최소의 메모리 비용을 갖는 구성을 나타낸다. 그러나 만약에 우리가 읽기 연산 *read_D*[i]를 클락 스텝 1에서 클락 스텝 2로 재스케줄링(rescheduling)한다면, 그림 2(b)에서 보는 바와 같이 전체 메모리 비용을 26.55에서 22.46으로 더 줄일 수 있게 된다. 이 간단한 예제는 스케줄을 어떻게 하느냐에 따라 메모리 탐색의 질에 얼마나 달라질 수 있는가를 잘 보여주고 있다.

2 메모리 탐색 알고리즘

이 장에서 우리는 메모리 구성을 보다 *정확하고 효과적으로* 탐색하기 위해 스케줄링 효과를 고려하는 메모리 탐색 기법을 제시한다. 제시된 알고리즘은 스케줄 되지 않은 데이터 흐름 그래프(data flow graph, DFG)를 입력으로 받는다. $\mathcal{L}$을 메모리 라이브러리라 하고, DFG의 모든 연산은 T 클락 싸이클 안에 수행해야 한다고 하자. 여기서 나타나는 최적화 문제는 라이브러리 $\mathcal{L}$에서 메모리 모듈을

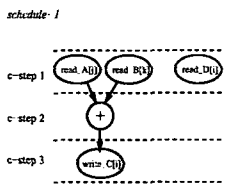
```
reg [15:0] A[1023:0], B[1023:0], C[1023:0];
reg [15:0] D[1023:0];
...
C[i] = A[j] + B[k];
t = D[i];
...
```

(a)

메모리 모듈	크기 (bits×words)	포트 수 r	w	r/w	비용 (면적: mm^2)
M1	16×1024	0	0	1	7.94
M2	16×1024	1	1	0	8.66
M3	16×2048	0	0	1	11.23
M4	16×2048	1	1	0	12.25
M5	16×2048	1	0	1	15.32

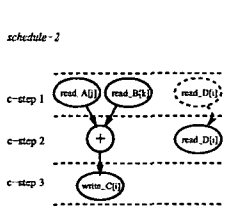
(b)

그림 1: (a) 행동적 기술의 일부분, (b) 메모리 모듈 라이브러리.



메모리 구성 (배열 그룹 : 메모리)	면적 (mm^2)
A:M1, B:M1, C:M1, D:M1	31.76
A:M1, B:M1, CD:M3	27.11
A:M1, C:M1, BD:M5	31.20
...	...
AB:M5, CD:M3	**26.55**
...	...

(a) *schedule-1*과 가능한 메모리 구성들



메모리 구성 (배열 그룹 : 메모리)	면적 (mm^2)
A:M1, B:M1, C:M1, D:M1	31.76
A:M1, B:M1, CD:M3	27.11
A:M1, C:M1, BD:M5	31.20
C:M1, D:M1, AB:M3	27.11
...	...
AC:M3, BD:M3	**22.46**
AB:M5, CD:M3	26.55
...	...

(b) *schedule-2*와 가능한 메모리 구성들

그림 2: 메모리 구성의 탐색에 있어 스케줄링 효과가 미치는 영향에 대한 예제

선택하여 이들을 주어진 DFG의 배열에 할당하되, T 클락 싸이클 안에 수행되어야 한다는 제약조건을 만족시키는 동시에 전체 메모리 비용을 최소화 시키는 것이다. 임의적으로 혹은 현존하는 메모리 합성 시스템에 의해 주어진 초기 스케줄과 초기 메모리 구성에 대해서, 우리는 동시에 재스케줄링과 재배정을 반복함으로써 결과의 질을 향상시킨다. 여기서 각 배열은 서로 다른 메모리 모듈에 배정되어 진다고 가정한다. 여러 개의 배열을 그룹핑 하여 하나의 메모리에 배정하는 메모리 클러스터링(clustering)은 [4]에서의 방법을 사용하여 후처리(post-processing) 과정으로 수행될 수 있다.

그림 3의 예제는 우리의 알고리즘이 매 수행마다 어떻게 연산들을 재스케줄링하고 새로운 메모리에 재배정 하는지 보여준다. 그림 3(a)은 초기 스케줄과 초기 메모리 구성을 보여준다. 여기서 각 엣지(edge)에 부여된 숫자는 두 메모리 연산을 수행함에 있어 그 사이의 비-메모리 연산을 수행하기 위해 필요한 최소한의 실행 시간을 나타낸다. (편의를 위해 그림 3(b)에서는 메모리 연산만을 표시하였다.) 각 메모리 연산에 대해 우리는 (각 엣지의 숫자들에 대한) 어떠한 타이밍 위반(timing violation) 없이 연산을 재스케줄링 할 수 있는 클락 스텝들의 집합 *schedule_zone*을 구할 수 있다. 각 연산 옆에 두꺼운 선으로 표시된 구간은 해당 *schedule_zone*을 나타낸다. 예를 들어, 그림 3(c)에서의 op_1에 대한

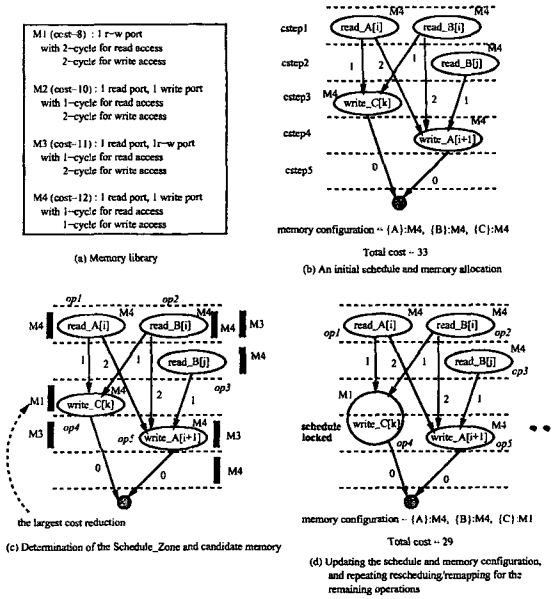


그림 3: 동시에 재스케줄링과 재배정을 하는 과정에 대한 예제

*schedule_zone*은 $\{cstep1\}$이며 (op_1을 *cstep2*로 재스케줄링 하는 것은 op_1과 op_4, op_1과 op_5사이의 타이밍 위반을 초래한다.), op_4에 대한 *schedule_zone*은 $\{cstep3, cstep4\}$이다. 각 연산 op_i에 대한 *schedule_zone*의 각 클락 스텝에 대해, 어떠한 타이밍 위반 없이 재배정 가능한 모든 메모리 모듈을 찾아 낼 수 있다. 이 때, 이 메모리 모듈 중에서 우리는 가장 적은 비용을 갖는 것을 선택한다. 어떤 메모리 구성 R에 대해 $X(i)$를 연산 op_i에 할당된 메모리 모듈, $Y(i,j)$를 앞서 말한 절차에 따라 op_i를 클락 스텝 j에 재스케줄링 했을 때 선택된 최소한의 비용을 갖는 메모리 모듈이라 하자. 그러면 우리는 다음의 값을 최대화 하게 만드는 연산과 클락 스텝을 선택할 수 있다.

$$\Delta C = C(X(i)) - C(Y(i,j)) \quad (1)$$

여기서 $C(M)$은 메모리 M의 비용 (혹은 면적)을 나타낸다. 그림 3의 예제에서, op_4를 선택하여 cstep3에 재스케줄링 하는 것이 가장 큰 메모리 비용 절감 효과를 가져온다. 이제 op_4는 고정되어 더 이상 재스케줄링과 재배정의 대상이 되지 않으며, 남은 연산들에 대한 재스케줄링과 재배정이 모두 이루어지기 까지 이 과정은 반복된다.

제시된 알고리즘의 전체 절차는 그림 4에 나타나 있다. 우선 입력으로 주어진 DFG에 대해 메모리 라이브러리 $\mathcal{L}$을 사용하여 초기 스케줄과 초기 메모리 구성을 생성한다. 바깥쪽 루프(outer-loop)를 돌면서 초기해(initial solution)는 점차적으로 개선되게 된다. 매 수행 마다, 우리는 수행 시간 T를 넘지 않으며 어떠한 타이밍 위반도 하지 않는 모든 (연산, 클락 스텝) 쌍에 대해서 메모리 비용을 최대한 절감시키는 것을 선택하여 재스케줄링과 재배정을 수행한다. 한 번 재스케줄링과 재배정이 수행되고 난 후에는 이 연산은 고정되어, 이 연산에 대한 재스케줄링과 재배정은 안쪽의 *while-loop* 안에서는 더 이상 허용되지 않는다. 바깥쪽 *while-loop*은 사용자가 정의한 횟수인 K번 까지 반복되고, 그 후로는 현재 까지 찾은 최소 비용의 해 보다 더 나은 해를 찾을 수 없을 때 까지 수행된다.

```
Memory Exploration with Scheduling (DFG, L, T) {
• DFG의 메모리 연산에 대한 초기 스케줄 S_init을 생성한다;
• S_init에 대한 초기 메모리 구성 R_init을 생성한다;
• COST_min = mem_tot_cost(R_init);
• S_best = S_init; R_best = R_init;
• i = 0;
while (R_best가 변경 되었다 OR i < K) { /* K: 반복 횟수 */
    • i = i + 1;
    while (고정되지 않은 배열이 있다) {
      foreach 고정되지 않은 배열 a_i {
        foreach 메모리 M_j {
            • a_i를 M_j에 재배정한다; /* 재배정 */
            • 재스케줄링을 시도한다; /* 재스케줄링 */
        } endforeach
        • 재스케줄링과 재배정을 원상복귀 시킨다;
      } endforeach
      • 테스트된 스케줄과 메모리 구성에 대해,
          최소 비용을 갖는 것을 선택한다;
      • 선택된 스케줄과 메모리 구성에 맞게
          재배정과 재스케줄링을 수행한다;
      if (COST_current < COST_min) {
        • COST_min, S_best, R_best를 업데이트한다;
      } endif
      • 선택된 배열을 고정시킨다;
    } endwhile
    • 고정된 모든 배열을 해제시킨다;
}endwhile
• Return (R_best, S_best);
```

그림 4: 스케줄링 효과를 고려하는 제안된 메모리 탐색 알고리즘

회로	|L|	제안된 알고리즘		Exhaustive		차이
		비용 (mm^2)	시간 (sec)	비용 (mm^2)	시간 (sec)	(%)
kalman	180	10.772	0.3	10.718	77	0.5
sor	200	371.134	0.1	371.134	20	0.0
phoneme	180	132.050	0.7	-	-	-
heat	180	17.103	0.2	17.103	27	0.0
ddpoly	200	8.349	0.1	8.349	32	0.0
tridag	200	24.816	0.1	23.309	24	6.1
평균						1.32

표 1: 우리의 알고리즘과 EXHAUSTIVE와의 메모리 구성 결과 비교

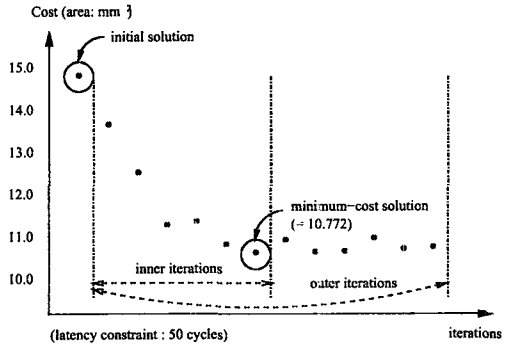


그림 5: kalman 회로를 우리의 알고리즘에 적용하였을 때, 매 수행에 따라 전체 메모리 비용이 감소하는 모습

3 실험 결과

제안된 메모리 탐색 알고리즘은 C++ 를 이용하여 구현하였으며, 실험은 Sun Sparc20 워크스테이션 환경에서 수행되었다. 우리의 탐색 기법이 얼마나 효율적인가를 알아보기 위하여 몇 가지 응용[7, 8, 9]에 대해서 실험이 설계되었다. 표 1은 우리의 알고리즘과 소모적인 탐색 방법인 EXHAUSTIVE와의 결과를 비교한 것이다. EXHAUSTIVE는 몇 가지 프루닝 테크닉(pruning technique)을 사용하여 최적성(optimality)를 잃지 않으면서도 빠른 시간 안에 수행되도록 구현하였다. 메모리 비용의 측면에서 우리의 방법은 최적해와 비교하여 평균적으로 1.32%의 근소한 차이를 보였으나, 수행 속도의 측면에서는 EXHAUSTIVE에 비해 매우 빠른 속도를 보여주었다. EXHAUSTIVE의 경우 메모리 라이브러리와 회로의 크기가 늘어날수록 수행 속도는 느려지게 되며, 표에서 phoneme회로의 경우 6시간 안에도 해를 구할 수가 없었다. 그림 5는 kalman회로에 대해 우리의 알고리즘을 적용시켰을 경우 매 수행에 따라 전체 메모리 비용이 어떻게 감소하는지를 나타낸다. 그림 5의 바깥쪽 *while-loop*은 2번 수행되었으며 안쪽 *while-loop*은 6번 수행되었다. 동그라미 쳐진 두 개의 점은 각 처음과 마지막 메모리 구성을 나타내는데, 이 경우 처음에 비해 20%의 비용 절감 효과를 얻었다.

4 결론

본 논문에서 우리는 상위 합성 단계에서의 스케줄링 효과를 고려한 메모리 탐색 문제를 풀기 위한 알고리즘을 제안하였다. 우리는 잘 설계된 비용 측정 방법과 타이밍 위반 체킹 기법을 사용하여 스케줄링과 메모리 배정을 동시에 반복적으로 수행함으로써 이 문제를 해결하였다. 실험 결과는 제안된 알고리즘의 수행 속도가 소모적인 탐색에 기초한 방법에 비해 매우 빠르면서 결과로서 얻은 메모리 구성의 질은 최적해에 비해 평균적으로 1.32%의 차이밖에 없다는 것을 보였다.

참고 문헌

[1] W.-T. Shiue, S. Tadas, and C. Chakrabarti, "Low Power Multi-Module, Multi-Port Memory Design for Embedded Systems", *Proc. of Signal Processing Systems*, pp. 529-538, 2000.

[2] N. Holmes and D. Gajski, "Architectural Exploration for Datapaths with Memory Hierarchy", *Proc. of EDAC*, pp. 340-344, 1994.

[3] L. Ramachandran, D. Gajski, and V. Chaiyakul, "An Algorithm for Array Variable Clustering", *Proc. of the EDAC*, pp. 262-266, 1994.

[4] P. R. Panda, "Memory Bank Customization and Assignment in Behavioral Synthesis", *Proc. of ICCAD*, pp. 477-481, 1999.

[5] P. R. Panda, N. D. Dutt, and A. Nicolau, "Exploiting Off-chip Memory Access Modes in High-Level Synthesis", *Proc. of ICCAD*, pp. 333-340, 1997.

[6] P. R. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg, "Data and Memory Optimization Techniques for Embedded Systems", *ACM Trans. on Design Automation of Electronic Systems*, Vol. 6 No. 2, pp. 149-206, April 2001.

[7] P. R. Panda and N. D. Dutt, "1995 high level synthesis design repository", *Proc. of International Symposium on System Synthesis*, pp. 170-174, 1995.

[8] Schmit, H., Thomas, D.E. "Synthesis of application-specific memory designs", *IEEE Trans. on VLSI Systems,*, Vol. 5, No. 2, pp. 101-111, March 1997.

[9] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, *Numerical Recipes in C*, Cambridge University Press, 1988.