MINIMIZING MEMORY REQUIREMENTS FOR CHAIN-STRUCTURED SYNCHRONOUS DATAFLOW PROGRAMS[1]

Praveen. K. Murthy, Shuvra. S. Bhattacharyya, and Edward. A. Lee

EECS Department, University of California, Berkeley, CA 94720
{murthy, shuvra, eal}@eecs.berkeley.edu

Abstract

This paper addresses trade-offs between the minimization of program memory and data memory requirements in the compilation of dataflow programs for multirate signal processing. Our techniques are specific to the synchronous dataflow (SDF) model [6], which has been used extensively in software synthesis environments for DSP; see for example [7][8]. We focus on programs that are represented as chain-structured SDF graphs. We show that there is an $O(n^3)$ dynamic programming algorithm for determining a schedule that minimizes data memory usage among the set of schedules that minimize program memory usage. A practical example to illustrate the efficacy of this approach is given. Some extensions of this algorithm are also given; for example, we show that the algorithm applies to the more general class of well-ordered graphs.

1. Introduction

We say that an *m*-node directed graph is *chain-structured* if it has $(m-1)$ arcs, and there are labels $N_1, N_2, \ldots, N_m$ and $\alpha_1, \alpha_2, \ldots, \alpha_{m-1}$ for the nodes and arcs, respectively, such that each α_i is directed from N_i to N_{i+1}.

With large sample-rate changes, careful code-scheduling techniques must often be applied to keep program memory requirements manageable when efficient in-line code is desired [1]. For chain-structured SDF graphs, although minimum-code-size schedules can easily be constructed, the number of distinct minimum-code-size schedules increases combinatorially with the number of nodes, and these schedules can have vastly differing data memory requirements.

In this paper, we focus on determining the schedule that minimizes data memory requirements from among the set of schedules that minimize code size. We show that this problem is similar to the problem of most efficiently multiplying a chain of matrices [5], for which a cubic-time dynamic programming algorithm exists. We show that this dynamic programming technique can be adapted to our problem to give an algorithm with time complexity $O(m^3)$. Finally, we present a heuristic having quadratic time-complexity that usually performs well. Thus, the heuristic can be used as a quick first-attempt, and if the resulting solution does not satisfy the given memory constraints, the dynamic programming algorithm can be applied to obtain an optimal solution.

2. Memory Requirements for Chain-Structured SDF graphs

In dataflow, a program is represented as a directed graph in which the nodes (*actors*) represent computations and the arcs specify the flow of data. An actor is allowed to execute (*fire*) only after it has sufficient data on all input arcs. Synchronous dataflow (SDF) is a restricted form of dataflow in which the number of data values (*tokens*) produced and consumed by each actor is fixed and known at compile-time. Fig. 1 shows a simple chain-structured SDF graph. The numbers on the ends of the arcs specify the number of tokens produced and consumed by the incident nodes.

We compile an SDF graph G by first constructing a *periodic admissable sequential schedule* (PASS), also called a *valid schedule* — a sequence of actor firings that executes each actor at least once, does not deadlock, and produces no net change in the number of tokens on each arc. Corresponding to each actor in the valid schedule, we insert a code block that is obtained from a library of predefined actors. The minimum number of times that each actor must execute in a valid schedule can be computed efficiently [6]. We represent these minimum numbers of firings by a vector q_G, indexed by the actors in G, and we refer to q_G as the *repetitions vector* of G (we often suppress the subscript if G is understood from context). For fig. 1, $\mathbf{q} = \mathbf{q}(A,B,C,D) = (9, 12, 12, 8)$.

For example, (2ABC) DABCDBC (2ABCD)A(2BC) (2ABC) A (2BCD) gives a valid schedule for fig. 1. Here a parenthesized term $(n\ S_1\ S_2 \ldots S_k)$ represents *n* successive firings of the subschedule $S_1\ S_2 \ldots S_k$, and we translate such a term into a loop in the target code. A more compact schedule for fig. 1 is

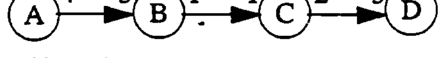

Figure 1. A chain-structured SDF graph.

1. This research was sponsored by Star Semiconductor Corporation, the State of California MICRO Project, the National Science Foundation (MIP-9201605), the Advanced Research Projects Agency and the United States Air Force.

(9A)(12B)(12C)(8D). We call this schedule a *single appearance schedule* since it contains only one appearance of each actor. Neglecting loop overhead, any valid single appearance schedule gives the minimum code space cost for in-line code generation.

The amount of memory required for buffering along arcs may vary greatly between different schedules. We define the *buffering cost* of a schedule S as (Σ *max_tokens*(α, S)), where the sum is taken over all arcs α, and *max_tokens*(α, S) denotes the maximum number of tokens that are simultaneously queued on α during an execution of S. For example, the schedule (9A)(12 B)(12C)(8D) has a buffering cost of 36+12+12+24 = 84.

The buffering cost gives the amount of memory required to implement the arcs if each arc is implemented as a separate block of memory. Optimally combining the advantages of buffer overlaying and nested loops is a topic for further study.

Corresponding to each valid schedule, there is a positive integer *J*, called the *blocking factor*, such that each actor N is executed *J*q(N) times by the schedule. It can be shown that given any SDF graph and a valid schedule with $J > 1$, there is a valid schedule of unity blocking factor that has equal or smaller buffering cost. Thus for our purposes, it suffices to consider only the unity blocking factor case. In the remainder of the paper, by a "schedule" we mean a valid unit blocking factor schedule.

Note that for simplicity, we assume that the arcs in a chain-structured SDF graph have no delay associated with them; however, the techniques presented in this paper can easily be extended to handle delays.

3. A Recursive Scheduling Approach

Let G be a chain-structured SDF graph with nodes $N_1, N_2, \ldots, N_m$, and arcs $\alpha_1, \alpha_2, \ldots, \alpha_{m-1}$ such that each α_k is directed from N_k to N_{k+1}. In the trivial case, $m = 1$, we immediately obtain N_1 as a schedule for G. Otherwise, given any $i \in \{1, 2, \ldots, m-1\}$, define *left*($i$) to be the subgraph formed by $\{N_1, N_2, \ldots, N_i\}$ and $\{\alpha_j \mid j < i\}$; similarly define *right*(i) to be the subgraph formed by $\{N_{i+1}, N_2, \ldots, N_m\}$ and $\{\alpha_j \mid j > i\}$. Let $q_L = gcd(\{q(N_j) \mid 1 \le j \le i\})$, and let $q_R = gcd(\{q(N_j) \mid i < j \le m\})$ where *gcd* is the greatest common divisor. If S_L and S_R are valid single appearance schedules for *left*(i) and *right*(i) respectively, then (q_L S_L)(q_R S_R) is a valid single appearance schedule for G [2].

For example, suppose that α_2 denotes the center arc in fig. 1, and suppose $i = 2$. It is easily verified that $q_{left(i)}$ (A, B) = (3, 4), and $q_{right(i)}$ (C, D) = (3, 2). Thus, S_L = (3A)(4B) and S_R = (3C)(2D) are valid single appearance schedules for *left*(i) and *right*(i), respectively, and (3(3A) (4B))(4(3C) (2D)) is a valid single appearance schedule for fig. 1.

We can recursively apply this procedure of decomposing a chain-structured SDF graph into left and right subgraphs to construct a schedule for the graph. However, different sequences of choices for *i* will in general lead to different schedules. For an *m*-node graph, the number of distinct schedules obtainable from this recursive process is

$$\frac{1}{m}\binom{2m-2}{m-1}.$$

This expression is $\Omega(4^m/m)$ [1], and its values for all *m* are known as the *Catalan numbers* [4].

It can be shown that the set of valid single appearance schedules obtainable from this recursive scheduling process, which we refer to as the set of *R-schedules*, always contains a schedule that achieves the minimum buffer cost over all valid single appearance schedules. For example, the R-schedules for fig. 1 are (3(3A)(4B))(4(3C)(2D)), (3(3A)(4(1B)(1C)))(8D), (3(1(3A)(4B))(4C))(8D), (9A) (4(3(1B)(1C))(2D)), (9A)(4(3B)(1(3C)(2D))); the corresponding buffering costs are, respectively, 30, 37, 40, 43, 45; and it can be verified that (3(3A)(4B))(4(3C)(2D)) has the lowest buffering cost over all valid single appearance schedules for this example. Note that the 1-iteration loops can be ignored during code generation.

The combinatorial growth in the size of the set of R-schedules precludes exhaustive evaluation. The following section presents a dynamic programming algorithm that obtains an optimal schedule in polynomial time.

4. Dynamic Programming Formulation

The problem of determining the R-schedule that minimizes buffering cost for a chain-structured SDF graph can be formulated as an optimal parenthesization problem. A familiar example of this problem is matrix chain multiplication [4][5]. In matrix chain multiplication, we must compute the matrix product $A_1 A_2 \ldots A_n$. These matrices are assumed to have compatible dimensions so that the product is computable. There are several ways in which the product can be computed. For example, with $n = 4$, one way of computing the product is $(A_1(A_2 A_3))A_4$ where the parenthesizations indicate the order in which the multiplies occur. Suppose that the matrices have dimensions 10x1, 1x10, 10x3, 3x2. It is easily verified that computing the product as $((A_1 A_2) A_3) A_4$ requires 460 multiplications whereas computing it as $(A_1(A_2 A_3))A_4$ requires only 120 multiplications (assuming that we use the standard algorithm for multiplying two matrices). Thus, we would like to determine the optimal way of placing the parentheses so that the total number of multiplies is minimized. This can be done using a dynamic programming approach. The observation is that any optimal parenthesization splits the product $A_1 A_2 \ldots A_n$ between A_k and A_{k+1} for some k in the range $1 \le k < n$. The cost of this optimal parenthesization is therefore the cost of computing the product $A_1 A_2 \ldots A_k$, plus the cost of computing $A_{k+1} \ldots A_n$, plus the cost of multiplying them together. In an optimal parenthesization, the subchain $A_1 A_2 \ldots A_k$ must itself be parenthesized optimally. Hence, this problem has the optimal substructure property and is thus amenable to a dynamic programming solution.

1. $f(x)$ is $\Omega(g(x))$ ($O(g(x))$) if $f(x)$ is bounded below (above) by a positive real multiple of $g(x)$ for sufficiently large x.

Determining the optimal R-schedule for a chain-structured SDF graph is similar to the matrix chain multiplication problem. For example, consider fig. 1. Here, q(A,B,C,D) = (9,12,12,8). An optimal R-schedule is (3((3A)(4B)))(4 ((3C)(2D))), and the associated buffering cost is 30. Therefore, as in the matrix chain multiplication case, the optimal parenthesization contains a break in the chain at some k, $1 \le k < n$. Because the parenthesization is optimal, the chains to the left of k and to the right of k must both be parenthesized optimally also. Hence, we have the optimal substructure property.

The problem can be formulated as follows: denote by $b[i,j]$ the cost for optimally parenthesized nodes $N_i, N_{i+1}, \ldots, N_j$, and let o_k denote the number of tokens produced by each invocation of N_k. Then we wish to compute $b[1, m]$. If $i \neq j$, then

$$b[i,j] = \underset{i \le k < j}{MIN}\{b[i,k] + b[k+1,j] + c_{ij}[k]\} \qquad \textbf{(EQ 1)}$$

where $b[i, i]=0$ for all i, and $c_{ij}[k]$ is the memory cost at the split if we split the chain at k. It is given by

$$c_{ij}[k] = \frac{q(N_k)\, o_k}{gcd(q(N_i), q(N_{i+1}), \ldots, q(N_j))} \qquad \textbf{(EQ 2)}$$

The gcd appears in the denominator because the repetitions vector $q_{SC(i,j)}$ of subchain $\{N_i, N_{i+1}, \ldots, N_j\}$ satisfies $q_{SC(i,j)}(N_k) = q(N_k) / gcd(\{q(N_i), q(N_{i+1}), \ldots, q(N_j)\})\ \forall\, k \in \{i, \ldots j\}$ [2].

It can be shown that the running time of the implementation of this dynamic programming formulation is $O(m^3)$.

5. Example: Sample-Rate Conversion

The recently introduced Digital Audio Tape (DAT) technology operates at a sampling rate of 48 khz while compact disk (CD) players operate at a sampling rate of 44.1 khz. Interfacing the two, for example, to record a CD onto a digital tape, requires a sample rate conversion.

The naive way to do this is shown in fig. 2. It is more efficient to perform the rate-change in stages. Rate conversion ratios are chosen by examining the prime factors of the two sampling rates. The prime factors of 44100 and 48000 are $2^2 3^2 5^2 7^2$ and $2^7 3^1 5^3$, respectively. Thus the ratio 44100:48000 is $3^1 7^2 : 2^5 5^1$ or 147:160. One way to perform the conversion in three stages is 4:3, 8:7, and 5:7. Fig. 3 shows the multistage implementation. Explicit upsamplers and downsamplers are omitted; it is assumed that the FIR filters are general polyphase filters [3].

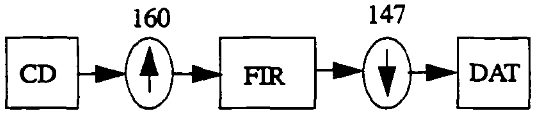

Figure 2. CD to DAT rate-change system

Here q(A,B,C,D,E) = (147,49,28,32,160). The optimal nesting for this graph as given by the dynamic programming approach is (49((3A)B)) (4((7C)(8(D(5E))))), and the associated buffering cost is 260. In contrast, the naive schedule (147A)(49B) (28C)(32D)(160E) has a buffering cost of 729. The data-memory requirement for a graph like this consists of two parts: inter-actor buffering induced by the schedule, and memory required to hold variables and states contained in the actors themselves. Clearly, the first component of the requirement is significant in this case; hence, it pays to do intelligent scheduling.

6. A Quadratic-Time Heuristic

The time-complexity of our dynamic programming solution is cubic in the number of nodes. For cases in which the running time of this exact algorithm is unacceptable or a quicker solution is desired, we have devised a quadratic-time heuristic that performs quite well in practice. However, as with all heuristics, examples can be constructed where the heuristic will perform poorly.

The heuristic is simply to introduce the parenthesization on the arc where the minimum amount of data is transferred. This is done recursively for each of the two halves that result. The running time of the heuristic is given by the recurrence

$$T(n) = T(n-k) + T(k) + O(n) \qquad \textbf{(EQ 3)}$$

where k is the node at which the split occurs. This is because we have to compute the gcd of the repetitions of the k nodes to the left of the split and the gcd of the repetitions of the $n-k$ nodes to the right. This takes $O(n)$ assuming that the repetitions vector is bounded. Computing the minimum of the data transfers takes a further $O(n)$ time since there are $O(n)$ arcs to consider. The worst-case solution to this recurrence is $O(n^2)$, but the average case running time is $O(n \cdot \log n)$ if $k = O(n)$. It is easily seen that this heuristic gives the R-schedule with minimum buffering cost, (3(3A)(4B))(4(3C)(2D)), for fig. 1.

We have tested the heuristic on 10,000 randomly generated 50-node chain-structured SDF graphs, and we have found that on average, it yields a buffering cost that is within 60% of the optimal cost. For each random graph, we also compared the heuristic's solution to the worst-case schedule and to a randomly generated R-schedule. On average, the worst case schedule had over 9000 times higher cost than the heuristic's solution, and the random schedule had 225 times higher cost. Furthermore, the heuristic outperformed the random schedule on 97.5 percent of the trials.

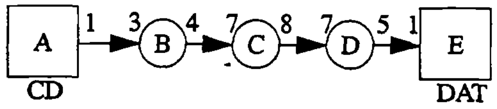

Figure 3. Multi-stage implementation of CD to DAT rate-change system:

7. Extensions

There are three simple extensions of our dynamic programming solution to the optimum R-schedule problem for chain-structured SDF graphs. In this section, we use the following notation: given an SDF arc α, we denote the source node and sink node of α by $source(\alpha)$ and $sink(\alpha)$, and we denote the number samples produced on α per invocation of $source(\alpha)$ by $produced(\alpha)$. First, our dynamic programming technique applies to the more general class of *well-ordered* SDF graphs. A directed acyclic graph is well-ordered if it has only one topological sort (a topological sort of a directed graph is an ordering of the nodes such that for each arc α, $source(\alpha)$ occurs earlier in the ordering than $sink(\alpha)$. All chain-structured graphs are well-ordered but the converse is obviously not true.

The dynamic programming technique for chain-structured SDF graphs can be applied to the topological sort corresponding to a general well-ordered graph with the modification that $c_{ij}[k]$, the memory cost of splitting subchain $\{N_i, N_{i+1}, \ldots, N_j\}$ between the nodes N_k and N_{k+1}, now gets computed as

$$c_{ij}[k] = \frac{\sum_{\alpha \in S_{i,j,k}} (\mathbf{q}(source(\alpha)))\,produced(\alpha)}{gcd(\{\mathbf{q}(N_i), \mathbf{q}(N_{i+1}), \ldots, \mathbf{q}(N_j)\})} \quad \textbf{(EQ 4)}$$

where $S_{i,j,k} = \{\beta \mid (source(\beta) \in \{N_i, N_{i+1}, \ldots, N_k\})$ and $(sink(\beta) \in \{N_{k+1}, \ldots, N_j\})\}$, the set of arcs directed from one side of the split to the other side.

The dynamic programming technique of section 3 can also be applied to reducing the buffering cost of a given single appearance schedule for an arbitrary acyclic SDF graph (not necessarily chain-structured or well-ordered). Suppose we are given a valid single appearance schedule S for an acyclic SDF graph and again for simplicity, assume that arcs in the graph contain no delay. Let $\Psi = N_1, N_2, \ldots, N_m$ denote the sequence of lexical actor appearances in S (for example, for the schedule (4 A(2 CD))F, Ψ = A, C, D, F). Now it can easily be verified that since S is a valid schedule, Ψ must be a topological sort of the associated SDF graph. The technique of section 3 can easily be extended to optimally "re-parenthesize" S into the optimal single appearance schedule (with regards to buffering cost) associated with the topological sort Ψ. The technique is applied to the sequence Ψ, with $c_{ij}[k]$ computed as in equation 4.

Thus, given any topological sort Ψ^* for an acyclic SDF graph, we can efficiently determine the single appearance schedule that minimizes the buffering cost over all valid single appearance schedules for which the sequence of actor appearances is Ψ^*.

Another extension is to use shared buffers. There are several ways in which buffers can be shared; the simplest is to have one shared buffer of size

$$cs_{ij} = \frac{MAX_{i \le k < j}(o_k \mathbf{q}(k) + o_{k+1}\mathbf{q}(k+1))}{gcd(\mathbf{q}(i), \ldots, \mathbf{q}(j))} \quad \textbf{(EQ 5)}$$

for the sub-chain consisting of nodes $i, \ldots, j$. We modify equation 1 by

$$b'[i,j] = MIN(b[i,j], cs_{ij}) \quad .$$

Note that if $j = i+1$, then

$$cs_{ij} = \frac{o_i \mathbf{q}(i)}{gcd(\mathbf{q}(i), \mathbf{q}(i+1))} .$$

This gives us a combination of shared and non-shared buffers for different sub-chains. Sharing buffers in a more general way is beyond the scope of this paper.

Acknowledgment

Tom Parks at UC Berkeley conceived, designed, and implemented the rate-change system of fig. 3.

References

[1] S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee, "A Scheduling Framework for Minimizing Memory Requirements of Multirate Signal Processing Algorithm Expressed as Dataflow Graphs", *VLSI Signal Processing VI*, IEEE Special Publications, 1993.

[2] S. S. Bhattacharyya and E. A. Lee, "Looped Schedules for Dataflow Descriptions of Multirate Signal Processing Algorithms", to appear in *Formal Methods in System Design*, 1994.

[3] J. T. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, "Multirate Signal Processing in Ptolemy", Proceedings of *ICASSP*, Toronto, April 1991.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to Algorithms", McGraw Hill 1990.

[5] S. S. Godbole, "On Efficient Computation of Matrix Chain Products", *IEEE Trans. on Computers*, September 1973.

[6] E. A. Lee and D. G. Messerschmitt, "Synchronous Dataflow", *Proceedings of the IEEE*, September, 1987.

[7] P.K.Murthy, "Multiprocessor DSP Code Synthesis in Ptolemy", Memo No. UCB/ERL M93/66, ERL, UC-Berkeley, Ca 94720

[8] J.Pino, S.Ha, E.A.Lee, and J.T.Buck, "Software Synthesis for DSP Using Ptolemy", to appear in *Journal of VLSI Signal Processing*, 1993.