SCHEDULING FOR MINIMIZING THE NUMBER OF MEMORY ACCESSES IN LOW POWER APPLICATIONS

Rafi Saied and Chaitali Chakrabarti
Department of Electrical Engineering
Telecommunications Research Center
Arizona State University
Tempe, AZ 85287-5706
rafi.saied@asu.edu, chaitali@asu.edu

Abstract - Power consumption in memory-intensive operations can be reduced by minimizing the number of memory acceses. In this paper we describe two scheduling schemes under fixed hardware resource constraints which reduce the number of memory accesses by minimizing the number of intermediate variables that need to be stored. While the first scheme achieves this by post order traversal of the DFG, the second scheme achieves this by judiciously delaying the scheduling of some of the nodes. Experimental results show that these schemes require significantly fewer memory accesses compared to existing scheduling schemes.

1 INTRODUCTION

The increasing demand for portable electronics has caused power consumption to be a critical issue in the design process. Reducing the total power consumption in portable systems is important in order to maximize the run time with minimum requirements in size and weight of the batteries. In the past few years a lot of work has been done to optimize power at all levels of the design process – from algorithmic and architectural levels to logic and circuit levels [1]. Most of the optimization is based on reducing the switching activity, since power is consumed during the charging or discharging of capacitors. It has been shown that power consumption can also be reduced by transformations such as subexpression elimination, operation substitution, resource utilization, transition activity reduction etc [2]. Most of the previous work on power optimization has been geared towards low power data path design. However it is well known that for many real-time signal processing applications, the power consumption in the memory (RAMs, register files) is larger than 50% of the total power consumption.

To reduce power consumption in memory, a proper choice of algorithm and architecture is essential. Algorithmic level decisions have been shown to reduce computational complexity and the number of memory accesses by a very large factor [3]. Memory optimization procedures such as storing the most frequently accessed data in small caches, obtaining simpler addressing (such as Gray code addressing) such that the number of transitions in the address lines is minimum, optimizing the order of memory accesses to obtain related addresses as immediate successors, etc. have been proposed in [6]. Architectural level optimizations include accessing data from the memory in parallel, thereby enabling throughput to be traded for reduction in voltage [1], or organizing the memory in a distributive manner since such a configuration switches less capacitance than centralized memory architectures [3].

The scope of this paper supplements the existing work on reducing the power consumption due to memory by developing new scheduling schemes which reduce the number of memory accesses. The rationale behind this is that since memory systems draw current when an access takes place, reducing the number of memory accesses will reduce the power consumed by charging and discharging memory elements. The proposed scheduling schemes operate under fixed hardware resource constraints and reduce the number of memory accesses by minimizing the number of intermediate variables that need to be stored. While the first scheme reduces the number of memory accesses by post order traversal of the DFG, the second scheme achieves this by judiciously delaying the scheduling of some of the nodes. Both schemes try to schedule siblings at the same time. As a result, the parent can be scheduled in the next cycle and the intermediate variables generated by the siblings do not have to be stored. It is to be noted that the same schemes can be used to reduce the number of register file accesses (We use the term 'memory accesses' and 'register accesses' interchangeably in this paper). Experimental results show that these scheduling schemes require fewer memory accesses compared to existing scheduling schemes. This paper also presents an allocation scheme which takes into account data correlation.

2 SCHEDULING SCHEMES

We begin by describing the model of the hardware unit used. We assume that each multiplier/adder has a latch at its output. In addition there are 8 to 12 general purpose registers in a register file that are used to store the operands and intermediate results. The output of a multiplier/adder can either be fed to another (or same) multiplier/adder in the next cycle or can be stored in a general-purpose register via a bus. If the number of general-purpose registers is insufficient, data is stored in the RAM via an additional data bus.

The above model suggests that a scheduling scheme which minimizes the number of register/memory acceses would not only minimize the switching in the registers but also the switching in the busses. In the rest of this section we describe two such scheduling schemes. It is to be noted that not all the memory accesses can be minimized. For instance, the inputs to the computation unit may have to be read out from the memory. Also, some of the inter-iteration delays cannot be minimized.

It is worth mentioning that the worst schedule with respect to minimizing the number of memory accesses is one in which all the nodes of a given level are scheduled before the nodes of the next level. Fig. 1a illustrates this schedule for the fourth order Chebyshev filter. Note that this schedule also requires a larger number of control cycles (12 versus 10). Of the existing schedules, the ALAP (as late as possible) schedule attempts to minimize the number of memory accesses. This is because the later a node is scheduled, the shorter is the time that the corresponding intermediate variable will have to be stored.

The two scheduling schemes that we propose here aim at minimizing the number of memory accesses that are caused by the generation of intermediate variables. In both the schemes we assume that the number of hardware resources (multipliers, adders) is fixed. While the first scheduling scheme (Algorithm A) reduces the number of memory accesses at the expense of a possible increase in timing, the second scheme (Algorithm B) tries to balance both the reduction in the number of memory accesses and the timing constraint. Algorithm A is based on post order traversal of the DFG, while Algorithm B is based on a modified version of the ALAP schedule.

It is worth mentioning that, in general, (i) the DFG is not a tree but a directed acyclic graph and the post order traversal has to be done on the graph, (ii) the siblings are nodes that not only have the same parent but also do not have any dependencies between them, (iii) the nodes of the left and right subtrees may not be disjoint. In addition, we assume that each node in the graph has only two incoming edges (and thus each node has two siblings). This is not restrictive this each hardware unit operates on only two operands at a time.

2.1 Algorithm A:

Algorithm A schedules the nodes in the order obtained by the post order schedule. It schedules siblings at the same time if there are enough hardware resources available. The rationale behind scheduling the siblings at the same time is that the parent can then be scheduled immediately after the siblings and thus the intermediate variables generated by the siblings do not have to be stored.

Scheduling Algorithm A:

1. For each node of the DFG compute NUM, the length of the longest path from the node to the root. The length of the path is the weighted sum of the nodes, $\sum_{i \in path} w(node_i)$, where $w(node_i)$ is the computation delay of node i.

2. If the DFG is a tree, modify the DFG so that each node has its longer path in the left subtree and its shorter path in the right subtree.

3. Obtain the post order schedule of the DFG.

4. Obtain the siblings of each node. This can be done using various techniques such as by using a combination of post-order and in-order schedule. Compare the NUM values of the siblings and order them such that

the node with higher NUM value precedes the node with the lower NUM value.

5. Use the post order schedule to schedule the nodes in the following way.

   (a) Schedule siblings with the same value of NUM (referred to as SIB) at the same time, if possible. If $SIB \leq$ *number of hardware units available in that cycle* then assign all the siblings in that cycle. Else if $SIB >$ *total number of hardware units* then assign as many siblings as possible in that cycle and the remaining siblings in the next cycle. Else if $SIB >$ *number of hardware units available in that cycle* then assign the siblings in the next cycle.

   (b) If the siblings do not have the same value of NUM, then assign the sibling with the highest value of NUM, followed by the sibling with the next highest value and so on. If the sibling with the highest value of NUM is scheduled at time t, then the sibling with the next highest value of NUM is scheduled at time $t - \Delta$, where Δ is the difference in the NUM values of the two siblings.

   (c) If the current node i is a parent of the previous node $i-1$, then schedule node i, $w(node_i)$ cycles after node $i-1$.

   (d) If the current node i is not a sibling of the previous node $i-1$ or parent of node $i-1$ (in the ordering obtained by post order traversal), then schedule all the nodes of the right subtree in the following way. Note that the right subtree consists of nodes i through parent(i-1) in the post order traversal. First find a 'relative' schedule (ignoring the conflict that may arise due to the scheduling of the left sub tree) of all the nodes in the right subtree. Then the final schedule is a shifted version of the relative schedule, where the amount of shift is such that there is no conflict with the schedule of the left subtree.

The main drawback of Algorithm A is that it has the potential of requiring more control cycles. This is because in Step 5d, the nodes of the right subtree are scheduled without taking into consideration the hardware vacancies that have been generated by the schedule of the left subtree. While this scheme clearly reduces the number of memory accesses, the best fit in Step 5d may result in larger number of control cycles. The operations in Algorithm A are explained with the help of the following example.

Example 1: Fourth order Chebyshev Filter (see Fig. 1)
Assume that the hardware available consists of two multipliers pipelined to two levels and one non-pipelined adder. Thus the multiplier takes two control cycles and the adder takes only one control cycle. The DFG of the fourth order Chebyshev filter is shown in Fig. 1. The longest path is eight cycles. Next the NUM values are computed for each node. The post order schedule of this DFG is: 16,13,12,9,8,5,4,1,2,14,17,15,10,11,6,7,3. The siblings are (16,13), (9,12), (5,8), (14,17), (10,15), (6,11), and (2,7).

In Fig. 1b, nodes 13 and 16 have the same value of NUM, so assign both of them in the same cycle (rule 5a). Since the multipliers are pipelined to two levels, node 12 is assigned in cycle 3 (rule 5c). Next node 9 which is a sibling of node 12 is assigned in cycle 3-1=2 (rule 5b). Node 8 is assigned in cycle 2+2=4 (rule 5c). Node 5 which is a sibling of node 8 is assigned in cycle 3 (rule 5b). Next node 4 is assigned in cycle 5. Node 1 is then assigned in cycle 6 and node 2 in cycle 7. In this way the left subtree of node 3 is scheduled.

Since nodes 2 and 14 are not siblings, rule 5d is used to assign the nodes of the right subtree (all nodes between 14 and 7 in the post order traversal). First a relative schedule of these nodes is found using rules 5a, 5b and 5c. In the relative schedule, nodes 14 and 17 are assigned in cycle 1, node 15 is assigned in cycle 3, node 10 is assigned in cycle 2, node 11 is assigned in cycle 4, node 6 is assigned in cycle 3 and node 7 is assigned in cycle 5. Next the final schedule is found such that there is no conflict with the left subtree schedule. In this example, the final schedule is obtained by delaying the relative schedule of the right subtree by 4 cycles.

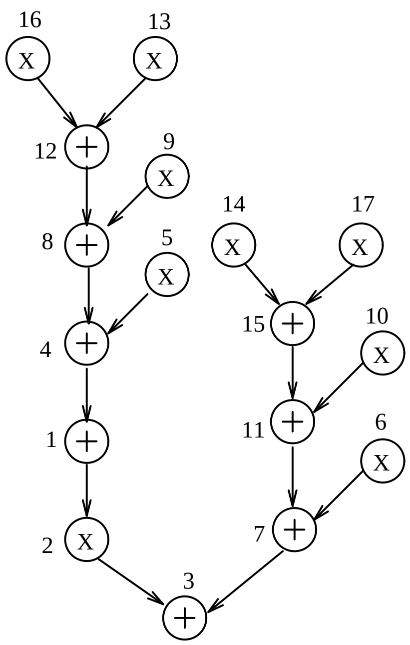
Finally node 3 is assigned in cycle 10. Note that in this schedule, the result of node 2 is available in cycle 8 and is not used till cycle 10 by node 3. Thus the result of node 2 has to be stored for two cycles. This is shown in the schedule of Fig 1b by an asterix (*). Note that the schedule shown in Fig. 1b takes 10 control cycles and only one memory access in addition to the essential accesses. □

Combining force directed schedule with minimum memory access schedule:

Our aim here is to combine force directed schedule [4] with Algorithm A in an effort to achieve both balanced use of hardware and minimum number of memory accesses. This can be achieved by breaking ties in force directed schedule by the minimum memory access schedule. This problem is currently under investigation.

2.2 Algorithm B:

Algorithm B is a modified version of the ALAP schedule which tries to minimize both the number of control cycles and the number of memory accesses. It operates on one control cycle at a time and tries to assign all the 'ready' nodes to the hardware units if possible. The 'ready' nodes include the nodes whose ALAP is the same as the control cycle number and the nodes which have been deferred from the previous cycle. In each cycle, some of the 'ready' nodes may have to be deferred since the number of hardware units may be insufficient. The nodes are deferred in a way that tries to minimize the number of intermediate variables. This is done by taking into consideration the NUM values and the sibling information. For instance, if the 'ready' nodes have the same NUM value, nodes which are not siblings or which have fewer children assigned in the previous cycle are deferred.



Clock cycle	Multipliers	Adder
1	14, 17	--
2	13, 16	--
3	10, 9*	15 *
4	6*, 5*	12 *
5	--	11 *
6	--	8 *
7	--	7 *
8	--	4
9	--	1
10	2	--
11	--	--
12	--	3

(a)

Clock cycle	Multipliers	Adder
1	13, 16	--
2	9	--
3	5	12
4	--	8
5	14, 17	4
6	10	1
7	2* ,6	15
8	--	11
9	--	7
10	--	3

(b)

Clock cycle	Multipliers	Adder
1	13, 16	--
2	9	--
3	5	12
4	14, 17	8
5	10*	4*
6	6*	15*
7	--	1
8	2	11
9	--	7
10	--	3

(c)

Figure 1: Fourth order Chebyshev filter. (a) Worst case schedule (b) Schedule using Algorithm A (c) Schedule using Algorithm B. The hardware resources include two multipliers pipelined to 2 levels and one adder.

Scheduling Algorithm B:

1. Steps 1 through 4 of Algorithm A which include computing NUM (the length of the longest path from the node to the root), and the siblings. The post order schedule need not be computed here unless it is used to identify the siblings.

2. For all nodes of the DFG, compute the latest time (ALAP) at which a node can get scheduled. This can be determined by subtracting NUM from 1+ the length of the critical path.

3. for c-step from 1 to Set(Ready)=null do:
(i) Determine 'ready' nodes in c-step. These include the nodes with ALAP=c-step and the nodes which have been deferred in the previous c-step.
(ii) While (number of hardware units < number of 'ready' nodes) do:

   (a) For nodes with differing NUM values, defer the nodes with lower NUM.

   (b) For nodes with same NUM values, defer the nodes which are not siblings (in an attempt to assign the siblings to the same cycle).

   (c) For nodes with same NUM values, defer nodes which have fewer children assigned in the previous cycle.

   (iii) Schedule the remaining 'ready' nodes in the current c-step.

Example 2: Fourth order Chebyshev Filter (see Fig. 1)

Assume that the hardware consists of two multipliers pipelined to two levels and one non-pipelined adder. The ALAP schedule, NUM values and the sibling information are obtained as in Algorithm A.

The "for" loop is repeated until all the nodes have been scheduled. When Tc=1, there are 2 'ready' nodes: 13,16. Since the number of multipliers is equal to the number of 'ready' nodes, both 13 and 16 are assigned in cycle 1. When Tc=2, 9 is the only 'ready' node and is assigned in cycle 2. When Tc=3, the 'ready' nodes 5 and 12 are assigned in cycle 3. When Tc=4, all three 'ready' nodes 14, 17 and 8 are assigned in cycle 4. When Tc=5, both the 'ready' nodes are assigned in cycle 5. When Tc=6, the 'ready' nodes are 1, 15 and 6. Since there is only adder available, and since nodes 1 and 15 have the same value of NUM, node 1 is deferred since it has only one child (whereas node 15 has two children that were assigned in a previous cycle) (rule 3(ii)(c)). Nodes 15 and 6 are assigned in clock 6. When Tc=7, the 'ready' nodes are 1 and 11. Since there is only one adder available and node 1 has a higher NUM value than node 11, defer node 11 (rule 3(ii)(a)). Node 1 is thus assigned in clock 7. When Tc=8, there are sufficient number of hardware units and nodes 2 and 11 are assigned in cycle 8. When Tc=9, the 'ready' operation 7 is assigned in cycle 9. When Tc=10, the 'ready' operation is assigned in cycle 10. Note that this schedule has only 4 memory accesses

Filter type	Number of control cycles	ASAP schedule	ALAP schedule	Schedule A	Schedule B
4th order Chebyshev	10	6	5	1	4
2nd order IIR	6	3	3	1	2
10th order IIR (all pole)	12	7	0	0	0
5th order FIR	7	2	5	2	4

Figure 2: Comparison of the number of memory accesses required for the different schedules. The hardware resources include two multipliers pipelined to 2 levels and one adder.

compared to the 5 memory accesses that would be required by the ALAP schedule. □

Results:

Fig. 2 lists the number of memory accesses that are needed for the proposed schedules with those of the ASAP and ALAP schedules for a fourth order Chebyshev filter, a second order IIR fiter, a tenth order all pole IIR filter and a fifth order FIR filter. Note that Schedule A requires the fewest number of memory accesses and that Schedule B requires more memory accesses than Schedule A but less than the ALAP schedule. In all these examples, all four scheduling schemes require the same number of control cycles. However, in general, Schedule B requires fewer control cycles. For instance, for the case when the number of pipelined multipliers is 3 and the number of non-pipelined adders is 2, Schedule B requires only 9 control cycles compared to the 10 cycles that would be required by Schedule A. In addition, Schedule B requires no memory accesses.

Inter-iteration delays:

As mentioned earlier, the memory accesses that are caused by inter-iteration delays cannot be reduced as easily. However if the computations of the iterations can be overlapped, then some of the inter-iteration delays can be reduced. We propose a method where the schedule for all iterations is the same and the computation of consecutive iterations is overlapped wherever possible. We choose to have the same schedule for all iterations in order that the control circuitry be simple.

1. Find the schedule for a particular iteration.
2. Check to see the earliest time during which the next iteration can be scheduled. Since the same schedule is used for all the iterations, finding the best fit between the vacancies in iteration i and hardware requirements in iteration $i+1$, is quite straight forward.

It is to be noted that retiming can change the number and position of inter-iteration delays, and consequently the number of memory accesses due to inter-iteration delays.

3 ALLOCATION

The next step in the procedure is to assign the operations to the hardware units. The exact assignment is based on the observation that power consumption can be reduced if the hardware unit operates on data that is correlated in consecutive cycles [5]. The proposed algorithm is sketched below.

1. Run simulations with random (or prespecified) inputs and compute the correlation between data that is input to similar hardware units in consecutive cycles. Construct a graph whose nodes are the operations and whose edges are weighted by the correlations, that is, edge (i, j) is weighted by the correlation between the inputs of operations i and j, and operations i and j occur in two consecutive time steps.
   Example: If operations 1 and 2 are assigned to the multipliers in Step 1 and operations 3 and 4 in Step 2, then the graph consists of 4 nodes with edges (1,3), (1,4), (2,3) and (2,4) weighted by their corresponding correlations.

2. Establish a correspondence between the nodes in two consecutive time steps (say k and $k+1$) such that the sum of the weights between nodes in step k and the corresponding nodes in step $k+1$ is minimum. If n be the number of nodes in each of the time steps, then $n!$ combinations will have to be checked for correspondence. While the search can be made more efficient by utilizing branch and bound techniques, the number of operations per step is usually small enough that it is not required.
   Example: If $(1,3)$ has a weight of 0.2, $(1,4)$ has a weight of 0.4, $(2,3)$ has a weight of 0.4 and $(2,4)$ has a weight of 0.7, then the combinations $w(1,3) + w(2,4) = 0.9$ and $w(2,3) + w(1,4) = 0.8$ will have to be checked. Since $w(2,3) + w(1,4)$ is smaller, a correspondence is established between nodes (1 and 4) and (2 and 3). Thus nodes 1 and 4 will be assigned to one multiplier while nodes 2 and 3 will be assigned to another multiplier.

3. Repeat Step 2 for all the time steps.

It is to be noted that a correspondence between steps k and $k+1$ is established independent of the correspondence between steps $k+1$ and $k+2$.

4 CONCLUSIONS

Power consumption has become a big issue in design with the increase in the demand for portable computing. Power consumption in memory can be reduced by minimizing the number of memory accesses. In this paper we have described two scheduling algorithms that minimize the number of memory accesses by minimizing the number of intermediate variables. Algorithm A achieves this by post order traversal of the DFG. This may result in an increase in the number of control cycles. Algorithm B tries to reduce the number of control cycles at the expense of a mild increase in the number of memory accesses. Experimental results show that both schemes significantly reduce the number of memory accesses compared to existing scheduling schemes.

ACKNOWLEDGMENTS

The authors would like to acknowledge the support of the Center for Low Power Electronics, UofA-ASU, and the Telecommunications Research Center, ASU.

References

[1] A. P. Chandrakasan and R. W. Brodersen, "Minimizing power consumption in digital CMOS circuits," *Proceedings of IEEE*, pp. 498-523, April 95.

[2] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey and R. Brodersen, "Optimizing power using transformations," *IEEE Trans on Computer Aided Design*, vol. 14, Jan 95.

[3] D. B. Lidsky and J. Rabaey, "Low-Power Design of Memory Intensive Functions," *Proc. 1994 IEEE Symposium on Low Power Electronics*, pp. 16-17, 94.

[4] P. Paulin and J. Knight, "Force directed Scheduling for the Behavioral Synthesis of ASICs," *IEEE Trans on Computer Aided Design*,pp. 661-679, July 89.

[5] A. Raghunathan and N. K. Jha, "Behavioral Synthesis for Low Power," *Proc. of the International Conference on Computer Design*, pp. 318-322, Oct 94.

[6] S. Wuytack, F. Catthoor, F. Franssen, L. Nachtergaela, H. De Man, "Global communication and memory optimizing transformations for low power systems," *Proc. 1994 Workshop on Low Power Design*, pp. 203-208, April 94.