

Energy-efficient Code Generation for DSP56000 family

Sathishkumar Udayanarayanan
Center for Low Power Electronics
Department of Electrical Engineering
Tempe, AZ 85287-5706
usathish@asu.edu

Chaitali Chakrabarti
Center for Low Power Electronics
Department of Electrical Engineering
Tempe, AZ 85287-5706
chaitali@asu.edu

ABSTRACT

This paper presents a procedure to generate energy-efficient code for the Motorola DSP56K processor based on increasing the packing efficiency and minimizing the number of address instructions. The key features are a novel scheduling algorithm that reduces the dependencies between instructions, a register allocation algorithm that spills variables based on their *packability*, and an address code generation algorithm that minimizes the number of additional instructions. The size of the code generated by this procedure is on the average 45% (25%) smaller than that generated by Motorola's g56K (SPAM).

Categories and Subject Descriptors

2.3 [Software and System Design]: Compilers, DSP and embedded systems

General Terms

Code Generation, Low Power

1. INTRODUCTION

In order to design a system for low power applications, it is important to analyze and optimize power in all the components of the system. A large portion of the functionality of today's systems is in the form of software. Thus it is important to estimate and reduce the software component of the power cost.

Code generation is the process of generating the best set of target instructions that implement the source code. Traditionally, code generation algorithms have tried to optimize for code size or performance (time it takes to execute the program). Recently researchers [6, 1] have added energy to the performance metric. Instruction-level energy models [6, 4] for medium to high-end processors (like Intel 486DX2, Fujitsu SPARClite, Intel i960) indicate that there is very little variation in energy between different instructions of the same type. As a result, it is valid to assume that optimizing

for performance optimizes for energy in most cases, if not all.

Code generation for Digital Signal Processors (DSPs) is more involved than that of general purpose processors. This is because the DSP processors have non-homogeneous register sets, number of very specialized registers, very specialized functional units, restricted connectivity, limited addressing, and highly irregular datapaths. Our approach to energy-efficient code generation for DSP processors is to increase the number of *packed*¹ instructions and to generate as small an address code as possible. Increasing the number of packed instructions results in lower energy, since the energy consumed by a packed instruction is significantly lower than when the instructions are executed in sequence [6]. Reducing the number of address generation instructions reduces the code size and hence the energy consumption.

In this paper we first present a procedure for instruction scheduling and register allocation that increases the packing efficiency for the Motorola DSP56K processor. The scheduling algorithm, referred to as zigzag scheduling, reduces the dependencies between instructions, thereby allowing MOVE instructions to be packed efficiently. Register allocation and packing are done in an integrated manner; variables are spilled according to the *packability* of the spill. While these algorithms are presented for the DSP56K family of processors, there are several other DSPs including the ADSP 2100 family, NEC uPD7701x family which have a similar architecture and for which these algorithms can be adapted easily. The address generation algorithm exploits the features of the specialized address generation units that allow manipulation of address registers in parallel with ALU computation and memory accessing. The address generation procedure is very general, and could be used for most DSPs without much change. A comparison of the code size generated by our procedure with those generated by Motorola's g56K and SPAM [5] show significant improvements with average code size reductions of 45% and 25% respectively.

The rest of the paper is organized as follows: Section 2 discusses the algorithms involved in increasing the packing efficiency. Address code generation is described in Section 3. The code size comparisons are given in section 4.

¹Packing refers to grouping an ALU-type instruction with data transfer instructions into a single instruction word for simultaneous execution.

2. INCREASING PACKING EFFICIENCY

2.1 Motorola DSP56K architecture

DSP56K supports packing two data transfer instructions with a MAC/ALU instruction. It has two memory banks. There are 4 24-bit “input” registers (X0, X1, Y0, Y1) and 2 56-bit accumulators (A, B). The MAC/ALU always writes to the accumulator and the multiplier always takes one of its input from X0/X1 and the other from Y0/Y1. There are 2 Address Generation Units with 4 address registers each.

2.2 Zigzag scheduling

Zigzag scheduling is a scheduling procedure that directly supports packing. It achieves this by scheduling nodes such that a gap is created between dependent nodes and MOVES from ACC to Data registers can be inserted in parallel. (In comparison, in a contiguous schedule, the dependencies disallow MOVES to be done in parallel.) Zigzag scheduling is effective in architectures that have multiple memory banks, support parallel data MOVE operations and have a heterogeneous register set. The Zigzag scheduling algorithm has a worst case complexity of $O(n^2)$, where n is the number of nodes in the DFG.

2.3 Register allocation with packing

In this section, we describe the procedure to do register allocation and packing in an integrated manner. A detailed description can be found in [7]. First, the instructions are packed using a greedy algorithm. The algorithm looks at the i^{th} instruction and the two successive instructions, $i + 1$ and $i + 2$. If there are no dependencies and if there is a free (empty) data move operation, then packing is done. Next, register allocation is done in a way that chooses the variables to be spilled according to the *packability* of the spill (i.e. if a spill MOVE can be packed or not). Third, the instructions caused by the spilled variables are packed. There could be conflicts during register assignment between X and Y register sets. These are resolved by assigning a physical register from the opposite register set, if possible. Otherwise, a new instruction is added.

Register allocation can potentially introduce a large number of MOVE operations due to spills. So the cost of spilling a variable has to include whether the resulting store and load could be packed. Our cost function is a function of load/store costs, packability of store, packability of load, the number of cycles affected by this spill. The cost for parallel operations would possibly come from the instruction-level energy model for the processor. The cost for a new instruction is a measure of the number of cycles required for execution and the energy consumed by the instruction.

Complexity analysis: Let k be the number of spills, and let j be the average number of variables considered for spilling at each cycle. The complexity of the algorithm is $O(kjn)$. Typically, j and k are much less than n .

3. ADDRESS CODE GENERATION

Address computation constitutes a large fraction of the execution time for most programs and thus a good address generation scheme is clearly important. We illustrate the importance of address assignment with the help of the following example.

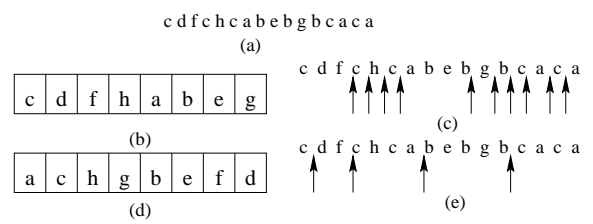


Figure 1: Example: (a) Access Sequence (b) An assignment based on first-use (c) The places requiring a jump in address value (d) An intelligent assignment (e) The places requiring a jump in address value

Example: Consider the access sequence shown in figure 1(a). The assignment shown in figure 1(b) is obtained on the basis of first-use of variables. Now consider the assignment shown in figure 1(d), where variables that are accessed consecutively several times have been placed in neighboring locations. The positions in the access sequence where auto-increment or auto-decrement cannot be done (i.e., a jump is required) are shown by arrows in figure 1(c) and (d). The number of jumps has reduced from 10 to 4, and since each jump requires an additional instruction it is clear that the placing the variables in proper locations is very important for efficient compiled code.

After register allocation, some variables get fixed to X-memory bank (S_x) and some to Y-memory bank (S_y) but some variables remain free to be allotted to either of the banks (S_f). Dual Access sequence (DAS) is the sequence of variables with two memory accesses. Using the information from DAS, we have to assign the free variables to a memory bank.

3.1 Memory bank partition

From the DAS, we obtain two access graphs(AGs). The nodes of the AG are the variables, and there is an edge between two nodes if they are accessed consecutively. A weight is placed on each edge which is equal to the number of times the two variables are accessed consecutively. Assuming that all the variables in S_f are put in X-memory(Y-memory), we obtain an access graph, let us call it $G_x(G_y)$. The weights on the edges could be variables like cx ; cx is 1 if variable c is put in X-memory and is 0 otherwise. The degree and weight of a node are the important parameters affecting the following optimizations. We perform a partition such that the increase in degree, increase in weight, are kept as low as possible. This makes sure that both X and Y partitions (P_x, P_y) have good access sequences.

Complexity Analysis: The procedure takes $O(n) + O(mk)$ where n is the length of the dual access sequence, m is $|S_x \cup S_y \cup S_f|$ and k is $|S_f|$.

3.2 General Offset Assignment

Definition: Given a variable set (P_x or P_y), an access sequence and the number of address registers (k), assign an address for each variable in the set such that the total addressing cost is minimized.

When $k = 1$, the problem is called Simple Offset Assignment(SOA). SOA can be mapped to the maximum weighted

Table 1: Code size comparisons, Number of packed instructions and Number of address instructions

Program	n	Code size comparisons			Number of packed instructions			Number of address instructions		
		g56K	SPAM[5]	ZP	g56K	SPAM	ZP	g56K	SPAM	ZP
complex_mult	6	21	18(14%)	10(52%)	4(19%)	4(22%)	3(30%)	3	2	2
MSE	11	37	36(2%)	16(56%)	15(40%)	9(25%)	10(62%)	6	3	2
SGAL filter	16	71	66(7%)	27(57%)	21(30%)	19(29%)	17(63%)	12	8	5
4_{th} order IIR	17	32	37(-15%)	26(18%)	15(46%)	8(21%)	13(50%)	9	7	4
8pt. DCT	34	168	105(37%)	73(56%)	22(13%)	23(21%)	37(50%)	48	16	9
8pt. IDCT	57	212	148(30%)	100(52%)	45(21%)	52(35%)	52(53%)	36	25	10

path covering problem. Since MWPC is NP-complete, Liao has described a heuristic algorithm that is similar to Kruskal's minimum spanning tree algorithm[3]. GOA is more complex and it has been suggested that the variables be partitioned into k sets and then SOA could be applied to each of them. Liao[3] and Leupers[2] have suggested heuristics to partition the variables into different ARs. While Liao's heuristic may lead to badly balanced partitions, Leupers' heuristics could be quite time consuming since it solves SOA for each unassigned variable k times.

Our heuristic tries to generate balanced partitions at considerable speed. The heuristic is based on Liao's observation that if a node in an access graph has more than two edges, then it might be better to move the corresponding variable to a different partition to be addressed by another AR. Each node v has a penalty $P(v)$, which is the sum of the weights on all edges except the two highest weights. Nodes with higher penalty have a higher priority of being moved to another partition. Only one node is moved at a time. The move operation is stopped if the new partition's penalty is more than that of the old partition(s). The algorithm iteratively increases the number of partitions (upto the limit of available address registers) if there is scope for improvement in cost.

Complexity Analysis: The algorithm takes $O(nm)$ time where n is the number of variables and m is the length of the access sequence.

3.3 Exploiting offset registers

The offset register (OR) has to be used effectively to reduce the number of additional instructions. For each AR, the sequence of values by which it needs to be modified, called the jump sequence S_j , is obtained. The values of different jumps are given in the set, V_j . The problem is to find the values to be loaded into the OR at specific times such that the number of additional instructions is minimized. We solve this problem using dynamic programming. This algorithm is applied to all available ARs.

Complexity Analysis: The complexity of the solution is $O(nk^2)$, where n is $|S_j|$ and k is the number of different jump values.

4. RESULTS

We implemented the algorithms and experimented with some typical DSP kernel algorithms, the results of which are given in Table 1. ZP refers to our procedure. In Table 1, the number of packed instructions in each program is given along

with the percentage of packed instructions in parenthesis. It can be seen that a higher percentage of instructions are packed by our procedure. It can also be seen from the table that our procedure reduces the number of additional addressing instructions. Code size (number of instructions) comparisons are also given in table 1. Percentage improvements in code size when compared to that generated by g56K are given in parenthesis. For the limited set of examples presented here, the average code size reduction is 45% compared to g56K and 25% compared to SPAM. Since the number of packed instructions obtained by our algorithm is comparable to that of SPAM, and the total number of instructions is much smaller than SPAM, the claim that our algorithm generates energy-efficient code is justified.

5. FUTURE WORK

An instruction-level energy model has to be developed. The algorithms presented have to be extended to procedures. The set of benchmarks has to be extended to real applications.

6. REFERENCES

- [1] C. H. Gebotys. An efficient model for dsp code generation: Performance, code size, estimated energy. In *International Symposium on System Level Synthesis*, pages 41–47, September 1997.
- [2] R. Leupers and P. Marwedel. Algorithms for address assignment in dsp code generation. In *International Conference on Computer Aided Design*, 1996.
- [3] S. Liao. *Code Generation and Optimization for Embedded DSPs*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1996.
- [4] J. T. Russell and M. F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *International Conference on Computer Design*, 1998.
- [5] SPAM. Spam home page. <http://www.ee.princeton.edu/spam>.
- [6] V. Tiwari, S. Malik, A. Wolfe, and M.-C. Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, pages 1–18, 1996.
- [7] S. Udayanarayanan and C. Chakrabarti. Energy-efficient code generation for dsp56000 family. Technical Report CLPE-TR-2-2000-27, Center for Low Power Electronics, 2000.