ACKNOWLEDGMENT

The authors would like to thank M. Wax for suggesting that we also apply our approach to proving the uniqueness results in [2].

REFERENCES

[1] M. Wax, "On unique localization of constrained-signals sources," *IEEE Trans. Signal Processing*, vol. 40, pp. 1542–1547, June 1992.
[2] M. Wax and I. Ziskind, "On unique localization of multiple sources by passive sensor arrays," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 37, pp. 996–1000, July 1989.

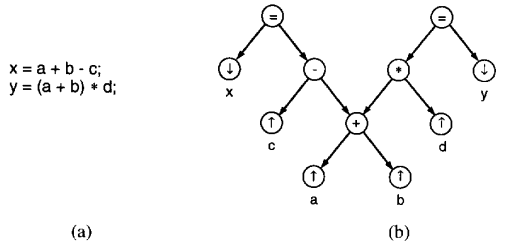


Fig. 1. (a) A fragment of C code; (b) its DAG representation. Up and down arrows represent fetch and write operations, respectively.

Efficient Scheduling and Instruction Selection for Programmable Digital Signal Processors

Kin H. Yu and Yu Hen Hu

***Abstract*— We present an efficient method for optimized instruction selection and scheduling for Programmable Digital Signal Processors. Our approach uses artificial intelligence techniques to yield code that is comparable to that of hand-written assembly codes by DSP experts. Several examples which demonstrate the feasibility of the approach, targeted to the TMS32020/50 architecture, are presented.**

I. INTRODUCTION

To fully utilize the available computing power of the modern programmable digital signal processor (PDSP) [1], software designers must face the difficult task of programming in assembly language. Although high-level language (HLL) compilers for PDSP's exist [2], most of them are based on technology developed with general purpose applications in mind. Genin *et al.* [3] estimated that assembly codes written by human DSP experts perform 5 to 50 times faster than those obtained from conventional HLL compilers of just a few years ago. Although the performance of current optimizing compilers has improved significantly, for real-time DSP applications with stringent constraints on execution time and/or code size, careful manual coding, typically with several fine-tuning iterations, is still the only effective approach.

This correspondence deals with code generation for PDSP's with nonuniform register sets. More specifically, we focus on the subproblems of *scheduling* and *instruction selection*. Other issues such as the handling of conditionals and branches, the use of circular buffers and special instructions are under study and will be reported in the future. In this correspondence, we describe an approach in which scheduling and instruction selection are handled concurrently, instead of in separate passes as often implemented in conventional compilers. Our measure of efficiency is the size and execution time of the generated assembly code. Many embedded applications in the DSP area depend directly on the efficiency of the executable code. Such applications may require a minimum code execution speed (for example, a certain computation must be finished before the next input data arrive) and/or a maximum code size (for example, to fit in the limited on-chip memory of the PDSP). For such applications, longer compilation time and larger memory requirement can still be more attractive than manual assembly coding. A prototype code generator targeted for a subset of the TMS3202x/C5x architecture and instruction set has been implemented. Experimental results reveal that our approach can yield codes that are comparable to that of hand-written assembly codes by DSP experts.

Manuscript received August 18, 1993; revised May 31, 1994. The associate editor coordinating the review of this paper and approving it for publication was Dr. Teresa H. Meng.

The authors are with the Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, WI 53706-1692 USA.

IEEE Log Number 9406018.

II. PROBLEM FORMULATION

An HLL program can be represented by a directed acyclic graph (DAG). Fig. 1 depicts a fragment of C code and its corresponding DAG. Nodes in the DAG represent operations to be performed and arcs indicate the data or control dependency among those operators. Similarly, each assembly instruction can be represented by a DAG-like pattern. Each pattern specifies, among other information, which registers it needs to read from and what state flags it will use before execution of the associated instruction. Each pattern also specifies which registers it will write to and what state flags it will modify after execution. We call the former *pre-conditions* and the latter *post-conditions*. In this context, it is easy to see that each pattern can "cover" part of the program DAG.

Code generation can be described as determining a sequence of instructions and their ordering, with compatible *pre-conditions* and *post-conditions*, from the given instruction set to realize all the operations specified in the DAG, subject to the DAG data dependency constraints. If each instruction also carries an associated cost, optimal code generation can be defined as determining the minimum cost cover for the program DAG. Unfortunately, it has been shown that optimal code generation is NP-complete [4].

Two important subproblems of code generation are *scheduling* and *instruction selection*. For example, the DAG in Fig. 1 indicates that the "+" node must be covered before either "−" or "*" nodes can be evaluated. However, there are no constraints on the covering order of the latter two nodes. Determining a valid evaluation order for all nodes, consistent with the partial order specified by the arcs, is known as *scheduling*.

Many PDSP's are based on the CISC (complex instruction set computing) concept, characterized by a nonorthogonal set of complex instructions. Consequently, certain parts of the DAG may be coverable by more than one instruction. Determining a set of instructions that can completely cover the program DAG, and for those parts that are coverable by many instructions, determining which instruction to

use, is known as *instruction selection*. The problem is complicated by the fact that the best instruction to cover a given DAG operator is *run-time context dependent*. Typical PDSP's have multipartitioned memory and nonuniform register sets in which certain registers and memory blocks are specialized for specific usage. In such architectures, scheduling and instruction selection affect each other [5]. In the next section we present a unified and context sensitive approach for scheduling and instruction selection.

III. Approach

A. Scheduling

At the topmost level, our scheduling algorithm is a heuristic list scheduler. First, a DAG is built from the input program. Next, the DAG is augmented to include flow-of-control information, followed by as-late-as-possible (ALAP) scheduling. This initial scheduling, however, does not determine the actual order in which nodes are covered. In the ALAP schedule, if node M must be covered before node N, M is scheduled as close to N's schedule as possible. The rationale for using ALAP is as follows. If M must be covered before N, then there must be an arc from N to M. If it is a *data dependency* arc, then N depends on M's result and executing M immediately before N makes the least commitment of resources, yielding the positive effect of reducing load/store instructions. If it is a *control dependency* arc, then any scheme that schedules M before N can be applied and ALAP provides a valid initial guess. At each iteration, all coverable nodes are collected; each coverable node is evaluated and the cheapest node is selected. In case more than one node evaluates to the minimum cost, the one with the lowest initial schedule is selected. The selected node is covered and removed from the DAG. The process recurs until the entire DAG has been covered.

Node coverage involves *pattern matching* while node evaluation involves *heuristic search*; both node coverage and node evaluation also involve *means-ends-analysis* (MEA) [6] and *hierarchical planning* [7], as we describe in [8].

TABLE I
Quality (Measured by Size and Execution Time) of the Assembly Code Generated by the Prototype Code Generator Compared to that Generated by a Conventional Optimizing Compiler and Handwritten Assembly Code

	PROTOTYPE			TI COMP.†	DSP EXPERTS	
	code size (# inst.)	exec. time‡ (ms)	compil. time* (s)	code size (# inst.)	code size (# inst.)	exec. time‡ (ms)
FFT	23	4.6	23.2	63	22[a]	4.4[a]
IIR	16	3.2	12.5	43	17[b]	3.4[b]
FIR	14	2.8	14.7	37	15[c]	3.0[c]
BIQUAD[d]	13	2.6	1.7	49	13	2.6
ROTATION	36	10.8	93.5	128	31[e]	14.4[e]
INVLATTICE[f]	24	4.8	302.5	56	26	5.2
FWDLATTICE[f]	25	5.0	42.7	58	25	5.0

Number of instructions excludes assembler directives and routine initialization instructions.
† TI Optimizing C Compiler, version 6.4, with all optimizations enabled (level 2)
‡ Estimate for TMS32020 at 200 ns per instruction cycle.
* Average user CPU time on a SUN SPARC IPX for 3 runs.
[a] Results taken from [12] page 77.
[b] Results taken from [12] (straight code implementation) page 39.
[c] Results taken from [12] (straight code implementation) page 32.
[d] *Example taken from [13].*
[e] Results taken from [12] page 297.
[f] Example adapted from [14] page 12-79.

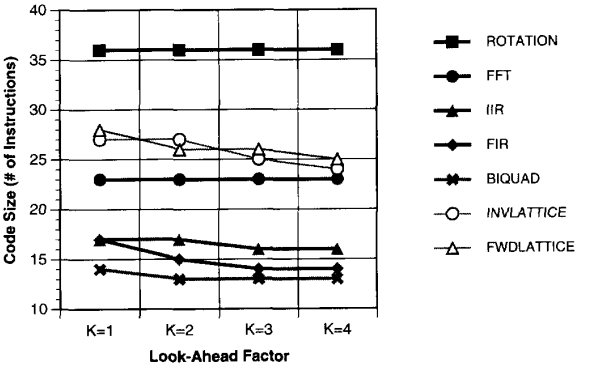


Fig. 2. Effects of the look-ahead factor K on the resultant code size.

B. Instruction Selection

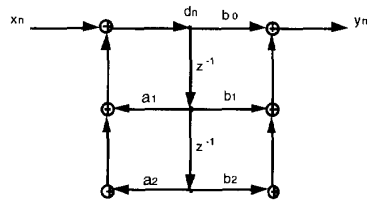
In most code generation techniques, instruction selection is statically predetermined, context insensitive and hard coded in form of small routines. For instance, in threaded code generation [9], each operation in the intermediate representation (IR) is replaced with a call to a subroutine that translates the IR operator into assembly instructions. Compact implementation and fast execution have made threaded code generation popular among some DSP synthesis systems [10], [11].

Many CISC PDSP's have a rich set of addressing modes and complex instructions that perform many operations in a single instruction. This leads to two potential problems when statically macroexpanding individual IR operators. Sometimes a single assembly instruction can cover more than one IR operator. Conversely, there may be many complex instructions with overlapping functionality that can cover an IR operator.

In our approach, each DAG operator is covered one at a time. If a complex pattern can cover more than one node, all involved nodes in the DAG are marked covered. If many patterns can cover a given node, heuristic search is used to determine the best one for the current context. However, a simple static evaluation function is not sufficient. In an analogy to game playing, it is necessary to look ahead several moves to discover that a seemingly bad move is in fact the best one. Our node evaluation algorithm implements a K-step look-ahead with a recursive heuristic search to determine the covering cost of candidate patterns. A *resource table* (RT) contains the status of all available resources (registers, stacks, memory partitions, etc.). A candidate pattern can be applied only if its preconditions are all satisfied by the RT. In case a pattern is not immediately applicable, a modified means-ends-analysis [6] combined with hierarchical planning [7] are utilized to formulate a plan of action. That plan consists of additional patterns, and their best application order, to completely satisfy the original pre-conditions. Once a pattern is applied, the RT is updated to reflect the pattern's post-conditions [8].

To illustrate the instruction selection process, let us consider the example IIR listed in Table I. Fig. 3 presents its flow graph and C implementation. A sample trace of the code generation illustrates hierarchical planning and MEA in action. The code generated by two runs of the code generator are presented side-by-side, one with $K = 1$ and the other with $K = 3$. The portion of the DAG affected is also shown. At the time step of interest, nodes 32 and 33 have been covered, with results stored in register **P** and **AC**, respectively. The algorithm must now evaluate the two expandable nodes, 34 and 35, and determine which one should be covered next. Node 34 is evaluated first. The best template for covering node 34 is $\mathbf{MPY_{dma}}$, which requires the template $\mathbf{LTA_{dma}}$ to satisfy its preconditions. On the other hand, node 35 can be immediately covered by template **APAC**. Hence, covering node 35 is cheaper then covering node 34

Flow Graph:



C Code:

```
#include <ioports.h>
IIR(int *dn; int *yn; int dnm1; int dnm2) {
  int xn;
  const a1, a2, b0, b1, b2;
   inport(0, &xn);
   *dn = xn + dnm1*a1 + dnm2*a2;
   *yn = (*dn)*b0 + dnm1*b1 + dnm2*b2;
   outport0, &yn);
}
```

Sample Trace from Code Generator

```
Enter input file name ('q' to exit): IIR
Enter Look Ahead Factor (K > 0): 3
Enter Cap on Branching Factor (BC > 0): 3
Enter Trace Level (0=None, 1=Minimum, 2=Detailed): 1
    ⋮
Searching •.....•..... Node 34 selected.
Covering operator *
  Expanding Node 34 (candidate 1)
    MPY_dma not applycable. Reducing ...
    PLAN: LTA_dma
        LTA_dma applied
    MPY_dma applied
  Node expanded.
  Expanding Node 34 (candidate 2)
    MAC_pma_dma not applycable. Reducing ...
    PLAN: TBLW_dma
        TBLW_dma applied
    MAC_pma_dma applied
  Node expanded.
**** 88% done ****
    ⋮
```

Influence of Look Ahead

```
Enter input file name ('q' to exit): IIR
Enter Look Ahead Factor (K > 0): 1
Enter Cap on Branching Factor (BC > 0): 3
    ⋮
Generated Optimized Code:
    var   EQU   dma
    var   EQU   dma
    var   EQU   dma
    var   EQU   dma
    var   EQU   dma
        IN    dma   pa
    var   EQU   dma
    var   EQU   dma
    var   EQU   dma
    var   EQU   dma
        LT    dma
        MPY   dma
        LTP   dma
        MPY   dma
        ADD   dma
        APAC
        SACL  dma
        MPY   dma
    var   EQU   dma
        LTP   dma
        MPY   dma
        APAC
        LT    dma
        MPY   dma
        APAC
        SACL  dma
        OUT   dma   pa

End of Job.
Code Size: 17 (non-free) instructions
Estimated execution time: 3.4 msec
```

```
Enter input file name ('q' to exit): IIR
Enter Look Ahead Factor (K > 0): 3
Enter Cap on Branching Factor (BC > 0): 3
    ⋮
Generated Optimized Code:
    var   EQU   dma
    var   EQU   dma
    var   EQU   dma
    var   EQU   dma
    var   EQU   dma
        IN    dma   pa
    var   EQU   dma
    var   EQU   dma
    var   EQU   dma
    var   EQU   dma
        LT    dma
        MPY   dma
        LTP   dma
        MPY   dma
        ADD   dma
        APAC
        SACL  dma
        MPY   dma
    var   EQU   dma
        LTP   dma
        MPY   dma
        LTA   dma
        MPY   dma
        APAC
        SACL  dma
        OUT   dma   pa

End of Job.
Code Size: 16 (non-free) instructions
Estimated code execution time: 3.2 msec
```

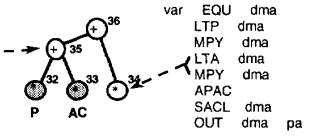


Fig. 3. Second-order direct form IIR filter.

and with $K = 1$ (no look-ahead), that is exactly the conclusion. Only with $K = 3$ (looking ahead 2 plays) one can see the advantage of investing a bit more in covering node 34 first: template $\mathbf{LTA}_{\mathrm{dma}}$ not only satisfies the preconditions of $\mathbf{MPY}_{\mathbf{dma}}$, but it also adds the contents of $\mathbf{P}$ to $\mathbf{AC}$ as a side effect, which in effect covers node 35.

IV. Results

A prototype code generator has been implemented in object-oriented $C++$. We used a target processor based on a simplified version of the TMS32020/C50's architecture.

Seven examples involving common DSP algorithms have been compiled with this prototype code generator and the results listed in Table I. As one can notice from Table I, the code generated by our prototype is much better (up to 3.8 times smaller) than that obtained with a commercial compiler such as the TMS320C2x/C5x Optimizing C Compiler from Texas Instruments. Furthermore, our prototype is capable of generating code whose quality is comparable to that generated by human DSP experts. Note that in example ROTATION, although the prototype generated more instructions, the program executes faster than the handwritten version. This is due to our heuristic function, which minimizes the sum of code size and

execution time. Incidentally, by varying the weights in the heuristic function, one can generate code that executes fastest (e.g., for real-time applications) or has the smallest size (e.g., to fit into a limited size PROM).

To study the effects of the look-ahead factor we varied K for all seven examples. As expected, the quality of the generated code, in general, improves when K increases, as shown in Fig. 2. For those cases in which the code size remained constant, we speculate that the heuristics used were the limiting factors. Fig. 2 also shows that a relatively small value of $K \leq 4$ is sufficient to obtain handwritten-quality code.

V. Future Research

An algorithm reformulation scheme that has knowledge of the target processor's architecture is currently under development. Such a scheme can transform the input program into a form more suitable for optimal code generation. Without transformations, the HLL plays an important role in the quality of the machine code. General purpose HLL's do not give the programmer direct access to unique features of PDSP's such as modulo-addressing register arithmetic, special fixed-point scaling modes, dual data memory access, and fractional data types. Developing a dedicated HLL for DSP certainly has its advantages [3], [11]. The only inconveniences are the learning curve associated with a new language and the time to rewrite common DSP routines, which may have already been coded in a conventional HLL, into the dedicated HLL.

In our model, scheduling and instruction selection consider the entire program rather than proceeding basic block-by-basic block. Hence, it can gather operations not only inter-basic block but also intra-basic blocks. Of course, if computational resources are limited, a few basic blocks would have to be considered at one time. Interaction between different runs is not yet supported, so the resultant code may not be globally optimized.

References

[1] D. Bursky, "DSPs expand role as cost drops and speed increases," *Electronic Design*, vol. 39, no. 19, pp. 53–65, Oct. 1991.
[2] D. Shear, "HLL compilers and DSP run-time libraries make DSP-system programming easy," *EDN*, pp. 69–74, June 23, 1988.
[3] D. Genin *et al.*, "System design, optimization and intelligent code generation for standard digital signal processors," in *Proc. IEEE Int. Symp. Circuits Syst.* (Portland, OR), 1989, vol. 1, pp. 565–569.
[4] A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code generation for expressions with common subexpressions," *J. Assn. Comput. Machinery*, vol. 24, pp. 146–160, 1977.
[5] I. Kuroda, A. Hirano and T. Nishitani, "A knowledge-based compiler enhancing DSP internal parallelism," in *Proc. 1991 IEEE Int. Symp. Circuit Syst.*, vol. 1, pp. 236–239.
[6] G. Ernst and A. Newell, *GPS: A Case Study in Generality and Problem Solving*. New York: Academic, 1969.
[7] E. D. Sacerdoti, "Planning in a hierarchy of abstract spaces," *Artif. Intell.*, vol. 5, pp. 115–135, 1974.
[8] K. H. Yu and Y. H. Hu, "Artificial intelligence in scheduling and instruction selection for digital signal processors," *Appl. Artif. Intell.*, 1994.
[9] J. R. Bell, "Threaded code," *Commun. ACM*, vol. 16, pp. 370–372, 1973.
[10] COMDISCO Systems Inc., "CGS-Code Generation System for SPW," CS-1005 3/90, 1990.
[11] E. A. Lee *et al.*, "Gabriel: A design environment for DSP," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 37, pp. 1751–1762, 1989.
[12] K.-S. Lin, Ed., *Digital Signal Processing Applications with the TMS320 Family*. Englewood Cliffs, NJ: Prentice-Hall, 1987, vol. 1.
[13] D. B. Powell, E. A. Lee, and W. C. Newman, "Direct synthesis of optimized DSP assembly code from signal flow block diagrams," in *Proc. IEEE Int. Conf. ASSP*, 1992, vol. 5, pp. V.553–V.556.
[14] Texas Instruments, *Third-Generation TMS320 User's Guide*, 1988.