DYNAMIC TRANSFORMATIONS IN OPTIMIZED CODE GENERATION FOR DIGITAL SIGNAL PROCESSORS

Kin H. Yu
University of Wisconsin - Madison
Department of Electrical and Computer Engineering
jimy@eckert.wisc.edu

ABSTRACT

In this paper we present an approach for performing dynamic context-dependent transformations (DCDT) to improve code generation for programmable digital signal processors. Unlike static optimizations, DCDT can guarantee to improve the quality of the generated code, at the expense of longer computational time. For many embedded DSP applications, hand coding in assembly is still the only effective approach. We show that our code generation approach, when combined with DCDT, can yield code of quality comparable to that of hand-written codes by DSP experts and many times superior to that generated by a conventional optimizing compiler.

1. INTRODUCTION

High level languages (HLLs) are attractive to programmers because they simplify the task of programming. Unlike assembly codes, HLL programs are readable, maintainable and portable to other processors. These features contribute to increase productivity and reduce development cost.

A HLL compiler translates the instructions present in a HLL program into assembly instructions, more easily understood by the processor. HLL compilers for programmable digital signal processors (PDSPs) have existed for several years [1]. Unfortunately, the performance of commercially available compilers for PDSPs is acceptable only to a few non-critical applications [2]. For embedded DSP applications with stringent constraints on execution time and code size, careful manual coding, typically with several fine-tuning iterations, is still the only effective approach.

A typical compiler can be viewed as a front end (FE) feeding a back end (BE). The FE translates the input HLL program into an intermediate representation (IR). The BE translates the IR into the output assembly code. The mapping of HLL instructions to IR operators is processor independent and is well defined using models such as context-free grammar [3]; many tools exist to automate that task. In contrast, the mapping of IR operators to assembly instructions, also known as code generation, is highly machine dependent and leaves much room for ad hoc approaches.

Even if a BE could generate optimal code for a given IR, that code may still be inferior to hand-written assembly code. The reason is simple. Although the IR can uniquely represent a given sequence of HLL instructions (program), the latter is not a unique implementation of a desired algorithm. In fact, there are infinite programs that evaluate a given expression. For example, $(a \times b)$ could also be implemented as $[(a+b)^2 - a^2 - b^2]/2$.

The use of HLLs emphasizes the issue of *algorithm transformation*. Unlike assembly languages, HLLs are, in principle, processor-independent. Without target architecture information, HLL programmers cannot bias the program towards certain constructs, as experienced assembly programmers often do. Hence, it is the responsibility of the compiler to perform such transformations.

In static transformations the compiler uses static analysis to determine the merits of a transformation. For instance, the apparently more complex implementation $[(a+b)^2 - a^2 - b^2]/2$ for $(a \times b)$ may actually make sense if the target processor is an analog computer. In such a machine, there may not be direct hardware support for multiplication. On the other hand, addition, subtraction and division can be easily implemented with resistors and operational amplifiers, and the exponential characteristics of certain analog elements can be exploited to implement the squaring operation. In such an environment, the transformation $(a \times b) \rightarrow [(a+b)^2 - a^2 - b^2]/2$ is context-independent, and knowing that, the compiler could simply replace, at parse time, all instances of $(a \times b)$ with $[(a+b)^2 - a^2 - b^2]/2$.

In contrast, context-dependent transformations are those which depend on either the values of the operands or the contents and status of the available resources such as registers and memory addresses, at run time. Such transformations are usually undecidable at parse time. For example, consider a processor in which subtraction and right shift take one machine cycle each, while multiplication takes 4 machine cycles. In normal circumstances, implementing $(a \times b)$ as $[(a+b)^2 - a^2 - b^2]/2$ would not make sense in such a machine. However, if the partial results $(a+b)^2$, a^2 and b^2 have been evaluated in previous statements and are still available in appropriate registers, the code for $[(a+b)^2 - a^2 - b^2]/2$ could execute in 3 machine cycles as opposed to 4 machine cycles for $(a \times b)$.

0-7803-2431-5/95 $4.00

In previous papers [4], [5], we presented a novel approach for optimized code generation for PDSPs called OASIS (Optimized Allocation, Scheduling and Instruction Selection). Results were encouraging overall. The quality of the generated code was many times better than that of a commercially available optimizing compiler and comparable to that of hand-written assembly code by DSP experts. In this paper we propose a set of DCDTs that complement that work.

2. ALGORITHM TRANSFORMATIONS

Algorithm transformation in software compilation is the process of rewriting a given implementation of an algorithm (or parts of it) into another implementation, which allows the generation of more efficient code. For our purposes, efficient means compact and fast. Twaddell [7] writes "the long-term trend in optimizations is for the compiler to effectively rewrite the code however it pleases with the user's code representing just and expression of intent."

In the framework for code generation presented in [4], [5] the HLL program is first parsed into a directed acyclic graph (DAG) [3]. Our system makes use of artificial intelligence techniques (means-ends analysis, hierarchical planning, expert system and heuristic search) to mimic the reasoning and planning processes used by human assembly programmers. Similar to template pattern matching (TPM) [6], all evaluations are performed at compile-time. Unlike TPM, however, template costs are dynamically obtained. This means that a template may have different costs for different program contexts. Performance evaluation showed that a prototype code generator targeted for the TMS3202x/5x PDSP can generate codes of comparable quality to those written by experienced assembly programmers and many times superior to those generated by a commercial optimizing compiler [8]. Nevertheless, the ability of human assembly programmers to modify certain parts of the algorithm being coded to suit the target architecture was not supported. We now present a methodology for algorithm transformations and describe how it can be integrated in our system.

2.1. Context-Dependent Transformations

In this paper, we extend OASIS' framework to support algebraic and boolean transformations. Many such transformations are performed by conventional optimizing compilers. In those compilers, however, transformations are static, context-insensitive. For example, if multiplication executes much slower than addition in the target processor, subtree $(a \times 2)$ could be replaced by subtree $(a + a)$. This transformation can be done at parse time and is context-independent. However, if both operations perform in the same number of cycles, the advantage of such transformation is unrecognizable at parse-time.

A common approach in those compilers is to parse the IR before code is generated and to perform transformations wherever possible, assuming that those transformations will always benefit code generation. If that assumption proves to be incorrect, the user must disable specific transformations and recompile. Such approach has two shortcomings. First, determining which transformations are responsible for the resulting inefficiency of the code is not trivial. Second, it cannot address the issue of a given transformation being advantageous in certain parts of the program but undesirable in others. We call the latter dynamic context-dependent transformations (DCDT).

Another difficulty with DCDT is that the code generator must decide not only if a certain transformation is worth performing, but also, in case several alternative transformations exist, which one to perform.

Our heuristic search framework can be easily extended to support this issue. Currently, each search node contains a copy of the remaining DAG to be covered. A pointer called CurrentNode points to the node in that DAG currently being covered. If N alternative patterns exist for the node pointed to by CurrentNode, the search node is replicated N times. Each replica contains a copy of the DAG, with the sub-DAG pointed to by CurrentNode replaced by one of the alternative patterns. A look-ahead heuristic search is then conducted for each replica and the cheapest among all replicas is preserved while others are discarded.

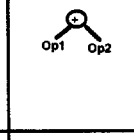
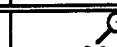
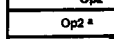
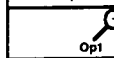
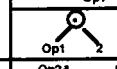
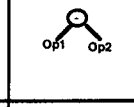
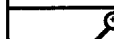
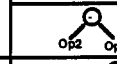
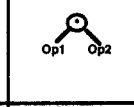
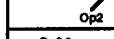
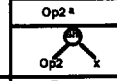
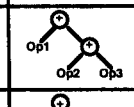
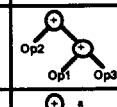
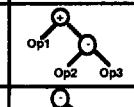
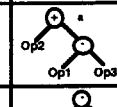
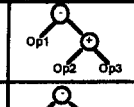
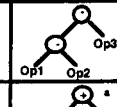
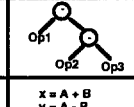
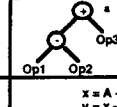
2.2. Reducing Search

Table 1 lists the transformations currently implemented in our prototype. The column Condition in Table 1 conveys heuristics to minimize the search space. A transformation is considered only if the listed condition is satisfied. If the condition is not satisfied, the transformation most likely yields either less compact or slower code, and hence is ignored.

For example, in transformation 6, there are two alternatives: $Op1 - (Op2 + Op3) \Rightarrow (Op1 - Op2) - Op3$ and $Op1 - (Op2 + Op3) \Rightarrow (Op1 - Op3) - Op2$. First, before these transformations can be even considered, the required condition that the accumulator contains Op1 must be satisfied. Second, if the condition is satisfied, which one of the two alternative transformations will yield better code depends on the run-time context. Both alternatives are evaluated (with K-step look-ahead) together with the original expression and the cheapest among the three implementations is retained while the others are discarded.

In another example, transformation 7 also has two possible alternatives: $Op1 - (Op2 - Op3) \Rightarrow (Op1 - Op2) + Op3$ and $Op1 - (Op2 - Op3) \Rightarrow (Op3 - Op2) + Op1$. The content of the accumulator dictates which alternative (if any) should be

Table 1: Some examples of DCDT performed by OASIS during instruction selection.

Transf.	Original Pattern	Transformed Pattern	Condition	Level
1				1
			a Op1=0 b Op2=0	1
			RegP = -Op2 || ACC = -Op2	1
			Op1 = Op2	2
2			a Op1=0 b Op2=0 c Op1=Op2	1
			Op1 ≠ Op2 & (RegP = -Op2 || ACC = -Op2)	1
			ACC = Op2	2
3				1
			a Op1=1 b Op2=1 c Op1||Op2=0	2
			Op1 = 2^x || Op2 = 2^x	2
4			ACC = Op1	2
5			a ACC = Op1 b ACC = Op3	2
6			ACC = Op1	2
7			a ACC = Op1 b ACC = Op3	2
8	x = A + B y = A - B	x = A + B y = x - 2B	ACC = x	2
9	x = A - B y = A + B	x = A - B y = x + 2B	ACC = x	2

used. The search process then determines if the original or the transformed expression will yield better code.

2.3. Dynamic Common Subexpression Elimination

Common subexpression elimination (CSE) is a popular technique applied by many optimizing compilers. In CSE, the compiler computes the values of expressions that always yield the same result, saves the result as a temporary value and uses that value instead of recomputing the expressions each time it encounters them.

In performing context dependent optimization, we must deal with three important issues. First, applying transformations *during* code generation can lead to what we call *dynamic CSE*. For example, consider the code in Example 1.

Example 1:

$(S1)$ $\quad f = a + (b - c);$

$(S2)$ $\quad g = a - (b + c);$

Static CSE would not detect any common subexpressions. However, if we apply transformations 5 and 6 from Table 1, the above code would result in

$(S1')$ $\quad f = b + (a - c);$

$(S2')$ $\quad g = (a - c) - b;$

and the common subexpression $(a - c)$ is created. In OASIS, performing dynamic CSE is a simple matter of traversing the DAG after each transformation is applied and merging common-subexpressions as they are encountered. This leads to the second issue.

Depending on the subexpression, the resource status and the target architecture, recomputing the expression could be cheaper than storing and reloading a value from register. The correct decision, to perform CSE or not, can only be made after both alternatives are evaluated. OASIS supports such evaluations through heuristic search.

The first two issues are undetermined at parse time. Evaluating all possible alternatives during code generation can help make that decision. The third issue is that of *look-ahead*, which must be used to further refine the evaluation. Let us consider Example 2 bellow.

Example 2:

$(S1)$ $\quad f = b + (a - c);$

$(S2)$ $\quad g = a - (b + c);$

Suppose we are processing statement $(S1)$ and that transformation 5 is applicable, i.e., the accumulator holds the value of variable b. Indeed, transforming $(S1)$ into

$(S1')$ $\quad f = a + (b - c);$

is advantageous because it eliminates the need to store b into a temporary location and reloading the accumulator with the value of a to compute $(a - c)$. A naive evaluation function would opt to transform $(S1)$. However, only by looking ahead one can see that if transformation 5 is not applied, and transformation 6 is applied on $(S2)$ instead, the result would be

$(S1)$ $\quad f = b + (a - c);$

$(S2')$ $\quad g = (a - c) - b;$

and performing the dynamic CSE on $(a - c)$ could be potentially more advantageous, especially if a and c are complicated subexpressions.

OASIS supports heuristic search with user specified look-ahead levels.

3. RESULTS

The code generator reported in [4], [5] was extended to implement the ideas presented in this paper. Table 2 summarizes the results. Along with the size of the generated code, it also lists the smallest look-ahead factor K necessary to obtain those results.

Table 2 - Performance of OASIS compared with TI compiler and hand-written assembly code.

HLL Program	TI Compiler* (# instr.)	OASIS without DCDT (# instr.)	OASIS without DCDT K	OASIS with DCDT (# instr.)	OASIS with DCDT K	Hand-written (# instr.)
xform5	18	9	2	7	4	7
xform6	18	9	3	7	3	7
xform56	27	14	6	11	6	12
xform4	24	9	2	8	5	8
xform7	23	10	4	8	6	8
xform47	27	14	6	12	6	12
FFT2	63	23	1	21	2	22[a]
FFT4	190	107	3	89	3	77[b]

The number of instructions excludes assembler directives and routine initialization instructions.
* The TI compiler is capable of static arithmetic transformations. Results obtained with all optimizations enabled (-o2).
[a] Results obtained from [9].
[b] Results obtained from [10].

Without DCDT, the code generated by our prototype is already much smaller than that obtained from the TI compiler, but not as small as the hand-written version. The reason is that when coding in assembly language, the programmer has access and, indeed, takes advantage of all kinds of "tricks" (algorithm transformations) that one can envision. However, with context-dependent transformations enabled, the prototype generates codes that are at least as small as their hand-written counterparts, except for FFT4, where we exceeded the available computational resources before we reached the limitations of our code generator. In two cases (xform56 and FFT2) it is even smaller than the hand-written version. Careful examination of both assembly versions of xform56 reveled that the prototype took advantage of dynamic CSE while the programmer missed that optimization opportunity in the hand-written version.

4. CONCLUSIONS

The experimental results confirm the efficacy of context dependent transformations. The only drawback is that more look-ahead levels are required when transformations are involved. Although our heuristic search formulation executes in (second order) polynomial time with the program size, it is exponential with the look-ahead factor. For time-critical applications, transformations are well worth the increased computational time.

REFERENCES

[1] D. Shear, "HLL Compilers and DSP Run-Time Libraries Make DSP-System Programming Easy," *EDN*, vol. 33, no. 13, pp. 69-74, Jun. 23, 1988.

[2] M. Sabatella, "Barking Up The Wrong Tree: Why Optimizing Compilers Are Still Unable to Match Assembly Language," Berkeley, CA:Univ. of California - Berkeley, report UCB/CSD 88/428, Aug., 1988.

[3] C. N. Fischer and R. J. LeBlanc Jr., *Crafting a Compiler*. Menlo Park, CA:Benjamin/Cummings, 1988.

[4] K. H. Yu and Y. H. Hu, "Optimal Code Generation for Programmable Digital Signal Processors," in *Proc. of the International Conference on Acoustics, Speech and Signal Processing*, Minneapolis, MN, IEEE, 1993,

[5] K. H. Yu and Y. H. Hu, "Artificial Intelligence in Scheduling and Instruction Selection for Digital Signal Processors," *Applied Artificial Intelligence*, vol. 8, pp. 377-392, 1994.

[6] A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees," *Journal of the ACM*, vol. 23, pp. 488-501, 1976.

[7] W. B. Twaddell, "Optimizations Ignite New Battle in the Compiler Wars," *Personal Engineering and Instrumentation News*, vol. 10, no. 2, pp. 25-32, February, 1993.

[8] Texas_Instruments, "TMS320C2x/C5x ANSI C Compiler," Texas Instruments, Houston, TX, version 6.40, 1992.

[9] P. Papamichalis and J. So, "Implementation of Fast Fourier Transform Algorithms with the TMS32020," in *Digital Signal Processing Applications with the TMS320 Family*, K.-S. Lin, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1987, pp. 69-168.

[10] C. S. Burrus and T. W. Parks, *DFT/FFT and Convolution Algorithms Theory and Implementation*. New York, NY: John Wiley & Sons, 1985.