Novel Code Optimization Techniques for DSPs

Rainer Leupers

University of Dortmund
Department of Computer Science 12
44221 Dortmund, Germany
e-mail: leupers@ls12.cs.uni-dortmund.de

ABSTRACT

Software development for DSPs is frequently a bottleneck in the system design process, due to the poor code quality delivered by many current C compilers. As a consequence, most of the DSP software still has to be written manually in assembly language. In order to overcome this problem, new DSP-specific code optimization techniques are required, which, in contrast to classical compiler technology, take the detailed processor architecture sufficiently into account. This paper describes several new DSP code optimization techniques: maximum utilization of parallel address generation units, exploitation of instruction-level parallelism through exact code compaction, and optimized code generation for IF-statements by means of conditional instructions. Experimental results indicate significant improvements in code quality as compared to existing compilers.

1. INTRODUCTION

More and more DSP system designs are based on software running on programmable processors rather than on dedicated hardware [1]. This trend towards software-based implementation is due to the fact, that software provides higher flexibility and better opportunities for reuse than hardware.

Today, however, software development for DSPs frequently is a bottleneck in the system design process. It is well-known that many of the currently available C compilers for DSPs cause a significant overhead in code size and performance as compared to hand-written assembly code. This is confirmed by numerous software developers and recent empirical studies from academia and industry. According to [2], the compiler overhead may be in the order of several hundred percent. Such an overhead can hardly be tolerated in presence of real-time constraints and limited program memory size. Therefore, time-consuming assembly-level programming is still predominant in the area of DSP, and better compilers are among the development tools most urgently demanded by embedded system designers [1]. As a consequence, efficient code generation techniques for DSPs have received high attention during the last years (cf. [3, 4, 5] for overviews).

The overhead of compiler-generated code is mainly due to the special architectural features of DSPs, to which classical code optimization techniques can hardly be applied. This includes the presence of special-purpose registers, special addressing modes, and instruction-level parallelism. In order to make the use of high-level language compilers feasible for more DSP applications, new DSP-specific code optimization techniques are required, which take into account the detailed processor architecture. An important constraint in this context is, that high compilation speed is not necessarily an issue for DSP compilers. Instead, many compiler users are willing to trade higher compilation times against better code quality. This allows to explore the use of code optimization algorithms of a comparatively high computational complexity.

The purpose of this paper is to present several new DSP-specific code optimization techniques. Experimental results indicate that the use of such techniques may significantly reduce the overhead of compiler-generated code. The organization of the paper is as follows. Section 2 describes techniques for utilization of special addressing modes in DSPs. Section 3 is focused on exploitation of instruction-level parallelism through code compaction. Section 5 deals with optimized code generation for if-statements by using conditional instructions. Finally, section 6 provides experimental results obtained by applying the proposed techniques to different DSPs.

2. ADDRESS GENERATION

As compared to CISC processors, DSPs show very restricted memory addressing modes. Frequently, only direct (via the instruction word) and indirect addressing (via special address registers) modes are supported. However, address generation units (AGUs) DSPs usually provide support for auto-increment and auto-decrement of address registers in parallel to operations of the central data path. Examples are the TI TMS320C25, the Motorola 56k, and the Analog Devices ADSP-210x. This feature allows for parallel next-address computation, whenever the accessed program variables are appropriately mapped to memory locations. In addition, many AGUs comprise modify (or index) registers intended to store frequently required address modification constants. Fig. 1 shows the general architecture of such an AGU, which contains a file of k address registers (ARs) and m modify registers (MRs).

In the following, we outline optimization techniques

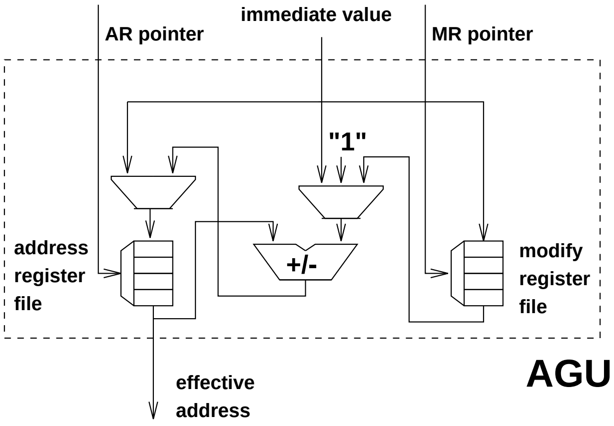

Fig. 1. *Address generation unit (AGU) model*

that aim at allocating ARs and arranging variables in memory, such the the use of auto-increment address computations is maximized.

2.1 Scalar variables

After code generation, the exact order of accesses to scalar program variables is known. Usually, source languages, such as C, do not prescribe any specific order of local variables in memory. Therefore, a compiler may compute a good layout of variables in memory, tailored towards the variable access sequence.

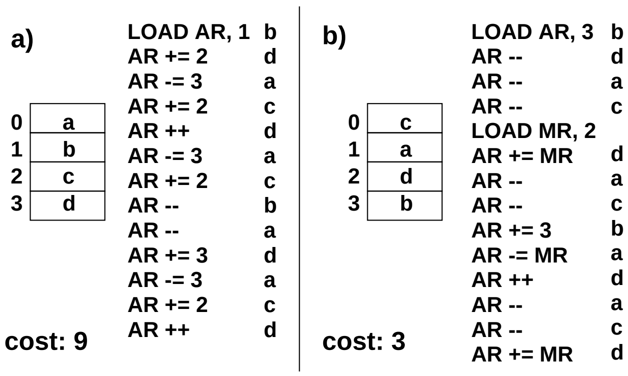

Fig. 2. *Addressing of scalar variables*

As an example, consider a variable set $V = \{a, b, c, d\}$ and a variable access sequence $S = (b, d, a, c, d, a, c, b, a, d, a, c, d)$. Suppose, one AR and one MR are available for generating the required memory addresses for S. Fig. 2 a) shows (in C-like notation) the corresponding sequence of AGU operations for a "naive" memory layout, where variables are mapped to memory cells in lexicographic order. Since only 4 out of 13 address computations are implemented by parallel auto-increment/decrement operations, there is a cost value of 9, i.e., 9 extra instructions are required for explicit address computations. A better memory layout is shown in fig. 2 b). Additionally, modify register MR is used in the AGU operation sequence to store the multiply required address modification value 2. Since the use of MR values as address modifiers neither incurs overhead, in total, this address generation scheme only requires 3 "costly" AGU operations, while the others can be executed in parallel to other machine instructions.

We have designed both heuristic graph-based and genetic algorithm based techniques, which construct good memory layouts. These techniques are capable of constructing close-to-optimum scalar address generation schemes for arbitrary numbers of ARs and MRs in the AGU.

2.2 Arrays

In contrast to scalar variables, the memory layout for arrays is typically fixed, so that only the allocation of ARs for accesses to array elements can be optimized. Again, the goal is maximum utilization of auto-increment addressing, so as to optimize code size and performance. Consider the following array access pattern within a for loop:

```
for (i = 2; i <= N; i++)
{ /* a_1 */   A[i+1]
  /* a_2 */   A[i]
  /* a_3 */   A[i+2]
  /* a_4 */   A[i-1]
  /* a_5 */   A[i+1]
  /* a_6 */   A[i]
  /* a_7 */   A[i-2]
}
```

If only a single AR were used, the AGU operation sequence for computing the addresses in the loop body would look as follows:

```
AR1 = &A[3]  /* initialize AR1 with &A[2+1] */
for (i = 2; i <= N; i++)
{ /* a_1 */   AR1 --     /* access A[i+1] */
  /* a_2 */   AR1 += 2   /* access A[i]   */
  /* a_3 */   AR1 -= 3   /* access A[i+2] */
  /* a_4 */   AR1 += 2   /* access A[i-1] */
  /* a_5 */   AR1 --     /* access A[i+1] */
  /* a_6 */   AR1 -= 2   /* access A[i]   */
  /* a_7 */   AR1 += 4   /* access A[i-2] */
}
```

However, this scheme would involve 5 costly address computations per loop iteration. Another naive approach would be to allocate a separate AR for each of the 7 accesses. In this case, all address computations could obviously be covered by auto-increment, but this would imply a waste of ARs. Our optimization technique minimizes the number of allocated ARs while avoiding address computation overhead. For instance, consider the array access pairs (a_1, a_2) and (a_1, a_3). Since in both cases the absolute address distance is 1, sharing of an AR would be possible without introducing costly address computations. This relation can be modeled by a "distance graph" (fig. 3), which contains a node for each array access and an edge between nodes v and w, if the the address for w can be computed from the address of v by auto-increment or decrement.

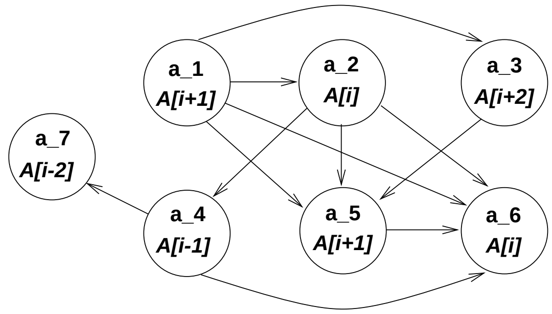

Fig. 3. *Distance graph for array accesses in loops*

One can show that the problem of optimal AR allocation is equivalent to a path covering problem on the distance graph. We have designed a branch-and-bound technique to compute optimal path covers. In case of the above example, the following addressing scheme with 3 ARs is optimal, which requires only auto-increment/decrement operations:

```
AR1 = &A[3]  /* initialize AR1 with &A[2+1] */
AR2 = &A[2]  /* initialize AR2 with &A[2+0] */
AR3 = &A[0]  /* initialize AR3 with &A[2-2] */
for (i = 2; i <= N; i++)
{  /* a_1 */    AR1 --   /* access A[i+1] */
   /* a_2 */    AR2 --   /* access A[i]   */
   /* a_3 */    AR1 --   /* access A[i+2] */
   /* a_4 */    AR2 ++   /* access A[i-1] */
   /* a_5 */    AR1 ++   /* access A[i+1] */
   /* a_6 */    AR2 ++   /* access A[i]   */
   /* a_7 */    AR3 ++   /* access A[i-2] */
}
```

3. CODE COMPACTION

Most DSPs show a certain degree of instruction-level parallelism (ILP). A TI 'C25, for instance, can execute a multiply-accumulate operation in parallel to an address computation within a single instruction cycle. Obviously, exploitation of ILP is a major source for code optimization. A popular technique for this purpose is code compaction. Code compaction reads a piece of sequential machine code, and assigns instructions to a minimum number of control steps, such that all inter-instruction dependencies and restrictions imposed by the instruction format are obeyed.

As an example, consider the expression tree shown in fig. 4. Sequential assembly code implementing this tree on a TI 'C25 DSP is shown in fig. 5 a). The 'C25 instruction set allows to combine different instruction pairs to single instructions. For instance, an APAC (add P register to accumulator) and an LT (load T register) can be compacted to an LTA instruction, and an APAC and a MPY (multiply) can be compacted to MPYA, whenever not prevented by data dependencies. However, the most efficient compaction scheme is usually far from obvious.

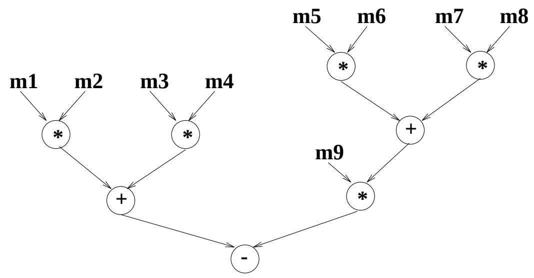

Fig. 4. *Expression tree*

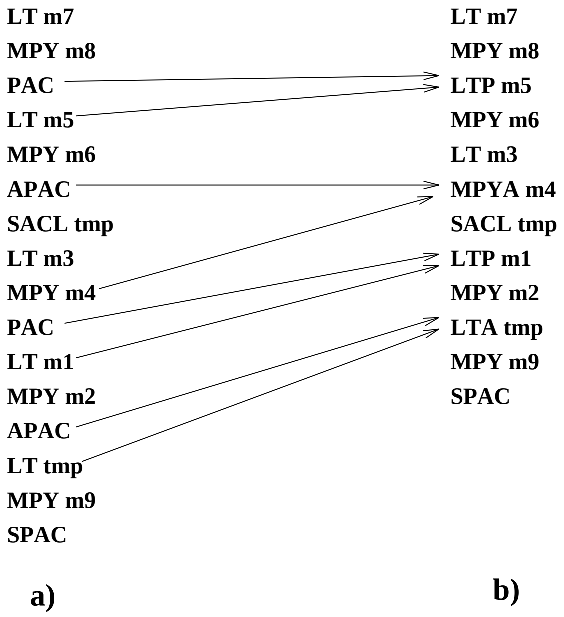

Fig. 5. *Sequential and compacted 'C25 assembly code*

Fig. 5 b) shows an optimal compaction for the sequential assembly code. In this case, a reduction from 16 to 12 instructions (25 %) is achieved.

One problem in code compaction for DSPs is that classical heuristic code compaction techniques [6], mainly developed for VLIW machines, can hardly be applied directly. The instruction format of DSPs sometimes permits alternative encodings for the same instruction, and also undesired side effects in the compacted code have to be avoided. In order to overcome the limitations of earlier heuristic compaction algorithms, we are using a technique based on Integer Linear Programming. In this approach, only the problem constraints (such as inter-instruction conflicts and dependencies) are specified in the form of linear (in)equations. Then, the equation system is solved by a standard tool. This guarantees (locally) optimally compacted code. Alternatively, a cycle constraint can be imposed on the compacted code. The technique is flexible enough to cope with alternative encodings and possible side effects. Even though Integer Linear Programming is of exponential complexity, we empirically found that it is still fast enough to solve many compaction problems of small to medium size. For a 'C25, code blocks of a length up to 50 instructions can typically be compacted within one minute of SPARC-20 CPU time.

4. CONDITIONAL INSTRUCTIONS

The source code of control dominated applications typically contains a large number of if-then-else (ITE) statements. Classical compiler technology uses conditional jumps for implementation of ITE statements. However, a frequent change of control flow due to many conditional jumps in the machine code has a strongly negative impact on performance in particular for deeply pipelined and highly parallel VLIW-like processors. On a TI C62xx, for instance, any jump incurs up to 5 stall cycles resulting in a performance waste of up to 40 instructions (8 per cycle). Therefore, recent VLIW DSPs permit to replace conditional jumps by conditional (or predicated) instructions. A conditional instruction is a term `[C] I`, where the condition C is a Boolean variable stored in a register and I is any "regular" machine instruction, e.g., an arithmetic operation, a register move, or a jump. The semantics of a conditional instruction is that instruction I is *effectively* executed, if and only if the condition C evaluates to true at the point of time when the control flow in a machine program reaches instruction I. Otherwise, instruction I behaves like a no-operation.

The availability of conditional instructions leads to the presence of two alternative ITE implementation schemes for a compiler. We denote the schemes with conditional instructions and conditional jumps by c-exec and c-jump, respectively.

4.1 The c-jump scheme

Consider an ITE statement of the form

```
if <cond> then <B_T> else <B_E>
```

where `<cond>` denotes a condition, and `B_T` and `B_E` are the then and else blocks of the statement. The standard replacement scheme using conditional jumps looks as follows:

```
                 c := evaluate(cond)
             [c] goto then_label
                 B_E
                 goto join_label
then_label:      B_T
join_label:      ...
```

The condition is evaluated into a register c, and dependent on the value of c, either B_T or B_E are executed. Then, control flow joins at the next instruction. Let $T(B)$ denote the time to execute a basic block B, and let J denote the (machine-dependent) jump penalty, including the time for executing the jump instruction itself. If the conditional jump is taken (i.e., condition `c` is true) then the execution time for the ITE statement S is $T_T(S) = J + T(B_T)$. If the jump is not taken, then $T_E(S) = 2 \cdot J + T(B_E)$. The worst-case execution time is $T(S) = \max(T_T(S), T_E(S))$.

4.2 The c-exec scheme

A semantically equivalent implementation using conditional instructions is:

```
      c := evaluate(cond)
[c]   B_T
[!c]  B_E
```

The notation "`[c] B_T`" denotes the conditional execution of all instructions in block B_T. The worst-case execution time when using c-exec is $T(S) = T(B_T \circ B_E)$, where "$\circ$" denotes the concatenation of basic blocks. In total, c-exec leads to a shorter worst-case execution time than c-jump, exactly if

$$T(B_T \circ B_E) < \max(J + T(B_T), 2 \cdot J + T(B_E))$$

A potential advantage of c-exec lies in the fact, that in VLIW processors, $T(B_T \circ B_E)$ is frequently much less than $T(B_T) + T(B_E)$, because the instructions in B_T and B_E may be partially executed in parallel. On the other hand it is obvious that c-exec is not guaranteed to be the fastest alternative in any case.

4.3 Implementation selection

We select the fastest implementation (w.r.t. the worst-case execution time) for an ITE statement by means of estimations and a dynamic programming algorithm. The estimation functions essentially count the number of statements in the then and else block of an ITE statement. In case of nested ITE statements, some additional instructions have to be inserted into the c-jump and c-exec schemes shown above, which ensure the correct propagation of *preconditions* to lower-level ITE statements. Preconditions reflect the fact, that some nested ITE statement must only be executed, if the condition of the "surrounding" ITE statement has been evaluated to true. This additional code is also taken into account during estimation.

The main problem is to select the fastest ITE implementation schemes across all nesting levels, because there is a cyclic dependence of the execution speed of ITE statements at different nesting levels. The dynamic programming algorithm breaks this cyclic dependence while exploiting the estimations as subroutines. In a bottom-up fashion, four cost estimation values are computed for each ITE statement, which depend on whether the statement is implemented by c-jump or c-exec, and whether or not a precondition has to be passed the the next nesting level. Afterwards, a top-down pass actually selects the fastest implementations for the ITE statements at all levels.

5. EXPERIMENTAL RESULTS

We have experimentally evaluated the techniques outlined in the previous sections for different DSPs. The

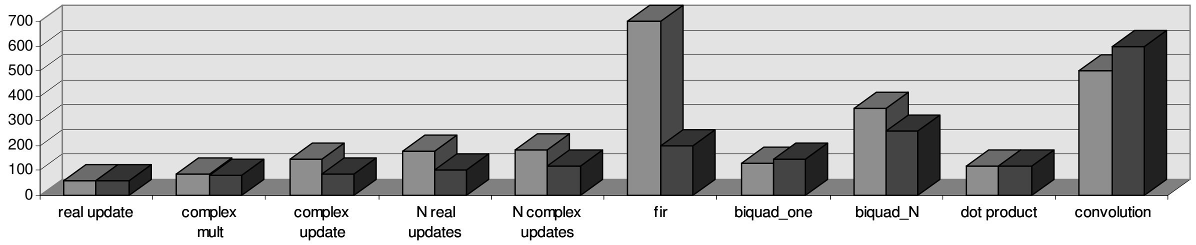

Fig. 6. *Experimental results: relative code size for DSPStone benchmarks and TI 'C25 DSP*

address generation and code compaction techniques described in sections 2 and 3 have been implemented in RECORD, a retargetable compiler for a class of fixed-point DSPs [5]. We have used RECORD to compile the DSP-Stone benchmarks [2] into machine code for a TI 'C25 DSP and compared the code size with the machine code generated by the TI 'C25 ANSI C Compiler. The results are shown in fig. 6.

The left columns show the overhead (in percent compared to hand-written assembly code) produced by the TI compiler, while the right columns show the corresponding results produced by RECORD. On the average, RECORD was able to halve the overhead as compared to the TI compiler. However, this achievement comes at the price of an increase in compilation time. Due to the use of comparatively time-intensive optimization techniques, the compilation speed is in the order of 2-5 generated instructions per CPU second. As mentioned above, however, high compilation speed frequently is not the most critical resource in the area of DSP, but better code quality justifies lower compilation speed.

The optimization of IF-statements described in section 4 has been evaluated for a TI C62xx VLIW DSP (table I). We have extracted 10 control-intensive pieces of C source code from an ADPCM transcoder and an MPEG package. These program fragments have been compiled by means of the ITE implementation selection algorithm and the TI *assembly optimizer* (column "opt"). Again the results have been compared to those directly produced by the TI C6x ANSI C compiler (column "TI"). Even though we are currently using rather simple estimation functions, faster code has been generated in most cases. This is due to the fact, that the proposed technique makes more intensive use of conditional instructions (across several nesting levels) than the TI compiler. However, also an increase in code size has been measured. So, the applicability of the optimization technique from section 4 depends on the code optimization goal (size or speed).

source	opt	TI
adapt_quant	11	15
adapt_predict1	13	13
adapt_predict2	22	27
diff_comp	12	10
outp_conv	24	21
code_adj1	23	30
code_adj2	49	51
code_adj3	30	41
detect_pos	27	29
find_mv	30	28

TABLE I
Experimental results: worst-case execution time (instruction cycles) for TI C62xx DSP with and without optimization of IF-statements

6. CONCLUSIONS

In this paper, we have proposed several new DSP code optimization techniques beyond the scope of classical compilers, and we have experimentally shown their practical applicability. Many of the techniques are efficient and easy to implement, so that they could be integrated into commercial compilers.

Future work will concentrate on further optimization techniques, with emphasis on VLIW DSPs. The main goal is to provide compiler technology, that is capable of replacing assembly-level programming of DSPs by the use of high-level languages and compilers, so as to enable higher productivity in DSP software development.

REFERENCES

[1] P. Paulin, M. Cornero, C. Liem, et al.: *Trends in Embedded Systems Technology*, in: M.G. Sami, G. De Micheli (eds.): *Hardware/Software Codesign*, Kluwer Academic Publishers, 1996

[2] V. Zivojnovic, J.M. Velarde, C. Schläger, H. Meyr: *DSPStone – A DSP-oriented Benchmarking Methodology*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994

[3] P. Marwedel, G. Goossens (eds.): *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995

[4] C. Liem: *Retargetable Compilers for Embedded Core Processors*, Kluwer Academic Publishers, 1997

[5] R. Leupers: *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, 1997

[6] S. Davidson, D. Landskov, B.D. Shriver, P.W. Mallett: *Some Experiments in Local Microcode Compaction for Horizontal Machines*, IEEE Trans. on Computers, vol. 30, no. 7, 1981, pp. 460-477