COOPERATIVE REGISTER ASSIGNMENT AND CODE COMPACTION FOR DIGITAL SIGNAL PROCESSORS WITH IRREGULAR DATAPATHS

Werner Kreuzer and Bernhard Wess

Institut für Nachrichtentechnik und Hochfrequenztechnik
Technische Universität Wien, A-1040 Vienna, Austria
email: Werner.Kreuzer@tuwien.ac.at

ABSTRACT

We address the phase ordering problem of code compaction and register assignment in a data flow graph compiler. During register assignment, we take into account the instruction-level parallelism available. Symbolic variables in straight-line code are allocated to register set/memory location pairs which maximally preserve the freedom available for code compaction. Whenever necessary, spill code is inserted during final register assignment and scheduled during code compaction. Register assignment is performed taking into account its impact on code compaction. This strategy results in final code of high quality.

1. INTRODUCTION

The increasing use of application-specific instruction set processors, like digital signal processors (DSPs), and the necessity to meet stringent cost constraints in digital systems design have caused new demands for code generation techniques which are not yet accomplished efficiently by traditional compiling techniques [1]. Usually, integrating the processor core, program ROM and RAM, and an ASIC circuitry on a single IC results in a severe limitation of program ROM size. Therefore, there is an urgent need for compilers which generate most dense code obtainable with highest performance. Unfortunately, high irregularities in the data path of many DSP architectures make optimal utilization of processor resources a highly complicated task.

In the course of generating assembly code for DSPs, solutions for various highly interdependent and exponentially complex problems have to be found, e.g., instruction selection, scheduling, register allocation, register assignment, and compaction. Solving these tasks simultaneously would result in a computationally intractable optimization problem. On the other hand, achieving optimal or near-optimal solutions for each subtask separately does not guarantee optimal or near-optimal final code. In this paper, we focus on the integration of register assignment and code compaction for a data flow graph (DFG) compiler for DSPs with irregular datapaths. In the front end, DFGs are transformed into intermediate straight-line code. In this intermediate representation, the results of instruction selection and scheduling of ALU/MAC operations are specified. Additionally, intermediate results which can reside in result or feedback registers are assigned to minimize register-register transfers.

Our approach to the phase ordering problem of register assignment and code compaction is to handle them in two separate phases, but to make register allocation sensitive to its impact on subsequent code compaction. This organization of the individual phases has the following advantages: (1) By keeping separate phases, the complexities of attempting to perform register assignment and code compaction simultaneously are avoided. (2) By applying code compaction after register assignment, spill code inserted during register assignment will be carefully scheduled. Although the compaction and register assignment algorithms have to be based on heuristics to solve these NP-hard problems in a reasonable amount of time, pursuing the proposed strategy results in assembly code of high quality.

In the remainder of this paper we describe our register assignment algorithm (Section 2) and we give a brief overview of the implemented compaction algorithm (Section 3). Section 4 provides experimental results for the Analog Devices ADSP-21xx general purpose DSP family. Section 5 gives a summary.

2. REGISTER ASSIGNMENT

Register assignment in code generation is the procedure of finding the best use of a fixed set of registers under hardware constraints; i.e., it attempts to map the registers such that the number of memory references is minimized. Register assignment is classified under the NP-complete problems [2] and can be considered as a graph-coloring problem [3].

Since this technique and its improvements [4, 5] were originally developed for global optimization for architectures with regular register files, they do not directly address dedicated architectures with special purpose registers. DSP architectures may have irregular datapaths and therefore new approaches are required considering the heavy interdependence between code compaction and register assignment.

This work was supported by the Fonds zur Förderung der wissenschaftlichen Forschung under research grant P10701-ÖTE.

2.1. Register Sets, Register Set Allocation

Considering architectures with irregular datapaths, in a conventional approach, allocating registers has to include the selection of a valid register for each appearance of a variable in the straight-line code. Therefore, register allocation not only has to decide which variables can reside in a register but also has to select a specific register for each variable. Thus, register allocation and assignment are performed actually in one step. To put it optimistically, integration of register allocation and assignment may result in better overall performance of code generation. However, the potential improvement is more or less compensated by the difficulty of solving both problems simultaneously. In [6], a graph labeling method which applies simulated annealing for memory bank and register assignment is proposed. This method results in high compilation time even for medium-sized programs.

To avoid the increased computational complexity caused by the enlarged solution space, we employ *register set allocation*. Register sets contain all registers which can be used equivalently in an instruction. Register set allocation means to select the best fitting register set for a symbolic variable under the assumption that it can reside in a register. The decision either in which specific register of that set a variable will reside or if it has to be spilled to memory, is made during final assignment. The concept of register sets for DSP code generation is not new [7]. However, we approach register assignment not as an separate problem. In contrast to [7], we allocate symbolic variables to register sets taking into account potential instruction-level parallelism.

```
 1: AR  = y27.1 + y25
 2: MR  = AR * c4
 3: y22 = MR + y25
        :
11: MR  = y12 * c2
12: AR  = y23 + y12
        :
26: AR  = AF - AR
27: y26 = AR
28: AR  = AR + y28
29: y29 = AR
30: AR  = y19 + y14
31: y21 = AR
32: AR  = AR + y15
33: y   = AR
34: AR  = y17 + y9
35: y16 = AR
```

Figure 1: Intermediate straight-line code for a 2^{nd} order WDF filter and the ADSP-21xx family.

Figure 1 shows a segment of intermediate straight-line code for a 2^{nd} order WDF filter. The target architecture is the Analog Devices ADSP-21xx family. As can be seen, register assignment is specified only for result (`AR`, `MR`) and feedback (`AF`) registers. Any operand which has to be loaded from memory is still labeled by a symbolic variable. Additionally, results which have to be written to memory are specified: inputs for delay elements (`y26`,`y29`,`y21`,`y16`) and the filter output (`y`).

Register set allocation starts with the first unassigned symbolic variable used as an operand for an ALU/MAC operation in the straight-line code. If there are no user-defined limitations for the selection of an appropriate register set, that register set is chosen which offers the highest degree of potential instruction-level parallelism. For example, the ADSP-21xx can execute two load instructions concurrently and also concurrently with an ALU/MAC operation if some resource constraints are met[1]. If these constraints are violated, only one load instruction can be executed concurrently with an ALU/MAC operation. In a first step, for symbolic variables which are only used once in a basic block, register sets are allocated to meet register constraints for the highest degree of instruction-level parallelism (`c4` in instruction 2 will be allocated to register set `MY` and memory bank `PM`; conf. Fig 3).

For symbolic variables which are used more than once, lifespan and reusage properties are exploited. Our approach is similar to [8] where a probabilistic method is used to exploit lifespan and number of reuse to allocate registers for register file architectures. For DSPs with irregular datapaths, however, we also have to consider each instruction where a symbolic variable is used. A usage in instructions which do not have operand registers in common causes a natural split of lifespan. Irrespective of free registers during the original lifespan, this symbolic variable has to be allocated to two different register sets (`y12` in instructions 11 and 12 will be allocated to register sets `MX` and `AY`, respectively; conf. Fig 3).

If a symbolic variable is used in instructions which share an operand register set but the variable is not specified as the same operand type (X/Y-operand) in each instruction, swapping operands in some of these instructions may enable the variable to reside in a register. However, swapping operands can be prohibited by resource constraints, e.g., result registers can only serve as X-operands whereas feedback registers can only serve as Y-operands. Thus, for symbolic variables which are used together with result or feedback registers, operand types cannot be changed (Fig. 2). This may result in additional load instructions.

```
AR  = x1 + x2        AR = x1 + AF
    :                    :
AR  = x3 + x1        AR = AR + x1

     (a)                  (b)
```

Figure 2: Reused operands; (a) operands x1 and x3 can be swapped, (b) swapping of operands is prohibited

Considering these constraints, symbolic variables are allocated to that register set/memory pair which offers the highest degree of potential instruction-level parallelism. This strategy is pursued also for variables with a single appearance in an operation with only one symbolic variable unassigned.

[1]Only certain register set/memory bank combinations are allowed for parallel execution.

2.2. Final Assignment

During register set allocation, a set of equivalent registers which meet the constraints imposed by the corresponding ALU/MAC operation is selected for each symbolic variable. The decision whether variables can reside in one of these registers is made during final assignment. Since these register sets are homogeneous subsets of a non-homogeneous register set, the previously mentioned techniques for optimization for architectures with regular register files can be applied. Due to the limited number of equivalent registers (for the ADSP-21xx, each register set contains just two registers), even optimal algorithms can be performed in a reasonable amount of time. Variables which cannot reside in registers are spilled to memory. The additionally inserted memory accesses are scheduled during code compaction. Fig. 3 shows a segment of intermediate straight-line code after register assignment and spill code insertion.

```
    AX0 = DM(y27.1)
    AY0 = PM(y25)
 1: AR  = AX0 + AY0
    PM(y28) = AR
    MY0 = PM(c4)
 2: MR  = AR * MY0
    DM(y23) = MR
 3: y22 = MR + AY0
         :
    MX1 = PM(y12)
    MY1 = PM(c2)
11: MR  = MX1 * MY1
    AX0 = DM(y23)
    AY1 = PM(y12)
12: AR   = AX0 + AY1
         :
    AY1 = PM(y28)
28: AR  = AR + AY1
29: DM(y29) = AR
    AX1 = DM(y19)
    AY0 = PM(y14)
30: AR  = AX1 + AY0
31: DM(y21) = AR
    AY0 = PM(y15)
32: AR  = AR + AY0
         :
```

Figure 3: Intermediate straight-line code for a 2^{nd} order WDF filter and the ADSP-21xx family; registers assigned and spill code inserted.

3. CODE COMPACTION

DSPs are similar to horizontally microcoded machines in that multiple functional and data addressing units can be controlled in a long instruction word. This allows microcode compaction techniques to be applied to straight-line intermediate code to exploit instruction-level parallelism of the target machine. We consider branch-free blocks of code, i.e. blocks of code with a single entry and a single exit point. Therefore, local compaction techniques can be applied to exploit the instruction-level parallelism of the target architecture. Despite local compaction is an exponentially complex problem, optimal or near-optimal results can be found in a reasonable amount of time [9].

Our compaction algorithm is based on the critical path algorithm [10]. Compaction starts with a data dependency analysis which is based on an examination of the data interactions between input and output resources for each operation in the straight-line code. Direct data interactions between operations o_i and o_j occur in one of the following cases (We assume that o_i precedes o_j in the straight-line code.):

- An output resource of o_i is also an input resource of o_j. If the order of the operations is reversed, the input resource of o_j does contain an outdated value.
- An input resource of o_i is also an output resource of o_j. In this case, a reversed order causes an update of the input resource of o_i before it has been used.

These direct data interactions restrict the potential reordering performed to exploit the compaction capability of the target system. Shifting an operand beyond the limit caused by such a restriction causes a data conflict. Resource conflict analysis on the other hand determines which instructions can be executed concurrently without conflicting over a hardware resource. These conflicts are caused by restrictions imposed by the target architecture. Therefore, a compaction algorithm must operate within the framework of a machine model. To specify all concurrently executable instructions and corresponding resource constraints, we use a target architecture description file as proposed in [11].

The results of data dependency analysis and resource conflict analysis provide the information required to compact operations into multifunction instructions. Therefore, an algorithm for forming complete multifunction instructions examines a set of operations of the straight-line code and constructs conflict-free operation bundles. An operation bundle is a set of operations which are desired to be executed together in one instruction cycle. An operation bundle is considered as a complete multifunction instruction with respect to a set of operations if no other members of the set can be added to the bundle without violating data dependency or resource constraints. Figure 4 shows a segment of intermediate code after register and memory bank assignment, spill code insertion, and code compaction.

4. RESULTS

As a test set, we generated assembly code for Analog Devices' ADSP-21xx for four different digital filter structures. Table 1 summarizes the number of ALU and MAC operations for each filter, the size of hand-coded and thoroughly optimized code, and the size of automatically generated code. In [12], we examined the impact of register assignment on code compaction if both processes are performed independently. That approach resulted in large code size variations for different register assignments (up to 62%)

```
AX0 = DM(y27.1), AY0 = PM(y25);
AR  = AX0 + AY0, AX1 = DM(y23),  MY0 = PM(c4);
PM(y28) = AR;
MR  = AR * MY0,  MX0 = DM(c1),   MY1 = PM(y2);
DM(y23) = MR,    AR = AX1 + AY0;
        :
AR  = AX1 + AY1, AX1 = DM(y19);
AY0 = PM(y14);
AR  = AX1 + AY0, AX1 = DM(y21),  AY0 = PM(y15);
DM(y21) = AR;
AR  = AX1 + AY0, AX1 = PM(y17);
AY0 = DM(y9);
        :
```

Figure 4: Segment of intermediate code for ADSP-21xx after register assignment, spill code insertion, and code compaction.

and the necessity to consider all assignments for code compaction to achieve optimal final code if backtracking is not applied.

Filter	a	b	c	d
2^{nd} order norm. ladder	13	≤ 16	17	17-25
2^{nd} order lattice	11	≤ 16	18	18-19
2^{nd} order state space	9	≤ 13	13	13-21
4^{th} order WDF	20	≤ 32	34	34-38

a ... Number of ALU/MAC operations
b ... Program size of hand-coded code
c ... Program size of automatically generated code
d ... Code size variation discovered in [12]

Table 1: Compilation results for Analog Devices' ADSP-21xx.

By coupling register assignment and code compaction as proposed in this paper, we circumvent the problem of generating all assignments possible; neither do we have to apply backtracking. Nevertheless, code generation based on our new method results in final code of the same quality as by exploiting all possible assignments without increasing computational complexity. Additionally, we compared our results with hand coded and thoroughly optimized programs to see if near-optimal solutions can be achieved. Although our register assignment and compaction algorithms are based on heuristics to solve these NP-hard problems, assembly code of high quality is achieved.

5. SUMMARY

In this paper, we presented a method for register assignment and code compaction for DSPs with irregular datapaths. Register assignment, which is made sensitive to its impact on code compaction, is performed in two steps. For the first step, we utilize the concept of register sets to group equivalent registers. During register set allocation, that register set is chosen which meets the resource constraints imposed by the corresponding ALU/MAC operation and which maximally preserves the freedom available for code compaction. Together with allocation of register sets, memory locations for variables are selected to maximize potential instruction-level parallelism. In the sequel, symbolic variables are assigned to registers contained in the previously allocated register sets. For variables which cannot reside in registers due to the limited number of registers in each set, spill code is inserted. Thus, scheduling of the spill code can be performed efficiently during code compaction. For code compaction, we utilize the critical path algorithm which provides optimal or near-optimal results in a reasonable amount of time. As our results show, the size of the automatically generated programs is about the same size as for hand-coded and thoroughly optimized programs.

6. REFERENCES

[1] V. Zivojnovic, S. Ritz, and H. Meyr, "Retiming of DSP programs for optimum vectorization", in *Proceedings of the IEEE ICASSP'94*, 1994, vol. 2, pp. 465–468.

[2] R. Sethi, "Complete register allocation problems", *SIAM J. Computing*, vol. 4, no. 3, pp. 226–248, September 1975.

[3] G. J. Chaitin, "Register allocation & spilling via graph coloring", *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 98–105, 1982.

[4] F. C. Chow and J. L. Hennessy, "The priority-based coloring approach to register allocation", *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 4, pp. 501–536, January 1990.

[5] Preston Briggs, Keith D. Cooper, and Linda Torczon, "Improvements to graph coloring register allocation", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 428–455, May 1994.

[6] A. Sudarsanam and S. Malik, "Memory bank and register allocation in software synthesis for ASIPs", in *Proceedings of the ICCAD'95*, San Jose, November 1995.

[7] C. Liem, T. May, and P. Paulin, "Register assignment through resource classification for ASIP microcode generation", in *Proccedings of ICCAD'94*, San Jose, CA, Nov. 6-10 1994.

[8] T. A. Proebsting, *Code generation techniques*, PhD thesis, University of Wisconsin - Madison, 1992.

[9] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett, "Local microcode compaction techniques", *ACM Computing Surveys*, vol. 12, no. 3, pp. 261–294, September 1980.

[10] C. V. Ramamoorthy and M. Tsuchiya, "A high-level language for horizontal microprogramming", *IEEE Transactions on Computers*, vol. C-23, no. 8, pp. 791–801, August 1974.

[11] W. Kreuzer, M. Gotschlich, and B. Wess, "A retargetable optimizing code generator for digital signal processors", in *Proceedings of the IEEE ISCAS'96*, Atlanta, May 1996, vol. 2, pp. 257–260.

[12] W. Kreuzer and B. Wess, "Optimized code compaction for digital signal processors", in *Proceedings of the ICSPAT'95*, Boston, October 1995, vol. 2, pp. 1753–1757.