Minimization of Spill Code Insertion by Register Constraint Analysis for Code Generation for Application Specific DSPs

Tatsuo WATANABE Nagisa ISHIURA

Department of Information Systems Engineering,
Faculty of Engineering, Osaka University,
Yamada-Oka, Suita, Osaka, 565-0871 Japan
Tel.: +81(6)6879-7808
FAX: +81(6)6875-5902
E-mail: {watanabe, ishiura}@ise.eng.osaka-u.ac.jp

Abstract

Application specific DSPs often employ irregular datapath structures with distributed registers. In the scheduling phase of retargetable compilation, resolution of register usage conflicts comes to be a new constraint for such datapaths. This paper presents a method of register constraint analysis which attempts to minimize the number of the spill codes required for resolving the register usage conflicts. It searches for a set of ordering restrictions among operations which sequentialize the lifetimes of the values residing in the same register as much as possible and thus minimize the number of the register conflict. Experimental results show that a combination of the proposed register constraint analysis and list-based scheduling reduces the number of the register spills into 25%.

1 Introduction

Application specific embedded processors are increasingly used recently, for they provide an advantageous trade-off between flexibility and cost. For the purpose of expediting the development of hardware and software for embedded systems, a lot of researches have been conducted on processor synthesis, compiler development, and hardware/software codesign [Aka93, Mar97]. Especially, a retargetable compiler is a useful tool in the design of systems using application specific instruction processors, for it enables (1) efficient software development using programming language, (2) software development in concurrence with hardware design, and (3) easier software transportation from an architecture to newly designed or modified ones.

The application specific DSPs often have irregular datapath structures which contain multiple functional units which can operate in parallel. They also employ distributed registers of small capacity, instead of a central register file, aiming at increased band width among the functional units. In such datapaths, many of the compilation phases such as instruction selection, binding, and scheduling comes to involve difficult combinational problems. Particularly in the scheduling phase, reconciliation of usage conflicts of the distributed registers as well as the functional units is a new challenge. [Mes98] solves the scheduling problem by introducing register constraint analysis before scheduling. It generates a set of ordering restrictions among the operations to sequentialize the usage of the same registers, which is passed to the scheduler. This works well as long as a feasible scheduling without register conflicts exists, but there is no guarantee for such a scheduling for a given dataflow graph (DFG) and a binding. In that case, [Mes98] resorts to recomputation of the binding, which do not always succeed and may be computationally unaffordable for complex datapaths and large DFGs.

On the other hand, the register conflicts can be resolved by inserting spill codes. If the number of the additional register saves/restores to avoid the register conflicts is small, this alternative is attractive since binding and scheduling are determined without repetitive computation. Thus, this paper proposes a register analysis method that tries to find a set of ordering restrictions that minimizes the number of the spill codes necessary for resolving the register conflicts.

This paper is organized as follows. Section 2 describes a flow of retargetable compilation and the basic idea of the register analysis. Section 3 explains how the register conflict is resolved by spill code insertion. The details of our method are described in Section 4 and experimental results are shown in Section 5. Section 6 summarizes the contribution with concluding remarks.

2 Retargetable Compilation

2.1 Flow of Compilation and Scheduling Problem

We assume that target processors may have multiple functional units which can be activated simultaneously by VLIW type instruction formats.

The general flow of retargetable compilation for such processors consists of (1) conversion of a given source program into an intermediate data structure, (2) selection and binding of functional units and registers to the operations and the variables, (3) scheduling of the operation, and (4) code generation. A given program is partitioned into basic blocks each of which is represented in the form of a DFG (dataflow graph), etc. In the selection and the binding phases, the hardware resources to execute each operation in the DFGs are determined. Operations may be merged or split if necessary. If the datapath has a multiply-and-accumulate (MAC) unit, for example, successive multiplication and add operations may be combined into a single operation which are executed by the MAC unit. If there are multiple adders, each add operation in the DFGs is assigned to one of the adders. A register is assigned to each intermediate value. Necessary operations to transfer data between registers are inserted. In the scheduling phase, the nodes of the DFG are allocated to control steps so that there are no conflicts of hardware resource usage within the same control step and yet the number of the total control steps is minimized.

Although a feasible scheduling without conflicts of the functional units is easily computed by the list-based scheduling method, register conflicts are difficult to deal with in the distributed register architec-

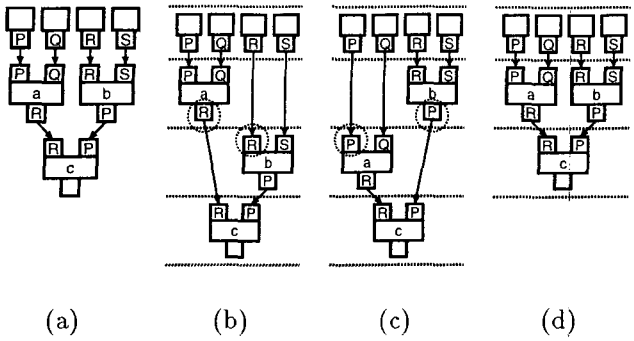

Fig. 1: Register Conflict.

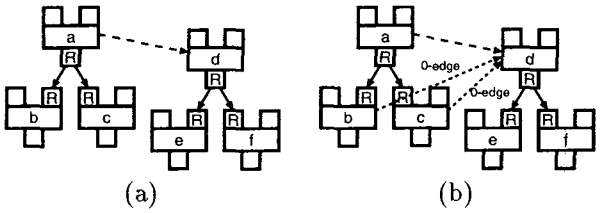

Fig. 2: 0-edges to sequentializing register usage.

ture. Let us see an example of scheduling of the DFG shown in Fig. 1 (a). If operation node a is scheduled into an earlier step than node b (Fig. 1 (b)), the result of a is written into register R before b reads the value in R. In converse, if b is scheduled earlier (Fig. 1 (c)), register P is overwritten before b reads it. The only solution to avoid the conflict is to schedule a and b into the same cycle (Fig. 1 (d)). This type of scheduling is difficult to find in the conventional list-based algorithm in which the nodes in the DFG are assigned to time slots from top to bottom.

2.2 Register Constraint Analysis

An efficient way of solving this problem is to introduce register constraint analysis before list-based scheduling [Mes98]. Necessary conditions for sequentializing the lifetimes of the different values stored in the same registers is analyzed and it is represented in the form of a set of new edges which force ordering of the nodes during scheduling. By handling the newly introduced edges in the same way as the existing data dependency edges, list scheduling can find a feasible scheduling without register conflicts. Fig. 2 shows the basic idea of the register constraint analysis. Two operations, a and d in (a), writes register R and two succeeding operations for each are reading register R. Suppose there is a restriction such that a should be executed earlier than d, which comes from a combination of existing data dependency relations. In this case, execution of d must wait for the completion of b and c, otherwise d will destroy the content of R before b and c read it. Thus the register constraint is reduced into predecessor-successor relations between

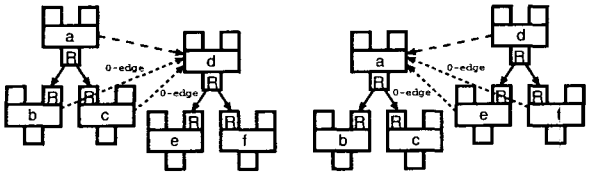

Fig. 3: which may be done in advance

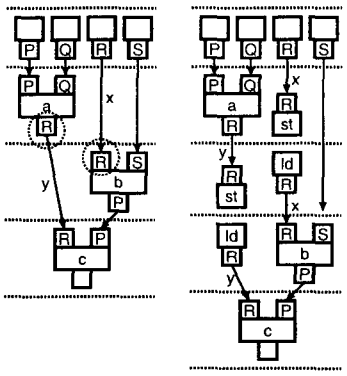

Fig. 4: scheduling with spill code insertion.

nodes as shown in Fig. 2 (b). A difference between data dependency edge and the newly introduced edge is that the latter allows the simultaneous execution of the two operations. Namely, in this example, d can be scheduled into a cycle same as or later than b and c. We refer to this type of edge as a *0-edge*. In the example of Fig. 1, the first phase analysis adds two 0-edges $a \rightarrow b$ and $b \rightarrow a$, which forces a and b to be scheduled into the same cycle.

Register constraint analysis is formulated as a problem of finding a set of 0-edges that sequentializes the usage of the same registers. The order between two nodes may be determined uniquely by the existing relations, as was shown in Fig. 2. If this is not the case however, we must examine the both cases. If there is no restriction between a and d in Fig. 2 (a), for example, we must examined the both cases as shown in Fig. 3. The register constraint analysis fails when addition of new 0-edges forms an invalid loop, which requires both node x should be scheduled later than y and y should be scheduled later than x, for example. In this case, a feasible ordering is searched by backtracking.

3 Resolution of Register Conflict by Spill Code Insertion

If there exists a scheduling free of register conflict free scheduling, the combination of the register constraint analysis and list scheduling can always find the scheduling. However, there are cases where no feasible scheduling exists for a given binding. In that case, the analysis algorithm of [Mes98] is designed to return a set of the DFG nodes which cause the conflict, so that the binding can be recomputed. However, the recomputed binding does not always yield a feasible scheduling. Repetitive backtracking may be required and this becomes infeasibly costly for large DFGs.

However, the search for a combination of feasible binding and scheduling is not the only solution. Register conflict can be settled by using spill codes. In Fig. 4, for example, the conflict on register R is resolved by storing value x before operation a writes y into R, storing also y so that x can be reloaded to execute b, and reloading y to execute c.

For irregular datapaths, the spill code insertion is also a difficult problem, for there are case where storing of a register requires storing of other registers, but a technique has been proposed in [Ish00] which enables storing and reloading of a register in irregular

```
P={(o1, o2) | operation o1, o2 write the same register}
R={(o1, o2) | o1 can be scheduled before o2}
Z=φ    /* set of 0-edges */
foreach (o1, o2) ∈ P{
  if ((o1, o2) ∈ R && (o2, o1) ∉R)
    Z=ZUnew_0edge(o1, o2);
  else if ((o1, o2) ∉R && (o2, o1) ∈ R)
    Z=ZUnew_0edge(o1, o2);
  else if ((o1, o2) ∈ R && (o2, o1) ∈ R){
    if(ASAP(o1) <ASAP(o2))
      Z=ZUnew_0edge(o1, o2);
    else
      Z=ZUnew_0edge(o1, o2);
  }
  else {}
}
```

Fig. 5: Algorithm of analysis without backtracking

paths in $2n$ clock cycles (where n is the number of the register in the datapath). Thus, by using spill code insertion, we can always find a feasible scheduling for a given binding, though the insertion of spill codes may increase the total execution cycles.

4 Register Constraint Analysis to Minimize Register Spills

The conventional register analysis method, which searches for a nonconflicting set of 0-edges, outputs no 0-edges when a given binding does not have a feasible scheduling. Although we can generate a feasible scheduling by the spill code insertion even if no 0-edges are given by the register analysis, the resulting code may be inefficient. Thus our goal now is to find, for an infeasible case, a set of 0-edges that leads to a scheduling with as few register spills as possible. This paper proposes two methods. One is a fast method that generates a set of 0-edges without backtracking. The other is an optimizing method that searches for the combination of 0-edges that minimizes the number of necessary spills.

4.1 Register Analysis without Backtracking

In the conventional method, backtracking is invoked when the search encounters a conflicting set of 0-edges. On the other hand, in our first method, search is continued by removing the 0-edges that have caused the inconsistency the most recently. Although there is no guarantee that the number of the spills is minimum, 0-edges that sequentialize the register usage as much as possible are quickly computed. Fig. 5 shows the outline of the algorithm. P is the set of the all pairs of operations (o_1, o_2) where o_1 and o_2 write to the same register. R is the scheduling constraints where the $(o_1, o_2) \in R$ means that o_1 can be scheduled before o_2. The resulting 0-edges are accumulated into set Z. When only one of (o_1, o_2) and (o_2, o_1) is in R, 0-edge is added so that writing of the register by o_1 and o_2 are sequentialized as was shown in Fig. 2. Function "new_0edge(o_1, o_2)" returns the set of 0-edges which are introduced to sequentialize o_1 and o_2, while updating R by removing impossible ordering. If both (o_1, o_2) and (o_2, o_1) are in R, we have two choices as was shown in Fig. 3. We break this tie by the ASAP (As Soon As Possible) values of o_1 and o_2: 0-edges are introduced so that the operation with the smaller ASAP value will be scheduled earlier. If neither (o_1, o_2) or (o_2, o_1) is in R, the search is continued without adding any 0-edges.

```
P={(o1, o2) | operation o1, o2 write the same register}
R={(o1, o2) | o1 can be scheduled before o2}
Z=φ;
nspill=0;
best_Z=φ;
best_nspill=∞;
minspill(nspill, Z, P);

minspill(nspill,Z, P){
  if(P = φ){
    best_nspill = nspill; best_Z = Z;
  }
  else if (nspill < best_nspill){
    get (o1, o2) ∈ P;
    if ((o1, o2) ∈ R && (o2, o1) ∉R)
      minspill(nspill,ZUnew_0edge(o1, o2), P-{(o1, o2)});
    else if ((o1, o2) ∉R && (o2, o1) ∈ R)
      minspill(nspill,ZUnew_0edge(o2, o1), P-{(o1, o2)});
    else if ((o1, o2) ∈ R && (o2, o1) ∈ R){
      minspill(nspill,ZUnew_0edge(o1, o2), P-{(o1, o2)});
      minspill(nspill,ZUnew_0edge(o2, o1), P-{(o1, o2)});
    }
    else
      minspill(nspill+1,Z, P);
  }
}
```

Fig. 6: Algorithm of analysis minimize register spills

4.2 Minimization of Register Spills

The second analysis method searches for a set of 0-edges that minimizes the conflicts of the register usage. The first feasible solution is computed by the same way as in the first method, but the number of the failures (conflicts) with this solution is recorded. Then search is continued, looking for other solutions with the smaller failure count by backtracking.

Fig. 6 shows this algorithm. "nspill" is the current number of failures, and "best_nspill" is the number of the failures of the best solution found so far. "best_Z" keep the set of 0-edge of the best solution. If the search reaches the bottom (P=ϕ), the best solution is updated. If both (o_1, o_2) and (o_2, o_1) belong to R, we examine the both cases by recursion. When neither (o_1, o_2) or (o_2, o_1) is in R, we increment "nspill" and continue the search while nspill<best_nspill.

Futhermore, with a view to descreasing the number of the clock cycles for the spill code, the pairs of operations in P is sorted by their cost functions. First, the cost of a register is defined as the number of the clock cycles for spill and reload the value stored in the register. The cost of a pair of operations is equal to the cost of the register to which the both operations are written into. In the search procedure, pairs of larger costs are processed first, so that write conflicts to registers with larger costs are more likely to be resolved.

The computational complexity of this algorithm is inherently exponential.

5 Experimental Result

A register constraint analysis program has been implemented on an Ultra-80 workstation (450 MHz) in C++ language. The target is a DSP which has a datapath dedicated to G723.1 speech codec [Fuj98].

Table 1: Experimental Results.

name	DFG	no analysis			analysis 1				analysis 2			
	#n/#e	#spill	#cs	CPU	#0-e	#spill	#cs	CPU	#0-e	#spill	#cs	CPU
Init_Decod.c	117/101	17	88	0.55	49	0	69	0.88	58	0	69	1.07
Wght_Lpc.c	436/411	109	450	1.85	97	78	439	24.76	247	28	297	20.67
Gen_Trn.c	485/448	96	413	1.74	202	48	337	14.86	235	22	299	9.85
Error_Wght.c	826/781	201	814	3.24	398	82	571	33.29	471	51	560	31.55
Upd_Ring.c	1306/1252	356	1397	5.47	678	130	906	619.85	748	95	873	295.09

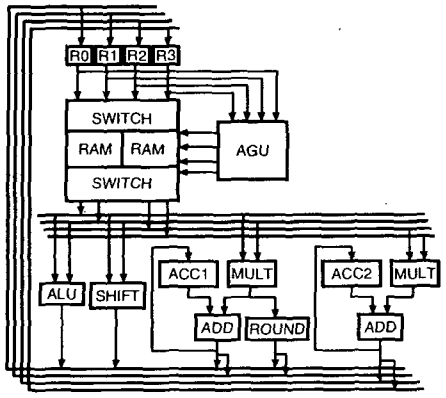

Fig. 7: G723.1 processor datapath.

Experimental results are shown in Table 1. For a binding computed by the method in [Ish00], list-based scheduling and spill code insertion are performed for the following three cases:

1) No register analysis is done ("no analysis").
2) The regiter analysis without backtracking is done before the list-based scheduling ("analysis 1").
3) The register analysis to minimize register spill count is done before list scheduling ("analysis 2").

The column "DFG" is the size of the DFG after binding in terms of the number of the node (#n) and the number of the edges (#e). "#spill" is the number of the required spill codes, "#cs" the total number of the clock cycles (original operation and spill codes). "#0-e" the number of the 0-edges introduced by the analysis. "CPU" in column is the execution time of analysis and scheduling.

Comparison of "no analysis" and "analysis 1" shows that the number of the spills is significantly reduced by the register constraint analysis and the resulting number of the control steps is also reduced accordingly. In "analysis 2", the number of the spills and the clock cycles are further reduced. As compared with "no analysis", the number of the pill is reduced to 25% and the number of the clock cycles to 67%. Since the number of the clock cycles corresponds to the number of the VLIW instructions, this analysis contributes much to the speed efficiency of the generated code. Our program successfully finished the analysis for DFGs consisting of as much as 1306 nodes in a practical amount of CPU time.

6 Conclusion

We have presented two methos of register constraint analysis which contribute to the minimization of the register spills to resolve register conflicts. With the analysis, the number of the register spills and the total number of the clock cycles to execute the program are significantly reduced.

Acknowledgment

The authors would like to thank Prof. Isao Shirakawa of Osaka University for his support and advice on this research. We would like to thank Dr. Masayuki Yamaguchi, Mr. Mizuki Takahashi, Dr. Hiroyuki Okuhata, and Mr. Sinya Hashimoto for their discussion and constructive comments.

References

[Mar97] P. Marwedel: "Compilers for Embedded Processors," in *Proc. SASIMI '97*, pp. 201–208, (Dec. 1997).

[Aka93] H. Akaboshi and H. Yasuura: "COACH: A computer aided design tool for computer architects," in *IEICE Transactions on Fundamentals*, vol. E76-A, pp. 1760–1769 (1993).

[Ish00] N. Ishiura, T. Watanabe and M. Yamaguchi: "A Code Generation Method for Datapath Oriented Application Specific Processor Design," in *Proc. SASIMI 2000*, pp. 71–78 (Apr. 2000).

[Mes98] B. Mesman, M. Strik, A. H. Timmer, J. L. van Meerbergen, and J. A. G. Jess: "A Constraint Driven Approach to Loop Pipelining and Register Binding," in *Proc. IEEE DATE*, pp. 377–383 (Feb. 1998).

[Fuj98] H. Okuhata, Morgan H. Miki, T. Onoye, I. Shirakawa: "A Low-Power DSP Core Architecture for Low Bitrate Speech Codec," in *IEICE Trans. Fundamentals*, vol. E81-C, pp. 1616–1621, (Aug. 1998).