LETTER

Register Constraint Analysis to Minimize Spill Code for Application Specific DSPs

Tatsuo WATANABE[†], *Student Member and* **Nagisa ISHIURA**[†], *Member*

SUMMARY This letter presents a method which attempts to minimize the number of spill codes to resolve usage conflicts of distributed registers in application specific DSPs. 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. Experimental results show that the proposed analysis method reduces the number of register spills into 28%.
key words: application specific DSP, spill code insertion, embedded systems, scheduling, register constraint analysis

1. Introduction

Application specific embedded processors are recently increasingly used in digital systems, for they provide an advantageous trade-off between flexibility and cost. Such processors often have irregular datapath structures containing distributed registers of small capacity, instead of a central register file, aiming at increased band width among functional units (Fig. 1). For those datapaths, many of the compilation tasks such as binding and scheduling become computationally difficult. Especially, in scheduling, reconciliation of usage conflicts of the distributed registers as well as the functional units becomes a new challenge.

Mesman et al. [Mes98] proposed an efficient way of solving the scheduling problem by introducing a register constraint analysis before scheduling. However, there is not always a feasible scheduling without register conflict for a given DFG and binding. In that case, it resort to recomputation of the binding, which requires large computation time. On the other hand, the register conflicts can be resolved by inserting spill codes. This paper proposes a new register analysis method which attempts to minimize the number of the register spills necessary to avoid the register conflicts.

2. Scheduling and Register Constraint Analysis

The scheduling problem is, given a DFG after binding (Fig. 2 (a)), to assign each nodes (operation) in the DFG to a control step so that the total number of the control steps is as small as possible (Fig. 2 (d)). For each operation node, the functional unit and the read/write registers are determined in the binding phase. In Fig. 2 (a), node a reads the data from register P, computes with `ALU1`, Q, and writes the result into register R. The same functional unit must not be used more than once in the same control step (a resource constraint), and a register, whose capacity is one, must not be overwritten until its value is not referenced any more.

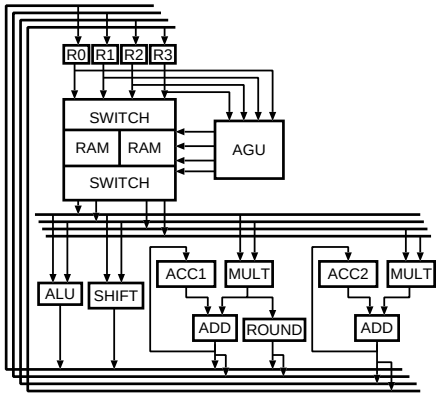

Fig. 1: G723.1 processor datapath.

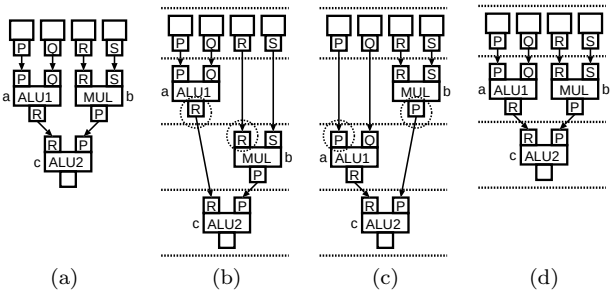

Fig. 2: Scheduling considering register conflicts.

The register constraints are difficult to deal with in the distributed register architecture. In the scheduling of Fig. 2 (a), neither (b) nor (c) satisfies the register constraints and the only solution is to schedule a and b into the same cycle ((d)). This type of scheduling is difficult to find by the conventional list-based algorithm.

The method of [Mes98] analyzes the given DFG before list-based scheduling and converts the register constraints into an equivalent set of new ordering edges. In Fig. 3, suppose there are dependency edges from a

Manuscript received Oct. 4, 2000.
Manuscript received Dec. 22, 2000.
[†]The author is with the Graduate School of Engineering, Osaka University.

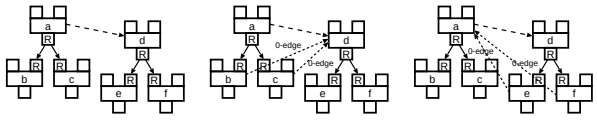

Fig. 3: 0-edges to sequentialize register usage.

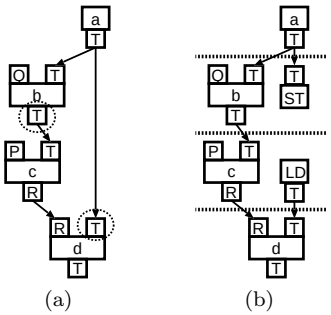

Fig. 4: Scheduling with spill code.

to d. 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 conflict is reduced into predecessor-successor relations between nodes as shown in (b). We refer to this type of edge as a *0-edge*, which is different from data dependency edge in that the 0-edge allows the simultaneous execution of the two operations it connects.

The register constraint analysis is formulated as a problem of finding a set of 0-edges that sequentialize the usage of the same registers. This becomes a search problem, because we must basically examine, for every pair of nodes writing to the same register, the two possibilities of sequentializing the usage of the register, unless some sort of ordering is forced (as in Fig. 3). Search fails if both of the two possible orderings of some node pair are infeasible. The infeasibility of the ordering is detected by finding a loop: In Fig. 3, ordering (b) is feasible, but (c) is not because (c) contains a loop [†] .

3. Minimization of Register Spills

If the register analysis fails, [Mes98] resorted to the recomputation of the binding, which was computationally expensive and yet did not always yield a feasible scheduling. On the other hand, register conflicts can be settled by using spill codes. For example, no scheduling can sequentialize the usage of register T in Fig. 4 (a), but insertion of `ST` and `LD` operation resolve the conflict, as shown in Fig. 4 (b).

By the spill code insertion, we could generate a feasible scheduling even if the analysis fails, but the resulting code may be inefficient. Thus, we propose an analysis method that attempts to find a set of 0-edges that leads to a scheduling with as few register spills as possible. It basically attempts to find a feasible solution. When the search encounters an invalid situation, however, it does not backtrack at that point. Instead, a failure count, which approximates the number of the necessary register spills, is increased by one and the search is continued. When the search reaches the bottom, the failure count and the set of 0-edges for that configuration is recorded. Then search is continued, looking for other solutions with the smaller failure count by backtracking.

[†]A loop consisting only of 0-edges are not invalid, if all the nodes associated with the loop can be scheduled into the same control step.

```
P={(o1, o2) | operation o1 and o2 write the same register}
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){
    if ((o1, o2) ∈ P s.t. R(Z, o1, o2) && !R(Z, o2, o1))
      /* case 1 */
      minspill(nspill,Z∪new_0edge(o1, o2), P − {(o1, o2)});
    else if ((o1, o2) ∈ P s.t. !R(Z, o1, o2) && R(Z, o2, o1))
      /* case 2 */
      minspill(nspill,Z∪new_0edge(o2, o1), P − {(o1, o2)});
    else if ((o1, o2) ∈ P s.t. R(Z, o1, o2) && R(Z, o2, o1)){
      /* case 3 */
      minspill(nspill,Z∪new_0edge(o1, o2), P − {(o1, o2)});
      minspill(nspill,Z∪new_0edge(o2, o1), P − {(o1, o2)});
    }
    else minspill(nspill+1,Z, P − {(o1, o2)}); /* case 4 */

  }
}

new_0edge(o1, o2){
return the set of 0-edges sequentializing register usages of o1
and o2
}
R(Z, o1, o2){
return 1 if Z∪new_0edge(o1, o2) doesn't form invalid loops
}
```

Fig. 5: The algorithm of proposed method

Fig. 5 describes the algorithm. P is the set of all the pairs of operations (o_1, o_2) where o_1 and o_2 write to the same register. The resulting 0-edges are accumulated into set Z. "nspill" is the failure count. "best_nspill" and "best_Z" memorize the "nspill" and "Z", respectively of the best solution found so far. Recursive procedure "minspill" searches for the solution. If the search reaches the bottom (P=ϕ), the best solution is updated. $R(Z, o_1, o_2)$ tests whether the ordering of o_1 before o_2 is possible. When only one of $R(Z, o_1, o_2)$ and $R(Z, o_2, o_1)$ is 1 ("case 1" and "case 2"), new_0edge(o_1, o_2) or new_0edge(o_2, o_1) is invoked to generate the 0-edges that sequentialize o_1 and o_2 in this order. If the both orderings of o_1 and o_2 are possible ("case 3"), we examine the two cases by recursion. When neither $R(Z, o_1, o_2)$ nor $R(Z, o_2, o_1)$ holds, we increment "nspill" and continue the search

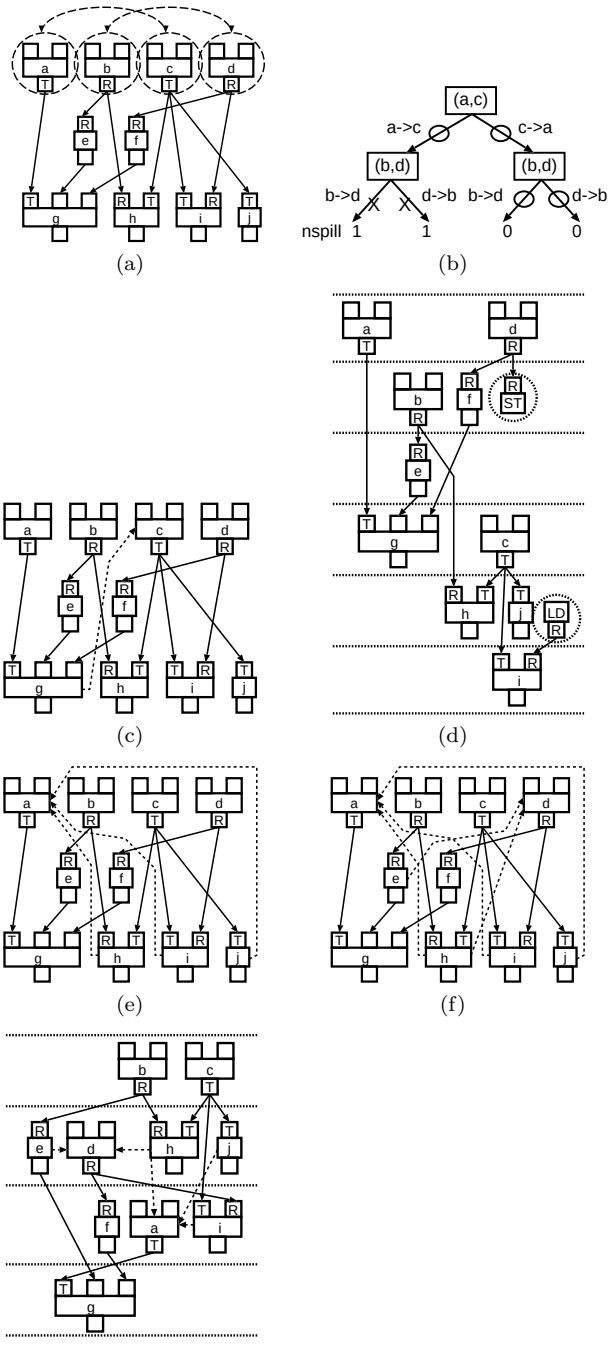

Fig. 6: Example of the algorithm.

while nspill<best_nspill. The computational complexity of this algorithm is inherently exponential.

Fig. 6 shows an example. Fig. 6 (a) is a given DFG where P=$\{(a,c),(b,d)\}$. We must determine the ordering of (a,c) and (b,d). Since all of $R(Z,a,c)$, $R(Z,c,a)$, $R(Z,b,d)$, and $R(Z,d,b)$ are 1, "minspill" will examine the "case 3". Fig. 6 (b) illustrates the search tree. Firstly, the case where a is executed before c ("a–> c" in Fig. 6 (b)) is examined and new_0edge(a,c)=$\{(g,c)\}$ is added to the DFG (Fig. 6 (c)). Then, "minspill" is called recursively with Z=$\{(g,c)\}$ and P=$\{(b,d)\}$ to determine the ordering of b and d. Now that both $R(Z,b,d)$ and $R(Z,d,b)$ have become 0 by the addition of 0-edge (g,c), neither branch "b–> d" nor "d–> b" is possible. If we try to force "b–> d" ("d–> b"), 0-edge (h,d) (0-edge (i,b)) will create a loop, respectively. Since this implies that the usage of register R by b and d cannot be sequentialized and we must resort to spill code insertion, we increment the failure count "nspill". The search reaches the bottom and the set of 0-edges in Fig. 6 (c) becomes the first feasible solution with nspill=1. Fig. 6 (d) is the result of a list-based scheduling for this solution, where a pair of store (ST) and load (LD) is inserted to resolve the usage conflict of register R between b and d. The search is continued for better solutions on the case of "c–> a" by backtracking. In this case, new_0edge(c,a)=$\{(h,a),(i,a),(j,a)\}$ are inserted into the DFG (Fig. 6 (e)). This time, both of $R(Z,b,d)$ and $R(Z,d,b)$ remain 1. If we select "b–>d", new_0edge(b,d)=$\{(e,d),(h,d)\}$ is added (Fig. 6 (f)). Since nspill=0, we can schedule the DFG without spill codes (Fig. 6 (g)).

We assume that the number of clock cycles to spill and to reload the register from/to RAMs may be different from register to register. In the datapath of Fig. 1, for example, storing of data for R0–R3 and reloading of all the registers takes one cycle but two cycle for storing of ACC1 and ACC2. With a view to decreasing the total number of the clock cycles for the spill codes, we sort the pairs of operations in P by the number of the clock cycles to spill and reload the register. In the search procedure, pairs of larger costs are processed first, so that write conflicts on registers with larger costs are resolved in earlier stage of the search and the spills may happen on registers with smaller costs.

4. 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 dedicated to G723.1 speech codec [Oku98]. Source programs are decomposed into basic blocks (sequences of operations without branches), each of which is converted into a DFG. The binding is computed by the method in [Ish00], and list-based scheduling and spill code insertion are performed after the register analysis.

Table 1 shows the result for various sizes of basic blocks. The columns "DFG #n" is the size of the DFG after binding in terms of the number of nodes. The columns "without analysis" shows the compilation results without any register analysis, where "#spill/#cs" are the number of the spill codes and the total number of the clock cycles (original operation and spill codes), and "CPU" is the computation time for analysis and scheduling. The columns "with analysis" shows the

Table 1: Experimental results I.

BB	DFG #n	without analysis		with analysis			with analysis (+register ordering)		
		#spill/#cs	CPU (sec)	#spill/#cs	(#0-e)	CPU (sec)	#spill/#cs	(#0-e)	CPU (sec)
Par2Ser_BB	29	5/24	0.15	2/16	(8)	0.16	2/16	(10)	0.16
Init_Vad_BB	32	7/28	0.18	2/21	(13)	0.22	2/19	(13)	0.20
Init_Decod_BB	41	12/42	0.15	4/31	(14)	0.23	2/25	(16)	0.33
Ser2Par_BB	53	11/45	0.28	9/45	(23)	0.98	5/33	(25)	0.75
Dec_Sidgain_BB	88	24/97	0.52	12/71	(47)	1.82	5/54	(51)	1.22
Comp_En_BB	91	26/98	0.74	10/61	(37)	1.84	6/55	(45)	1.33
Rem_Dc_BB	104	31/127	1.09	16/90	(52)	3.76	13/78	(52)	2.19
Filt_Pw_BB	208	71/293	1.58	25/237	(141)	33.08	21/213	(141)	8.83

Table 2: Experimental results II.

function	size		previous method			proposed method	
	#BB	#n	#BB_ok	#spill/#cs	CPU (sec)	#spill/#cs	CPU (sec)
Wght_Lpc.c	9	423	2	97/386	2.98	29/270	3.70
Gen_Trn.c	15	483	9	68/371	3.32	22/297	3.74
Error_Wght.c	18	825	10	141/726	5.94	46/597	7.66
Upd_Ring.c	21	1229	6	295/1196	9.14	84/882	17.16

compilation results with the register analysis without register ordering and the columns "with analysis (+register ordering)" is that considering register ordering. "#0-e" indicates the number of the 0-edges introduced by the analysis. The number of the spills is significantly reduced by our register constraint analysis and the resulting number of the control steps is also reduced. With register ordering, the number of the spills and the clock cycles are further reduced.

Table 2 shows a simple comparison with the previous method. This time, functions each of which consists of multiple basic blocks are compiled. "#BB" and "#n" are the number of the basic blocks and the number of the nodes in the function. "#BB_ok" shows the number of the basic blocks for which previous method successfully found the solution. If the analysis failed for a basic block, register conflicts are resolved by spill code insertion. The number of the spills and the number of the control steps are significantly reduced by our method. Our program successfully finished the analysis for DFGs consisting of as much as 1229 nodes in a practical amount of CPU time.

5. Conclusion

We have presented a register constraint analysis method which attempts to minimize 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

[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).

[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).

[Oku98] 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).