Optimal Register Assignment to Loops for Embedded Code Generation[*]

David J. Kolson Alexandru Nicolau Nikil Dutt
Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425

Ken Kennedy
Dept. of Computer Science
Rice University
Houston, TX 77251

Abstract

One of the challenging tasks in code generation for embedded systems is register assignment. When more live variables than registers exist, some variables are necessarily accessed from data memory. Because loops are typically executed many times and are often time-critical, good register assignment in loops is exceedingly important since accessing data memory can degrade performance. The issue of finding an optimal register assignment to loops, one which minimizes the number of spills between registers and memory, has been open for some time. In this paper we address this issue and present an optimal, but exponential, algorithm which assigns registers to loop bodies such that the resulting spill code is minimal. We also show that a heuristic modification performs as well as the exponential approach on typical loops from scientific code.

1 Introduction

Currently, much research has focused on code generation for embedded systems [13, 14, 15]. One of the challenging tasks in generating code for an embedded processor is that of register assignment. In this assignment process, program values are mapped to the processor's registers so that values are available and in the appropriate registers for computation. When the number of simultaneously live variables is larger than the number of registers available, some of these values will have to reside in the data memory (i.e., "spilled" to memory), requiring data transfers between memory and registers when those values are updated or necessary for computation.

Typically, embedded processors or processor "cores" have a small number of registers. Any mapping of variables to registers which contains poor choices for spill code will adversely affect performance. Thus, quality register assignment is exceedingly critical, especially for innermost loops which are executed many times and often time-critical.

In the compiler domain, optimal register assignment solutions have been extensively studied [7, 8, 9]. Although these approaches are effective for straight-line code, they do not address the issue of an optimal assignment of registers to loops—innermost loops probably being the only place such extreme methods are practical. Thus, adaptation and extension of this work to the problem of assigning an embedded processor's registers to program values requires that we overcome the fundamental difficulty that these previous techniques did not address satisfactorily—that of matching the register usage at the entry and exit of loop iterations. That is, for loop code to be correct, the mapping of variables to registers at the beginning of an iteration and at the end of that iteration must be equivalent (i.e., the "right" values must be in the "right" places) in order to correctly iterate over that loop code.

In this paper we demonstrate that the algorithms for register assignment to basic blocks given in [8, 9] can be extended to assign registers to loops by incorporating loop unrolling techniques into the algorithm. Thus, a distinguishing characteristic of our approach is that the register assignment produced may span multiple iterations of the original loop. We also present a heuristic derived from our algorithm that, in practice, seems to perform as well as its exponential counterpart.

2 Related Work

In the compiler domain, the predominant approach to register assignment is the heuristic graph coloring approach [1, 3]. Heuristics for selection of spill candidates have received attention [2] along with coloring methods [4]. Also, [6] addresses loops but without regard to the number of register transfers potentially

[*]This work supported in part by ONR grant N000149311348.

required by their technique at iteration boundaries to match register usages. In contrast, optimal assignments have been studied for particularly critical code segments, such as the innermost loops of time-sensitive applications. Horwitz *et al.* present a method in [7] for obtaining an optimal register assignment to *index* registers which minimizes the number of loads and stores in a basic block. Further work extends this algorithm to deal with simple loops [9], but in doing so, loses optimality. More recent research [8] extends the basic block algorithm to include the assignment of *general purpose* registers to straight-line code.

In High-Level Synthesis the problem of register assignment traditionally refers to determining the number of registers necessary to save values between time-steps [12]. To reduce the interconnect and multiplexor cost of scattered registers, some researchers have focused on grouping registers into memory modules [5, 10]. Other research has addressed the assignment of registers to loop variables [16, 17] by splitting each cyclic variable into two variables. Register-to-register transfers are inserted at loop end when these "coupled" variables are not assigned to the same register. These traditional techniques were developed for *register allocation*, and thus, do not consider the storage of variables within various levels of a memory hierarchy.

Work in code generation for embedded systems has extended the left-edge algorithm and incorporated register classes for register assignment [14] or formulated the problem of register assignment as an ILP formulation [18]. However, these techniques introduce register-to-register transfers at loop boundaries to match register usage in subsequent iterations. Also, [13] uses a complex searching scheme to navigate a large search space with many trade-offs, one of which is register assignment.

None of this previous work has addressed the issue of finding an *optimal* assignment of registers to loops (i.e., an assignment of variables to registers which requires no register-to-register transfers to match usages at loop top and bottom *and* minimizes the cost due to added spill code).

3 Problem Description

In our approach the task of register assignment follows that of code selection and scheduling of operations. We define the problem of register assignment as determining a specific mapping of variables to an embedded processor's registers. When resource shortages occur, spill code (explicit data transfer operations) between the registers and data memory are necessary.

```
Function OPT-Assign (REGS : Initial mapping;
                     VA : Variable access stream)
Begin
   Set curr_states set to REGS
   Foreach variable access V in VA do
      Foreach mapping N in curr_states set do
         If V ∈ N then
            Copy N to new_states set
         Otherwise
            Forall registers R do
               N' = copy_state(N)
               Replace variable, V', currently in R with V
               Cost(N') = Load-Cost(V) + Store-Cost(V')
                          + Cost(N)
               Add N' to children of N
               Add N' to new_states set
            Enddo
         Endif
      Enddo
      Set curr_states set to new_states set
   Enddo
   Return new_states set
End OPT-Assign
```

Figure 1: A register assignment algorithm.

Our goal is to minimize the number of these spill operations repeatedly executed within a loop.

Using a variant of the algorithm presented in [8], we can derive an optimal (i.e., spill minimizing) algorithm that assigns variables to registers for a straight-line code stream. This algorithm is found in Fig. 1. Given a scheduled flowgraph, the *variable access stream* is derived and corresponds to the accessing of variables in execution—a read is denoted by the variable itself while a write is denoted by the variable concatenated with a '*'. For example, for the operation $\mathbf{a} = \mathbf{b} + \mathbf{c}$ the variable access stream is **bca*** (reads before writes). A *variable mapping* or *configuration* corresponds to a particular assignment of variables to registers at a particular point in the code. For example, if there are two registers and the variable **a*** occupies register **R0**, the configuration is $\{\mathbf{a*}, \phi\}$.

OPT-Assign takes as input the variable access stream for a code segment and the register mapping immediately preceding that segment. Then an assignment tree is built where the root is the given (initial) configuration, and each path from the root to a leaf represents a (unique) mapping of variables to registers.

The algorithm proceeds by a breadth-first expansion of the assignment tree, examining each mapping in the current level to determine if it contains the variable(s) accessed by the code in the current step. After the last variable accesses are considered, the leaves are examined to find the lowest cost node. Tracing the path from the root to this node gives an assignment of registers to variables that results in the minimal cost of generated spill code as it has exhaustively generated every possible assignment. Heuristics can be (and have been) used to prune this search space [7, 8, 9].

4 Assigning Registers to Loops

By applying the OPT-Assign algorithm to the body of a loop, we get an optimal assignment *for a single execution* of that code. Since this code is contained within a looping construct, the register mappings at the beginning and end of that code must match to iterate correctly. However, in general, the assignment produced by OPT-Assign will not satisfy this criteria. To remedy this, register-to-register transfers and/or spill code can be added to enforce a match. However, since the cost of this additional spill code may vary greatly from each conceivable leaf node to the root, and would vary further by unrolling the loop some number of times, OPT-Assign's results, which ignore this effect, cannot be optimal.

4.1 Our Algorithm

It is not immediately obvious how many iterations suffice to produce an assignment which results in the minimal amount of spill code. In fact, this is why this problem has been an open issue. If the process of unwinding a loop and applying OPT-Assign is continued, the cost may be decreased. By iteratively unrolling one loop iteration and applying OPT-Assign to the resulting code, we can find a new loop body, potentially spanning several iterations of the original loop, such that: a) the cost of spills per iteration in the loop body is minimal; and b) the entry and exit register mappings of the new loop match.

Our algorithm for assigning registers to loop code, called OPT-Assign-LOOP, is found in Fig. 2. The general structure of our algorithm is to iteratively unroll the loop one iteration and to apply OPT-Assign to that new iteration using, as initial mappings, the register mappings found at the end of the previous iteration. The resulting mappings are checked against those generated previously. Matching mappings correspond to legal register assignments over the unrolled loop and are, thus, removed from future consideration. The process of unrolling and assignment is repeated with those mappings which did not match.

When matches are found, the average cost per iteration is computed. Note that, if the loop were fully unrolled, the assignment with the lowest average cost per iteration would be the optimal assignment for the

```
Function OPT-Assign-LOOP (REGS : Initial mapping;
                          VA : Variable access stream;
                          K : number of iterations)
Begin
    Set $MIN$ to an empty configuration with $\infty$ average cost
    Set $i$ to 0
    Set curr_states set to REGS
    Loop
        Set save_states set to null
        Foreach state S in the curr_states set do
            new_states set = OPT-Assign(S, VA)
            Foreach state N in new_states set do
                If N matches an ancestor A then
                    Direct N to A
                    Delete N from new_states set
                    $AveCost(N) = \frac{Cost(N)-Cost(A)}{Iter(N)-Iter(A)}$
                    If $AveCost(MIN) > AveCost(N)$ then
                        $MIN = N$
                    Endif
                Endif
            Enddo
            Set save_states set to save_states set $\cup$ new_states set
        Enddo
        Set $i$ to $i+1$
        Set curr_states set to save_states set
    Until $i$ = K
    Return $MIN$
End OPT-Assign-LOOP
```

Figure 2: A loop register assignment algorithm.

loop. Since full unrolling of the loop is not necessarily practical, the parameter K denotes the maximum number of unrollings to perform. The lowest cost mapping found with this "cut-off" scheme is a local minimum, but is "global" over K iterations.

Note that this algorithm must always derive an average cost less than or equal to what OPT-Assign would derive because we deal strictly with the costs calculated by OPT-Assign and add nothing more—beyond unrolling.

4.2 Heuristic Pruning

Although our algorithm may be computationally prohibitive even for moderately long loops, it does provide a strong starting point for determining good heuristics. The computational complexity in this algorithm arises from the replacement of each register in the current configuration when a variable read or write miss occurs. Our heuristic modification is a simplistic pruning strategy where only the m best configurations are kept for future expansion once all mappings at a particular level are generated.

5 Convergence and Optimality

Previously it was not known whether optimal register assignment for a loop could be accomplished, regardless of the efficiency of the algorithm. The difficulty was due to the fact that in order to ensure optimality for the overall loop, matching of register usages at the top and bottom of the loop body may require additional spills. To optimally minimize these spills, loop unwinding with different register assignment in each unwound iteration may be needed. Furthermore, it was not known whether any finite unwinding can be guaranteed to converge and result in an optimal assignment.

To answer these questions, in [11] we introduce the notion of a *configuration graph*. Essentially, a node in this graph corresponds to a specific mapping of variables to registers and weighted, directed edges correspond to the minimal cost of using the source node as an initial mapping to an iteration and the sink node is a resultant exit mapping. Fig. 3 illustrates the notion of a configuration graph.

In this simple example, there are two registers for the loop variable access stream in (a). Using an initial mapping of register one containing **a** and register two containing **b***, the assignment tree with root {**a**,**b***} in (b) is produced. Note that, since this algorithm exhaustively generates all possibilities, one would expect six mappings corresponding to all of the permutations of the variables in the registers. However, the mapping {**b***,**a**} does not appear due to the nature of the variable accesses and the initial mapping.

A partial configuration graph is constructed from the assignment tree in (b). Traversing a path from the root configuration, labelled P, to each leaf configuration gives directed edges from P to those nodes with a weight equal to the cost of the respective paths. For instance, the path from the root to the leftmost leaf node, also labelled P, has a cost of two. Thus, an edge in the configuration graph from P to itself is added with a weighted-edge of two. Other edges are added similarly and the partial configuration graph in (c) results. To construct the complete graph requires that the assignment trees for each possible exit configuration be built.

5.1 Convergence

In order to guarantee that our algorithm converges, it must shown that by unrolling, new exit configurations (i.e., mappings of variables to registers) that previously did not exist are not generated. Because our algorithm exhaustively replaces registers each time a variable access miss occurs, all conceivable mappings of variables to registers are generated. When an unrolling of the loop body and assignment to that iteration is performed, the costs associated with going from the initial to the derived exit mappings become known. Thus, the edges in the configuration graph which connects the initial configuration with all of the possible exit configurations are generated. If the assignment algorithm is again applied to each of these nodes (e.g. unroll the loop body for another iteration), directed edges from each of those exit configurations to one another are obtained. Convergence of our algorithm, therefore, is equivalent to finding a cycle in the configuration graph. Thus, our algorithm converges because the number of variables and the number of registers is finite and, therefore, the number of permutations of the variables in the registers is finite, although exponential.

5.2 Optimality

An optimal assignment is one in which the memory traffic is minimized over execution of the loop. When the loop body is unrolled, an optimal assignment is an assignment which has minimal spill cost *over the iterations* that are contained within the unrolled loop. Thus, in the optimal assignment, the ratio of the spill cost for the new unrolled loop body to the number of iterations is minimized. In the configuration graph this corresponds to the ratio of the total cost of a cycle to the number of nodes in that cycle. Therefore, an optimal assignment is found by examining the average costs of all possible cycles of all possible lengths in the configuration graph and taking the minimum. Note that this does not necessarily correspond to the minimal cycle of length one in the graph[1]. In the worst-case it is possible that the optimal cycle must make a complete tour of the graph.

6 Experiments and Results

Our benchmark suite consists of six numerical codes written in C and compiled into RISC-like code which is typical of the code executed by most embedded core processors. The variable access streams were derived and used as input to the OPT-Assign, OPT-Assign-LOOP and Heuristic OPT-Assign-LOOP algorithms. Two experiments were conducted: the first measured the difference in assignments produced by OPT-Assign and OPT-Assign-LOOP while the second compared our heuristic to the graph coloring approach implemented in the Gnu Standard Distribution C Compiler (gcc).

[1]A cycle of length one would imply that some assignment to the loop body is minimal *and* its initial configuration naturally (i.e. without spills or transfers) matches its exit configuration.

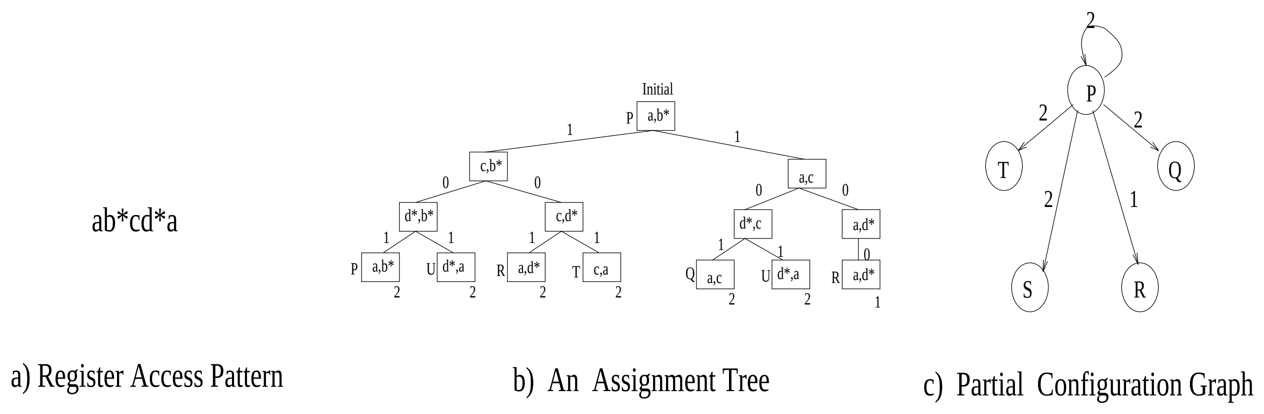

Figure 3: Building a configuration graph from the assignment trees.

6.1 Basic Block Vs. Loop Assignment

In order to create opportunity for OPT-Assign to do well, we used enlarged loop bodies constructed by unwinding the loops three times. Thus, some of our results for OPT-Assign are not whole numbers as they represent averages for a single iteration of the original loop. For OPT-Assign-LOOP, we set the control parameter K to seven iterations[2] and observed that the minimal assignments produced were usually found within four iterations.

Table 1 contains our observed results and contains the number of spills per iteration for OPT-Assign and OPT-Assign-LOOP, as well as the percentage improvement of OPT-Assign-LOOP over OPT-Assign. There is a general trend for the percentage improvement to increase as the number of registers increases (i.e., the loop algorithm produces better assignments as the number of registers increases) which can be attributed to the fundamental difference between OPT-Assign and OPT-Assign-LOOP: OPT-Assign assigns registers without regard to the effect of iterating on those register usages, while OPT-Assign-LOOP examines the iterating effects on register usages while naturally discovering a minimal assignment for a loop.

Another factor, which is an advantage for OPT-Assign-LOOP, is the presence of loop-carried dependencies. Because OPT-Assign-LOOP unrolls the loop and assigns registers based upon the exit mappings for the previous iteration, it examines the possibility of keeping a variable which has uses in future iterations in a register, thus reducing the number of loads of that variable in the future.

6.2 Heuristic Loop Vs. Graph Coloring

We compared our heuristic algorithm with the Gnu Standard Distribution C Compiler (gcc) which implements a graph coloring scheme. Because the SPARC architecture has a RISC instruction set similar to that found in many embedded core processors, gcc was configured to produce SPARC code. Also, the register assignment module was modified so that gcc would produce code which used four, six and eight registers[3]. For our heuristic version of OPT-Assign-LOOP we used a width of two. Table 2 summarizes the results of

Table 1: Comparison of basic block and loop methods.

Program	# Regs.	Basic Block	Loop	% Impr.
2D-Hydro	2	33.3	32	4%
	4	14.7	12	23%
	6	8	7	14%
	8	2.3	2	15%
Inner Product	2	10.3	9	14%
	4	4	2	100%
	6	2	1	100%
	8	1	0	∞
Linear Eqs. Solver	2	22.3	21	6%
	4	11.3	9	26%
	6	6	5	20%
	8	2	1	100%
Tri-diag. Elim. (below diag.)	2	52	52	—
	4	17.3	16	8%
	6	8	7	14%
	8	0	0	—
Tri-diag. Elim. (above diag.)	2	53	53	—
	4	19.3	17	14%
	6	9	8	13%
	8	0.3	0	∞
Prefix Sums (scan)	2	13	13	—
	4	5.7	4	43%
	6	3	2	50%
	8	0	0	—

[2] This setting could, of course, be varied.

[3] Gcc produced an internal compiler error when the real register count was set to two.

Table 2: Comparison of graph coloring and heuristic.

Program	# Regs.	Gcc	Our Heur.	% Impr.
2D-Hydro	4	19	14	36%
	6	16	8	100%
	8	12	3	300%
Inner Prod.	4	8	2	300%
	6	8	1	700%
	8	8	0	∞
Linear Eqs.	4	12	10	20%
Solver	6	10	6	67%
	8	8	1	700%
Tri-diag.	4	29	16	81%
Elim.	6	24	9	167%
(below diag.)	8	17	0	∞
Tri-diag.	4	27	17	59%
Elim.	6	22	8	175%
(above diag.)	8	19	0	∞
Prefix Sums	4	7	4	75%
(scan)	6	6	2	200%
	8	6	0	∞

the spill code produced by gcc as well as our heuristic algorithm.

In all cases, our heuristic produced assignments that were superior to gcc. In this graph coloring approach, variables are assigned to registers for their entire lifetime. In some areas of code where a variable assigned to some register is currently not being accessed, keeping it in that register causes high "register pressure" where more loads and stores of other variables (which currently *are* being accessed) are generated than necessary if the variable had been previously spilled to memory.

One interesting result is that our heuristic produces assignments which are better than OPT-Assign in a number of cases. Comparing Tables 1 and 2 shows that our heuristic results are better by an average of 8%. Although this is a heuristic version of the loop algorithm, it is able to derive better results than OPT-Assign because, as previously noted with the optimal loop algorithm, it derives register assignments which naturally match at loop top and bottom.

7 Conclusion

In this paper we have motivated and presented an algorithm which optimally assigns registers to loops. In this case an optimal assignment is one in which the memory traffic resulting from spill code is minimized. Our work answers the long standing question of whether it is possible to, in principle, achieve optimal (minimal) spill code in loops. We have demonstrated the feasibility of using our technique for the task of register assignment in embedded code generation by conducting experiments on RISC-like code typical of embedded core processors. In our experimentation, we observed that a heuristic version of our technique out-performs the graph coloring-based approach used by the Gnu C compiler (gcc). Because our approach is directly applicable to architectures having special-purpose and restricted register usages, future work will extend the approach discussed here.

References

[1] P. Briggs. *Register Coloring via Graph Coloring.* PhD thesis, Rice University, April 1992.

[2] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring Heuristics for Register Allocation. *PLDI*, June 1989.

[3] G. Chaitin, M. Auslander, A. Chandra, J. Coocke, M. Hopkins, and P. Markstein. Register Allocation Via Coloring. *Computer Languages*, 6, January 1981.

[4] F. Chow and J. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12(4), October 1990.

[5] M. Balakrishnan *et al.* Allocation of Multiport Memories in Data Path Synthesis. *IEEE Trans. on CAD*, 7(4), April 1988.

[6] L. J. Hendren, G. R. Gao, E. Altman, and C. Mukerji. A Register Allocation Framework Based on Heirarchical Cyclic Interval Graphs. *Int. Conf. on Comp. Cons.*, April 1992.

[7] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index Register Allocation. *Jour. of the ACM*, 13(1), January 1966.

[8] W. Hsu, C. Fischer, and J. Goodman. On the Minimization of Loads/Stores in Local Register Allocation. *IEEE Transactions on Software Engineering*, 15(10), October 1989.

[9] K. Kennedy. Index Register Allocation in Straight Line Code and Simple Loops. In R. Rustin, editor, *Design and Optimization of Compilers.* Prentice-Hall, 1972.

[10] T. Kim and C. L. Liu. Utilization of Multiport Memories in Data Path Synthesis. *30th DAC*, 1993.

[11] D. J. Kolson, A. Nicolau, and K. Kennedy. An Algorithm for Minimizing Spill Code in Loops. Technical Report 94-43, U.C. Irvine, October 1994. Also available as Rice University Technical Report: CRPC-TR94482.

[12] F. J. Kurdahi and A. C. Parker. REAL: A Program for Register Allocation. *24th DAC*, 1987.

[13] D. Lanneer, M. Cornero, G. Goossens, and H. De Man. Data Routing: a Paradigm for Efficient Data-Path Synthesis and Code Generation. *Int. Symp. on HLS*, May 1994.

[14] C. Liem, T. May, and P. Paulin. Register Assignment through Resource Classification for ASIP Microcode Generation. *ICCAD-94*, November 1994.

[15] P. Marwedel. Tree-Based Mapping of Algorithms to Predefined Structures. *ICCAD-93*, November 1993.

[16] C. Park, T. Kim, and C. L. Liu. Register Allocation for Data Flow Graphs with Conditional Branches and Loops. *Euro-DAC '93*, 1993.

[17] L. Stok. Interconnect Optimisation During Data Path Allocation. *EDAC*, 1990.

[18] T. Wilson, G. Grewal, B. Halley, and D. Banerji. An Integrated Approach to Retargetable Code Generation. *Int. Symp. on High-Level Synthesis*, May 1994.