UC Irvine
ICS Technical Reports

Title
Optimal register assignment to loops for embedded code generation

Permalink
https://escholarship.org/uc/item/0w3605wb

Authors
Kolson, David J.
Nicolau, Alexandru
Dutt, Nikil

Publication Date
1995-07-04

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

SLBAR
Z
699
C3
no. 95-46

Optimal Register Assignment to Loops for Embedded Code Generation *

Technical Report #95-46

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

Notice: This Material may be protected by Copyright Law (Title 17 U.S.C.)

4 July 1995

Abstract

One of the challenging tasks in code generation for embedded systems is register assignment. When more live variables than registers exist, some variables will necessarily be accessed from data memory. Because loops are typically executed many times and are often time-critical, good register assignment in loops is exceedingly important as accessing data memory can degrade performance. The issue of finding an optimal register assignment to loops has been open for some time. In this paper, we present a technique for optimal (i.e., spill minimizing) register assignment to loops. First, we present a technique for register assignment to processor cores which are characterized by a consolidated register file. Then, we extend the technique to include architecture styles which are characterized by the partitioning of registers into multiple register files and/or a combination of general- and special-purpose registers. Experimental results demonstrate that, while the optimal algorithm may be computationally prohibitive, heuristic versions obtain results with performance better than that of an existing graph coloring approach.

*This work supported in part by ONR grant K000042879921.

1 Introduction

Typically, an embedded system consists of an embedded, programmable processor interconnected with some memory and specialized "accelerators" (application-specific components). This embedded processor can be realized either by a *processor core* or by an *application-specific instruction-set processor (ASIP)*. Processor cores offer the core functionality and datapath regularity of a general purpose processor and represent an "off-the-shelf" solution, examples of which are the MIPS RC4000 and the microSPARC-II. Conversely, ASIPs offer an application-specific instruction set and some degree of irregularity in the datapath (for efficient implementation of the application-specific instructions/features) and represent a semi-custom approach. Examples of ASIPs are fixed-point DSPs such as Texas Instruments' TMS series, Motorola's 56000 and floating-point DSPs such as Texas Instruments' TMX320C44.

Currently, much research has focused on code generation for these embedded systems [16, 18, 21, 22, 27]. 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 architecture'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 have a small number of registers, with, perhaps, some registers having restricted or specialized uses. Because of these limitations, register assignment is exceedingly critical, especially for innermost loops which are executed many times and often time-critical. Thus, any mapping of variables to registers which contains poor choices for variable spills will adversely affect performance.

In the compiler domain, *optimal* register assignment solutions have been extensively studied [11, 12, 13]. 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) so that it is correct to iterate over that loop code.

In this paper we demonstrate that the algorithms for register assignment in basic blocks given in [12, 13] can be extended to assign registers in loops by incorporating loop unrolling techniques into the algorithm. We also present a heuristic derived from our algorithm that, in practice, seems to perform as well as its exponential counterpart.

In Section 2 we discuss related work and in Section 3 we describe the problem we are addressing. Section 4 discusses the optimal assignment of registers in basic blocks for architectures with consolidated register files,

while Section 5 extends this technique to loops. Section 6 discusses the convergence and optimality of the loop algorithm. Section 7 extends the loop algorithm to assign registers for architectures with multiple register files and special purpose registers. Section 8 gives our experimentation and observed results and Section 9 concludes this paper.

2 Related Work

The register assignment problem is an important issue and has become pervasive in many areas: compiler design, where, typically, the number of registers is fixed and uniform access to all registers is available; high-level synthesis, where the number and interconnection of registers is being synthesized; and code generation for embedded systems, where embedded processors have a very limited number of registers, with, perhaps, partitioned register files and special-purpose registers.

In the compiler domain, the most popular approach to register assignment is the heuristic graph coloring approach [4, 6]. In assigning registers by graph coloring, a graph is constructed where each node represents a variable and the edges between nodes represent the overlapping of the respective variable's lifetimes. The task is then to "color" the graph with the number of colors equal to the number of physical registers. If a coloring is not found, some variable is spilled to memory and the process is repeated. The key to good register assignment in this scheme is the selection of a particular variable to spill—heuristics for selection have received attention [5] along with methods of coloring the graph [7]. Also, [10] addresses loops but without regard to the number of register-to-register transfers potentially required by their technique at the end of an iteration.

Many researchers have felt that for particularly critical code segments, such as the innermost loops of time-sensitive applications, an optimal assignment is necessary. Horwitz *et al.* present a method in [11] for obtaining an optimal register assignment to *index* registers which minimizes the number of loads and stores. Further work either improves upon the efficiency of the Horwitz algorithm [19] or extends the basic algorithm to deal with simple loops [13], but in doing so loses optimality and degrades performance. More recent research [12] extends the basic idea in Horwitz's algorithm to include register assignment for *general purpose* registers.

In High-Level Synthesis the problem of register assignment traditionally refers to determining the number of registers necessary to save values between time-steps [15, 24]. In order to reduce the interconnect and multiplexor cost of scattered registers, some researchers have focused on grouping registers into memory modules [1, 3, 14, 20]. Other research has addressed the assignment of registers to loop variables whose lifetimes are cyclic in nature [23, 26]. These approaches (arbitrarily) break a cyclic variable's lifetime at loop boundaries, creating two "coupled" variables which the assignment process tries to assign to the same register. If the coupled variables are not assigned to the same register, register transfers are necessarily inserted at the end of the loop to correctly set-up the next iteration. Because these techniques were developed for *register allocation*, they do not consider cases where variables are stored 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 [18] or formulated the problem of register assignment as an ILP formulation [28].

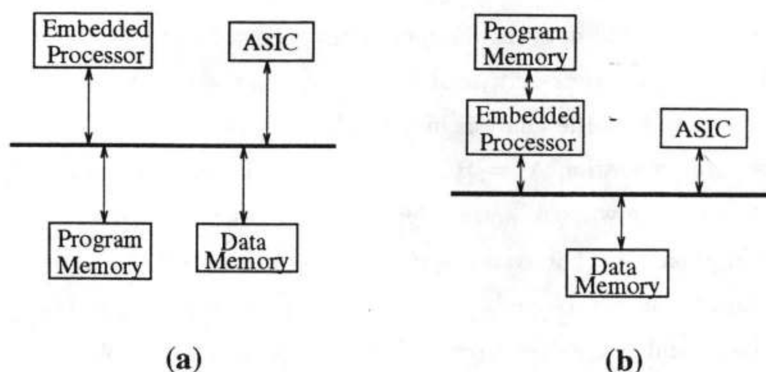


Figure 1: Target architecture organization.

However, these techniques introduce register-to-register moves at loop boundaries. In [16] a complex searching scheme is used to navigate a large search space with many trade-offs, one of which is register assignment. Some work has been done in minimizing the number of spills for an accumulator-based architecture [17].

However, 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 *and* minimizes the cost due to added spill code).

3 Target Architecture and Problem Description

An example of the architectural organization that we are targeting is found in Figure 1. In this architecture, an embedded processor is interconnected with program memory (typically, *read-only memory*), data memory and one or more ASICs. The embedded processor can be realized either by a processor core or by an ASIP. As previously mentioned, a processor core offers the functionality and datapath regularity of a general-purpose processor (GPP), and compares with a GPP in the following: a narrower datapath bit-width, a smaller instruction set, and a fewer number of registers which are consolidated into one register file, examples of which are the MIPS RC4000 and Sun Microsystem's microSPARC-II. By contrast, an ASIP offers an application-specific instruction set and some degree of irregularity in the datapath. Due to the application-specific nature of an ASIP, the available registers can be partitioned into multiple register files. Examples of which can be found in fixed-point DSPs such as Texas Instruments' TMS series and Motorola's 56000 and floating-point DSPs such as Texas Instruments' TMX320C44. Thus, the task of register assignment for a processor core corresponds to determining a mapping of program variables to the registers contained within one register file and register assignment for an ASIP corresponds to determining a mapping of program variables to the registers of multiple register files while honoring any access (i.e., port) restrictions to the register files.

In our approach the task of register assignment follows that of code selection and scheduling of operations into time steps. When resource shortages occur during the register assignment phase (i.e., when more live variables than registers exist, thus requiring multiple variables to share registers) spill code, or explicit data transfer

operations between the registers and data memory, becomes necessary. Our goal is to minimize the number of transfer operations between the registers and the data memory that will be repeatedly executed within a loop.

Register assignment begins with the analysis of variable accesses in execution to derive the *variable access stream.* For example, for the operation **A = B + 1**, the variable access stream is B, A* (reads of variables before writes), where '*' denotes a write to a variable. To denote concurrent accesses, parentheses bracket those reads or writes performed in parallel. For example, if the operations **A = B + 1** and **C = D + E** are executed concurrently, then the variable access stream is (BDE)(A*C*), as the variables B, D and E are read concurrently after which the variables A and C are concurrently written. Once the variable access stream is derived, it is input to the assignment algorithm.

4 Optimally Assigning Registers in Basic Blocks

Using a variant of the algorithm presented in [12], we can derive an optimal (i.e., spill minimizing) algorithm that assigns variables to registers in basic blocks. This algorithm, which we call OPT-Assign-BB, is found in Figure 2. OPT-Assign-BB takes as input the variable access stream for a code segment and the mapping of variables to registers which immediately precedes that segment (which could be *null*, signifying that all registers are initially free). This algorithm then builds an assignment tree where the nodes in the tree correspond to a *variable mapping* or *configuration* representing the contents of each register found at some particular point in execution and the root of the tree is the given (initial) mapping of variables to registers. Each path in the tree from the root to a leaf is a (unique) mapping of variables to registers.

As the assignment tree is built, each successive level in the tree is derived by examining the variable stream and all of the current configurations to determine if they contain the variable under consideration. If the variable is contained within a configuration, that node is duplicated at the next level of the tree and a zero-cost edge connects the two. When a variable is not contained within a node, a *variable access miss* occurs and spill code might be necessary. For any configuration causing an access miss, each variable currently in that configuration is replaced in turn by an access to the faulting variable. An edge joining the access miss node with each of the newly created nodes represents the cost, in spill code, of going from the first mapping to the second. This cost is composed of the cost of (possibly) storing the replaced register if it is live and dirty[1] (the Store-Cost(V') in our algorithm) and/or the cost of (possibly) loading the faulting variable (the Load-Cost(V) in our algorithm) if this is a variable read. Setting the load and store costs both to one gives a total cost equal to the number of memory operations[2]. Thus, if there are r registers, a faulting configuration in the current level will generate r configurations in the next level, resulting in an optimal, but exponential method. Heuristics can be (and have been) used to prune this search space [11, 12, 13].

Once the last variable access is considered, all the leaves of the assignment tree are examined for the lowest

[1] Every variable is assumed to have a unique memory location where it may be kept if a spill of that variable is necessary. Dirty refers to the case where the value in a register is inconsistent with the value stored in the memory location.

[2] Priorities can be given to loads or to stores simply by changing these costs.

```
Function OPT-Assign (REGS : Initial register configuration;
                     VA : Variable access pattern)
Begin
   Set curr_states set to REGS
   Foreach varible access V in VA do
      Foreach config. N in curr_state 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 2: A register assignment algorithm.

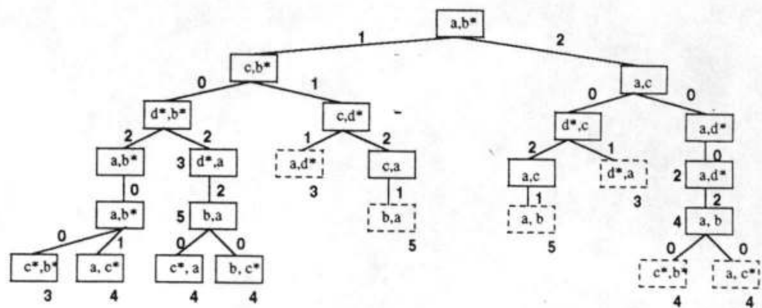
b = a + 1
d = c + 7
c = a + b

b, d live

a) Code Segment

ab*cd*(ab)c*

b) Variable Access Stream



c) Assignment Tree

Figure 3: Building an assignment tree.

cost node. Tracing the path from the root to this lowest cost node will yield an assignment of registers to variables that results in the minimal cost in terms of memory loads and stores due to spill code, because it has exhaustively generated every possible assignment.

In Figure 3(a), an example code segment appears and Figure 3(b) contains the variable access stream for this segment. For this example, there are two registers which have been initially assigned R1 to **a** and R2 to **b**. The OPT-Assign-BB algorithm begins with this (initial) mapping and constructs the assignment tree in Figure 3(c).

The first two variable accesses, to the variables **a** and **b***, are to variables contained in the current configuration, therefore, no spill code is necessary. However, the next access to **c** causes a variable access miss and spill code becomes necessary. Two configurations are generated at the next level, corresponding to assigning **c** to R1 or to R2. The left child assigns **c** to R1, displacing the variable **a**. Since **a** does not need to be stored (its value is consistent with that in memory), the spill code generated by this is the load of **c**, for a cost of one. The other possibility, assigning **c** to R2, is represented by the right child and displaces the variable **b**. Since **b** is dirty (it has previously been written), **b** must be stored and then **c** is loaded, for a cost of two.

This process, examining the next variable access(-es) and checking whether they are contained within the current configurations, continues for the remainder of the variable access stream and the full tree in Figure 3(c) is generated. In Figure 3(c), there are some nodes which have dashed outlines. These nodes can be pruned from the tree as there are identical nodes at the same level which will generate identical sub-trees. Within a group of identical nodes, only the one with lowest cost need be kept, breaking ties arbitrarily.

5 Extending the Basic Block Algorithm to Loops

By applying the OPT-Assign-BB 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, it is necessary for the register mappings at the beginning and end of the code segment to match in order to correctly iterate over that segment. In general, the assignment produced by OPT-Assign-BB will not satisfy this criteria (i.e., the lowest cost config-

uration at a leaf of the assignment tree does not necessarily match the root). Thus, this basic algorithm is not adequate to optimally assign registers to loop code.

To remedy this, one might try simply to add register-to-register moves and/or spill code (loads and/or stores) to enforce a match. However, since the cost of this additional spill code may vary greatly from each conceivable mapping to another, and would vary further by unrolling the loop some number of times, OPT-Assign-BB's results, which ignore this effect, cannot be optimal. Another sub-optimal approach is to 'force' a match between loop top and bottom, i.e., to choose from the exponential tree derived by OPT-Assign-BB the least cost leaf node which is identical to the initial configuration (leaf configurations which match the root configuration are not necessarily guaranteed to be those with lowest cost).

5.1 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-BB is continued, the cost may be decreased. By iteratively unrolling one loop iteration and applying OPT-Assign-BB 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 configurations of the new loop match.

Our algorithm for assigning registers to loop code, which we will refer to as OPT-Assign-LOOP, is found in Figure 4. The general structure of our algorithm is to iteratively unroll the loop one iteration and then to apply OPT-Assign-BB to the new iteration once for each possible previous iteration exit mapping. Then the algorithm analyzes each resulting exit mappings of that new iteration to determine if matches between those nodes and iteration ancestors (i.e., a node in the assignment tree that lies on the path from the root to this node and also lies on an iteration boundary). If so, a legal register assignment to the unrolled loop has been found. If not, then that exit mapping becomes one of the mappings which will be used as an initial configuration to the next iteration.

Each time that a match is found, our algorithm computes the average cost per iteration for that assignment (since the assignment may span multiple iterations). If the loop were fully unrolled, the assignment with the lowest average cost per iteration would be the optimal assignment for the loop. Since full unrolling of the loop is not necessarily practical, we have parameterized our algorithm with K, the number of unrollings of the loop body to perform. The lowest cost mapping found with this "cut-off" scheme is a local minimum, but is "global" over the number of iterations unrolled so far (K). Note that this algorithm must always get an average cost less than or equal to what OPT-Assign-BB would get because we deal strictly with the costs calculated by OPT-Assign-BB and add nothing more—beyond unrolling.

```
Function OPT-Assign-LOOP (REGS : Initial mapping;
                          VA : Variable access pattern;
                          K : number of iterations)
Begin
  Set MIN to an empty configuration with ∞ average cost
  Set i to 0
  Set curr_states set to REGS
  Loop
    Set save_state_set to null
    Foreach state S in the curr_state set do
      new state set = OPT-Assign(S, VA)
      Foreach state N in new state set do
        If N matches an ancestor A then
          Direct N to A
          Delete N from new_states set
          AveCost(N) = (Cost(N) - Cost(A)) / (Iter(N) - Iter(A))
          If AveCost(MIN) > AveCost(N) then
            MIN = N
          Endif
        Endif
      Enddo
      Set save_state_set to save_set_state ∪ new_set_state
    Enddo
    Set i to i + 1
    Set current register state set to save_state_set
  Until i = K
  Return MIN
End OPT-Assign-LOOP
```

Figure 4: A loop register assignment algorithm.

5.2 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. That is, for each node in the current level, when an access miss occurs, all possibilities for spills are considered. Then, of those newly generated nodes, the m lowest cost nodes are retained for consideration.

6 Convergence and Optimality of the Loop Algorithm

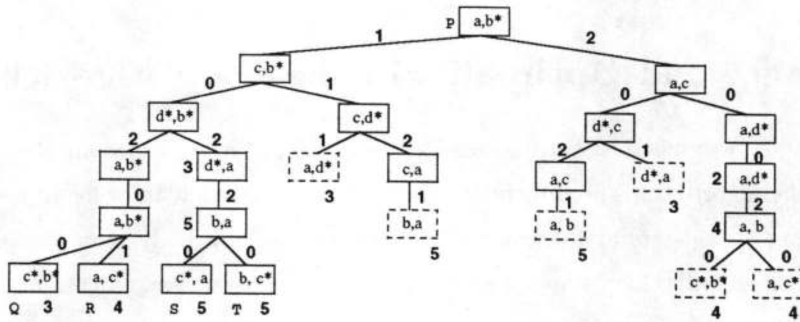
Previously it was not known whether optimal register assignment for a loop could be accomplished, regardless of the efficiency of the algorithm. The difficulty was due to the fact that in order to ensure optimality for the overall loop, matching of registers at the top and bottom of the loop body may require additional spills. To optimally minimize these spills, loop unwinding with different register assignments 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, we introduce the notion of a *configuration graph*. A node in the configuration graph corresponds to a specific mapping of variables to registers found at an iteration boundary and a directed edge in the configuration graph corresponds to the cost in spill code of using the source node as the initial mapping to an iteration, applying the loop algorithm and having the sink node as one of the resultant nodes. Thus, the edge represents the cost of spill code with the source node as the initial register assignment to and the sink node resulting from an iteration of the loop.

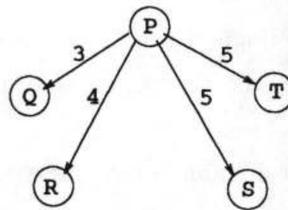
Figure 5 illustrates the method of building a configuration graph. We use the same assignment tree from Figure 3 and have labelled the leaf nodes. A partial configuration graph, shown in (b), can be constructed from the assignment tree in (a). Traversing a path from the root configuration, which has been labelled P, to each leaf configuration gives a directed edge in the configuration graph from P to that respective node with a weight equal to the cost of the path. For instance, the path from the root to the first leaf node on the left, labelled Q, has a cost of three. Thus, an edge in the configuration graph from P to Q is added with that edge having weight three. Similarly, other edges are added to the configuration graph by traversing the various paths. The partial configuration graph in (b) results. To construct the complete graph requires that we build the assignment trees for each possible exit configuration.

6.1 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



a) Assignment Tree



b) Partial Configuration Graph

Figure 5: Building a configuration graph from the assignment trees.

exhaustively replaces registers each time a variable access miss occurs, all conceivable mappings are generated. Stated another way, 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 connect 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.

6.2 Optimality

An optimal assignment is one in which the memory traffic is minimized. When the loop body is unrolled, an optimal assignment is an assignment which has minimal memory traffic or 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 it contains, is minimized. In the configuration graph this corresponds to the ratio of the total cost of some 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[3]. In the worst-case it is possible that the optimal cycle must make a complete tour of the graph.

7 Extending the Model to Heterogeneous Register Usages

The algorithm presented earlier for register assignment in loops has the underlying assumption that access to all available registers is equivalent, as is found, for instance, in general-purpose processors and processor cores. That is, all registers are consolidated into one register file and any variable mapped to a register is uniformly available to any operation using that variable. However, in the case of ASIPs, or any architecture where the available registers have been partitioned into disjoint register files or some of the available registers have specialized purposes, this assumption must be modified to generate feasible register assignments. Also, previously there was the assumption that enough ports on the register file exist to support the reading and writing of all variables accessed in a particular step. However, in the case of ASIPs, it is possible that restrictions are present on the number of registers that are concurrently accessible (i.e., the number of read/write ports on a register file constrains the number of reads/writes to that register file).

In this section we discuss the extension of our algorithm to loop register assignment when the target architec-

[3] 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 moves) matches its exit configuration.

ture contains multiple register files and/or special-purpose registers and specific restrictions on register accessing exist. First, we start by discussing the addition of register classes to the model. Then, we consider separate modifications necessary to our algorithms to handle special-purpose registers and multiple register files.

7.1 Adding Register Classes to the Model

To extend our algorithms, we introduce the notion of *register classes*. Register classes have been used in compilers [2, 25] and in microcode synthesis [8, 18] to denote functional equivalences between registers. However, combining all registers having the same (potential) usages into one class is not precise enough for register assignment to our target architecture class. To see this, consider a simple case where two register files are composed of "general-purpose" registers, each register file connected to a different ALU. Clearly, any operation scheduled on either ALU must have its operands present in the respective register file. However, if the collective registers are grouped into one register class (called "general-purpose"), it is possible that the necessary operands have been assigned in such a way as to honor the register classes, but be invalid for execution, thus, making that register assignment invalid. The main cause of this problem is not due to registers being grouped by their equivalency, but, rather, how the equivalency is established.

In our approach, two types of register classes are defined: *connectivity register classes* and *operation register classes*. The connectivity register class (conn_RC) defines the equivalency between registers as a function of the architecture's connectivity, while the operation register class (oper_RC) defines the equivalency between registers as a function of an operation's semantics. The motivation for deriving both of these classes is that the connectivity of the architecture defines which registers may be read from or written into by some functional unit, while the semantics of a particular operation executing on a particular functional unit may preclude the use some of the connected registers (a load operation, for instance, may require that the memory address reside in a specific register, while the functional unit that executes that load operation may be connected to many registers which do not serve the same purpose).

Figure 6 contains an algorithm to derive the register classes for a given architecture. A connectivity register class is derived for each of the inputs and outputs of each functional unit in the architecture based upon which registers may be accessed by that input or output. Operation register classes are derived by examining which operations a functional unit can execute and selecting all of the readable (and writable) registers imposed by an operation's semantics. In a large number of cases, the conn_RC and oper_RC will be equivalent.

7.2 Extension to Special-Purpose Registers

The algorithm OPT-Assign-BB exhaustively generates variable mappings by placing a variable in each register either when a read miss occurs (requiring a load of the variable) or when a variable is written. When access to all registers is uniform, this strategy is correct. However, when some registers have specialized usages, this strategy generates some mappings which are invalid as variables have been assigned to registers which cannot perform the required specialized function. Thus, it is necessary to restrict the placement of variables into registers so

```
Procedure Derive-Register-Classes()
Begin
   Foreach FU, f in the architecture do
      Foreach Input, i of f do
         Set RegClass(f, i) to ∪ all regs connected to i
      Enddo
      Foreach Output, o of f do
         Set RegClass(f, o) to ∪ all regs that f may write
      Enddo
      Foreach operation, op, that f can execute do
         Set RegClass(f, op_input) to ∪ all regs that op may read
         Set RegClass(f, op_output) to ∪ all regs that op may write
      Enddo
   Enddo
End Derive-Register-Classes
```

Figure 6: An algorithm to derive register classes.

that variables only reside in registers which can perform the necessary functionality.

With the notion of register classes, we can extend the OPT-Assign-BB algorithm to handle registers which have specialized usages. When a variable causes an access miss, only those registers which perform the necessary functionality are considered. These are found by intersecting the *operation register class* for the accessing operation and the *connectivity register class* for the functional unit that is executing that operation. Recall that we perform register assignment on a scheduled dataflow graph. Thus, when performing register assignment the operations (and their types) which access variables, as well as the functional units that those operations execute on, are known—retrieving this information is a simple matter.

Figure 7 contains an extended version of the OPT-Assign-BB algorithm for register assignment with specialized register usages. The function op_of returns the operation which currently accesses the variable V. From this, the type of operation and the functional unit that executes the operation are found via calls to functions $OperationType$ and $FunctionalUnit$, respectively. Then, the appropriate operation register class and connectivity register class are found and intersected. RC_intersect, the intersection of these classes, defines the feasible registers in which a variable V may reside. If the variable is in one of those registers, then no spill code is necessary. If not, then all of the registers contained in RC_intersect are candidates for replacement and spill code is generated.

7.3 Extension to Multiple Register Files

To extend our algorithms to assign registers to multiple register files requires that the notion of a node in the assignment tree be altered. In assigning registers to an architecture with a consolidated register file, the semantics of a node are that all registers are uniformly available. For instance, if there are eight registers filled with the

```
Function OPT-Assign (REGS : Initial register configuration;
                     VA : Variable access pattern)
Begin
    Set curr_states set to REGS
    Foreach varible access V in VA do
        Set op_type to OperationType(op_of(V))
        Set fu to FunctionalUnit(op_of(V))
        Set RC_intersect to RegisterClass(op_type) ∩ RegisterClass(fu)
        Foreach config. N in curr_state set do
            If V ∈ RC_intersect then
                Copy N to new_states set
            Otherwise
                Forall registers R ∈ RC_intersect 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 7: Extending OPT-Assign-BB to special-purpose registers.

variables **a**–**h**, a mapping of variables to registers is represented as **{a,b,c,d,e,f,g,h}**, signifying that **a** is mapped to register one, **b** is mapped to register two, etc.

To model multiple register files, we change the information contained in a node to reflect the grouping of registers into a register file. Each node in the assignment tree is then composed of a number of *register sets* equal to the number of register files.

7.3.1 Assigning Variables

Figure 8 contains an extended version of the OPT-Assign-BB algorithm which assigns registers to multiple register files. The main modification required when multiple register files exist is, only the registers in the respective register file are examined to determine if a variable is resident. If a variable is not contained within the necessary register file, rather than loading it from memory, a check is first made to see if the variable is contained within one of the other register files. If so, then a move operation is used to transfer the value into the necessary register file if the necessary connections exist as this transfer is likely to have a lower latency than a load from (slower) memory. Otherwise, the variable is loaded from memory. Once a spill is considered[4], any access restrictions present on the register files, such as the number of registers which can be simultaneously accessed, are considered. If the restrictions are satisfied, then the assignment is valid and is maintained for future assignment, otherwise the mapping represents an assignment which causes an access conflict to exist and the node is removed from future consideration.

7.3.2 A Note on Optimality

With the addition of register classes and extension to special purpose registers, our algorithm derives optimal (i.e., spill minimizing) results. However, with multiple register files and the version of our algorithm presented, it may be possible that sub-optimal results are obtained. Previously, when a variable was assigned to a register, all registers were viewed as candidates for replacement. Extending this to cases where some registers have special purposes merely removes some number of registers as candidates (and, thus, serves to restrict the growth of the assignment tree). However, in the case of multiple register files, our algorithm may no longer derive an optimal solution. When one variable is displaced by another in the same register file, that displaced variable may need to be stored into data memory, requiring a load when it is needed in the future. However, if a free register exists in some other register file, then it might be possible to "store" the displaced value there temporarily until its future use. Further, even if there is no free register in the remote file, it still possible that some remote variable can be spilled without loss of performance, thus freeing a register. In general, this effect can have cascading effect and become quite complex, with a variable "hopping" from register file to register file until its future use. Extending our algorithms to handle this would be straightforward, but impractical.

[4] We can assign different costs for spilling to memory, fetching from memory and fetching from another register file.

```
Function OPT-Assign (REGS : Initial register configuration;
                     VA : Variable access pattern)
Begin
   Set curr_states set to REGS
   Foreach varible access V in VA do
      Foreach config. N in curr_state set do
         If /* V is already in proper RF */ then
               /* Copy the current state to the next level */
         Else Forall register files, RF, do
            If /* V is contained in another RF */ then
                  /* Generate a move of V to this RF */
                  /* Check access restrictions */
            Endif
         Otherwise /* V must be loaded from memory */
               /* Generate all possible spills of */
               /* variables contained in this RF */
               /* Check access restrictions */
         Endif
      Enddo
      Set curr_states set to new_states set
   Enddo
   Return new_states set
End OPT-Assign
```

Figure 8: Extending OPT-Assign-BB to multiple register files.

8 Experiments and Results

To examine the benefits of our technique, we conducted two sets of experiments. The first targeted architectures with consolidated register files (e.g., processor cores), while the second targeted architectures with partitioned register files and special use registers (e.g., ASIPs).

For both experiments, our benchmark suite consisted of six numerical codes written in C and then compiled into RISC-like code which is typical of the code executed by most embedded core processors and ASIPs. From those codes, the variable access streams were derived and used as input to our algorithms. For each experiment, we derived register assignments from the optimal basic block algorithm (OPT-Assign-BB), the optimal loop algorithm (OPT-Assign-BB-Loop), and a heuristic version of the loop optimal algorithm (heuristic OPT-Assign-BB-Loop) and counted the number of spills (i.e., loads and stores) for those assignments.

In generating assignments for the OPT-Assign-BB algorithm, the registers were assumed to be empty upon initial assignment to the loop. Because OPT-Assign-BB is not guaranteed to produce an assignment in which the initial and exit mappings match, we note the point at which the registers became full (i.e., the point where more live variables than registers exist) and introduce spill code and/or moves to match usage from the minimal (leaf) node to the previously noted (initial) node. Also, in order to create opportunity for OPT-Assign-BB to do well, we used enlarged loop bodies constructed by unwinding the loops three times. Thus, some of our results for OPT-Assign-BB are not whole numbers as they represent averages for a single iteration of the original loop.

8.1 Experimentation with A Consolidated Register File

We use the microSPARC-II as a target for code generation to an architecture with a consolidated register file as the microSPARC-II has a RISC instruction set similar to that found in many embedded core processors. Using the variable access streams, register assignments were produced by the OPT-Assign-BB, OPT-Assign-LOOP and heuristic OPT-Assign-LOOP algorithms. From those register assignments, the number of spill code operations were counted. In section 8.1.1, we compare the OPT-Assign-BB and OPT-Assign-LOOP results. Next, we compare our heuristic version of OPT-Assign-LOOP to the graph coloring approach implemented in the Gnu C Compiler[5] in section 8.1.2. Subsection 8.1.3 compares the optimal and heuristic loop versions and section 8.1.4 compares the number of iterations spanned by the optimal and heuristic loop assignments.

8.1.1 Comparison of OPT-Assign-BB and OPT-Assign-LOOP

Table 1 contains our observed results and contains the number of spills per iteration for OPT-Assign-BB and OPT-Assign-LOOP, as well as the absolute percentage improvement of OPT-Assign-LOOP over OPT-Assign-BB measured as: $\frac{Spills_{BB} - Spills_{Loop}}{Spills_{Loop}}$. There is a general trend for the percentage improvement to increase as the number of registers increases (i.e., the disparity between the loop assignments and the basic block assignments increases as the number of registers increases) which can be attributed to the fundamental difference between

[5] The code produced by this compiler is generally accepted to be of high quality [9].

Program	Number of Registers	OPT-Assign-BB Spills per Iteration	OPT-Assign-LOOP Spills per Iteration	% Improvement
2D-Hydrodynamics	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 Equations	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	—

Table 1: Comparison of basic block optimal and loop optimal for the microSPARC-II.

Program	Number of Registers	Gnu gcc Spills per Iteration	Heuristic OPT-Assign-LOOP Spills per Iteration	% Improvement
2D-Hydrodynamics	4	19	14	36%
	6	16	8	100%
	8	12	3	300%
Inner Product	4	8	2	300%
	6	8	1	700%
	8	8	0	∞
Linear Equations	4	12	10	20%
	6	10	6	67%
	8	8	1	700%
Tri-diag. Elim. (below diag.)	4	29	16	81%
	6	24	9	167%
	8	17	0	∞
Tri-diag. Elim. (above diag.)	4	27	17	59%
	6	22	8	175%
	8	19	0	∞
Prefix Sums (scan)	4	7	4	75%
	6	6	2	200%
	8	6	0	∞

Table 2: Comparison of graph coloring and our heuristic for the microSPARC-II.

OPT-Assign-BB and OPT-Assign-LOOP: OPT-Assign-BB 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. The minimal assignments produced by OPT-Assign-BB are not guaranteed to match at the loop entry and exit points. Therefore, some spill code becomes necessary to match the register usage at those two points. However, because OPT-Assign-LOOP explores the possibilities of keeping the variables in registers found at the loop end as it assigns registers to the next iteration, it discovers better register usages and places for inserting spill code. Essentially, OPT-Assign-LOOP produces superior results as it naturally (i.e., without additional loads, stores and/or register moves) finds a match between loop entry and exit configurations during the assignment process.

8.1.2 Comparison of Heuristic OPT-Assign-LOOP and Graph Coloring

Gcc was configured to produce SPARC code and the register assignment module was modified so that gcc would produce code which used four, six and eight registers[6]. For our heuristic version of OPT-Assign-LOOP we used a pruning factor parameter $m = 2$ best configurations[7]. Table 2 summarizes the results of the spill code produced by gcc as well as our heuristic algorithm. Percentage improvement is measured as: $\frac{Spills_{gcc} - Spills_{heur}}{Spills_{heur}}$.

In all cases, our heuristic produced assignments that were superior to gcc. In the graph coloring approach,

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

[7] Recall that our pruning strategy keeps the m best configurations after considering all possibilities each time a variable miss occurs.

variables are assigned to registers for their entire lifetime. In some segments of code, where a variable assigned to some register is currently not being accessed, keeping that variable in a register causes high "register pressure" where more loads and stores of other variables (which currently are being accessed) are generated than is necessary if the unaccessed variable had been previously spilled to memory during this segment.

Another interesting result is that our heuristic produces assignments which are better than OPT-Assign-BB (the optimal assignment for basic blocks) in a number of cases. Comparing Tables 1 and 2 shows that our heuristic results are better than optimal basic block assignment 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-BB because it has the ability to find matching register assignments over loop execution which the OPT-Assign-BB algorithm does not.

8.1.3 Comparison of OPT-Assign-LOOP and Heuristic OPT-Assign-LOOP

Table 3 contains the number of spills per iteration for the OPT-Assign-BB-Loop and heuristic OPT-Assign-BB-Loop algorithms in columns three and four, respectively. Column five contains the percentage within optimal that the heuristic results are, measured as: $\frac{Spills_{Heur} - SpillsOpt}{Spills_{Opt}}$. For this measure, lower numbers are better (i.e., the lower the number, the closer the heuristic is approximating the optimal). In half of the cases (12 of 24), the heuristic produced results that are equal to the optimal. Also, 71% of the cases (17 of 24) are within 10% of the optimal, while 92% (22 of 24) are within 20% of the optimal. These results demonstrate that, while the optimal may be computationally prohibitive, the simple heuristic version, which executes in a matter of seconds, produces results that are acceptably close enough to the optimal.

8.1.4 Code Size of Loop Register Assignments

One concern of our technique is the increase in code size that results from loop unrolling as our method typically produces register assignments which span multiple iterations. This is especially a concern in the context of embedded code generation where the program code resides in ROM, and, thus, directly affects the ROM size. In Table 4 we have noted the number of iterations spanned by the assignments produced by OPT-Assign-LOOP and heuristic OPT-Assign-LOOP[8] in columns three and four, respectively. Column five of Table 4 indicates whether the heuristic version derived assignments with the same amount of spill code as the optimal.

In the majority of cases (18 of 24), the number of iterations spanned by both versions is the same. Of those 18 cases, the heuristic version derived assignments with the same amount of spill code as the optimal in nine cases (50%). There are a few cases (5 of 24) where the heuristic spanned more iterations than the optimal, and of those, the same amount of spill code was produced in three cases. There is one case where the heuristic version spanned a fewer number of iterations than the optimal due to the nature of the heuristic. However, in this case, the heuristic version produced more spill code. Overall, the number of iterations spanned by the loop assignments ranges between two and five, which, we feel, is within acceptable limits for the performance gain

[8] Results for the two register case for the heuristic version appear in this table, but do not appear in Table 2 as, mentioned earlier, they are not available for gcc.

Program	Number of Registers	OPT-Assign-LOOP Spills per Iteration	Heuristic OPT-Assign-LOOP Spills per Iteration	% within Optimal
2D-Hydrodynamics	2	32	34	6%
	4	12	14	17%
	6	7	8	14%
	8	2	3	50%
Inner Product	2	9	10	11%
	4	2	2	0%
	6	1	1	0%
	8	0	0	0%
Linear Equations	2	21	22	5%
	4	9	10	11%
	6	5	6	20%
	8	1	1	0%
Tri-diag. Elim. (below diag.)	2	52	54	4%
	4	16	16	0%
	6	7	9	29%
	8	0	0	0%
Tri-diag. Elim. (above diag.)	2	53	55	4%
	4	17	17	0%
	6	8	8	0%
	8	0	0	0%
Prefix Sums (scan)	2	13	14	8%
	4	4	4	0%
	6	2	2	0%
	8	0	0	0%

Table 3: Comparison of loop assignments for the microSPARC-II.

Program	Number of Registers	# Iterations		Equal Spill Code?
		OPT-Assign-LOOP	Heuristic OPT-Assign-LOOP	
2D-Hydrodynamics	2	4	3	no
	4	3	3	no
	6	3	3	no
	8	3	3	no
Inner Product	2	4	4	no
	4	3	4	yes
	6	2	3	yes
	8	2	2	yes
Linear Equations	2	5	5	no
	4	4	5	no
	6	4	4	no
	8	3	3	yes
Tri-diag. Elim. (below diag.)	2	4	4	no
	4	3	4	yes
	6	3	3	no
	8	3	3	yes
Tri-diag. Elim. (above diag.)	2	3	4	no
	4	3	3	yes
	6	3	3	yes
	8	3	3	yes
Prefix Sums (scan)	2	4	4	no
	4	4	4	yes
	6	3	3	yes
	8	2	2	yes

Table 4: Comparison of loop assignment code sizes for the microSPARC-II.

resulting from fewer memory accesses.

8.2 Experimentation with Distributed Register Files -

We use the TMX320C44 as an example of an architecture with distributed register files. Figure 9 shows a simplified view of the architecture and in section 8.2.1 we give a brief overview. For the purposes of code generation we use "scaled-down" versions of the TMX320C44 in which we parameterize the number of registers for the register banks. In our tables, the "# Registers" column denotes this number (i.e., a '2' means that each bank contained two registers—two extended precision, two general-purpose and two auxiliary registers—for a total of six). We then generated assignments with OPT-Assign-BB, OPT-Assign-LOOP and our heuristic version of OPT-Assign-LOOP. In section 8.2.2 we compare the results of basic block optimal and loop optimal. As we are unable to gain access to a commercially available compiler which generates code for the TMX320C44, a comparison between our heuristic and another heuristic is unavailable. However, in section 8.2.3, we compare the results of OPT-Assign-LOOP to the heuristic OPT-Assign-LOOP. Subsection 8.2.4 compares the code size of the loop assignments.

8.2.1 TMX320C44 Overview

Figure 9 shows a simplified view of the TMX320C44. In this architecture, there are three register files: Extended Precision Registers which are 40-bits wide and used for floating-point and long integer arithmetic; Auxiliary Registers which are 32-bits wide and used as address pointers with dedicated address generation hardware to auto-increment and auto-decrement address values; and General-Purpose Registers which are 32-bits wide. All register files are connected to the Reg1 and Reg2 busses and available to the Multiplier and ALU. The Multiplier and ALU may both write to the Extended Precision Registers or one of them may write to either the Auxiliary Registers or the General Purpose Registers. Additionally, an operand may be supplied to the Multiplier or ALU by the memory.

8.2.2 Comparison of OPT-Assign-BB and OPT-Assign-LOOP

Table 5 contains our observed results and contains the number of spills per iteration for OPT-Assign-BB and OPT-Assign-LOOP, as well as the absolute percentage improvement of OPT-Assign-LOOP over OPT-Assign-BB measured as $\frac{Spills_{BB} - Spills_{Loop}}{Spills_{Loop}}$. As before, there is a general trend for the percentage improvement to increase as the number of registers increases due to OPT-Assign-LOOP's ability to naturally match the register usages at loop top and bottom. Upon inspection of the assignments produced it was noted in some cases that the assignments produced by the basic block scheme assigned an address variable to a general purpose register near the end of the iteration. This variable was heavily used at the top of the loop, so more spill code (spills of other address variables currently within the Auxiliary Registers) than necessary was generated to accommodate that variable.

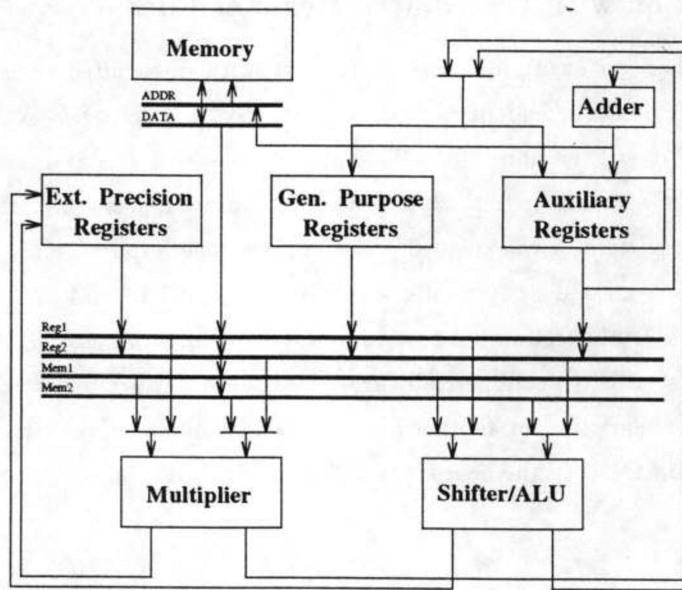


Figure 9: A simplified view of the TMX320C44.

Program	Number of Registers	OPT-Assign-BB Spills per Iteration	OPT-Assign-LOOP Spills per Iteration	% Improvement
2D-Hydrodynamics	2	9	7	29%
	4	2.7	2	35%
Inner Product	2	4	3	33%
	4	1.7	1	70%
Linear Equations	2	8	7	14%
	4	0	0	—
Tri-diag. Elim. (below diag.)	2	10	8	25%
	4	2.3	2	15%
Tri-diag. Elim. (above diag.)	2	11	10	10%
	4	3	2	50%
Prefix Sums (scan)	2	7	5	40%
	4	3	2	50%

Table 5: Comparison of basic block optimal and loop optimal for the TMX320C44.

Program	Number of Registers	OPT-Assign-LOOP Spills per Iteration	Heuristic OPT-Assign-LOOP Spills per Iteration	% Improvement
2D-Hydrodynamics	2	7	8	14%
	4	2	2	0%
Inner Product	2	3	4	33%
	4	1	2	50%
Linear Equations	2	7	8	14%
	4	0	1	—
Tri-diag. Elim. (below diag.)	2	8	8	0%
	4	2	2	0%
Tri-diag. Elim. (above diag.)	2	10	11	10%
	4	2	3	50%
Prefix Sums (scan)	2	4	4	0%
	4	2	3	50%

Table 6: Comparison of loop assignments for the TMX320C44.

8.2.3 Comparison of OPT-Assign-LOOP and Heuristic OPT-Assign-LOOP

Table 6 presents the results of the spill code produced by the optimal and heuristic algorithms in columns three and four, respectively. Column five contains the percentage withing optimal that the heuristic results are, measured as $\frac{Spills_{Heur} - SpillsOpt}{Spills_{Opt}}$. For this measure, lower numbers are better (i.e., the lower the number, the closer the heuristic is approximating the optimal. In a few cases (3 of 12), the heuristic produced results that are equal to the optimal. For 58% of the cases (7 of 12) results were produced that are within 15% of the optimal. For the rest of the cases (5 of 12), the percentage within optimal is higher, however, the actual difference in spill code produced is only one instruction.

8.2.4 Code Size of Loop Register Assignments

Again, because code size directly affects the size of the program ROM, we study the number of iterations produced by our loop assignments. Table 7 contains the number of iterations spanned by each of the loop methods for the given number of registers, as well as indication of whether the heuristic method produced an equal amount of spill code as the optimal. In the majority of cases (9 of 12) our heuristic derived assignments which spanned the same number of iterations as the optimal, and, of those, generated the same amount of spill code in three cases (33%). In the other cases (3 of 12), the heuristic assignments spanned one more iteration, producing the same amount of spill code in one case (33%). The overall range of the number of iterations spanned by the assignments is between two and four, which we believe is within acceptable limits.

Program	Number of Registers	# Iterations		Equal Spill Code?
		OPT-Assign-LOOP	Heuristic OPT-Assign-LOOP	
2D-Hydrodynamics	2	3	3	no
	4	2	2	yes
Inner Product	2	3	4	no
	4	3	3	no
Linear Equations	2	4	4	no
	4	3	4	no
Tri-diag. Elim. (below diag.)	2	3	3	yes
	4	2	2	yes
Tri-diag. Elim. (above diag.)	2	2	2	no
	4	2	2	no
Prefix Sums (scan)	2	2	3	yes
	4	2	2	no

Table 7: Comparison of loop assignment code sizes for the TMX320C44.

9 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. We have also extended our algorithm to assign registers to irregular datapaths, such as those found in many ASIPs, where some registers have specialized uses and/or registers are partitioned into multiple register files. Experimentation with our methods has demonstrated that heuristic methods obtain suitable performance, while out-performing the graph coloring-based approach used by the Gnu C compiler (gcc).

References

[1] I. Ahmed and C. Y. R. Chen. Post-processor for Datapath Synthesis Using Multiport Memories. *ACM/IEEE ICCAD*, 1991.

[2] A. H. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

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

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

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

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

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

[8] H. Feuerhahn. Data-Flow Driven Resource Allocation in a Retargetable Microcode Compiler. *Int. Symposium on Microarchitecture*, 1988.

[9] T. Granlund and R. Kenner. Eliminating Branches using a Superoptimizer and the GNU C Compiler. *Proceedings of SIGPLAN Conf. on Prog. Lang. Des. and Impl.*, 27(7), July 1992.

[10] 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.

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

[12] 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.

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

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

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

[16] 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.

[17] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Code Optimization Techniques for Embedded DSP Microprocessors. *32nd Design Automation Conference*, June 1995.

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

[19] F. Luccio. A Comment on Index Register Allocation. *Communications of the ACM*, 10(9), September 1967.

[20] P. Marwedel. The MIMOLA Design System: Tools for the Design of Digital Processors. *DAC-84*, June 1984.

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

[22] P. Marwedel and G. Goossens. *Code Generation for Embedded Processors.* Kluwer Academic Publishers, Norwell, MA, 1995.

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

[24] P. G. Paulin and J. P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. *IEEE Transactions on the Computer-Aided Design of Integrated Circuits and Systems*, 8(6), June 1989.

[25] R. M. Stallman. Using and Porting Gnu CC. *Free Software Foundation*, November 1992.

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

[27] M. Strik, J. van Meerbergen, A. Timmer, J. Jess, and S. Note. Efficient Code Generation for In-House DSP-Cores. *EDTC*, May 1995.

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