Post-Compaction Register Assignment in a Retargetable Compiler

Philip Sweany
Department of Computer Science

Steven Beaty
Department of Mechanical Engineering

Colorado State University
Fort Collins, Colorado, 80523

Abstract

We discuss graph-coloring register assignment in a retargetable compiler for Long-Instruction-Word architectures. Of specific concern is when, during the compilation process, should register assignment be performed. We conclude that, for best results, register assignment should follow compaction. We discuss methods of circumventing the implementation problems inherent in such late register assignment.

1 Introduction

Higher-level languages permitting abstraction from low-level timing and concurrency details are a major step toward alleviating the problem of mapping complex applications onto complex architectures. However, this approach is useful only if compilers for these languages can produce high-quality code and can be reliably targeted to new machines with modest effort. The ROCKET compiler, a retargetable C compiler for Long-Instruction-Word (LIW) and pipelined architectures, strives to meet these criteria.

A high-level language compiler must include mappings from programmer-defined operations and resources to actual operations and resources existing on the target architecture. This process includes supplying multiplexers with data from a bus, invoking the proper ALU operation, and, *register assignment*, mapping user-defined and compiler-generated variables to register resources.

While Sethi has shown optimal register assignment computationally intractable [Set75], good heuristics exist which produce near-optimal results in reasonable time. One popular method, developed by Chaitan, [Cha82, CAC+81] initially assumes an infinite number of available "symbolic" registers, allocates each scalar value to a distinct symbolic register, and later maps the symbolic registers to the finite target architecture register set using a graph coloring heuristic. We have included graph-coloring register assignment in the ROCKET implementation because of its proven effectiveness in other compilers and because its basis in well-founded mathematical principles allows easy retargetability from one architecture to another.

Having decided to use graph-coloring register assignment, we must determine when, during compilation, register assignment should be performed. This becomes an issue in the ROCKET compiler because, unlike most compilers using graph-coloring register assignment, ROCKET also includes instruction scheduling (compaction) to take advantage of increased parallelism available in LIW architectures. At least two other compilers [All88, MDSW88] include both compaction and graph-coloring register assignment, but both assign registers before compaction. We believe delaying register assignment until after compaction leads to more efficient code generation.

The remainder of this paper overviews the ROCKET compiler, describes the graph-coloring register assignment technique, provides motivation for delaying register assignment until after compaction and discusses implementation issues involved in post-compaction register assignment.

2 The ROCKET Compiler

ROCKET is an offshoot of the Horizon compiler [MDSW88], also developed at Colorado State University. Like Horizon, ROCKET focuses on machine resource usage as the primary issue in both retargetability and production of highly-optimized assembly code. ROCKET targets to a wide variety of architectures which are assumed to have a single control store and to operate synchronously (such machines correspond to what others have called "clocked micro-architectures" [Das84, DDMS86]). Beyond that, they may have arbitrarily wide control words; polyphase or monophase execution; pipelined fetch/execute; pipelined functional execution; permanent or transient storage elements with arbitrary (discrete) setup and hold times; machine operations with side-effects; and branches with arbitrary (discrete) branch delays.

2.1 Code Generation Phases

To translate C into highly-optimized code for LIW architectures, ROCKET first produces an abstract representation of an input C program and then performs *global optimization* which massages the intermediate representation to improve expected program speed; *code selection*, which replaces abstract representations of C statements with collections of machine operations; *parallelization*, which determines resource dependencies and timing; *local compaction*, which assigns machine operations to (a hopefully minimum number of) instructions to satisfy data-dependency and machine-resource constraints; and *register assignment*, which replaces symbolic references with specific machine register references.

ROCKET's global optimization includes common subexpression elimination, copy propagation, constant folding, constant propagation, algebraic simplification, induction variable simplification, and reduction in strength. Aho, et al. [ASU86]. describe these "traditional" compiler optimizations. Unfortunately, although many traditional optimizations, produce improved code efficiency on vertical architectures, they do not always do so on LIW machines. We attribute this difference mainly to the need to take advantage of the available concurrency in an LIW architecture during compaction. Often, traditional optimizations (such as common subexpression elimination, copy propagation, and constant folding) perturb code in a manner that inhibits compaction's ability to minimize the number of instructions required to execute a program. Rather than disregard these traditional optimizations, ROCKET includes modified versions which consider parameters in the target machine description to evaluate when an optimization is likely to provide improved code. Beaty et al. discusses a similar optimization scheme in [BDM+88].

After global optimization, code selection, in conjunction with a *serial-parallel coupler/decoupler*, builds a data-dependency graph (DDG) for each basic block in the abstract intermediate representation of the C program. A DDG's nodes represent target machine operations. A directed edge from node *x* to node *y* indicates that *x* must occur no later than *y* in the final (compacted) code generated for the DDG. In ROCKET's DDGs each edge from *x* to *y* is labeled with a pair of non-negative integers (min,max) indicating that *y* can execute no sooner than min cycles after *x* and no later than max cycles after *x*. For example, if *x* places a value on a bus that *y* reads, an edge from *x* to *y* establishs a "data dependency" with timing (0,0) (indicating that the read can and must follow the write in the same cycle). In contrast, if *x* assigns a value to a register subsequently read by *y* and the target machine does not permit reading a register after it is written in the same cycle, the edge from *x* to *y* would include timing $(1,\infty)$. In addition to nodes and edges DDGs contain analysis information (described below) in an easily accessed and manipulated form. The coupler/decoupler, originally implemented in the Horizon compiler [MDSW88, MS86], combines and manipulates DDGs, performing ROCKET's parallelization phase in the process.

Local compaction uses DDGs built by code selection. Given a data-dependency graph, compaction schedules nodes into the shortest sequence of instructions subject to (1) the graph's constraints, (2) the machine's resource dependencies (i.e., a machine resource can typically hold only a single value at any instant of time), and (3) the field-encoding conflicts that may exist should several operations share a common instruction field. In general, this optimization problem is NP-complete [Rob79]. However, in practice, heuristics can achieve excellent results. Landskov et al. provides a good survey of early code compaction algorithms [LDSM80], while Vegdahl [Veg82] and Allan [All86] give more sophisticated algorithms. Our compaction resembles Allan's.

Register assignment in the ROCKET compiler uses a graph coloring method which we detail in Section 3.

2.2 Analysis

In all the code generation phases mentioned above, information about the properties and use of both program variables and machine resources is paramount. Thus, ROCKET relies heavily on extensive analysis of the program being translated. High-level resource usage (*used* and *defined* as well as *live-in* and *live-out* sets) is easily and efficiently obtained through standard control and dataflow analysis (Aho et al. [ASU86] or Hecht [Hec77].)

Another analysis important for generating efficient compacted code is accurately determinating data dependencies among a program's operations. Data-dependence concepts and standard terminology are widely discussed in the literature [BSKT79, PKL80, Veg82, PW86, Ban88]. The three basic types of data dependence are:

- *Flow Dependence* — sometimes called true dependence or data dependence. An operation m_2 is flow dependent on operation m_1 if m_1 executes before m_2 and m_1 writes to some memory location read by m_2.
- *Anti-Dependence* — sometimes called false dependence. An operation m_2 is anti-dependent on operation m_1 if m_1 executes before m_2 and m_2 writes to some memory location read by m_1, thereby destroying the value read by m_1.
- *Output Dependence* An operation m_2 is output dependent on operation m_1 if m_1 executes before m_2 and m_2 and m_1 both write to the same location.

As discussed, compaction involves "scheduling" the DDG for each basic block in a program's control-flow graph. Since a DDG's edges (representing dependencies between operations) inhibit parallelism, we wish to build the graph with as few edges as possible while maintaining program semantics. Our difficulty is determining exactly which data dependencies are necessary to maintain program semantics. When, after available analysis, a compiler cannot determine whether two references access the same memory location, it generally must make the conservative assumption that they might. Because making conservative data dependency assumptions has a potentially devastating effect upon optimization opportunities, considerable research effort has gone into analysis techniques which accurately determine whether two operands can possibly access the same memory location. These methods are generally referred to as memory reference disambiguation [Nic84, Ell85, Ban88, BC86].

2.3 Machine Description

We patterned the ROCKET machine description files after the Horizon machine description [MDSW88]. The ad hoc machine description language indicates how data passes among storage resources, and how data is transformed by functional resources, as well as specifying resource timing data, and field attributes of the instruction word.

A ROCKET target machine description includes:

- **Machine Resources** — Each machine resource has an associated setup and hold time. ROCKET views this timing abstractly in terms of instruction cycles.
- **Target Machine Instruction Word Fields** — The compiler assumes a number of distinct encoding fields comprise the instruction word. A field's encoding values are symbolically represented in the specifications of target machine operations.
- **Target Machine Operations** — Each machine operation has a distinct identifier as well as a collection of instruction fields that invoke it during execution.
- **Target Machine Data Paths** — The code selector must extract information about how data can be moved among storage resources, how functional operations are invoked, and how instruction sequencing is invoked. This information forms the bulk of ROCKET'S machine description.

3 Register Assignment

The graph coloring method proposed by Chaitan [CAC+81, Cha82] performs register allocation and assignment at the same time. Allocation decides which variables will reside in physical (hard) registers at run time; assignment places those variables into their respective hard registers. Register assignment aims to place as many program variables into hard registers as possible. This increases the program's execution speed and reduces the code size, both desirable effects. Variables that are live at the same time can not share a hard register. A representation of and solution to this problem could greatly aid the final code's usability. The graph coloring view denotes variables as graph nodes and places arcs between nodes where those variables are live simultaneously. The solution then involves finding an n coloring of the graph, where n represents the number of the target machine's available hard registers. A graph is considered correctly colored if each node's color differs from each neighbor's. As an architectural paradigm, this implies each variable is assigned a register different from all other variables live during the same execution cycles.

It is well known that given a graph G and a natural number $n > 2$, the problem of determining whether G is n-colorable is NP-complete [HS80] There exist graphs that have a search space that grows exponentially based upon the number of nodes. An exhaustive method could randomly choose a color, from a pool of n, and assign it to a node. This process could continue until the graph is colored or adjacent nodes received the same color. If adjacent nodes received the same color, the process would backtrack in search of a different, correct, solution. This is a lengthly process.

Fortunately, criteria can be added to this process to speed the search for a solution. A node has a metric of urgency with the order of assignment based on each node's urgency. This speeds the search in graphs that are, in some sense, easy to color (non-pathological graphs.) Unfortunately, solutions based on urgency criteria will still create exponential search spaces in the worst case. There is no way to easily predict the onset of exponential search, leaving this method with certain flaws.

Another approach is possible with a simple observation about coloring graphs. If we remove a node from the graph that has degree less than n, no matter how its neighbors are colored, there will be at least one color left over for it. For example, in a graph where a four-coloring is being attempted, if a node of degree three is removed, each of its neighbors may be assigned any three colors leaving at least one color for the removed node. Nodes are removed in this manner until the graph is empty or no remaining nodes have degree less than n. This deterministic routine consumes non-exponential time and space. It is not guaranteed to find n-coloring if one exists; however, it does produce excellent results in practice. Once a symbolic register-interference graph has been shown to be n-colorable, nodes are colored

by determining which nodes they interfere with, in inverse order of removal. A hard register (color) not used by any of a node's neighbors is chosen for it. Two methods for choosing which hard register to assign next are

- pick the lowest numbered register not used by neighbors
- pick registers in a round-robin fashion. In this method, we would keep track of the last hard register allocated and allocate the next higher numbered register (mod the number of colorable registers, of course) not already allocated to a neighbor in the interference graph.

The first method reduces the overall number of hard registers used, possibly reducing the number of save and restores around call sites. The second increases the number of hard registers used, thereby reducing the number of anti-dependcies required, possibly, creating a more advantageous environment for compaction (see Section 5).

When a graph is not n-colorable, some program variables must be placed in a non-register resource. This resource is usually an off-chip read/write memory. This creates a definite speed penalty so variables must be chosen carefully for these locations. Code must be generated to "spill" the variable out after each of its definitions and to "spill in" before each use [ASU86]. This reduces the pressure on the internal register bank by reducing the length of any spilled variable's live tracks. This should reduce the interference caused by variables spilled, allowing the graph to be completely colored. Because the interference may not be reduced enough and hard registers are usually still temporarily needed for spilled variables, a new graph is built and the coloring/spilling cycle repeated as necessary. Variables are chosen to be spilled based on perceived cost. Those within nested loops or with a high number of uses will be spilled last.

When a program executes a call statement, values residing in hard registers may be destroyed by the called routine. A convention may be adopted that specifies which hard registers may be overwritten by a subroutine but this practice removes resources that could otherwise be well used. An alternative convention saves the values needed before they are overwritten and restores them when control is returned to the calling routine. Using this procedure, two choices exist: 1) the caller can save the values before transfer of execution and restore them upon return or, 2) the callee can save them immediately upon entrance and restore before exit. The caller-save method saves all those hard registers that are live before and after the call. On the other hand, the callee-save method saves any register that will be used during subroutine execution. The callee-save method has the benefit that the save/restore code is present only once whereas the caller-save has to place code around each call. A refinement to this process saves hard registers only before they are used within the subroutine and also restores only those. Control flow constructs within the subroutine may cause certain registers not to be destroyed every time it is called. The caller-save will save fewer registers if there are fewer registers live before and after the call in the calling routine than exist within the subroutine. The inverse is true for save by callee. We have chosen save by callee for its code-space reduction and its implementation simplicity.

4 Building the Interference Graph

To perform graph-coloring register assignment, we need to build a register-interference graph. Fortunately, the ROCKET's method works equally well with either the program's intermediate representation or final compacted form as input. Whether to do register assignment before or after compaction depends exclusively on other factors.

As mentioned, two symbolic registers interfere if they are live at the same time. This implies that they must reside in different hard registers at run time. What we need to know then, is which symbolic registers have intersecting lifetimes.

Before further describing how ROCKET builds the register-interference graph, however, we digress briefly to discuss *execution points*. Intuitively, an execution point can be thought of as a distinct step in the program's execution. Thus, the definition of an execution point differs slightly, depending on the program representation being considered. In C source code, each ";" might be considered an execution point. Program analysis (before compaction), considers each intermediate statement a distinct execution point. After compaction, each compacted instruction is an execution point.

Given this malleable definition, we determine which variables have intersecting lifetimes using two dataflow sets that the ROCKET compiler maintains for each execution point.

defined The set of symbolic registers redefined during the execution of this point.

live The set of symbolic registers which "contain" values needed at this or some later execution point.

We build the interference graph using the following observation:

> A symbolic register, R, interferes only with those symbolic registers which are live at the execution point(s) where R is defined.

Stating this notion in algorithmic form, we do the following to build the interference graph.

```
foreach execution point, P, in the program:

   foreach member, defR, of P->defined:
      foreach member, liveR, of P->live:

         add an arc between defR and liveR
```

All we require to build the register-interference graph is to compute *defined* and *live* sets for each program execution point and to maintain the ability to traverse those execution points. Since the ROCKET compiler maintains the *defined* and *live* sets at each execution point in both the intermediate and compacted representations of the program, it is equally easy to build the interference graph either before compaction (using intermediate statements), or after compaction (using compacted instructions).

5 Timing of Register Assignment

Having decided to perform register assignment using graph coloring, the important question we need to answer is when, during compilation, should register assignment take place.

In one sense, we would like register assignment to be done very late in the compilation process. Then we can maintain the myth of unlimited register resources until after optimizations (such as common subexpression elimination, copy propagation, and dead code removal.) Delaying register assignment provides several benefits. If an optimization calls for creation of a new register, (or expansion of a register variable's lifetime), we can perform the optimization content in the knowledge that a later register assignment will "make everything right" with respect to the allocation and assignment of register values needed. Similarly, if an optimization (e.g. dead code removal) removes the need for a register or shortens a register variable's lifetime, we can be sure that the lowered register interference will be noticed at register assignment time. Thus, the basic rationale for performing register assignment late: we wish to assign values to hard registers only after any optimizations that may change either the number of register values needed or those values' lifetimes. If we assign registers before one or more of these optimizations, we base assignment and spilling decisions on poor estimates of the register usage in the compiler's final product.

By delaying register assignment we encounter some difficulties, however. We cannot entirely ignore that no target architecture has an infinite number of registers. Common optimizations must consider the consequences of adding to a program's register interference. Indeed, if, to eliminate a common subexpression, we increase register interference to the point that register assignment spills, the cost of evaluating an expression multiple times must be great to outweigh the added spill cost. Thus we have a phase-coupling problem where intermediate optimizations and register assignment depend on one another. A common solution, discussed in Beaty et al. [BDM+88] and used in ROCKET, performs register assignment after common optimizations such as common subexpression elimination but includes a "register interference" parameter in the target machine description. This parameter estimates the probability that creation of a new register value would lead to register spilling.

So far we have discussed timing of graph-coloring register assignment only with respect to traditional optimizations. How does inclusion of a compaction phase affect the optimal placement of register assignment? While compaction itself will not create or destroy register values, it will most certainly alter the lifetimes of register values by changing the relative order of operations in intermediate code. Therefore, to use the most accurate dataflow information we delay register assignment until after the compiler's compaction phase. Interestingly, to the best of our knowledge, of the compilers which include both compaction and graph-coloring register assignment, only ROCKET postpones register assignment until after compaction. We assume this is due to implementation difficulties encountered, specifically

- in those cases requiring spilling, it is not clear how to add spill code to already compacted instructions.
- code for save/restoration of registers needs to be compacted. But this causes a phase-coupling problem, because, until after register assignment, we don't know how many registers need to be saved/restored.

We address each of these problems shortly, but first, we provide added motivation for our contention that performing register assignment before compaction leads to poorer compacted code. As stated, local compaction takes a DDG as input and attempts to schedule the DDG's operations in as few instructions as possible, subject to the DDG's data dependencies. When we map symbolic registers to hard registers before compaction, we must add anti-dependencies associated with reuse of hard registers. These dependencies are in a sense unnecessary since they do not represent actual dataflow of the program being compiled. Necessary or not, however, they restrict the possible movement of operations during compaction and, thus, reduce the likelihood of obtaining the minimal number of instructions.

To see how early register assignment can lead to unnecessary anti-dependencies, let us look at an example. Consider a hypothetical machine H, that provides both a floating point multiplier and a floating point adder, each a five-stage pipeline. Both the inputs and output of each pipe must be one of a bank of floating point registers (F0-F15). In addition, H includes 64k of RAM and an integer processor (including registers I0-I15) to compute RAM addresses. Figure 1 depicts our hypothetical architecture. Assembly language for H includes the following simple operations:

- $Ix = addr(A)$ — store the RAM address symbolically represented by A into an integer register.
- $Ix = Iy$ op Iz — arithmetic in the integer alu.
- $Fx = mem(Iy)$ — move a RAM value into a floating point register.
- $mem(Iy) = Fx$ — move a floating point register value into a RAM location.
- $Fx = Fy + Fz$ — floating point addition.
- $Fx = Fy * Fz$ — floating point multiplication.

In any one machine cycle of H we can concurrently perform the following:

- either initialize an integer register to a RAM address or perform a calculation in the integer alu.
- start data movement either from RAM to a floating point register or from a register to a RAM location. Such data movements are pipelined and take two cycles to complete.
- start a floating point addition. This is a pipelined computation and requires 5 cycles to complete.
- start a floating point multiplication. This is a pipelined computation and requires 5 cycles to complete.

As an example of code in which register assignment prior to compaction can lead to relatively poor compacted assembly language, consider the following C code fragment:

```
float sum, product;
float A[100], B[100];
   .
   .
   .
sum += A[i];
product *= B[i];
   .
   .
```

Bearing in mind H's architecture and our desire to have all scalars reside in registers, we might well expect to allocate both sum and product to (symbolic) floating point registers, but certainly the arrays A and B must be maintained in RAM. Since H's floating point adder and multiplier each require both inputs to be in registers, the intermediate code might look like (assuming t1 and t2 to be allocated to floating point registers):

```
      .
      .
      .
   t1 = A[i];
   sum += t1;
   t2 = B[i];
   product *= t2;
      .
      .
      .
```

Notice that the temporary "variables" t1 and t2 have disjoint lifetimes (i.e. t1's value is referenced for the last time before t2's value is set.) Thus, if we perform register assignment on this intermediate code, no reason exists why t1 and t2 should not be mapped to the same hard register. If that happens, however, we get the assembly code listed below. (For readability, nops are included to make the timing delays obvious. The assembly language statements are numbered only to facilitate this discussion.)

```
/*  assume that

  I0 holds the address of A
  I1 holds the address of B
  I2 is a temporary

  and the following register
  mappings have been made

  i -> I3
  sum -> F2
  product -> F3
  t1 -> F1
  t2 -> F1
*/
        .
```

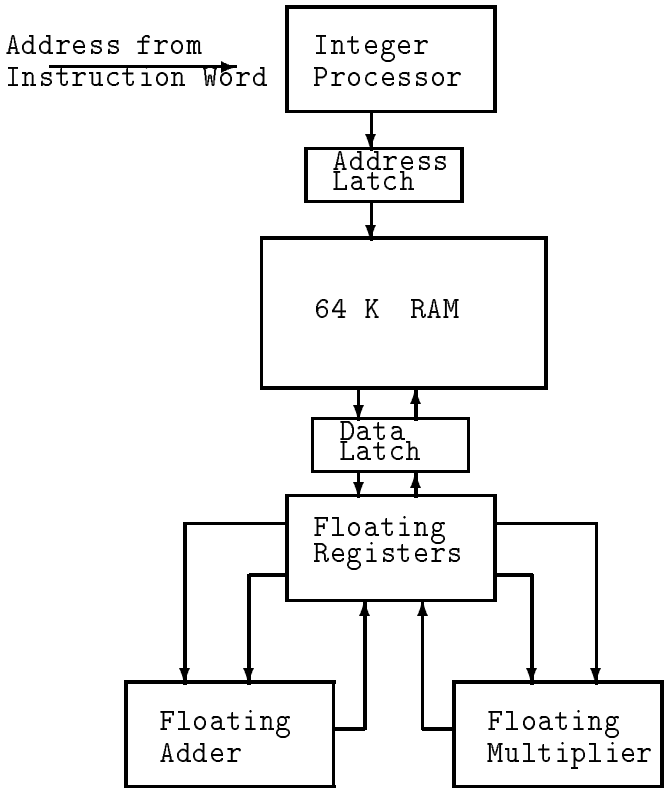

Figure 1: Hypothetical Architecture, H

```
        .
        .
(1)     I2 = I0 + I3;
(2)     F1 = mem(I2);
(3)     nop;
(4)     F2 = F2 + F1  &
             I2 = I1 + I3;
(5)     F1 = mem(I2);
(6)     nop;
(7)     F3 = F3 * F1;
(8)     nop;
(9)     nop;
(10)    nop;
(11)    nop;
        .
        .
        .
```

Counting the nop's needed to satisfy the pipeline timing requirements, this code requires eleven H instructions to execute. Assigning both variables t1 and t2 hard register F1 causes an unnecessary anti-dependency between statements (4) and (5). Because of this anti-dependency, compaction could not move the operation of statement (5) above the floating point addition in statement (4). If we wait until after compaction to assign hard registers, no such anti-dependency exists and we get the following compacted code (before register assignment):

```
/*  assume that
    Isym119 holds the address of A
    Isym117 holds the address of B
    Isym13 and Isym14 are temporaries.

    and the following symbolic register

    mappings have been made

    i -> Isym133
    sum -> Fsym107
    product -> Fsym93
    t1 -> Fsym83
```

```
     t2 -> Fsym105
   */
      .
      .
      .

(1)   Isym13 = Isym119 + Isym133;
(2)   Fsym83 = mem(Isym13)   &
          Isym14 = Isym117 + Isym133;
(3)   Fsym105 = mem(Isym14);
(4)   Fsym107 = Fsym107 + Fsym83;
(5)   Fsym93 = Fsym93 * Fsym105;
(6)   nop;
(7)   nop;
(8)   nop;
(9)   nop;
      .
      .
      .
```

In this simple example, by delaying register assignment until after compaction, we get code which executes in two less instructions. It should not be hard to imagine larger examples in which multiple unnecessary anti-dependencies could have a much more serious effect.

Having shown why it is useful to delay register assignment until after compaction, we return to the implementation issues mentioned previously, specifically the difficulty in spilling registers in compacted instructions and the uncertainties in saving and restoring registers around procedure calls.

One of the nice consequences of performing register assignment based upon intermediate statements is the ease with which we can insert spill code into the intermediate code. We need only (1) construct new intermediate statements to represent the actions of storing a register's value to memory and later loading a register from that memory location and (2) insert the new statements into the list of intermediate statements for the program being compiled. Sadly, such insertion will not generally work with compacted code. The compacted code probably will contain machine operations packed into different instructions but bound by restrictive timing. Arbitrarily inserting new instructions into the compacted code could violate the constraints under which the code was originally compacted.

Well, if we can't insert spill code into already-compacted instructions, we might somehow add spill code to the DDG from which we built the compacted code. While possible, this would be extremely messy. We have chosen an alternative. Rather than add spill code to either the compacted instructions or the DDG, we use the compacted instructions to choose candidates for spilling, but add spill code to the intermediate statements. With this approach we can easily insert spill code, and we base spill decisions on register usage in compacted code. The method's major disadvantage is compilation speed, or rather the lack of it. Recall from Section 3 that register assignment is actually a loop in which we assign, spill, assign, spill, ... until the assign phase can allocate all the (remaining) symbolic registers to hard registers. This can, of course, be a time-consuming process. By assigning registers after compaction but spilling in the intermediates, we have lengthed the loop. Now, in addition to adding spill code for each iteration of the assign/spill loop, we must also build DDGs from the intermediate and compact the DDGs for each iteration.

Delaying register assignment until after compaction may also lead to implementation problems in saving and restoring registers around each procedure call. What is the difficulty? We do not know how many registers we must save/restore until after register assignment. But we also want to compact the save/restore code. Thus, save/restore code must precede compaction but follow register assignment, playing havoc with our decision to compact before assigning symbolic registers. Luckily, our decision to include save/restore code in the callee rather than caller subprogram leads to an easy solution. A crucial observation in reaching this solution is that register save/restore code cannot itself reference symbolic registers. Therefore, register assignment can ignore the register save/restore code. To insert register save/restore code in a function, ROCKET inserts two dummy basic blocks, ENTRY and EXIT, into the function's control-flow graph. As the names imply, these blocks will be the first and last executed. Through most compilation these blocks contain no code. After compaction and register assignment, however, ROCKET inserts register save code into the ENTRY block and register restore code into the EXIT block, saving/restoring those hard registers referenced within the function. Finally, the ENTRY and EXIT blocks are compacted. We can neglect register assignment for these two blocks because they can contain no symbolic registers.

6 Conclusions

We have argued that the relative timing of compaction and register assignment has an impact upon the quality of code produced. If assignment occurs before compaction, the anti-dependence constraints can be more restrictive than necessary, limiting compaction's opportunities to exploit parallelism. If assignment occurs after compaction, these restrictions do not exist.

We have also addressed implementation difficulties which have previously discouraged late register assign-

ment. Using the method proposed, register spilling, when it occurs, requires a compaction/assignment loop, leading to increased compilation times relative to those achieved with earlier register assignment. We feel this slower compilation is acceptable since post-compaction register assignment can increase the quality of the resultant code, and spilling should be a rare occurrence. Finally, we have shown how callee save around call sites simplifies the register save and restore problem. Therefore, we conclude that post-compaction register assignment should be used when final code efficiency is paramount.

References

[All86] V.H. Allan. *A Critical Analysis of the Global Optimization Problem for Horizontal Microcode*. PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Colorado, 1986.

[All88] V.H. Allan. "Data Dependency Graph Bracing". In *Proceedings of the 21th Microprogramming Workshop (MICRO-21)*, San Diego, CA, December 1988.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[Ban88] U Banerjee. *Dependence Analysis for Supercomputing*. Kluwer, Boston, Ma, 1988.

[BC86] Michael Burke and Ron Cytron. "Interprocedural dependence analysis and parallelization". In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 162–175. SIGPLAN '86, 1986.

[BDM+88] S.J. Beaty, M.R. Duda, R.A. Mueller, P.H. Sweany, and J. Varghese. "Optimization issues for a retargetable optimizing microcode compiler". *IEEE MicroArch*, 3(1), December 1988.

[BSKT79] U. Banerjee, S. Shen, D.J. Kuck, and R.A. Towle. "Time and parallel processor bounds for fortran-like loops". *IEEE Transactions on Computers*, C-28(9):660–670, Sep 1979.

[CAC+81] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. "Register allocation via coloring". *Computer Languages*, 6, 1981.

[Cha82] G.J. Chaitin. "Register allocation and spilling via graph coloring". In *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, pages 201–207, June 1982.

[Das84] S. Dasgupta. "A Model of Clocked Micro-Architectures for Firmware Engineering and Design Automation Applications". In *Proceedings of the 17th Microprogramming Workshop (MICRO-17)*, pages 298–308, New Orleans, LA, November 1984.

[DDMS86] W. Damm, G. Doehmen, K. Merkel, and M. Sichelschmidt. "The AADL/S^* Approach to Firmware Design Specification". *IEEE Software*, 3(4):27–37, July 1986.

[Ell85] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, 1985. PhD thesis, Yale, 1984.

[Hec77] M.S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, New York, NY, 1977.

[HS80] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Potomac, MA, 1980.

[LDSM80] D. Landskov, S. Davidson, B.D. Shriver, and P.W. Mallett. "Local Microcode Compaction Techniques". *ACM Computing Surveys*, 12(3):261–294, September 1980.

[MDSW88] R.A. Mueller, M.R. Duda, P.H. Sweany, and J.S. Walicki. "Horizon: A Retargetable Compiler for Horizontal Micro-Architectures". *IEEE Transactions on Software Engineering (Special Section on Microprogramming)*, 14(5):575–583, May 1988.

[MS86] R.A. Mueller and P.H. Sweany. "Horizon Code Generator Series-Parallel DDG Coupler/Decoupler (Version 3.1)". Technical Report MAD-86-10, Firmware Engineering and Micro-Architecture Design Laboratory, Colorado State University, Fort Collins, CO, September 1986.

[Nic84] Alexandru Nicolau. *Parallelism, Memory Anti-aliasing, and Correctness Issues for a Trace Scheduling Compiler*. PhD thesis, Department of Computer Science, Yale University, New Haven, Conn, December 1984.

[PKL80] D.A. Padua, D.J. Kuck, and D.H. Lawrie. “High Speed Multiprocessors and Compilation Techniques”. *IEEE Transactions on Computers*, C-29(9):763–776, Sept 1980.

[PW86] D.A. Padua and M.J. Wolfe. “Advanced Compiler Optimizations for Supercomputers”. *Communications of the ACM*, 29(12):1184–1201, Dec 1986.

[Rob79] E.L. Robertson. “Microcode Bit Optimization is NP-complete”. *IEEE Transactions on Computers*, C-28(4):316–319, April 1979.

[Set75] R. Sethi. “Complete register allocation problems”. *SIAM Journal of Computing*, 4(3):226–248, 1975.

[Veg82] S.R. Vegdahl. *Local Code Generation and Compaction in Optimizing Microcode Compilers*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1982.