Building a Retargetable Compiler for Instruction-Level Parallel Architectures

Philip H. Sweany, Computer Science Department, Michigan Technological University, 1400 Townsend Drive, Houghton MI 49931-1295, (906) 487-2209, sweany@cs.mtu.edu

Steven J. Beaty, 3612 Chipperfield Court, Fort Collins, Colorado, 80525, (970) 204-0098 beaty@lance.colostate.edu

This work was partially supported by the National Science Foundation under grant CRR-9308348.

Keywords

Register Assignment, Instruction Scheduling, Instruction-Level Parallelism, Compilers

Abstract

Many modern architectures have instruction-level parallelism in order to increase the amount of computation performed during a given amount of time. While this is a noble goal, and can indeed increase performance, compilers that generate code for such architectures must take their peculiarities in account. We have built and extensively studied such a compiler and discuss what measures are necessary in order to generate efficient code for ILP architectures.

I. Introduction

Computer manufacturers are continually striving to make faster computers by a combination of faster circuitry and increasing the amount of simultaneous computation (parallelism) in their architectures. One popular method of increasing the degree of simultaneous computation is inclusion of instruction-level parallelism (ILP.) ILP computers exploit the implicit parallelism that most programs contain [63]. They overlap the execution of operations[1] that do not depend on one another. For example, a memory address can be found while the value to store there is computed. Today, typical ILP processors have a memory address, an integer, and several floating point computational units. Future processors will have more of each of these. Some ILP processors reorder operations based on the hardware knowing they depend on each other (usually called *Superscalar* processors). Some (usually called *(V)LIW* processors) do not reorder the instruction stream but instead rely on the instructions themselves to express parallelism. Newer processors have hardware support for speculative execution. Unlike most traditional multiprocessors that usually require explicit parallelism, ILP exploitation does not require users to rewrite programs to use the potential parallelism, relying instead upon compilation techniques to effectively map a program's ILP to the parallelism available for the target architecture.

While high-performance architectures have included some ILP for at least 25 years [63], recent computer designs have exploited ILP to a larger degree. This trend shows no sign of reversing. Effective use of ILP hardware requires that the instruction stream be ordered such that, whenever possible, multiple low-level operations can be in execution simultaneously. This ordering of machine operations to effectively use an ILP architecture's parallelism is typically called *instruction*

[1]We define an *operation* as an atomic computational function, such as an add, multiply, or memory access. An *instruction* is an abstract representation of the operations that can be issued during a single machine cycle. An instruction might contain more than one operation, and the operation(s) may not be performed in the order they are presented in the instruction.

scheduling.

We have long been interested in compilation techniques appropriate for ILP architectures and have built a highly-optimizing compiler (Rocket) that is retargetable to a wide range of ILP architectures in order to facilitate our study of such issues. This paper provides an introduction to the problems faced when writing a compiler for ILP architectures and focuses on problems we have faced over the years when producing compilers for a variety of ILP machines. It also describes our solutions to those problems so that others can learn from our experience.

In order to make our compiler as widely applicable as possible, we have striven toward two major goals within Rocket:

1. architecture independence, and
2. language independence[2].

These two goals add substantial complexity when creating a compiler, notably:

- the need to modify traditional optimizations,
- the need to have machine-independent:
  - code selection,
  - register assignment,
  - instruction scheduling,
- the need to have a flexible way of representing varied architectures.

We have created a number of novel techniques that effectively deal with these complexities. With our techniques, we have been able to quickly target our compiler to numerous architectures. We currently accept two languages as input[3], and can add others easily due to a flexible intermediate language and the relative lack of reliance we place on the parser.

This paper focuses first on the general problems that a compiler for ILP machines must attack. Then we shift focus slightly and look at how Rocket addresses each of these problems. More specifically we first discuss the type of architectures we have spent the most time analyzing (Section II.) These include many of today's most popular processors. In Section III-A we explore a number of different compiler optimizations and discuss why one cannot be naîve about applying them to ILP architectures. As instruction scheduling remains the heart of the optimizations available for ILP machines, we devote Section IV-B.5 to discussing instruction scheduling methods. Register

[2] within an imperative framework

[3] C, and FORTRAN

assignment also plays a crucial role in producing efficient ILP code, so we address that at length (in Section V-B.) We then leave overview of problems generic to ILP compilers, and dive into discussion of our Rocket compiler (Section VI-E), focusing on how we have attempted to address a number of the difficulties mentioned in the previous sections. We first list Rocket's code-generation phases and their important interactions, before turning attention to a detailed discussion of our chosen machine model and how that is used to describe a wide variety of architectures. We next discuss how we can generate code in a machine-independent manner. Then the Rocket instruction scheduler is discussed, both in terms of local and global scheduling methods. Finally we consider the Rocket's register assignment methods.

II. Machine Model

Before we can talk meaningfully about ILP compilation techniques, we need to identify the class of architectures for which such techniques are designed. Of course, we wish the model to be as general as possible to include as many ILP architectures as we can. Rocket's model focuses on machine resource usage as the primary issue in both retargetability and instruction scheduling for ILP architectures. The architectures are assumed to have a single control store and to operate synchronously. Beyond that, they may have arbitrarily wide instruction formats, pipelined functional execution, permanent and transient storage elements with arbitrary (discrete) setup and hold times, and branches with arbitrary (discrete) branch delays. This encompasses a broad range of ILP architectures. This section takes a brief look at some popular classes of ILP architectures. Later (Section VI-E) we look at how Rocket models the different architectural features mentioned here. A more complete discussion of different forms of ILP architectures can be found in Hennessy and Patterson's architecture text [44].

A key feature of ILP architectures is their ability to execute multiple machine operations during a single machine execution cycle. An architecture can provide this parallelism in several ways.

Long-Instruction-Word (**LIW**) computers are characterized by a wide instruction word that includes separate fields for each of several operations that can be initiated during an instruction execution. Instruction scheduling for LIW architectures consists of encoding the instruction fields to specify the machine operations to be performed during that instruction's execution. LIW architectures execute a single instruction during each machine cycle. To account for delays that

occur when no new operation can be executed, *NOP*s are added to the instruction stream. Examples of LIW computers include Pixar's Chap [54] and the ESIG-1000 from Evans and Sutherland [34].

Superscalar computers add instruction-level parallelism by retaining a simple (and short) instruction format but by allowing multiple instructions to begin execution during a single execution cycle. Examples of superscalar computers include the IBM RISC System/6000 (RS6000) [13], the Intel i860 [49], and the Intergraph Clipper [59]. Typically, superscalar architectures include additional hardware to determine at run time whether two or more instructions can be executed simultaneously. Thus, instruction scheduling for superscalar architectures involves ordering simple instructions to reduce dependences between adjacent instructions and, thus, maximize the ability to execute multiple instructions simultaneously. It has sometimes been held that compilers for superscalar architectures need not perform instruction scheduling, since the hardware reorders operations at execution time. Bernstein et al. [19] have shown, however, that even simple instruction scheduling by the compiler improves execution times for an RS6000 by better than 5% for a large sample of programs, with some programs exhibiting as much as 23% improvement in execution time. More significant gains would be expected if either more sophisticated scheduling techniques were used or a superscalar architecture with more ILP were tested.

Very-Long-Instruction-Word (**VLIW**) computers differ from LIW in kind as well as size. Their long instruction word (on the order of 1000-2000 bits for current machines) is a result of replicated simple processors whose instruction fields are concatenated into a single instruction word. As in LIW computers, instruction scheduling consists of specifying as many fields as possible for each instruction and a single instruction is executed during each machine cycle. The Multiflow TRACE series of computers [30] are VLIW computers, as are computers being built at IBM, described in [31].

SuperPipelined computers are defined by Jouppi and Wall [50] to be computers in which the machine cycle time is shorter than the latency of any functional unit. In such machines, multiple operations are overlapped because different operations can be at different stages of computation during the same cycle. Instruction scheduling for superpipelined architectures consists of ordering operations to minimize delays occurring when a needed operand is not yet available. The Cray architectures are examples of superpipelined machines.

Even though superpipelined machines are considered a separate class of ILP architecture, pipelining itself has been a common feature of architectures for a long time and is included in all high-

performance modern computers. Superscalar architectures like the RS6000 and i860 include pipelines for floating point operations, as do VLIW and LIW machines. When considering real architectures, a compiler needs to combine scheduling of pipelines with whatever other features provide for instruction-level parallelism.

III. Machine Dependence of Optimizations

Historically, enormous effort has been directed towards refining compilation techniques for diverse classes of computer architectures. This has resulted in a rich body of possible optimization methods, though not all are equally useful in every architectural context. This section investigates those compiler optimization techniques applicable to a highly-optimizing retargetable compiler for ILP architectures and addresses the degree of architectural independence they exhibit over the broad class of ILP computers.

A. Classical Optimizations

Since we wish to build a highly-optimizing retargetable compiler for ILP architectures, we might look first at well-known optimizations for more traditional machines, such as CISC and RISC architectures. Allen and Cocke [12], Aho et al. [1] and Chow [28] provide excellent summaries of these traditional techniques. In the realm of microprocessors and other non-ILP architectures, most of these optimizations (such as common subexpression elimination, copy propagation, and constant folding) are considered machine-independent optimizations. Are these techniques useful for ILP architectures as well as for the more conventional computers? If suited to ILP architectures, are they as machine-independent in this realm as for non-ILP machines?

Unfortunately, many traditional optimizations that produce improved code efficiency on non-ILP architectures do not always result in improved code on ILP machines. We can attribute this mainly to the need for instruction scheduling to take advantage of the available concurrency in an ILP architecture. It is often the case that these traditional optimizations perturb code in a manner that inhibits instruction scheduling's ability to minimize the number of machine cycles required to execute a program. Of course, this is not a problem for non-ILP architectures since instruction scheduling is not an issue.

The following optimizations are all considered "standard" in most highly-optimizing compilers

for non-ILP architectures.

Loop invariant code hoisting moves code out of loops, into less frequently executed parts of a program. The prospect of executing code fewer times motivates hoisting loop invariant code, based upon the assumption that the loop will execute more than once. For conventional microprocessors with little concurrency, this provides a sufficient reason to move invariant code outside of loops. Consider an ILP architecture, however. Loop invariant operations might be overlapped with other (loop variant) operations in the loop during instruction scheduling. Thus, within the loop, loop invariant code may well be executed for "free" (not require any additional instructions). When hoisted out of the loop, however, the loop invariant operations might require one or more extra instructions. Clearly, this would not improve code efficiency but, rather, make it worse. Of course, it is also possible that, by moving the invariant code, the operations could overlap operations outside the loop when concurrency within the loop proves impossible. Thus, while loop invariant code hoisting almost always improves code efficiency for non-ILP architectures, it may not do so for ILPs. As an architecture's available concurrency increases, the chance of overlapping loop invariant operations increases and the perceived value of loop invariant hoisting decreases. In fact, for machines with a high degree of concurrency, moving code into a loop might prove worthwhile. Such code would, by necessity, be loop invariant. However, it is possible that by overlapping such code with operations of the loop, we could execute the operations with no added loop instructions. Although this "optimization" of code motion into a loop seems counter-intuitive, it could actually result in improved efficiency for some ILP architectures.

Copy propagation is the process of identifying simple copy statements of the form "f = g" and replacing subsequent uses of this definition of f with uses of g, until either f or g is redefined in the program. At first glance, this "optimization" does not seem to produce more efficient code. In fact, copy propagation can make the original copy statement useless, allowing the removal of the copy statement. Programmers do not typically make such useless assignments. However, such assignments are exactly the kind generated by other compiler optimizations (e.g., common subexpression elimination). As a short example consider the program segment:

$$x = y;$$
$$t = x;$$

if we perform the copy propagation optimization on this segment we get

$x = y$;

$t = y$;

Now, the assignment to x is not needed and we can remove it.

Consider the general problem in which there is a copy "x = y" followed by n uses of this definition of x. We'll define a "pre-copy propagation cost" C1, to be the sum of the execution time costs of the copy statement and each of the n statements containing uses of x. Similarly, we'll define a "post-copy propagation cost" C2 to be the sum of the execution time costs of each of the n statements containing uses of x, but with y substituted in place of each occurrence of x.

When x and y can be used interchangeably with identical cost, Cl is C2 plus the cost of the copy statement. Such a scenario may once have been typical, but is certainly not generally true for modern load-store computers, whether they include ILP or not. Consider, for example, the case where y represents a slow memory device while x is a fast register. The program initially requires only one use of the slow memory, followed by n uses of the fast register, while the "optimized" code contains n uses of the slow memory.

The most extreme (but not uncommon in the ILP world) case is where copy propagation yields statements that cannot be executed on the target processor. For example, if one of the uses of x occurs in a two-address statement (e.g., "x += z"), we would replace a two-address statement with a three-address statement (in the given example, the three-address statement would be "x = y + z"), but the target processor may not support three-address statements. While most recent ILP architectures avoid these difficulties by supporting three-address instructions, certainly not all do.

The point is that heuristic cost analysis should be integrated into a copy propagation optimization that resides in a retargetable compiler.

Common subexpression elimination identifies expressions that are computed several times and replaces all but one expression evaluation with a temporary variable reference. For example, in the code segment,

$x+ = y * 4$;

$w = y * 4$;

we could avoid one of the multiplications of y and 4 by creating a new temporary, t and changing the code to be

$t = y * 4;$

$x+ = t;$

$w = t;$

thus reducing the number of arithmetic computations required. This will usually produce more efficient code for a microprocessor, but not necessarily do so for an ILP architecture. The difference occurs not so much because instruction scheduling may be able to overlap the extra multiplication operation with other instructions, although this might well be true. The key here is that by creating a new temporary (presumably in a register), we may make it more difficult to perform register assignment. Indeed, this can be a problem for a conventional architecture as well, but the relative merits of trading an extra register for the benefit of removing an arithmetic operation could be quite different for an ILP architecture than a microprocessor.

Constant propagation replaces references to program variables with a constant when the compiler can determine the variable's value. Both constant propagation and constant folding (replacing an operation whose operands are constants with a constant) are common optimizations for microprocessor compilers. Vegdahl [74] gives numerous examples of how these optimizations can lead to poorer code for LIW architectures. LIW machines often include an "immediate" field in the instruction word to represent constants. However, LIW architectures also quite often include different formats for the instruction word (e.g. one that allows for a branch address, another that does not) and the immediate field is often not included in all formats (or if included, it might differ in size among different formats). Vegdahl shows examples in which moving a constant to a register can lead to more parallelism. He actually outlines several "constant unfolding" optimizations that can lead to more efficient code for some architectures.

Operator strength reduction replaces "expensive" operations (ones that require several machine cycles) with cheaper ones. Multiplication, for example, often requires multiple instruction cycles in both conventional architectures and ILP machines. Addition and shifting operations, in contrast, usually require fewer cycles. Often, a compiler can replace an integer multiplication with either a shift or an addition as in the examples below.

$x * 2$ can become $x + x$ and

$x * 16$ can become $x << 4$

Indeed, some microprocessors convert any multiplication by a constant to a sequence of shift

and addition operations. For an ILP architecture with pipelined multiplication, however, it is not as clear that we want such manipulation. With a pipeline available, the multiplication's net effect can be considered a single cycle (if we can keep the multiplication pipeline full), while conversion to shift and/or addition operations may merely use other needed computer resources (restricting further concurrency) while failing to improve the multiplication's efficiency. Thus, strength reduction on operators, for ILP architectures at least, depends heavily upon both the target architecture and the source program structure.

Induction variable elimination attempts to eliminate induction variables by rewriting each of the loop's induction variables (of which there may be several) as a function of a single induction variable. A loop induction variable changes by a constant amount during each iteration of a loop. Typically, a loop will be written, in source code, with a single induction variable and the compiler will add additional induction variables as temporaries to hold indices of arrays. For example, consider the following C code fragment:

```
for (i = 0; i < 100; i++)
        A[i] = 0;
```

The intermediate code generated for this loop might include the following

$$t1 = 4 * i$$
$$A[t1] = 0$$
$$i = i + 1$$
$$if(i < 100) goto B3:$$

where $t1$ is used as a temporary to compute the offset from the start of array, A. Strength reduction and a pre-loop assignment of i to $t1$ will allow the compiler to rewrite the first intermediate statement as

$$t1 = t1 + 4$$

which leaves the test ("$i < 100$") as the only place where i's value is required. But we can rewrite the test to use $t1$ as well ("if t1 $<$ 400"), thus eliminating the need for i within the loop. Such an optimization is always useful for conventional computers, but sometimes not for ILP processors. Consider the not uncommon practice (for LIW architectures, at least) of including special branch hardware that can, in one instruction, decrement (or increment) a special register by 1, and also test that register, branching to one location on zero and to another

for any other value. If an architecture includes such a special instruction, that is almost always more efficient than the more general branch instruction(s) also provided, we want the compiler to take full advantage of it. To do so, we need to have a loop induction variable that increments (decrements) by 1 each iteration. If, using induction variable elimination, we remove i, our code no longer fits the necessary pattern for the special branch instruction and we must use the slower, more general branching mechanism.

Dead code removal removes code that computes values that the program never subsequently uses. Basically, if an operation computes a result but that result is never used throughout the rest of the program, then we can remove the unnecessary (dead) computation. It may seem strange to think that a program might include dead code, but quite often once "useful" operations become dead code due to other optimizations. Unlike the other traditional optimizations discussed here, removing dead code can never lead to slower code and so we should remove dead code whenever the opportunity arises. Thus, a compiler retargetable for ILP architectures requires no special machine-dependent information to perform dead code removal.

Considering the uncertainty of so many traditional optimizations, what should we do? Ignore optimizations such as common subexpression elimination completely? Make all optimizations for an ILP compiler machine dependent? Beaty et al. [17] suggest that we should include many traditional optimizations in a retargetable ILP compiler in spite of the uncertainty. However, the compiler should be modified slightly to make use of simple machine-dependent parameters. For example, the relative "cost" (in machine cycles) of different operators is not at all uniform for ILP architectures (due to pipes, etc.). To perform operator strength reduction, a compiler should include, in the target machine description, a cost estimate for each operator supported. Using this cost data, the compiler can determine the expected benefit of strength reduction. Similarly, the expected benefits of performing common subexpression elimination can be computed on a case by case basis. Again, we need the cost of performing an operation. In addition, we need the cost of accessing each operand type (in this case, type refers to where operands are stored in the architecture.) Finally, since common subexpression elimination applies pressure to register assignment by adding register references, the machine description should include a "fudge factor" (small on a machine with lots of registers and large on an architecture with more limited register resources.) Common subexpressions are eliminated only when the expected difference between the optimized code and the unoptimized exceeds the fudge factor. Including such cost estimates in

the target machine description of the retargetable compiler makes it relatively simple to tune many traditional optimizations to differing architectures, making such optimizations pseudo-machine-independent.

IV. Instruction Scheduling

To generate efficient code for an ILP architecture, a compiler must include instruction scheduling. Instruction scheduling is classified as *local* if it considers code only within a basic block[4] and *global* if it schedules multiple basic blocks at once. Local instruction scheduling's largest impediment is its inability to consider context from surrounding blocks. While local scheduling can find parallelism within a basic block, it can do nothing to exploit parallelism between basic blocks. Generally, global scheduling is preferred because it can take advantage of added program parallelism available when motion across block boundaries is allowed. Tjaden and Flynn [72], for example, found parallelism within a basic block quite limited. Using a test suite of scientific programs, they measured an average parallelism of 1.8 within basic blocks. In similar experiments on scientific programs in which inter-block motion was allowed, Nicolau and Fisher [57] found parallelism that ranged from 4 to virtually unlimited, with an average of 90 for the entire test suite.

This section defines the local instruction scheduling problem, and briefly surveys some popular global instruction scheduling methods.

A. Local Instruction Scheduling

Local instruction scheduling typically requires two phases. In the first phase, a data dependence DAG (DDD) is constructed for each basic block in the function. DDD nodes represent operations to be scheduled. The DDD's edges indicate that a node **x** preceding a node **y** constrains **x** to occur no later than **y**. More sophisticated systems [74, 5] label edges from **x** to **y** 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** placed a value on a bus that **y** read, an edge from **x** to **y** would establish a "data dependence" with timing (0,0) indicating that the *read* must follow the *write* in the same instruction. In contrast, if **x** assigned a value to a register subsequently read by **y** and the target machine did not permit reading a register after it is written in the same

[4] A basic block is a straight-line sequence of code that contains no branches except possibly at the end. Thus, each operation specified in a basic block will be executed once each time execution enters the block.

instruction, the edge from **x** to **y** would include timing (1,∞), specifying that **y** must follow **x** by at least one instruction, but can be placed any number of instructions after **x**.

Given a DDD, instruction scheduling attempts to order the nodes in the graph in the shortest sequence of instructions, subject to

1. the constraints in the graph, and
2. the resource limitations in the machine.

Resource limitations come in different forms, but most reflect the fact that when one operation is using a particular resource, another non-data-dependent operation cannot execute if it needs that particular resource. Busses, registers, functional units, and encoding fields are examples of resources that must be shared between operations. In general, this optimization problem is NP-complete [65]. However, in practice, heuristics can achieve good results. A good survey of early instruction scheduling algorithms occurs in [53], while [74] and [5] give more sophisticated algorithms. Beaty [15] provides a summary of current techniques, evaluating many often-used heuristics, and developing new ones. He also describes promising results applying genetic algorithms (GAs) to the instruction scheduling problem. He found that GA techniques produce schedules comparable to those produced by finely-tuned heuristics, without requiring the exhaustive targeting iterations necessary to fine-tune the heuristics. He also reports that the GA scheduler, more robust than a heuristic scheduler, can better cope with a variety of DDDs that are difficult to schedule.

B. Global Scheduling

While local instruction scheduling can find parallelism within a basic block, it can not exploit parallelism between basic blocks. However, several global scheduling techniques are available that extract parallelism from a program by moving operations across block boundaries and subsequently inserting compensation copies to maintain program semantics. Trace scheduling was the first of these to be defined. This section discusses trace scheduling in some detail and briefly defines *sentinel scheduling, percolation scheduling*, and *region scheduling* which are three other global scheduling methods that require compensation copies.

B.1 Trace Scheduling

Trace scheduling optimizes the most frequently used program paths at the expense of less frequently used paths. Fisher originally proposed using trace scheduling to improve microcode efficiency by

selectively moving operations around basic block boundaries [36]. Subsequently, Fisher designed a VLIW machine (Multiflow Trace) that relies heavily on the efficiency of its trace scheduler for high performance. In trace scheduling, the path to optimize is known as the *trace*, leading to the terms *on-trace* (program control paths to optimize) and *off-trace* (blocks in less frequented program paths). Trace scheduling's basic plan moves code from one block to another to reduce the number of instructions in the on-trace path, possibly at the expense of increasing the number of instructions in the program's off-trace segments.

Three principal steps occur in trace scheduling: trace selection, trace instruction scheduling, and bookkeeping. First, trace selection identifies frequently-executed paths. The compiler, using data derived by program analysis, can automatically perform some trace selection. For example, program dominators indicate looping structure, and we can safely assume that the statements in the innermost loop in a nested loop structure execute at least as often as those in an outer surrounding loop. Where we cannot automatically determine path execution frequency, other means of selecting traces exist. Another automated mechanism uses a simulator tool to provide profiling data on a carefully chosen set of benchmark programs. This data, provided to the trace selector, tunes the estimates of relative path-execution-frequency. Still another trace selection method allows the programmer to explicitly identify traces. Of course, one can mix these methods as appropriate.

The second major phase of a trace scheduling optimization is trace instruction scheduling, which treats the entire sequence of basic blocks on a trace as a single block, using local instruction scheduling to schedule the trace as a single unit. Scheduling multiple blocks on a trace is somewhat more complicated than scheduling a single basic block. To ensure that scheduling preserves the original program semantics, trace scheduling restricts movement of some operations from one basic block to another. Ellis [33] gives an excellent summary of the restrictions necessary to prevent "illegal" transformations.

Bookkeeping, the third major phase of trace scheduling, compensates for movement of operations across basic block boundaries, thereby preserving semantics. Such compensation is necessary because, when an operation, **O**, moves from one basic block, **B**, to another (on-trace) block during the multi-block instruction scheduling of phase two, there needs to be some guarantee that **O**'s effects will remain constant on all paths to and from **B**. This compensation requires that operations moving from one on-trace block to another be copied to an off-trace block as well. While bookkeeping's compensation copies will increase the code size required for a program,

the assumption is that program execution time will decrease because the off-trace blocks where compensation code has been placed execute relatively infrequently. Again, Ellis provides an excellent summary of bookkeeping transformations required to ensure semantic preservation during trace scheduling.

B.2 Other Techniques with Copies

While trace scheduling can, in some sense, be considered representative of a group of global scheduling techniques that require copies, several other popular methods exist. Several perturbations to trace scheduling have been defined with the express intent of reducing (but not eliminating) compensation copies. Su's work is indicative of these [66]. Another trace-scheduling like technique is Hwu's *sentinel scheduling* [55] which uses profiling information to schedule multiple basic blocks that he calls superblocks. The key addition of sentinel scheduling is the ability to accurately detect and report all program exception errors, which is a problem when operations move across block boundaries.

Nicolau and Aiken have defined *percolation scheduling* which is based upon a set of core transformations applicable to a program graph. Like trace scheduling, percolation scheduling requires compensation copies of operations that are moved by the core transformations. It is important to note that percolation scheduling's core transformations represent all "legal" code motions defined on a program graph. To use percolation scheduling, supplied heuristics indicate when the compiler applies core transformations. One such heuristic, suggested by Aiken and Nicolau [4], moves all code as "high" as possible in the program graph—that is as close as possible to the graph node that represents the entry to the function being compiled.

Gupta's *region scheduling* [43], yet another global scheduling method, computes the critical path for a region to be scheduled. It uses this information, in conjunction with program transformation rules similar to percolation scheduling to move code in an attempt "balance" the program dependence graph. The feature being balanced is program parallelism. When region scheduling finds a group of blocks (region) that has more operations than can be scheduled within the critical path[5] for that region, it attempts to move operations to another region that has relatively few operations compared to the critical path. Allan et al. describe an enhanced region scheduling method in [7]

[5]The critical path of a DDD is the longest dependence path in the DDD and represents the fewest possible instructions that can be used to schedule that DDD.

which adds transformations to the set defined by Gupta.

B.3 Global Scheduling without Copies

To address the issue of producing schedules without operation copies, Bernstein [20, 18, 19] defined a technique he calls *global instruction scheduling* (GS) which allows movement of instructions beyond block boundaries based upon the program dependence graph (PDG) [35]. Bernstein's method differentiates among inter-block movements that fall into three categories:

Useful motion moves code from a post-dominator block to the immediate dominator block.[6]

Speculative motion moves code from a block to that block's immediate dominator [7]

Code Duplication moves code from a block, B, to some other location that does not dominate B. Such motion requires compensation code.

While Bernstein considers *useful* motion to always improve code, he suggests that speculative motion may not since the dominator block to which code has been moved must execute more often than the block where the code originally resided. In the original GS work [20], only useful and speculative motion were considered. In a small test suite of four programs run on the RS6000, GS showed improvement over local scheduling of roughly 7% for two of the programs with no significant difference for the others. Subsequent work [18] discusses how code duplication can be added to the general GS scheme, and suggests that code duplication will be useful in compilers for future superscalar architectures with more ILP, but gives no comparison between code generated by GS with duplication and that without. The latest work [19] is less useful for a direct comparison of GS and local scheduling, since other factors (e.g machine-specific branch optimizations, and software pipelining) have been incorporated into Bernstein's scheduler.

B.4 Dominator-Path Scheduling

Another approach to global scheduling without copies is *dominator-path scheduling* (DPS) [70, 71]. DPS performs global instruction scheduling by treating a group of basic blocks found on a dominator tree path as a single block, scheduling them as a whole. This allows instruction scheduling to choose the most advantageous position for an operation that might "legally" be placed in any one of several blocks.

[6] B is a post-dominator of A iff A dominates B and all paths from A to the exit of the function must pass through B.

[7] A block B's immediate dominator is that dominator that is itself dominated by all other dominators of B.

In a sense DPS is much like trace scheduling. Both exploit inter-block parallelism by scheduling (using the local scheduler) a group of blocks as though they were a single block. But while trace scheduling requires a bookkeeping phase and compensation copies to ensure program semantics, DPS requires no such copies or bookkeeping. Instead, DPS disallows any operation's movement that would require a compensation copy. The same motivation that drives trace scheduling — namely that scheduling one large block allows better use of machine resources than scheduling the same code as several smaller blocks — applies to DPS as well. Much like $traces$ (groups of blocks to be scheduled together in trace scheduling), the dominator path's blocks can be chosen by any of several methods. Heuristically choosing a path based on length, nesting depth, or some other program characteristic is one method. Allowing the programmer to specify the most important paths is another. Actual profiling of the running program is a third.

The main attraction of DPS for ILP architectures with limited parallelism is that the expense of executing compensation copies is not incurred. This differentiates DPS from most global scheduling methods. Since *dominator-path scheduling* is a significant component of our compiler, we defer further discussion to Section VI-E where our Rocket compiler, including DPS, is discussed in detail.

DPS differs from GS in that DPS allows inter-block motion *only* when no duplicates are required. In GS code moves only in one direction — from dominated block to dominator, while DPS also allows motion into a post-dominator block. Like trace scheduling, GS defines inter-block motion only within the context of a loop. This is in contrast to DPS which allows motion across loop boundaries as well.

B.5 Software Pipelining

While local and global instruction scheduling can, together, exploit considerable parallelism for non-loop code, to best exploit instruction-level parallelism within loops requires software pipelining. Software pipelining can generate efficient schedules for loops by overlapping execution of operations from different iterations of the loop. This overlapping of operations is analogous to hardware pipelines where speed-up is achieved by overlapping execution of different operations.

To illustrate the potential of software pipelining, we'll investigate the problem of computing 4x4 matrix products on a machine that has separate add and multiply units that operate concurrently. Assume that in our simple machine an add requires a single instruction to produce a result while

the multiplier produces a result with a two-cycle pipe. Figure 1 shows the C code for our 4x4 product example.

If we simply count the additions and multiplications, we see that the innermost loop of the program in Figure 1 contains one add and one multiply, each of which are executed 4^3, or 64, times. A simple schedule (that abstracts the memory accesses, initialization of variables, etc.) would contain the following three instructions.

```
1.        t = a[i] [k]  * b[k] [j]
2.        nop
3.        c[i][j]  = t + c[i][j];
```

Note that instruction “3” must follow instruction “1” by two instructions because of the 2-instruction execution time for a multiply, coupled with the fact that “1” computes something (“t”) used in “3”. The result is $3 \times 4^3 = 192$ instruction cycles to perform these arithmetic operations.

If the program was compiled with local scheduling only (that is, the instructions were not moved around basic block boundaries), we would end up with the code as shown above. An opportunity to improve the efficiency of the program becomes evident when one sees that the next scalar multiply

```
int        a[4][4];
int        b[4][4];
int        c[4][4];

main()
{
    int i, j, k;

    for(i=O;  i<4; i++)
    {
        for(j=0; j<4; j++)
        {
            c[i][j] = 0;
            for(k=0; k<4; k++)
                c[i][j] = a[i][k] * b[k][j] + c[i][j];
        }
    }
}
```

Fig. 1 C Program for computing a 4x4 product

figure

can begin as soon as the addition at the bottom of the loop completes. Software pipelining would "fold" the innermost loop body to allow the adds to overlap with the multiplies. The result is shown below. The "#" delimiter separates operations that are executed in parallel.

```
PRELUDE (INNERMOST LOOP):
    1.   t[O]=a[i][O]*b[O][j];
    2.
INNERMOST L00P BODY
(executed  for  k=l  ...  3):
    1. c[i][j]=t[k-1]+c[i][j] # t[k] = a[i][k] *b[k][j];
    2. nop
POSTLUDE  (INNERMOST LOOP):
    1. c[i][j]=t[3]+c[i][j];
```

The result is now $(2 + (2)x3 + 1) \times 4^2 = 144$ instruction cycles to perform these arithmetic operations, representing a savings of 25% over local scheduling.[8] Note that the loop body now contains operations from what would have been two different iterations of the original loop, as indicated by reading $t[k-1]$ while defining $t[k]$. This overlapping of loop iterations is exactly what allows the more efficient code, but just as hardware pipelines need to be initialized before the first result is available and then drained after the last result has begun execution, software pipelining requires that the steady-state loop body be prefaced with initialization code, typically called the *prelude* and suffixed with a *postlude* to finish off the loop results.

To improve the code efficiency further, we can pipeline the middle loop. The result is shown below. The optimized loop produced using software pipelining now requires only $(1 + ((2) \times 3 + 2) \times 3 + 8) \times 4 = 132$ instruction cycles, a savings of 31.2% over the initial (locally) scheduled loop.[9]

```
PRELUDE   (MIDDLE     LOOP):
   1. t[0]=a[i][0]*b[0][0];

MIDDLE LOOP BODY
(executed  for  J=0  ...  2):
INNERMOST LOOP BODY
(executed for k=l ... 3):
    1.
    2.  t[k]=a[i][k]*b[k][j] # c[i][j]=t[k-l]+c[i][j];
```

[8] Actually, software pipelining could improve upon this by overlapping two iterations of the loop within the 2-instruction loop body, yielding a scheduling requiring only 96 cycles. We chose the presented schedule to simplify the discussion.

[9] Again, a better schedule is possible, one requiring 84 cycles.

```
POSTLUDE  (INNERMOST LOOP):
    1.  nop
    2.  t[0]=a[i][O]*b[0][j+l] # c[i][j]=t[3]+c[i][j];

POSTLUDE  (MIDDLE LOOP):
    1.  nop
    2.  t[1]=a[i][1]*b[1][3] # c[i][3]=t[0]+c[i][3];
    3.  nop
    4.  t[2]=a[i][2]*b[2][3] # c[i][3]=t[1]+c[i][3];
    5.  nop
    6.  t[3]=a[i][3]*b[3][3] # c[i][3]=t[2] +c[i][3]
    7.  nop
    8. c[i][3]=t[3]+c[i][3];
```

Software pipelining of loops was first reported by Charlesworth [27] who described an algorithm used in generating hand-written assembly code for the Floating Point System AP-120B family of array processors. Early software pipelining generally dealt with the inherent complexity of finding an optimal schedule by limiting the program constructs on which software pipelining is utilized. Touzeau [73] restricts software pipelining to loops of a single Fortran statement. Su et al. defines two algorithms, each with different restrictions: URCR [66] limits the number of loop iterations that can be overlapped to two; URPR [68] does not restrict the number of overlapped iterations in the loop but insists that the number of loop iterations be known at compile time. In addition, both URCR and URPR apply only to loops consisting of a single basic block. Neither algorithm can pipeline loops containing conditional statements.

More recent software pipelining efforts have removed the restriction that only single-block loops can be pipelined. With GURPR [67], Su et al. updated the URPR algorithm to consider arbitrary control flow. Lam [52] defines a method she calls *hierarchical reduction* that pipelines loops with any block-structured control flow. Hierarchical reduction schedules the blocks of an innermost loop, starting with the innermost control constructs. As each control construct is scheduled, it is replaced with a single node that represents all the resource and data dependence constraints of the entire control construct. This single node is then scheduled along with all other nodes in the surrounding control construct. This hierarchical reduction continues until a single node represents the entire loop.

Allan et al. [8] divide software pipelining techniques into two general categories called *kernel recognition* methods and *modulo scheduling* methods. In the kernel recognition technique, a loop

is unrolled an "appropriate" number of times, yielding a representation for N loops bodies that is then scheduled. After scheduling the N copies of the loop, some pattern recognition technique is used to identify a repeating kernel within the schedule. Examples of kernel recognition methods are Aiken and Nicolau's perfect pipelining method [2, 3] and Allan's petri-net pipelining technique [11].

In contrast to kernel recognition methods, modulo scheduling does not schedule multiple iterations of a loop and then look for a pattern. Instead, modulo scheduling selects a schedule for one iteration of the loop such that, when that schedule is repeated, no resource or dependence constraints are violated. This requires analysis of the data dependence graph (DDG) for a loop to determine the minimum number of instructions required between initiating execution of successive loop iterations. Once that minimum initiation interval is determined, instruction scheduling attempts to match that minimum schedule while respecting resource and dependence constraints. Lam's *hierarchical reduction* is a modulo scheduling method as is Warter's [76, 77] *enhanced modulo scheduling* which uses *IF-conversion* to produce a single super-block to represent a loop. Rau [62] provides a detailed discussion of an implementation of modulo scheduling, while Allan et al. [8] provide a thorough survey of software pipelining methods.

V. Register Assignment

Assignment of program values to registers is critical for generation of efficient code. Computers typically have a hierarchy of memory resources that includes (at a minimum) two levels: relatively fast registers and much slower RAM. It is as true for ILP architectures as for any other computers that a program's execution time can be cut dramatically by maintaining program values in registers rather than slower memories.

This section addresses the interactions between register assignment and instruction scheduling that effect their relative placement in the compilation process. Section V-A defines the register assignment problem and describes graph-coloring register assignment, a popular solution to the register assignment problem. Section V-B investigates the issue of when, during compilation, register assignment should be performed. While the discussion here will use graph-coloring register assignment as an example, the issues are the same no matter what register assignment techniques are used.

A. Graph-Coloring Register Assignment

The optimization of keeping program values in registers as much as possible consists of two (potentially) distinct problems: *register allocation* determines which program values will be placed into a register resource; *register assignment* maps those program values to be allocated to a register to the available machine register set.

A popular register assignment technique is register assignment via graph coloring, commonly attributed to Chaitan [26, 25]. Compared with other, more *ad hoc* register assignment methods, graph coloring provides a relatively simple, conceptually elegant, solution to the problem of mapping program values to a computer's register set. Graph coloring combines allocation and assignment by allocating each distinct scalar program value to a different imaginary (symbolic) register and then mapping the symbolic registers to the physical (hard) registers for the architecture. In this process, graph coloring denotes symbolic registers 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 of its neighbor's. As an architectural paradigm, this implies each symbolic register is assigned a hard register different from all other symbolic registers *live* during the same execution cycles.

It is well known that given a graph, **G**, and a natural number $\mathbf{n} > 2$, the problem of determining whether **G** is **n**-colorable is NP-complete [45], but several researchers have reported excellent results using simple heuristics to color the interference graph [26, 29, 23]. Chaitan's register assignment is based upon a simple observation about coloring graphs. If we remove a graph node of degree less than **n**, no matter how its neighbors are colored, at least one color will be left over for it. For example, consider a graph where we attempt a four-coloring. If we remove a node of degree three, each of its neighbors may be assigned any of three colors, leaving at least one color for the removed node. Nodes are removed in this manner until the graph is either empty or no remaining nodes have degree less than **n**. 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 the node. This deterministic routine consumes non-exponential time and space. It is not guaranteed to find an **n**-coloring if one exists; however, it does produce excellent results in practice.

When a graph is not **n**-colorable, or rather when the heuristic cannot demonstrate that the graph is **n**-colorable, some symbolic registers (program values) are stored in a non-register resource. Using this resource (usually an off-chip read/write memory) creates a speed penalty, so great care must be taken in choosing which symbolic registers should be so reallocated. Code must be generated to store the symbolic register after each of its definitions and to load it before each use. This reduces register contention by reducing the lifetime of the spilled symbolic registers, thereby reducing the interference caused by the spilled registers and allowing register assignment to completely color the graph. Because the interference may not be reduced enough and because hard registers are usually still needed temporarily for spilled symbolic registers, a new graph is built and the coloring and spilling cycle repeated as necessary. Symbolic registers are chosen to be spilled based on perceived cost. Those within nested loops or with a high number of uses will be spilled last.

B. When to Perform Register Assignment

Having decided to perform register assignment using graph coloring, the important question is when during compilation should register assignment be done? There are several possibilities as shown in Figure 2. Figure 2 pictorially represents the different compiler phases of Rocket, highlighting the possible possible places where register assignment could be performed.

Proceeding in "chronological" order of the compilation process, the first choice would be to perform register assignment on the machine-independent intermediate form. This placement is often used in register assignment for non-ILP architectures where the intermediate form is closely related to the final code produced. For ILP architectures, however, where instruction scheduling reorders operations, we shall see that performing register assignment this early in the compilation process leads to difficulties.

The next logical place to include register assignment would be during code selection. This approach has long been used in traditional compilers. Graph coloring register assignment, however, is designed to function as a separate pass of the compiler. This design feature is both a strength and a weakness. The decoupling from other compiler phases makes it more difficult to access information from other compiler phases when making register assignment decisions, but makes the algorithm used more independent.

One could perform graph coloring register assignment on the DDDs generated by code selection. In fact, several compilers do just that [14, 58, 42]. The advantage over earlier register assignment

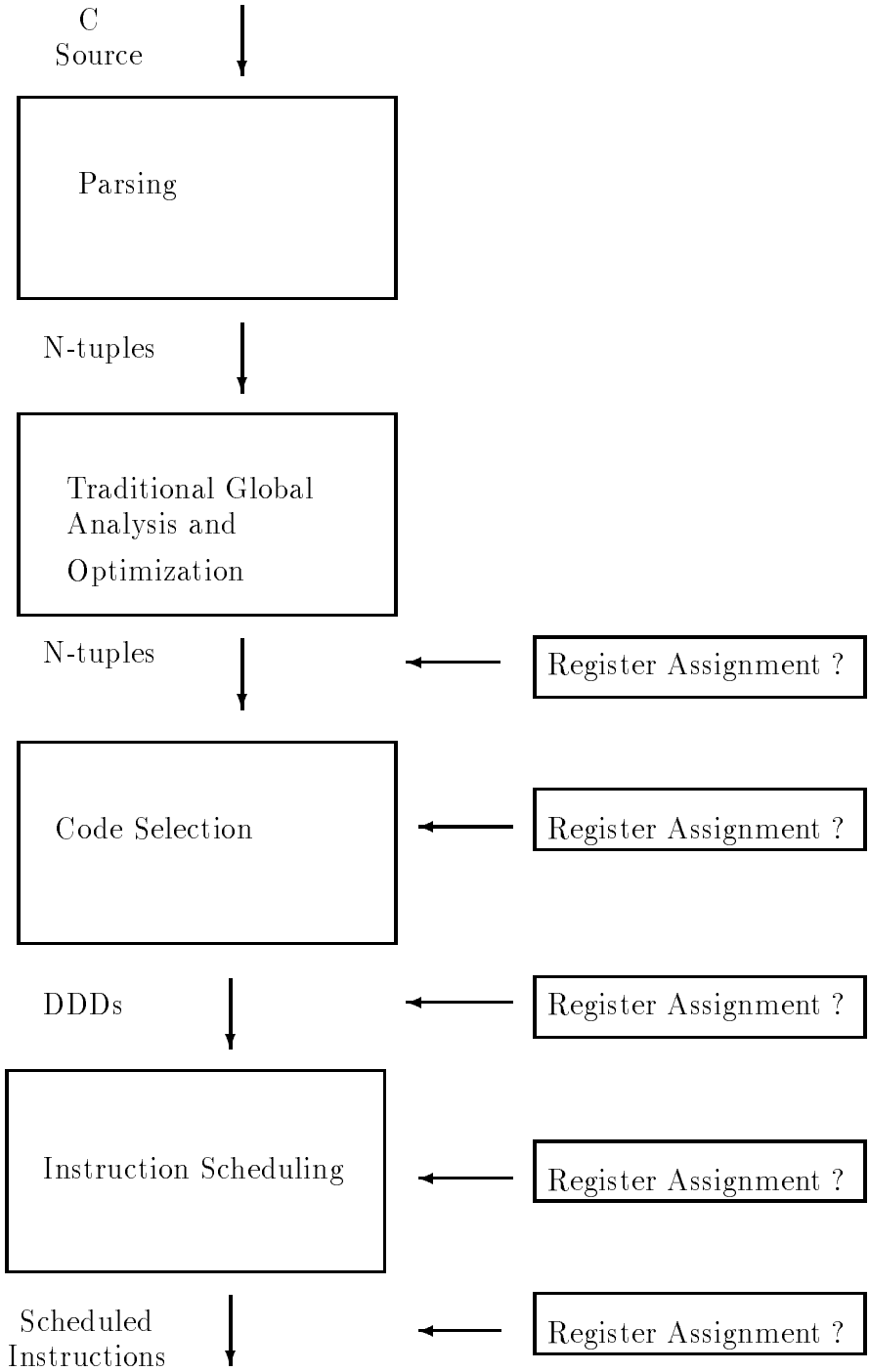

Fig. 2 Possible Placements for Register Assignment

figure

is that the DDDs expose more parallelism inherent in the program than the intermediate form.

Ignoring for the moment the possibility of combining register assignment and instruction scheduling, we could choose to delay register assignment as a separate phase until after instruction scheduling has been completed. This has the advantage, over earlier register assignment, of allowing more freedom to the scheduler. The problem with performing register assignment before scheduling is that, the very process of mapping the hypothetically infinite number of symbolic registers to the very limited physical registers available for the architecture must lead to reuse of physical registers that is not apparent at the symbolic register level. This reuse of registers in turn leads to additional dependences (anti-dependence and output dependence) in the DDDs to be scheduled. These added dependences limit the scheduler's ability to parallelize the final code. So, merely in terms of obtaining better schedules, performing register assignment after scheduling is advantageous. The disadvantage of delaying register assignment until after scheduling is that the greater scheduling flexibility will usually lead to higher register pressure. If this higher pressure leads to spilling, we are presented with the classic tradeoff between shorter instruction schedules possible by delaying register assignment and fewer spilled registers needed when performing register assignment before scheduling.

Consider again the alternatives of building the interference graph from DDDs or from scheduled instructions. In some sense, DDDs represent "too much" parallelism. They potentially include a greater degree of parallelism than scheduled instructions and thus could lead to even greater register interference than that obtained when building the interference graph from scheduled instructions. In some sense this added parallelism is "useless" to us, since it represents parallelism that the compiler will not be able to exploit. In contrast the scheduling phase of the compiler represents exactly that level of parallelism the compiler exploits from a program so it seems that building interference graphs from that program representation will give the most accurate assessment of register needs, in the best case.

Since there are well-defined pros and cons to doing register assignment early or late, it has been suggested that the two phases should be merged. A basic technique is to have a pool of registers being managed by the scheduler. In this way, the scheduler can take into account the number of available registers when it is scheduling. Several researchers have tried more sophisticated techniques.

Goodman and Hsu [42] compared two different methods of integration against both early register

assignment and late register assignment. In one method, which they call integrated prepass scheduling, they perform late register assignment while restricting the scheduler to use a fixed number of registers. Their other method manipulates the scheduler's DDD so that the "width" of the DAG is no greater than the number of registers available. Using the altered DDD, the method performs early register assignment. They found both techniques outperformed strictly early or strictly late register assignment.

Bradlee [21] discusses a method of mixing the two phases in which initial passes of an instruction scheduler get estimates of the schedule cost given a certain number of registers. The scheduler is run locally with a very limited number of registers and then again with the maximum number of registers in the machine. These values are then used to allocate a certain number of registers for each basic block.

Freudenberger [39] describes how register assignment is integrated into trace scheduling in the MultiFlow compilers. The scheduler drives the register assignment process to place the heavily used values (i.e. the values in the heavily-used traces) in registers. Since trace scheduling starts scheduling on the crucial traces first, the scheduler, which uses a pool of registers, takes as many registers from the pool as it needs. Since traces have multiple entry and exit points, information is stored about which registers contain which values at each entry and exit point. When other, less crucial, traces "hook up" to this trace, this information is used to minimize the amount of code that is needed for data movement.

Norris and Pollock [58] perform early register assignment, but add edges to the interference graph to estimate the re-ordering effect of instruction scheduling. Their basic goals are to construct a plan that does not require alteration of the scheduling phase of compilation, and to require only a single pass of instruction scheduling. They achieve these goals by building the interference graph from the DDD) rather than from either a linear listing of intermediate code (as "traditional" early assignment does) or the scheduled code (as late assignment does.) Since the DDD represents more parallelism than either the initial linear code or the final schedule, this could lead to interference graphs that would be hard to color. Norris and Pollock evaluate several heuristic techniques to limit DDD parallelism (and thus the number of registers needed) while hopefully retaining a good schedule. Their experiments show a significant improvement over strictly early register assignment for livermore loops.

Pinter's work [60] realizes that the register interference graph for early register assignment has

fewer edges, thus allowing extraneous anti-dependences to occur. This, in turn, leads to a schedule that is too conservative. To avoid this, her algorithm creates a parallelizable interference graph. To generate such a graph, the schedule graph (i.e. similar to a DDD except that nodes consist of only the destination for each operation) is analyzed and the transitive closure of all the directed edges are placed into a graph as undirected edges. Any machine dependences (resource conflicts) are then added to this graph. As an example, if a machine has only one divide unit, and two operations require the division unit, an edge is placed between the nodes corresponding to these operations. The graph's complement is then constructed and referred to as the false dependence graph. The union of the register interference graph and a false dependence graph is created. This new graph is the parallel interference graph, and represents all the true interference that exists between values. It can be colored to give a register assignment that does not retard any available ILP. Since it is likely that this graph is not colorable with the available registers, Pinter carefully chooses which edges to remove in order to avoid creating anti-dependences that might retard the final schedule. By generating a parallelizable interference graph, Pinter is approaching the problem of incurred false dependences by removing all of them and then adding them back only as needed. Scheduling is done after register assignment, but only after the register assignment process has effectively "massaged" the information in a way that allows the scheduler more freedom to re-order code.

One final solution to where register allocation and assignment should be in the compilation process is to delay it past compilation entirely, and perform at least part of the process during program linking. Wall [75] has reported good results with such link-time register assignment.

VI. Rocket

This section describes the Rocket compiler, a highly-optimizing compiler retargetable for a broad class of ILP architectures. Rocket is implemented in C++. Current targets include three commercial superscalar architectures (the Alpha 21164 Microprocessor [32], the IBM RS6000 [48] computer and a computer based on an Intel i860 chip [49]) as well as several hypothetical LIW processors, and a family of hypothetical superscalar processors [61] Rocket was designed and initially implemented as a C compiler. However, due to the small emphasis we place on transformations within the parser, it is relatively easy to add an additional front-end for a different language. In fact this has been done, and Rocket currently includes both C and FORTRAN parsers.

The most significant feature of any compiler for ILP architectures is the need to order machine operations to obtain maximum benefit of the architecture's available parallelism. This necessitates a series of analysis and optimization phases geared towards providing maximum flexibility in operation placement and, given such flexibility, a mechanism to make intelligent decisions in the operation placement. Section III-A discussed analyses and optimizations useful for generating high-quality code for ILP architectures. This section begins with a discussion of the relative placement of such analyses and optimizations within the compilation process and a brief description of the different representation Rocket uses for a program during compilation.

To build a retargetable compiler, whether for ILP architectures or not, the designer must provide a clean interface between the machine-independent algorithms used to generate code and the machine description necessary to specify architectural detail, so we next turn our attention Rocket's machine description as an example of such an interface, followed by a discussion of the machine-independent algorithm Rocket uses to select code.

Finally we turn to a discussion of the several different instruction scheduling methods used in Rocket and a brief account of Rocket's register assignment methods.

A. Code Generation Phases

To generate excellent code, an optimizing compiler normally uses several phases, typically representing the C program in different forms during the each phase. When compiling for ILP targets, all the phases necessary when compiling for traditional architectures are still required. In addition, we need phases for parallelization of intermediate code and scheduling of instructions.

During translation into highly-optimized code for ILP architectures, Rocket executes a number of distinct phases. These are summarized pictorially in Figure 2.

First, **Parsing** produces an abstract representation of the input C or FORTRAN program. Rocket uses the lcc compiler [38] as a C parser, building an intermediate form from lcc's code generator. Rocket's current FORTRAN parser comes from the ParaScope programming environment [24].

Next, **Global Analysis and Optimization** is performed on the abstract intermediate representation built by the parser. To adequately do both traditional optimization and instruction scheduling, a compiler must perform extensive dataflow analysis of the intermediate form(s) generated by the compiler. Rocket includes analysis that computes local and global dataflow, symbolic cover analysis based upon Reif and Tarjan's fast covers algorithm [64], live variable analysis required by

register assignment, and data dependence analysis necessary in the parallelization process included in code selection. In addition, to more accurately determine the program's actual data dependences, Rocket includes memory reference disambiguation for array and pointer references. This disambiguation method, based upon the analysis of diophantine linear equations, is well summarized in John Ellis' dissertation [33].

Rocket includes traditional optimizations described in any modern compiler book [1] and discussed in section III-A. These include common subexpression elimination, copy propagation, constant folding, constant propagation, algebraic simplification, reduction in strength, and dead code removal. Further, Rocket actually makes use of two different abstract intermediate forms that are discussed subsequently.

Code Selection replaces the abstract representations of an intermediate form with collections of machine operations. In compilers for traditional, non-ILP architectures, code selection can generate the final form of the compilation process — either assembly language or binary code, depending upon the compiler. In ILP compiler such as Rocket, however, more is required. The product of code selection in Rocket is a Data Dependence DAG (DDD) for each basic block of the control flow graph for the function being compiled.

Instruction Scheduling assigns machine operations to instructions to the satisfy data-dependence and machine-resource constraints of the DDD built by code selection. It is the need for such an instruction scheduling phase that distinguishes compilers for ILP architectures from those for more traditional computers.

Figure 2 does not identify where, during Rocket's compilation process, **register assignment** is performed. machine memory resources. In fact, as Figure 2 shows, there are several possible locations, during compilation, for the graph-coloring register assignment. Rocket's placement of register assignment is discussed later in this section (see VI-E.)

Of course, many compilers use different intermediate forms to represent the program being compiled, varying the representation to fit the needs of whichever phase the compiler is performing at the time. All of Rocket's phases are based on a control flow graph (CFG) for the compiling function, but within the CFG Rocket uses, at different times during compilation, four different representations of the function being compiled These are:

Intermediate Statements are the basic form of intermediate code used throughout the analysis and optimization phases of Rocket. Rocket's intermediate statements are a linear form of three-

address code called n-tuples Three-address code is so named because it is a representation of a generalized assembly code for a virtual three-address (two sources and one destination) architecture. N-tuples are a form of three-address code that includes fields for an operator, a destination and as many source operands as are needed. Since Rocket is written in C++, it is not necessary to actually include fields in the intermediate statement for each of an operator and three addresses. Instead, we defined a basic type, *Statement*, that includes dataflow sets useful in analysis. For each of several types of special statements, such additional data fields as are necessary are added to the basic Statement pattern. Although Rocket includes several types of intermediate statements, the two most important for purposes of our discussion are:

CopyStatement is a representation of a simple data movement, corresponding to a C statement such as

A = B

A CopyStatement is represented by two operands, the destination and the source.

OperatorStatement is a representation of a computation statement including a single operator and as many operands as are necessary for the computation. This could include standard unary or binary operators, or include a special operator that requires an arbitrary number of arguments. Most generally an OperatorStatement corresponds to an expression of the form

R = $f(S_1, ..., S_m)$

where R is a destination, f some functional operator, and each S_i, a source operand. A Rocket OperatorStatement consists of a destination, a functional operator, and a list of source operands.

A Symbolic Execution Dag for the entire function is constructed from the intermediate statements of the CFG. This symbolic execution DAG uses origin information supplied by the Tarjan covers analysis to provide global information and is modeled (with minor modification) after the DAGs described in algorithm 9.2 of the Dragon book [1]. The symbolic execution DAG serves two major functions in Rocket: it is the basis for constant propagation, algebraic simplification, and common subexpression elimination, and it also provides the basis for memory reference disambiguation.

DDDs are directed acyclic graphs that represent a parallel form of the machine operations to be performed by a basic block. Associated with each graph node are an identifier and a list of machine operations that are expressed as assembly language constructs that are

eventually written as the output of a Rocket compilation. Additionally, each DDD node contains dataflow information specifying which machine resources are used and which are defined by the operations specified in that node. Finally, to assist in instruction scheduling, each node includes a list of instruction fields that must be set in order to invoke the operation specified by the node.

Each edge E from node A to node B in a DDD has a label (min_time, max_time) associated with it. This indicates that execution of B must follow the execution of A by at least min_time and, at most, max_time clock cycles. Edges may represent data dependence or a control dependence (e.g., a branch in a basic block's DDD would generally be connected to the sink node by an edge with (n, n) timing, where n is the branch delay).

The DDD also contains a collection of *resource livetracks* that represent resource usage (or for brevity herein simply *livetracks*). A single *livetrack* for a resource R is a pair, (b, U), where b is the DDD node identifier in which R is written (defined) and U is the set of DDD node identifiers that read (use) the value of R written at b. When a value of R is live-in to the DDD, it has a livetrack with $b = source$. When a value of R is live-out from the DDD, the corresponding livetrack has the sink as a member of U. In general, several livetracks may be associated with a given resource in a DDD. For efficiency during livetrack manipulation, the livetracks are linearly ordered in terms of height in the DDD.

Lastly, the DDD contains the same dataflow information that is maintained for a basic block, namely *live_in*, *live_out*, *used*, and *defined* sets.

Scheduled Instructions which are simply lists of nodes from a DDD, representing the operations to be executed in a single instruction or cycle. The scheduled instructions represent the final product of a Rocket compilation, ready to be written to an assembly file.

B. Target Machine Description

To generate code for a particular architecture Rocket relies heavily on table-lookup techniques that access machine-dependent information. Rocket builds these tables from a machine description file using a configuration processor program. This section describes the machine description file format, and addresses issues related to targeting Rocket to a new architecture.

A Rocket machine description includes the following general categories, each discussed in more detail below:

Machine Resources includes information about each memory (general purpose registers, main memory, condition registers, etc.) in the target computer.

Assembler Syntax includes special assembler-dependent information such as label format, and constant formats.

Activation Record includes a description of the activation record[s] used for the target architecture.

Code Tables include a description of how to perform data movement and computations in the target architecture.

B.1 Example Architecture

To provide a more concrete explanation of the machine description file, we will use an example hypothetical load/store architecture, called H. For purposes of this discussion, we'll assume that H is an LIW architecture, though the configuration information for a superscalar ILP machine would be much the same. The H processor provides both integer and floating point function units, as well as a memory access (load/store) pipeline, allowing up to two computations and a memory operation to be started in each instruction. Loads and stores are from/to a single RAM memory, thus no cache is included. There are two sets of registers, 32 32-bit registers for integer computation and 16 64-bit registers for floating point computation.

In any one machine cycle (assembly language statement) 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. Integer operations require only one cycle to complete.
- start either a load or a store from/to RAM. Loads and stores require three cycles to complete.
- start a floating point operation. This is a pipelined computation and requires 5 cycles to complete.

B.2 Machine Resources

The machine resources portion of the configuration file specifies the characteristics of each memory included in the target architecture. Rocket distinguishes between two types of machine resources, those that are "colorable" and those that are not. Colorable resources, typically registers, are those that are replicated and for which Rocket is to perform graph-coloring assignment of individual

values to one of the members of the resource. As an example of a colorable resource specification, consider the following description of an I register for the integer processor of the H architecture:

```
"i_reg"                         // name
        "r?"                    // contents
        "ERROR"                 // address
        "0x0(r?)"               // indirection
        1 100000                // data_dep
        0 100000                // anti_dep
        1 100000                // output_dep
        ".set ?hard ?soft"      // register equate
        32                      // # regs
        3 - 31                  // color
        yes                     // multisave?
        rsave                   // operator to save registers
        rrestore,               // operator to restore registers
```

The specification starts with three fields (contents, address and indirection) that describe how an operand of this type is to be specified in the assembly language to be generated by Rocket. In each case the "?" will be replaced (in the assembly language) with the register number. Three different specifications provide for three types of operand access, corresponding to wanting (1) the contents of the register, (2) the address of it (not defined in this case), or (3) the contents of something "pointed to" by the register.

Next in the specification is the timing information code selection will access to include proper timing for operands of this resource in the DDDs. There are three pairs of timings indicated, corresponding to the three basic types of dataflow dependences.

- *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.

For each of the three timings both a *min* and *max* time are given to represent the earliest that an operation may follow the preceding operation and the latest that it may follow. So, when the data dependence time for our register example is (1, 100000), a read operation that follows a register write must be scheduled no sooner that 1 cycle later than the write of the register and can be scheduled as many as 100000 cycles later (100000 is used to indicate infinity in this case. It is assumed that no DDD will require 100000 instructions to be scheduled.) Similarly the anti-dependence times of (0, 100000) indicate that a DDD node writing a new value to a register can be scheduled in the same instruction (0 cycles later) as the last node that read the previous value or it can be scheduled up to 100000 instructions later.

The operand addressing specifications and the timing information are required for any type of machine resource specified in the target description. The rest of the fields in the *i_reg* memory specification are associated only with colorable resources. These fields include the number of registers in the register bank, the number that Rocket can use to color values allocated to an *i_reg* resource, and three fields associated with the saving and restoring of the registers. Differentiation between the number of registers in the register bank and those that can be colored is an important distinction. We must account for the fact that, in many cases, not all registers in a particular bank can be treated as general-purpose registers. Some may be reserved for special purposes like the stack pointer, the program counter, or a function return value. In our example, registers 0-2 are apparently reserved for some special purposes.

A necessary consequence of register assignment, whether by graph coloring or not, is the need to save-and-restore registers when functions are called. The issue of when and how to do this save and restore is discussed in section VI-E. For now, we're concerned with the kinds of information necessary to perform the register save and restore using a machine-independent algorithm. Three items are necessary for the Rocket specification of register save and restore — a boolean value that indicates whether a range of registers can be saved or restored with a single "command", and the two "operators" that perform the save and restore respectively. Operator in this sense is meant to be equivalent to a C operator. As we shall see in section VI-B.3, Rocket associates code table entries with each possible C operator. Additional non-C operators such as *rsave* and *rrestore* are recognized by Rocket as operators as well, so that references can be made to code packets to

perform special functions such as saving and restoring registers in the same way that “normal” C operations are specified.

In addition to the memory specifications just described, Rocket includes a separate description, called **default memories** that maps C types (*int*, *float*, etc) to their appropriate machine memory. Rocket accesses the default type specification to allocate program values to the proper machine resource. Each C type has several memories associated with it to account for the different types of C values we need to store in the target machine. The *default* specification is used to allocate a “normal” local scalar value in C that does not have its address taken. For each C type there are additional specifications for *spill* memory (when register assignment needs to spill a value out of a register), *global* that is used for global memory locations, and *local* which is used for stack variables which cannot be stored in a register (arrays, structures and scalars whose address is taken.)

B.3 Code Tables

To perform code selection, Rocket accesses information about how data can be moved among storage resources, about how functional operations are invoked and about how sequencing is performed. The code tables section of configuration file for a given architecture contains this information. Rocket includes two varieties of code tables entries; one corresponds to data movement between storage resources and one to the functional and sequencing operations. The foundation of both forms of code table entry is an abstract representation of a DDD to perform the necessary function — data movement or some functional operation. Rocket’s code table notation is syntactically similar to the programming language Prolog, no doubt due to the fact that an early precursor to the Horizon compiler [56] (of which Rocket is a descendant) was implemented completely in Prolog. Consider, for example, the following code table entry for loading a value from Ram to an integer register for our hypothetical H computer:

```
data_movement(i_reg, ram, // load i_reg
        (
                fields  (integer),
                reads   (i_reg1,indirect_addr,argv1),
                writes  (rw_pipe0),
                assembly("load |0,|1")
```

```
        ),
        (
                fields  (dummy),
                reads   (argv1,rw_pipe0),
                writes  (rw_pipe1),
                assembly("")
        ),
        (
                fields  (dummy),
                reads   (rw_pipe1),
                writes  (argv0),
                assembly("")
        )
).
```

This entry's first line indicates that this is a data movement code entry with an integer register destination and a Ram location as the source of the movement. The rest of the entry is the DDD description of the H operations necessary to perform the function. This abstract DDD consists of an ordered but unnamed list of DDD nodes For purposes of the following discussion only, we shall name the three nodes indicated in our example code table entry I1, I2, and I3 respectively. Each node (included in parentheses) is represented by four fields:

fields, which lists any instruction fields set by this DDD node. Instruction fields are used to represent resource constraints. In the given example, I1 uses the integer field, indicating the the integer pipeline is required, while I2 and I3 use "dummy" fields, meaning that they do not encumber any resources. Use of dummy fields is a convenience that allows additional DDD nodes to be used strictly for timing constraints. In this case the timing constraint is that the i_reg whose value will be defined by the load this DDD represents will not actually obtain its (new) value until 3 machine cycles after the load is initiated.

reads, lists those machine resources that are used (read) by this DDD node. Looking at I1's read information we see that i_reg1, argv1, and indirect_addr are all read. Of these resources, two (argv1, indirect_addr) are *pseudo-resources* in that they represent configuration "variables"

rather than actual machine resources. Following C-like conventions, argv1 is a "parameter" that will be replaced by a resource representing the actual memory location read. This feature allows at least the possibility to recognize that different memory locations do not actually conflict with one another. The indirect_addr pseudo-resource is used to indicate that any register used indirectly to access the RAM location should be considered as well. For example, if the actual assembly language statement that will be constructed (by code selection) from this data movement DDD is

load r4,0(r5)

we need to recognize that r5 is used to specify the RAM location being loaded. The i_reg1 read specification indicates that the actual machine resource, register 1 of the i_reg bank, is used by this DDD node. Perhaps this is due to i_reg1 being the stack pointer, and the concern that any RAM location read might depend upon the current stack pointer value.

writes, lists those resources defined by this DDD node. Again, liberal use of pseudo-resources adds flexibility. Looking again node I1, we see that it writes rw_pipe0, which will subsequently be read I2. This shows how Rocket specifies pipelined instructions, by the addition of DDD nodes with restrictive timing (1,1 in this case) that represent individual stages of a pipeline. The rw_pipe0 and rw_pipe1 reads and writes will ensure (due to the flow dependence timing of these resources) that the value defined by this load will be ready exactly 3 cycles after the load is initiated.

assembly, gives the parameterized assembly language associated with this DDD node. The assembly language string is parameterized by the "|0" and "|1" symbols that will be replaced by the appropriate assembly language specification for the two input arguments (argv0 and argv1) at code selection time. Notice again that the "dummy" nodes contain no assembly language since they don't contribute to the code for the instruction into which they are scheduled.

As mentioned in the description of the DDD nodes' fields, our example data movement DDD demonstrates how we can easily represent pipeline operations in the Rocket compiler. The pipeline of interest here, the read-write pipeline, requires three cycles to read a value from memory into a register. Thus, we need a way to specify that in the machine description. We could, of course, include a data dependence of $3,\infty$, but that would unnecessarily limit our ability to overlap reads or writes. Instead, we choose to include specific resources to represent pipe stages, making sure to include data dependence times of 1,1 between individual pipe stages. Notice that this ensures

that scheduling recognizes that the register being written to gets a new value exactly 3 instructions following the assembly language instruction that initialized the load. That our DDD has three nodes does not mean three separate instructions will appear in the assembly language generated by Rocket. Nodes I2 and I3 neither include any assembly language, nor do they encumber any instruction fields, and thus, they can share an instruction with any other DDD nodes, subject to timing constraints.

Note also the location of the *use* of the Ram resource and the *definition* of the integer register in node I2 and I3 respectively. This placement ensures two things: 1) that the Ram location will not be rewritten with a new value (due to some other operation) before it is last needed at the instruction *following* the one where it is used in the assembly language, and 2) that whatever register is being defined can be safely accessed in either the instruction where the load starts (wherever I1 is scheduled) or the next (where I2 *must* be scheduled) and, in either case, we can be sure that the "original" value (the value before the one defined by this load) will be the one used. This brings up a subtle distinction between scheduling for LIW architectures such as our hypothetical H processor and scheduling for superscalar machines. Consider the following code fragment, of unscheduled assembly language:

```
add r5,r4,r2          # add r5 and r4 and place in r2
load r4, X            # load a value from memory into r4
```

On an LIW architecture with timing such as the H machine these two instructions could be reversed and the same values would be computed because the "load r4, X" would not actually alter the value of r4 until three cycles after the instruction is executed. However, on a superscalar with the same 3-cycle load time, if we exchanged the two statements, we would have changed the code's meaning, because a superscalar, recognizing that r4 would not obtain a new value for three cycles, would insert a stall to wait for the new value. This means that for LIW architectures, it is possible, under some circumstances to access an "old" resource value that appears, from the assembly language, to have already been redefined, but it is not possible to do this on a superscalar machine. Thus, our code table above would need to be amended if the target machine were a superscalar architecture. If we did not rewrite the entry for a superscalar architecture we would run the risk that instruction scheduling would "break" the program. To fix the potential problem for a superscalar target, we could insert a superfluous data dependence on the register being written. Of

course, this would inhibit (in some sense unnecessarily) the scheduling possibilities but it would ensure correct code. With the new dependence, the updated superscalar form of the DDD would be:

```
data_movement(i_reg, ram,
        (
                fields  (integer),
                reads   (i_reg1,indirect_addr,argv1),
                writes  (rw_pipe0,argv0),
                assembly("l |0,|1")
        ),
        (
                fields  (dummy),
                reads   (argv0,argv1,rw_pipe0),
                writes  (rw_pipe1),
                assembly("")
        ),
        (
                fields  (dummy),
                reads   (rw_pipe1),
                writes  (argv0),
                assembly("")
        )
).
```

Notice that the difference between the superscalar and LIW implementations of this code table are that the superscalar has an additional write of argv0 in I1, and a read of argv0 in I2.

The other type of code table entry, that for a functional operation, is actually quite similar to a data movement code entry. Again, the basic information is included in the parameterized DDD. Consider the following code table entry that specifies how integer computations are performed on the H machine.

```
oper (three_address_alu, i_reg, (i_reg, i_reg),
```

```
        ((
                fields  (integer),
                reads   (argv1, argv2),
                writes  (argv3),
                assembly("|0 |3,|1,|2")
        )),
        (
                int      (add "addw", sub "subw"),
                short    (add "addw", sub "subw"),
                long     (add "addw", sub "subw"),
                unsigned (add "addw", sub "subw"),
                char     (add "addb", sub "subb")
        )
).
```

The parameterized DDD still includes all the information needed to generate code for the desired function (in this case an integer operation performed in i_regs. We now have more possible parameters than with the data movement DDD. Specifically, we need parameters for each possible source operand and the destination, and we include a parameter (number 0 by convention) for the operator as well. Using a parameter for the operation allows us to save code table space as the same DDD can be used for several operators.[10]

In addition to the DDD, we include a list of possible C types that can use this code table entry for some operator, and, within each C type, we include a list of the operators that are represented by this DDD. Each C operator is accompanied by the appropriate assembly language mnemonic, which explains the *addb* (for add byte) field for addition of *char* type while addition of an *int* type is specified by *addw*.

The first line of the function code entry assigns an arbitrarily determined name to the function specification (Rocket ignores this name, but it provides a documentation feature for the code tables.) Next, the entry lists the machine resource for the destination of this functional code entry and a list of the machine resources for the sources to the function. These resources are not ignored, but

[10] Actually, this table entry has been shortened quite a bit as we removed most of the operators from the operator list of this example to make it all fit nicely on the witdth of a page.

are rather an important part of the Rocket specification for a function code entry, as we shall see in the discussion of code selection in section VI-C. Rocket's code selection accesses code table specifications for functional operations by retrieving the operation code entry associated with the pair <C type, C operator> for the function to be performed.

B.4 Activation Record

To include machine-independent algorithms to process function calls, Rocket needs to access machine-dependent information such as how parameters are passed (stack or in registers), how return values are returned from a function, and the syntax of how functions are actually called. While the determination of how functions are called is treated similar to register save-and-restore code (special operators are hypothesized and functional operation code entries for those operators are included in the code table entries.) Rocket's activation record includes specification of how parameters are passed and what machine resources are used for return values.

Since C supports recursion, a general activation record must provide support for an activation stack. It is often possible, however, to provide a more efficient mechanism for function calls if a stack discipline is not required. Because of this possibility, Rocket recognizes special activation types that allow for more efficient calling sequences. Currently, Rocket allows three different activation records; default (recursive) calls, non-recursive calls, and leaf functions. This allows the possibility of faster calling mechanisms for special-case function calls.

B.5 Assembler Syntax

Since Rocket generates assembly language as output several features of the assembly language need to be specified in the configuration language as well. These include:

- Label format
- Prefix and postfix code for each program, function, basic block, and instruction. (Any of these may be empty, of course.)
- The format of constants in the assembler. A constant specification including the necessary information (whether supported, the number of bits allowed, syntax for statically allocation) is part of the Rocket configuration description.
- Any predefined constants used in the assembly language, such as constants used in type conversion between integer and floating point values.

In addition, the configuration specification includes a description of the instruction word's fields, and an indication how many instances of a particular field are allowed in any instruction. These fields, the same ones specified in the nodes of the DDD, are used by instruction scheduling to ensure that the architecture's resource constraints are enforced. For example, the format specification

1 integer, 2 rw_float, 2 fadd;

indicates that an instruction can include up to one instance of an operation encoding the *integer* field, two encodings of *rw_float*, and two of *fadd*.

C. Machine-Independent Code Selection Algorithm

The code selector generates code on a basic block level. It derives its simplicity from putting off the problem of synchronizing resource usage to a separate phase, the DDD coupling phase. The general form of the input to the code selector is a representation of a basic block as a sequence of intermediate statements. Rocket is based on the assumption that a DDD exists that evaluates each assignment without semantic transformations (e.g., the need to replace a subtract operation by a complement and add expression). Thus, to select code for a basic block $B =< Q_1, ..., Q_n >$, the code selector derives individual DDDs for each intermediate statement Q_i and uses the DDD coupler to combine the DDDs in sequence. Deriving a DDD for a statement, Q_i is divided into two phases. First, using an algorithm described below, Rocket builds an "abstract" DDD for Q_i. This abstract DDD is a parameterized form of a DDD from the code tables. Having obtained the proper abstract DDD from the code tables, the code selector builds a list of the actual operands of the statement and calls a unification function that substitutes the actual operand values for the DDD's parameters. Having done that, the code selector merely calls the coupler to combine the DDD being built for the basic block with the DDD just derived for Q_i.

To obtain the proper abstract DDD for intermediate statement, Q_i, we need consider only two cases, based on the form of the statement. The simple case is an assignment of the form $R = S$ (CopyStatement), where the source expression S is a machine resource. Then, the target machine representation will contain a DDD, D, for moving the contents of S to R (the (R, S) entry in the data movement matrix) that can be readily retrieved.

The more complex case applies to assignments of the form

$$R = f(S_1, ..., S_m)$$

(an OperatorStatement), where f is a functional operation and each of S_1 through S_m is a source operand for the computation to be performed. We begin by retrieving the DDD for evaluating f (say DDD_f) from the target machine representation. Included in the entry will be the immediate input resources ($I_1, ..., I_m$) and the immediate output resource (O) for the functional unit associated with f.

In a previous version of the compiler we then proceeded as follows:

1. Apply the code selector (recursively) to each of the assignments $I_1 = S_1, ..., I_m = S_m$ to obtain $DDD_1, ..., DDD_m$. We then couple these m DDDs to form a single DDD, DDD_I.
2. Couple DDD_I onto DDD_f to obtain DDD_{fI}.
3. Apply the code selector (recursively) to the assignment $R = O$ to obtain DDD_O. Then couple DDD_{fI} onto DDD_O to obtain the resulting DDD for

$$R = f(S_1, ..., S_m)$$

Summarizing, the solution calls for

1. (possibly) copying the source operands for the computation into variables allocated to the proper input resources for the functional unit,
2. generating code for the computation,
3. (possibly) copying the result of the computation to the destination of the intermediate statement, and
4. coupling all of the DDDs together in order.

But since code selection for a basic block already couples the DDDs for individual statements to get a DDD for the block, we can obtain the same effect by inserting (prior to code selection) new intermediate statements to perform the necessary data movements to set up the inputs and copy the output of the computation. Note that by performing all of these tasks during code selection itself, code selection must be able to generate new temporaries for the I_i and O operands. Since we wished to keep register assignment and code selection separate in the Rocket compiler, Rocket now performs code selection in two phases, one pass to perform the data movements necessary for each OperatorStatement, and a later pass to perform the actual code selection and coupling. This splitting of code selection allows us the possibility of completing register assignment before code selection actually builds DDDs.

C.1 DDD Coupler

Traditionally, code sequences in an LIW compiler have been represented and coupled vertically, requiring an additional procedure to "parallelize" them for subsequent scheduling. Vegdahl [74] pointed out problems with effective management of resource sharing when using such an approach and suggested the need to deal with *data anti-dependence* relations to facilitate more highly optimized code. To effectively manage resource sharing, Rocket uses a *DDD coupler* to provide a mechanism for joining the small DDDs generated by the code selector into one DDD for each basic block, with the intention of removing the need for separate coupling and parallelization procedures. While the original motivation for the DDD coupler was to provide support for the machine-independent code selection algorithm described above, the coupler provides a flexible method for combining DDDs while maintaining a maximal level of parallelism. As such the coupler will play an important role in dominator scheduling and trace scheduling, the global instruction scheduling methods described in sections VI-D.3 and VI-D.2.

In Rocket, the coupler connects "open" tracks (that occur when data dependences span from one DDD into another), and synchronizes the remaining resource usages by introducing edges that ensure that two tracks for a given resource cannot overlap after instruction scheduling.

The DDD's major parts, as far as coupling is concerned, are 1) the definition-use tracks for each variable with a value in the basic block the DDD represents, and 2) the variable sets *def, used,* and *used-before-defined* in the block and the set *live_out* exiting the block. Almost all of the coupler's function involves case analysis of definition-use tracks. To meaningfully discuss the algorithm we use a pictorial representation of the definition-use tracks of a basic block, borrowing the notation of Allan [5, 6]. An example of this graphical representation is shown in Figure 3. As we can see, each rectangular box represents a single definition-use track for a specific variable or resource. Each vertical column of boxes represents all tracks for that one variable. Thus, the basic representation is capable of showing tracks for only five variables in one basic block. Of course, a block will, in general, have tracks for more than five variables. The pictorial representation is meant only to demonstrate patterns, not represent every track in a block. This representation distinguishes between four types of definition-use tracks:

1. A box open at the top represents a definition-use track that is *used-before-defined*. Thus, only a set of uses is actually included in the block depicted. The definition of such a track (for

example, the box containing only the use node “a”) can be found in a previous block.

2. A box open at the bottom represents a track when only the definition is in the block depicted. In this case, the use(s) are found in a subsequent block of the program. The boxes containing only the nodes “j” and “k” are examples of such *live_out* tracks.
3. A completely enclosed box represents a definition-use track where the definition and all uses are included within the block being depicted. Examples in Figure 3 are the boxes containing 1) the definition “d” and use “e”, 2) the definition “f” and the use “g” and 3) the definition “h” and the use “i.”
4. A box without bottom or top but filling the entire space for one variable’s tracks is called a *spanning track*. This represents a track (value) that passes through the block represented by the picture. In such a case the track definition is found in a previous block and all uses are in subsequent blocks.

Note that even though the pictorial representation uses only a single letter for the use node(s), a single livetrack may include many use nodes. However, a track can contain only a single definition.

Coupling is performed with specifically two input DDDs. We call the two input DDDs “top” and “bottom” for reasons that will soon be apparent. Coupling is needed to combine two DDDs when at least one operation of one of the input DDDs (bottom) is data dependent upon an operation in

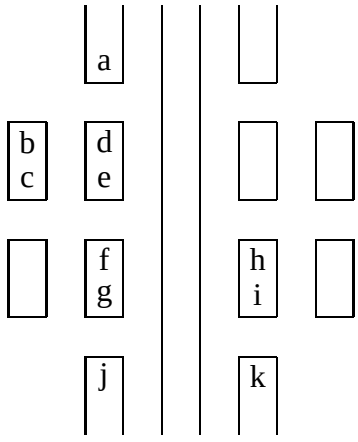

Fig. 3 Sample Definition-Use Tracks

figure

the other input DDD (top). Such a data dependency determines an execution order between the DDDs; bottom must follow top. Coupling can be thought of as adding DDD bottom onto DDD top, indicating that for any variable that has livetracks in both DDDs, those livetracks associated with top must precede those associated with bottom in the resultant DDD. In the coupling algorithm described below, the result of the coupling of DDDs top and bottom will be a DDD, result. The coupling algorithm is outlined in Figure 4.

D. Instruction Scheduling

Instruction scheduling forms the basis for any compiler for ILP architectures, and Rocket provides both local scheduling and two different forms of global scheduling, dominator-path scheduling (DPS) and a derivative of trace scheduling called sliding-window trace scheduling (SWTS) [47, 70]. In addition to both local and "general" global scheduling, Rocket includes software pipelining to deal specifically with generating efficient schedules for loops. The combination of these four techniques affords Rocket power and flexibility in scheduling not found in many ILP compilers.

1. Make a copy of DDD top and call the copy result. This allows the algorithm to couple top and bottom without altering either input DDD.
2. For each variable ("var") in bottom that has a livetrack, perform the following case statement.
   (a) If var is *used_before_defined* in bottom and *live_out* in top, then we need to combine the *use* fields of the first track (for var) in bottom with those of the last track (for var) in result. This is reasonable since var is *used_before_defined* in bottom, it must therefore get its first value from the previous DDD, top in this case. We have now processed only the first of possibly many tracks for var in bottom. For each subsequent var track in bottom, copy the track and append the copy to the end of the livetracks for var in result. (see Figure 5 for a pictorial view of this case.)
   (b) If var is *used_before_defined* in bottom and top has no livetrack for var, then copy all of var's tracks in bottom to result. (see Figure 6 for a pictorial view of this case.)
   (c) If var is not *used_before_defined* in bottom but has associated livetrack(s), then append all such tracks to result, placing them after any livetracks result already has for var. (see Figure 7 for a pictorial view of this case.)
3. Now, having properly coupled the livetracks of top and bottom in steps 1 and 2 above, we need to ensure that result includes the proper information for the *used_before_defined*, *live_out*, *use*, and *def* fields of the DDD structure. To do this we need only perform set union between each of these fields in bottom and top, placing the union in result. Finally, the node lists for the two DDDs are also unioned.

Fig. 4 Coupler Algorithm

figure

D.1 Local Scheduling

Rocket's local scheduler, which is itself the basis for Rocket's global scheduling techniques, is based on *list scheduling*. List scheduling derives its name from the fact that a list of *data-ready* nodes is maintained and the next node to schedule is heuristically chosen from the data-ready set (DRS.) Figure 8 gives the basic list scheduling algorithm.

The decision to adopt list scheduling as Rocket's scheduling paradigm leads to several practical considerations. The first is that we need to define heuristics by which nodes will be prioritized within the DRS. Because list scheduling uses heuristics to prune areas of the search space that appear uninteresting, the heuristics must be chosen with great care so unsearched spaces are truly uninteresting. The choice of giving one operation higher priority than another can have great influence on the final schedule. This is particularly true in the presence of multi-cycle operations. If an operation with a long latency is scheduled late, it may have a large negative influence on the overall schedule length by serializing the code.

A heuristic often cited [5, 53, 69, 78] as one necessary for efficacious list scheduling is that of *critical path*. A critical path in a DDD is defined to be a longest path from any of the roots to any of the leaves [40]. This definition is correct for unweighted DDDs, that is, those whose edges are of unit length. With the introduction of weights on the edges of the DDD, the definition must be slightly modified. A *schedule-critical path* is one with the greatest sum of the edge weights from all the roots to all the leaves.

Developing a set of heuristics that attempts to produce a valid schedule for a valid DDD can be challenging. Further, the most useful heuristics for assuring validity during list scheduling vary from architecture to architecture. This results from architecture-dependent features: one may have a restrictive branch delay while another may have a synchronous pipe that does not latch its output. Differing machine features make the generation and reuse of heuristics difficult when a machine-independent scheduler is desired.

We have investigated many heuristics in achieving valid schedules for a variety of architectures. Typically, considerable testing goes into choosing heuristics for a particular target, but some heuristics, such as critical path are almost always used. Allan and Mueller describe a *discriminative polynomial selection* [10] mechanism to to combine multiple heuristic factors into one heuristic ranking. Here is a list of heuristics that have been useful for at least one architecture we have

targeted:

1. `height`.
2. `schedule_height`.
3. `on_critical_path`.
4. `on_schedule_critical_path`.
5. `lexical_order` – ordering of nodes from source. Fisher [37] shows that program lexical order is not a good metric for list scheduling priority, our experience agrees with this. This can be used for non-ILP architectures for a default ordering.
6. `branch_node` – the node is a branch node, especially useful in the presence of delayed, restricted branching mechanisms. This has been used to increase the chance a branch node will be placed before other operations.
7. `schedule_spread` – the number of instructions an operation can be placed. The greater the spread, the more flexibility for placement.
8. `resource_usage_of_this_type` – the amount of use of this node's resource in this DDD. The more contention for resources, the earlier a node should be placed in order to free the resource as soon as possible for reuse.
9. `used_and_defined_resources` – as above, nodes that use more resources than others should be scheduled so they do not interfere with others needing those resources.
10. `least_recently_used_resource` – a method of forming round-robin reference to resources.
11. `field_usage_of_this_type` – as with resources, try to minimize field conflicts.
12. `fields_used` – as supra.
13. `least_recently_used_field` – as supra.
14. `successors` – the more successors a node has, the earlier it should be scheduled, allowing its successors to become data ready as early as possible. This exposes more parallelism to the scheduler.
15. `restricted_successors` – the more restricted successors a node has, the earlier it should be scheduled so timing is more flexible within the DDD. Once timing becomes increasingly limited, restricted successors become harder to place.
16. `total_restricted_successors` – total of $\Delta(e)$ for all restricted successors.
17. `shortest_restricted_successor` – restricted successors having a smaller $\Delta(e)$

reduce flexibility, and therefore the possibility for valid scheduling, diminishes.

18. `distance_from_succs` – a measure of how restricted the edges to the successors are.
19. `average_restricted_successor` – the average of $\Delta(e)$ for all the restricted successors.
20. identical heuristics for predecessors as 14 to 19 for successors.

Once the factors that will make up the heuristic evaluation have been chosen, we still need to address the issue of when to update the priority weightings for the DDD nodes. Two possibilities exist:

1. calculating the weights once, before the list scheduling algorithm begins (denoted *static* weighting), and
2. calculating after each node is placed in an instruction (denoted *dynamic* weighting).

Certainly, the first method requires the least computation time. It also gives a good estimate of the overall priorities present in the DDD. Its difficulty is that a DDD does not remain static throughout the scheduling process. As operations are placed into instructions, they are removed from the DDD, changing its shape and makeup. This is not reflected in the priorities if they are calculated only once. Of the heuristics listed above, 1, 3, 5, 6, 7, 8, 11, and 12 are not dynamic values and so if they are the only ones used, there is no need to compute the priorities dynamically.

One important heuristic that can change during scheduling is critical path. If nodes are scheduled from a critical path, chances are favorable that the path will become shorter than another remaining in the DDD. The decision as to when to generate the priorities on the nodes is one that must be considered carefully when producing a list scheduler. Empirical results usually drive the decision; if the static method works well, no reason exists to use the dynamic method. We have found that the dynamic method can add up to 50% more time in the scheduling phase. Rocket supports both dynamic and static heuristic evaluation. A command-line argument specifies which method to use for any individual compilation.

The direction a scheduler traverses a DDD can also have a large influence on the final schedule length. Thus far, forward traversals have been assumed. To schedule in the backwards direction, no changes are required to the algorithms thus far discussed. The change occurs exclusively in the data structure representing the DDD. Here, all the sources become sinks and vice versa, all the predecessors edges become successor edges and vice versa, and all the operations are placed in instructions beginning at the end of the schedule. Upon completion of scheduling, the instruction

list is reversed to reproduce the original program semantics.

Allan and Mueller [9] note that one direction will succeed in producing a valid schedule in a given architecture much more often than the other. Bias is predicted on the presence of restricted timing within the DDD. Reasons for this bias include:

1. Presence of restricted branch delays. If branches in an architecture have (n, m) (or more likely (n, n)) timing to the end of the DDD, there exist few (or one) instructions in which they may be placed. Reverse traversal will tend to place this type of operation in the correct instruction early in the scheduling process, increasing the chance for a valid schedule. Increasing the branch node's scheduling priority is another way of achieving this end.
2. Presence of restricted machine pipe stages. As above with branch delays, (n, m) pipe stages can cause failures when a pipe operation is not instantiated at the proper time. Traversal direction is dependent on whether
   (a) the inputs of the pipe are latched,
   (b) the outputs of the pipe are latched,
   (c) both are, or
   (d) neither are.
3. Use of transient condition code registers within the DDD.

It is also possible for the direction to have an effect on the length of the final schedule without considering the impact of restricted edges. The reason is simple: direction has an impact on the order nodes are chosen to be placed. This is because the formation of the data-ready sets differs between the two directions. It may be worthwhile to attempt both and choose the shorter. If one direction fails to produce a valid schedule, the other direction certainly should be tried. Rocket includes a flag to specify whether forward, reverse, or both directions of scheduling is to be performed.

In the context of building a list scheduler, it is convenient to associate both relative times and absolute times with each DDD node. While relative times (decorated on the DDD edges) indicate the timing relationship between two DDD nodes, the absolute time of a node indicates the possible range of instructions into which that DDD node might be scheduled. An operation is said to be *data-ready* if all of its predecessor operations have been scheduled. Similarly, an operation is said to be *timing ready* if it is data-ready and the minimum edge time, between each of the node's predecessors and the node, has elapsed.

Absolute timing utilizes the relative timing information and the location of previously scheduled nodes. If O_m has absolute timing (min_m, max_m), O_m may only be placed at an instruction with indices between min_m and max_m inclusive. Though min_m always has a finite value, max_m is often initially infinite, due to the infinite relative times of register resources. As predecessors are placed, the absolute time interval shrinks as min_m increases and max_m decreases. When O_m is placed at I_k, (min_m, max_m) is changed to (k, k). If a node has absolute timing such that $min_m > max_m$, scheduling fails as the node cannot be placed; no placement can satisfy the conflicting absolute timing requirements of other DDD nodes.

A new absolute timing pair for node m is computed having the largest possible range that satisfies the current absolute timing and the edge timing. Then, the old absolute timing and the new absolute timing are combined by using the intersection of the two ranges. The maximum of the minimum times and the minimum of the maximum times provides this overlap in range. The absolute timing algorithm is first used to produce initial absolute timing. As each operation is placed in an instruction during scheduling, the absolute timing algorithm propagates the effects of the placement by restricting timing assignments.

Note that relative times are assigned by the code generator, whereas the absolute times are assigned solely as an aid to scheduling. Relative times are determined from a small amount of contextual information; absolute times are assigned after examining the entire graph and are updated as operations are placed.

Another concern when scheduling is the possibility that scheduling will fail. This is a concern because we are using a non-exhaustive method and "difficult" timing may lead to scheduling failures. This is especially true when considering edges that represent restrictive timing (non-infinite max times). Several techniques have been suggested to decrease the risk of scheduling failure. Rocket makes use of the absolute timing computation to support a technique called *lookahead* scheduling. Lookahead scheduling relaxes the practice of scheduling a DDD node *only* after all of its predecessors have been scheduled, and instead uses computed absolute times to schedule DDD nodes at any point during scheduling in which a node's latest absolute times becomes non-infinite.

The foundation upon which lookahead scheduling is based is the fact that edges in a DDD only limit the ordering in the final schedule, not the order in which the schedule is created. So long as the partial order is preserved, the order of placing the nodes is irrelevant. The absolute timing

algorithm specifies the range in the final schedule where an operation can be placed. Because the lookahead routine examines instructions in this range for node placement, if lookahead succeeds in finding a valid place for an operation, then that placement will be valid in the final schedule. An alternative view is that not only *can* a node be placed where absolute timing predicts, it *should* be placed there. The original motivation for lookahead was to increase both the speed and the chances of creating valid schedules for a stochastic scheduling method [16]; it was then noticed it could speed up generic list scheduling as well.

Several minor changes to list scheduling with lookahead need to be noted. First, the definition of data-ready does not change, it is still those nodes in the graph that have no unscheduled predecessors. The computation of these nodes might be different. It is no longer enough to remove nodes from the DRS when they are placed by list scheduling; one must also add and remove nodes in the DRS based on those that lookahead places. Lookahead can remove any or all the nodes in the DRS; it can also make nodes further down in the graph data-ready by placing all their predecessors. The scheduler must also ignore all the nodes that are placed by lookahead during later stages of the scheduling process. Both of these conditions are handled in the compiler by the addition of a flag in the nodes that state whether or not the node has been placed, either by list scheduling or by lookahead. It is also important for lookahead to check nodes in a breadth-first manner so that no cycles develop during the procedure.

A decision must be made as to whether to pack only the nodes with $O_m = (x, x)$, or additionally to pack the nodes with $O_m = (x, y) when x < y < \infty$. In the first case, no choice exists as to when to pack the nodes, they must be placed in instruction x. The second case contains more flexibility and requires the analysis of a tradeoff. Having lookahead place them will result in a larger chance of generating a valid schedule. However, if lookahead does not immediately place them, the scheduler may be able to produce a more compact final sequence.

This tradeoff varies with the flexibility in the operation's schedule range in the current DDD, and in the architecture, making it difficult to analyze the tradeoff universally. For example, if the current DDD is wide (displaying a lot of parallelism), constrained operations might need to be placed immediately so that other parallel operations do not consume all needed resources. For machines with a large amount of available parallelism, final placement should probably be deferred, allowing the most flexibility for the scheduler. Placement decisions made between the time of finite constraint and final packing are less likely to have a deleterious effect as there is more "room" in

each instruction for operations in these types of architectures. A heuristic based on the range for any node can be tuned on a per-machine basis to control the amount of lookahead.

Note that any failure to place a restricted node with lookahead would result in a failure later in the scheduling process, thereby reducing the time spent scheduling an infeasible schedule. Lookahead also schedules nodes without having to topologically sort them. By doing so it reduces the number of nodes list scheduling must deal with and thereby increases the speed of scheduling.

Also note that naïve lookahead places nodes in the final schedule non-heuristically. That is, there is no order in examining the constrained nodes based on node weights built into lookahead. While this expedites the process, lookahead could be extended to deal directly with differing priorities in the constrained node set so that the final schedule length is optimized.

It is important to understand that using lookahead with list scheduling does not guarantee that no timing failures will occur; it only lessons the chances of encountering such failures. There still is the possibility that valid DDDs exist that cannot be scheduled due to poor choices made by the node priority heuristics. This is an inherent problem when only searching a small subspace of the possible solutions.

D.2 Sliding-Window Trace Scheduling

Rocket's trace scheduling is an implementation of the "sliding window" technique of Howland [46, 47]. The sliding window trace scheduling method (henceforth SWTS) is an adaptation of the "traditional" trace scheduling described in Section IV-B.1 and Ellis's dissertation [33]. SWTS was designed with two overriding goals:

- Reduce the time required to schedule large traces.
  Since list scheduling has time complexity $\geq (c * n^2)$, where n is the number of nodes in the DDD being scheduled, scheduling large DDDs can lead to a significant increase in the time required to schedule.
- Reduce the number of bookkeeping copies required.
  Trace scheduling's bookkeeping produces multiple copies of any operation that moves in relation to the branches of a program. Since the number of such copies produced can grow dramatically when operations move past several branches or when branches move in relation to one another, a method of limiting such copies is desirable.

To achieve these goals, SWTS schedules a trace of N blocks by performing $N - 1$ pairwise trace scheduling steps, with each step scheduling two successive blocks in the trace. As implied by the term "sliding window", the first pairwise step schedules the first two blocks on the trace, the

second step schedules the second and third blocks, and so on. More formally, to schedule a trace of N blocks, SWTS proceeds as follows:

FOR i = 1 to N - 1

trace schedule blocks B_i and B_{i+1}

modify block B_{i+1} so that it includes all operations not scheduled in block B_i

ENDFOR

SWTS achieves its goal of reducing the time required for scheduling because it schedules two blocks at a time. This leads to faster scheduling because the DDDs scheduled are smaller than those required for an entire trace. While this is a useful feature, it is not as important as the ability to reduce the number of bookkeeping copies required. Since Rocket is currently targeted to superscalar architectures with limited to moderate levels of instruction-level parallelism, the gain we expect from scheduling several blocks as one is much more limited than for architectures such as VLIW machines that have substantially more instruction-level parallelism. This means that for trace scheduling to produce an improvement over local scheduling, trace scheduling should severely limit the number of bookkeeping copies required. SWTS does this by not allowing branches to move in relation to other branches in the program. In addition, since trace scheduling is performed in a pair-wise fashion, the tendency of operations to move over several branches is reduced.

To perform a pair-wise trace scheduling step, SWTS performs the following steps:

1. Combine the DDDs for the two blocks to schedule into a single DDD.
2. Insert data dependences (DDD arcs) to prevent any inter-block motion that might change the program semantics beyond bookkeeping's ability to compensate.
3. Schedule the combined DDD.
4. Assign the instructions of the scheduled DDD to the appropriate blocks.
5. Add any required bookkeeping copies.

To satisfy these requirements, SWTS relies on the insertion of special DDD nodes to limit illegal motion and makes extensive use of the *coupler/ decoupler* (See Section VI-C.1) to combine and split DDDs.

In the following discussion of SWTS's pair-wise trace scheduling method, the two blocks in the window being scheduled will be referred to as **U** (upper block) and **B** (bottom block). **U**'s off-trace successor will be denoted **OT**.

The basis for pair-wise trace scheduling is the insertion of a special *pivot node* in the combined DDD for **U** and **B**. This pivot node is used to prevent illegal motion as well as to separate scheduled code between **U** and **B**, and to determine when bookkeeping copies are required. To show how SWTS uses the pivot node (*PN*) in pair-wise trace scheduling we revisit the list of steps above:

1. **Combine the DDDs for the two blocks to schedule into a single DDD.**
   SWTS uses the DDD coupler to a) couple the DDD for **U** with the "Pivot DDD" which is a DDD containing the single node, PN, and then b) couple the result of a) with the DDD for **B**. This yields a single combined DDD in which PN represents the "boundary" between the DDDs for **U** and **B**.

2. **Prevent any inter-block motion that might change the semantics of the program being scheduled.**
   Given an operation, O, in a DDD that defines *x* and uses *y*, O can be moved between **U** and **B** subject to the following restrictions.
   - O cannot move from **U** to **B** if **B** has multiple predecessors and either *x* or *y* is live-in to **B**. To prevent such an illegal inter-block motion, SWTS identifies each node, D , in **U** that represents the last definition of *x* or a use of *y*. SWTS then adds a dependence arc between each such D and PN. Since PN represents a boundary between **U** and **B**, these arcs prevent the illegal motion into **B**.
   - O cannot move from **B** to **U** if *x* is live-in to **OT**. To prevent such an illegal inter-block motion, SWTS adds a dependence arc between PN and that node which represents the first definition of *x* in **B**, thus preventing O's motion into **U**.

3. **Schedule the combined DDD.**
   SWTS simply calls the local scheduling routine for the combined DDD.

4. **Assign the instructions of the scheduled DDD to the appropriate blocks.**
   SWTS traverses the scheduled instructions for the combined DDD, assigning those that come before the instruction that includes PN to **U** and the rest to **B**. Since those DDD nodes currently assigned to **B** will subsequently be part of the next pair-wise trace scheduling step, SWTS alters the combined DDD by removing those DDD nodes that are scheduled into **U**. It does this by using the DDD *decoupler*. After removing **U**'s nodes the resulting combined DDD is ready for the next pair-wise trace scheduling step.

5. **Add any required bookkeeping copies.**
   Any node that moves from **U** to **B** or from **B** to **U** during the pair-wise scheduling potentially needs to be copied by bookkeeping. Rocket thus maintains sets of those nodes that moved during scheduling and calls the bookkeeping phase.

As previously emphasized, the distinguishing feature of SWTS is the highly reduced number of bookkeeping copies required. For each pair-wise schedule, at most two new blocks must be inserted into the control flow graph, one to copy operations moving from **U** to **B** and another for copies of operations moving from **B** to **U**. Quite often, no bookkeeping copies are required even when operations move across block boundaries. For example, operations moving from **U** to **B**

need not be copied if **B** is **U**'s only successor. Similarly, when operations move from **B** to **U** no bookkeeping is required if **U** is **B**'s only predecessor.

D.3 Dominator-Path Scheduling

In their 1981 paper, [64] Reif and Tarjan provide a fast algorithm for determining the *approximate birthpoint*s of expressions in a program's flow graph. An expression's birthpoint is the first block in the control flow graph where the expression can be computed while guaranteeing the value computed will be the same as in the original program. Their technique is based upon fast computation of the *idef* set for each basic block of the control flow graph. The *idef* set for a block, **B** is that set of variables that are defined on some path between **B's** immediate dominator and **B**.

Reif and Tarjan's expression birthpoints are not sufficient to allow us to safely move entire operations from a block to one of its dominators because they address only the movement of expressions, not definitions. Operations in general include not only a computation of some expression but the assignment of the value computed to a program variable. For an operation $A \leftarrow E$, in addition to computing the birthpoint of the right-hand-side expression, we must concern ourselves with the variable being assigned to as well. To ensure a "safe" motion for an expression, we need only ensure that no expression operand move above any *possible definition* of that operand, thus changing the program semantics. We need to make a similar requirement for the variable being assigned to, but we must do more. As well as not moving A above any previous definition of A, we must ensure that A does not move above any $possible\ use$ of A. Otherwise, we run the risk of changing A's value for that previous use. Thus, dominator analysis computes the *iuse* set for each basic block as well as the *idef* set. Using the *idef* and *iuse* sets, dominator analysis computes an approximate birthpoint for each operation in a function being compiled.

To measure the motion possible in C programs Sweany [70] defined *dominator motion* which moved each intermediate statement to its birthpoint as defined by dominator analysis and counted the number of dominator blocks each statement jumped during such movement. Sweany's choice of intermediate statements is attributed to the lack of machine resource constraints at that level of program abstraction. He envisioned dominator motion as an upper bound on the motion available in C programs when compensation code is not included. In the test suite of twelve C programs compiled, more than 25% of all intermediate statements moved at least one dominator block upwards towards the root of the dominator tree. One function allowed more than 50% of the

statements to be hoisted an average of nearly eight dominator blocks. This considerable amount of motion (without copies) available at the intermediate statement level of program abstraction provides motivation for using similar analysis techniques to facilitate global instruction scheduling.

Given dominator analysis information, we would like to use it to perform global instruction scheduling without compensation copies, much like dominator motion used the information to move intermediate statements. Dominator-path scheduling (DPS) is, in fact, such a use of dominator analysis. DPS performs global instruction scheduling by treating a group of basic blocks found on a dominator tree path as a single block, scheduling them as a whole. In this regard it resembles trace scheduling which schedules adjacent basic blocks as a single block. Unlike trace scheduling, however, DPS requires no semantic-preserving operation copies, to "clean up" effects of code motion. DPS's foundation is scheduling instructions while moving operations among blocks according to both the opportunities provided by and the restrictions imposed by dominator analysis.

The question arises as to how to exploit dominator analysis information to permit code motion at the instruction level during scheduling. DPS is based on the observation that we can use $idef$ and $iuse$ sets to allow operations to move from a block to one of its dominators during instruction scheduling. This allows instruction scheduling to choose the most advantageous position for an operation that we might "legally" place in any one of several blocks. Because machine operations are incorporated in nodes of the data dependence DAG (DDD) used in scheduling and, like intermediate statements, DDD nodes are represented by def and use sets, the same analysis performed on intermediate statements can be applied to a basic block's DDD nodes as well. One could say that DPS is, in fact, an instantiation of dominator motion where instruction scheduling provides the heuristics used to determine which of (potentially) several possible dominator blocks to move an operation to.

The same motivation that drives trace scheduling — namely that scheduling one large block allows better use of machine resources than scheduling the same code as several smaller blocks — applies to DPS as well. In addition, because DPS does not allow motion of DDD nodes when a copy of that node would be required, it does not incur the potential code explosion due to copies that trace scheduling can produce. On the other hand, the restrictions to operation movement inherent in DPS will probably limit movement more than trace scheduling does, so we can expect DPS to find less available parallelism than trace scheduling. Still, for architectures that exhibit a moderate amount of instruction-level parallelism, DPS may well produce better results than trace scheduling,

because the more limited motion may be sufficient to make good use of machine resources and, unlike trace scheduling, no machine resources will be devoted to execution of semantic-preserving operation copies.

Much like $traces$ (groups of blocks to be scheduled together in trace scheduling), the dominator path's blocks can be chosen by any of several methods. Heuristically choosing a path based on length, nesting depth, or some other program characteristic is one method. Allowing the programmer to specify the most important paths is another. Actual profiling of the running program is a third.

Once we select a dominator path to schedule, we need a method of combining the blocks' DDDs into a single DDD for the entire dominator path. In Rocket this task is performed by the DDD coupler (See Section VI-C.1) that is designed for just such a purpose. Given the DDD coupler, DPS proceeds by repeatedly

- choosing a dominator path to schedule,
- using the DDD coupler to combine each block's DDD on the dominator path so chosen, and
- scheduling the combined DDD as a single block.

Of course, this short description of DPS, while providing a basic notion of how DPS works, does not include detail of how "appropriate" inter-block motion of DDD nodes will be allowed while "illegal" motion is prohibited.

Because the combined DDD for a dominator path includes control flow we need, when scheduling a group of basic blocks represented by a single DDD, some mechanism to map the scheduled instructions to the correct basic blocks. We can easily accomplish this by adding two special nodes to each block's DDD. These nodes are called BlockStart and BlockEnd. They represent the basic block boundaries. Since BlockStart and BlockEnd are nodes in the eventual combined DDD, they are scheduled just like all the other nodes of the combined DDD. After scheduling, all scheduled instructions between the instruction containing the BlockStart node for a block and the instruction containing the BlockEnd node for that block will be considered instructions for that block. The only remaining chore is to ensure that the BlockStart and BlockEnd DDD nodes remain ordered (in the scheduled instructions) relative to one another and to the BlockStart and BlockEnd nodes for any other block. To do so, we add use and def information to the nodes to represent a pseudo-resource, BlockBoundary. By initializing each BlockStart node to define BlockBoundary and each BlockEnd node to use BlockBoundary, we ensure that no BlockEnd node can be scheduled ahead

of its associated BlockStart node (due to *flow dependence*.) We also ensure that no BlockStart node can be scheduled before its dominator block's BlockEnd node (because of *anti-dependence*). By establishing these imaginary dependences, DPS ensures that the DDD coupler will add arcs between all BlockStart and BlockEnd nodes.

When combining two DDDs, the coupler adds arcs for any data dependences that exist between the two DDDs. DPS uses this coupler feature by adding "dummy" nodes to the DDD for each basic block. These dummy nodes include uses and/or definitions of appropriate dataflow values to prevent any potentially hazardous motion of DDD nodes across block boundaries within the DDD being scheduled. When the basic blocks of a dominator path are combined, the coupler automatically inserts those arcs necessary to prevent nodes' "illegal" motion from one block to another. Looking back to dominator analysis we see that interblock motion is prohibited if the operation being moved defines something that is included in either the *idef* or *iuse* set or 2) uses something included in the *idef* set for the block where the operation currently resides. To obtain the same effect in the combined DDD we again turn to using the use and def sets for the BlockStart nodes. By adding the $idef$ set for a basic block B to the def set B's BlockStart node, and similarly adding the $iuse$ set for B to the use set of B's BlockStart node, we can enforce the same restriction on movement that dominator analysis imposed upon intermediate statements, thus ensuring that any interblock motion preserves program semantics.

One requirement of dominator analysis which does not exist in DPS is the need to update the $idef$ and $iuse$ sets when operations are moved from a block to one of its dominators. This is necessary when statement motion completes the motion in one pass over the entire control flow graph for a function. DPS schedules an entire path, then repeats dominator analysis for the whole flow graph before scheduling another path. Of course, in adopting such a policy, DPS incurs the compile-time cost of performing dominator analysis for each path scheduled. Still, because the analysis is quite fast, this extra cost will not be significant when compared to the extra time required to schedule the larger combined blocks.

DPS is complicated by some factors that are not relevant for dominator motion of intermediate statements. Foremost is the added complexity imposed by the bidirectional motion of operations that instruction scheduling allows. In dominator motion, intermediate statements moved in only one direction — towards the top of the function's control flow graph. There was no concept of a statement moving from a dominator block to a dominated one. This is reasonable when attempting

to move intermediate statements as one statement's movement will likely open possibilities for more motion in the same direction by other statements. If, however, statements move in different directions, one statement's motion might well inhibit another's movement in the opposite direction. During instruction scheduling, however, we are less interested in moving operations as far as possible than we are in using the fewest number of instructions to schedule the available operations. Thus our goal has changed. To gain the full benefit from DPS, we wish to allow operations to move past block boundaries in either direction. To permit bi-directional motion, we use the post-dominator relation, which says that a basic block **PD** is a post-dominator of a basic block **B** if all paths from **B** to the function's exit must pass through **PD**. Using this strategy, we similarly define *post-idef* and *post-iuse* sets. In fact, it is not difficult to compute all these quantities for a function. The simplest way is to reverse (logically) the direction of all the control flow graph arcs and perform dominator analysis on the resulting graph. Having computed the post-dominator tree, we wish to choose dominator paths such that the dominated node is a post-dominator of its immediate predecessor in a dominator path. This allows operations to move "freely" in both directions. Of course, this may be too limiting on our choice of dominator paths. To allow for the possibility that nodes in a dominator path will not form a post-dominator relation we need a mechanism to limit bi-directional motion when we must. Again, we rely on the technique of adding dependences to the combined DDD. In this case (assuming that we are scheduling paths in the forward dominator tree), for any basic block, **B**, whose successor, **S**, in the forward dominator path does not post-dominate **B**, we add **B**'s def set to the use set of the BlockEnd node associated with **B**. In similar fashion, we add **B**'s *use* set to **B**'s BlockEnd node's *def* set. This will prevent any DDD node originally in **B** from moving downward in the dominator path. Since this addition of dependences will hamper instruction scheduling's ability to find a good schedule for the combined DDD, we usually wish to choose dominator paths to schedule such that the paths chosen are inverses of some path in the post-dominator tree.

One other consideration in DPS that did not occur in dominator motion of intermediate statements is the requirement that we not allow any transient DDD arcs to traverse a block boundary. A transient DDD arc (one whose max time is less than infinity) of value n indicates that the successor node must follow the predecessor node by no more than n instructions. It is quite reasonable (and relatively easy) to schedule nodes having these restrictions within basic block boundaries. In general, however, we cannot allow the predecessor of such an arc to be placed in one block

when the successor is placed in another. This is because we do not, in general, know how many instructions (or, indeed, how many basic blocks) may exist in the control flow path between any basic block and its successor in a specific dominator path. To guarantee that no transient arcs will cross a basic block boundary in the final schedule for the combined DDD, DPS inserts "full" dummy instructions between the BlockEnd node of one block and the BlockStart of the next in the dominator path. "Full" in this context means that we specify the instructions in such a way that no real node can be placed there.

By always adding such dummy nodes to the DDD, we are being more conservative than necessary. In many cases we will know that the successor of a specific block in the dominator path is also an immediate successor in the control flow graph. For those blocks that are immediate flow graph successors of their dominator, we can allow transient timing across the block boundary because we know that no intervening instructions will be executed between leaving the dominator and entering the dominated block. In these cases, no dummy nodes need be added to the transition DDD. The dominator-path scheduling algorithm is summarized in Figures 9 and 10.

D.4 Software Pipelining

Rocket's software pipelining, patterned after Warter's *enhanced modulo scheduling* [76], uses a modulo scheduling algorithm. While Warter's method provides a general framework for our software pipelining, the actual modulo scheduling technique implemented in Rocket closely follows Rau [62]. Modulo scheduling assumes that a single data-dependence graph (DDG) can be built for a loop. To build a single DDG for a loop requires some method of treating the entire loop as a single basic block. Like Warter, we use IF-conversion to transform a loop with arbitrary control flow into a single block and then construct the DDG for that "super-block" using standard dependence analysis. Since we intend to overlap different loop iterations we need to consider loop-carried dependence as well as loop-independent dependence, but well-known algorithms provide this information [51, 41].

E. Register Assignment

As discussed in Section V-B, the relative placement of register assignment and instruction scheduling has profound consequences on the quality of code generated. Performing register assignment first restricts the freedom available to scheduling, leading to less efficient schedules, while schedul-

ing first typically increases register pressure and thus can lead to spilling in functions for which register assignment before scheduling requires none. Rocket supports both early register assignment (register assignment before scheduling) and late register assignment (assignment after scheduling). In addition, Rocket provides a "combined" register-assignment/scheduling technique called *Combining Register Assignment Interference Graphs* (CRAIG) [22].

CRAIG mediates the "tug-of-war" between register assignment and instruction scheduling by providing a mechanism to decrease anti-dependences (thus increasing scheduling freedom) even to the extent of adding spill code. CRAIG incorporates this "mediation" as a schedule cost considering both schedule efficiency and register pressure. This schedule cost is a heuristic designed to meet the goals of the code generator. If the initial schedule cost is too high, CRAIG goes back to the original linear code and attempts **early** register assignment. The intuition is that the original code will have a less busy interference graph and will therefore have a lower cost due to register pressure. If this schedule cost is still too high, CRAIG accepts this schedule based upon the assumption that it is the best that we can do under the circumstances. If, however, the schedule cost is not too high it is likely that anti-dependences have been added, and thus, the schedule can be improved. CRAIG will attempt to reclaim some of this lost efficiency by removing as many of these anti-dependences as possible, up to the point where the schedule cost is too high. By adding edges found exclusively in the late interference graph CRAIG is adding interferences edges between two values that the scheduler forced to be in different registers and are thus "tuning" the added parallelism to that actually extracted by our instruction scheduler. Rocket's current implementation of CRAIG, that could be called CRAIG_0, attempts to generate the best schedule it can without generating spill code. Therefore, a schedule cost is determined to be "too high" when spill code is inserted. CRAIG_0 results [22] show a significant improvement over either early or late register assignment alone.

Another register-assignment issue related to function calls is how and when registers will be saved and restored around function calls. The two basic techniques for saving and restoring registers are *caller-save* and *callee-save* conventions. As the name implies, in the caller-save convention the compiler will identify those registers in the caller function that need to retain values across the function call, and insert code (in the caller function) to perform the save and restore operations. In contrast, the callee-save convention requires each function to save all registers it will use at the entrance to the function and then restore the original values at the end of the function. Rocket explicitly supports callee-save by including (in the configuration file) information about how to

save and restore registers (by the callee.) However, caller-save can be implicitly supported by Rocket as well, merely by including no specification for (callee) save/restore in conjunction with indicating that *all* registers are redefined by each function call. A popular hybrid solution is to partition the register set into a group of caller-saved and a group of callee-saved registers, in hopes of getting the best of both conventions. Rocket supports this technique as well.

VII. Conclusions

In this paper, we've discussed many issues involved in building an efficacious retargetable compiler for instruction-level parallel architectures. Many traditional optimizations compete and conflict when one considers applying them to ILP machines. We have spent considerable effort analyzing how to add to and modify these optimizations so that they continue to produce better code for a variety of architectures. Among the issues we have found critical are:

Modification of the traditional optimizations. By slightly altering traditional optimizations such as common subexpression elimination and copy propagation to consider details of the target architecture, we can tune these traditional optimizations to the specific target.

The target machine description. This is an important aspect of generating good code as it is used as the basis of all phases of compilation. The description must be rich enough to encompass all features in all the targets. Rocket can generate code for a number of different types of architectures, so we feel good about the richness of our language. Using machine resources, and adding relative times between resources has proven not only to be effective in generating efficient code, but it is a strong foundation that we can build many code-improving transformations on.

Machine-independent code selection. Our method of coupling DAGs together not only solves the machine-independence problem, it also is the basis for the powerful global instruction scheduling algorithms Rocket implements.

Local instruction scheduling. At the heart of producing efficient code for ILP architectures is instruction scheduling. We have put a lot of work into making this as fast and as effective as possible. The generation of a valid final schedule given a valid DDD has received a lot of attention. The tuning of the plethora of heuristics, the development of the lookahead method, and the use of the absolute timing algorithm are examples of this work.

Global instruction scheduling. In conjunction with local scheduling, global scheduling plays an important role. The ability to place operations in a variety of basis blocks can greatly increase the run-time speed of a program. Rocket can perform several global methods, and is based on both our DDD representation of the program, and our local instruction scheduler.

Register assignment can have a large influence on the final program, so we have spent a lot of time improving our methods. Rocket can assign registers at a number of different times during compilation. We have also studied methods to help mitigate the deleterious interaction register assignment can have with instruction scheduling.

We have described the approaches we use in the Rocket compiler to address these issues, and recommend them to those planning to build an optimizing compiler that is retargetable to many ILP architectures.

Acknowledgments

We would like to thank the following graduate students who have aided in the development of Rocket: Brett Huber, Chen Ding, Tom Brasier, and Mike Bourke. We would also like to thank the National Science Foundation for helping to fund this work through Grant CRR-9308348.

References

[1] B. R. Rau and J. A. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective," *The Journal of Supercomputing*, vol. 7, pp. 9–50, 1993.

[2] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 1990.

[3] A. Levinthal and T. Porter, "Chap: A SIMD graphics processor," *Computer Graphics*, vol. 18, no. 3, 1984.

[4] Evans and Sutherland, *The Breadth of Visual Simulation Technology*, 1989.

[5] H. Bakoglu, G. Grohoski, and R. Montoye, "The IBM RISC System/6000 processor: Hardware overview," *IBM Journal of Research and Development*, vol. 34, pp. 12–22, January 1990.

[6] Intel, *i860 64-bit Microprocessor Programmer's Reference Manual*, 1990.

[7] K. Patch, "Intergraph issues superscalar Clipper," *Digital Review*, September 10 1990.

[8] D. Bernstein, D. Cohen, Y. Lavon, and V. Rainish, "Performance evaluation of instruction scheduling on the ibm RISC system/6000," in *Proceedings of the 25th International Symposium on Microarchitecture (MICRO-25)*, (Portland, OR), pp. 226–235, December 1992.

[9] R. Colwell, R. Nix, J. O'Donnell, D. Papworth, and P. Rodman, "A VLIW architecture for a trace scheduling compiler," *IEEE Transactions on Computers*, vol. 37, August 1988.

[10] K. Ebcioglu and A. Nicolau, "A global resource-constrained parallelization technique," in *Proceedings of the Third International Conference on Supercomputing*, pp. 154–163, 1989.

[11] N. Jouppi and D. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Architectural Support for Programming Languages and Operating Systems*, (Boston, MA), pp. 272–282, April, 1989.

[12] F. Allen and J. Cocke, "A catalogue of optimizing transformations," in *Design and Optimization of Compilers* (R. Rustin, ed.), pp. 1–30, Prentice-Hall, 1972.

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

[14] F. Chow, *A Portable Machine-Independent Global Optimizer—Design and Measurements*. PhD thesis, Electrical Engineering Department, Stanford University, Stanford, California, 1984.

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

[16] S. Beaty, M. Duda, R. Mueller, P. Sweany, and J. Varghese, "Optimization issues for a retargetable microcode compiler," *MicroArch*, vol. 3, pp. 5–15, December 1988.

[17] G. Tjaden and M. Flynn, "Detection and parallel execution of independent instructions," *IEEE Transactions on Computers*, vol. C-19, no. 10, pp. 889–895, Oct 1970.

[18] A. Nicolau and J. Fisher, "Measuring the parallelism available for very long instruction word architectures," *IEEE Transactions on Computers*, vol. 33, no. 11, pp. 968–976, Nov 1984.

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

[20] E. Robertson, "Microcode bit optimization is NP-complete," *IEEE Transactions on Computers*, vol. C-28, no. 4, pp. 316–319, April 1979.

[21] D. Landskov, S. Davidson, B. Shriver, and P. Mallett, "Local microcode compaction techniques," *ACM Computing Surveys*, vol. 12, no. 3, pp. 261–294, September 1980.

[22] S. Beaty, *Instruction Scheduling Using Genetic Algorithms*. PhD thesis, Mechanical Engineering Department, Colorado State University, Fort Collins, Colorado, 1991.

[23] J. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478–490, July 1981.

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

[25] B. Su, S. Ding, and L. Jin, "An improvement of trace scheduling for global microcode compaction," in *Proceedings of the 17th Microprogramming Workshop (MICRO-17)*, (New Orleans, LA), pp. 78–85, Nov 1984.

[26] S. A. Mahlke, W. Y. Chen, W. mei W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for VLIW and superscalar processors," in *asplos*5, vol. 27, (Boston, MA), pp. 238–247, oct 1992.

[27] A. Aiken and A. Nicolau, "A development environment for horizontal microcode," *IEEE Transactions on Software Engineering*, vol. 14, no. 5, pp. 584–594, May 1988.

[28] R. Gupta and M. Soffa, "Region scheduling: An approach for detecting and redistributing parallelism," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 421–431, April 1990.

[29] V. Allan, J. Janardhan, L. Lee, and M. Srinivas, "Enhanced region scheduling on a program dependence graph," in *Proceedings of the 25th International Symposium on Microarchitecture (MICRO-25)*, (Portland, OR), pp. 72–80, December 1992.

[30] D. Bernstein and M. Rodeh, "Global instruction scheduling for superscalar machines," in *Conference on Programming Language Design and Implementation*, (Toronto), pp. 241–255, SIGPLAN '91, June 1991.

[31] D. Bernstein, D. Cohen, and H. Krawczyk, "Code duplication: An assist for global instruction scheduling," in *Proceedings of the 24th Microprogramming Workshop (MICRO-23)*, (Albuquerque, NM), pp. 103–113, November 1991.

[32] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, July 1987.

[33] P. Sweany, *Inter-Block Code Motion without Copies*. PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Colorado, 1992.

[34] P. Sweany and S. Beaty, "Dominator-path scheduling — a global scheduling method," in *Proceedings of the 25th International Symposium on Microarchitecture (MICRO-25)*, (Portland, OR), pp. 260–263, December 1992.

[35] A. Charlesworth, "An approach to scientific array processing: The architectural design of the AP_120B/FPS_164 family," *Computer*, pp. 18–27, Sept. 1981.

[36] R. Touzeau, "A FORTRAN compiler for the FPS-164 scientific computer," in *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pp. 48–57, 1984.

[37] B. Su, S. Ding, and J. Xia, "URPR – an extension of URCR for software pipelining," in *Proceedings of the 19th Microprogramming Workshop (MICRO-19)*, (New York, NY), pp. 94–103, December 1986.

[38] B. Su, S. Ding, J. Wang, and J. Xia, "GURPR - A Method for Global Software Pipelining," in *Proceedings of the 20th Microprogramming Workshop (MICRO-20)*, (Colorado Springs, CO), pp. 97–105, December 1987.

[39] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," *SIGPLAN Notices*, vol. 23, pp. 318–328, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[40] V. Allan, R. Jones, R. Lee, and S. Allan, "Software Pipelining," *ACM Computing Surveys*, vol. 27, no. 3, September 1995.

[41] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Conference on Programming Language Design and Implementation*, (Atlanta Georgia), pp. 308–317, SIGPLAN '88, June 1988.

[42] A. Aiken and A. Nicolau, "Perfect Pipelining: A New Loop Optimization Technique," in *Proceedings of the 1988 European Symposium on Programming, Springer Verlag Lecture Notes in Computer Science, #300*, (Atlanta, GA), pp. 221–235, March 1988.

[43] V. Allan, M. Rajagopalan, and R. Lee, "Software Pipelining: Petri Net Pacemaker," in *Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, (Orlando, FL), January 20-22 1993.

[44] N. Warter, G. Haab, and J. Bockhaus, "Enhanced Modulo Scheduling for Loops with Conditional Branches," in *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)*, (Portland, OR), pp. 170–179, December 1-4 1992.

[45] N. J. Warter, S. A. Mahlke, W. mei W. Hwu, and B. R. Rau, "Reverse if-conversion," *SIGPLAN Notices*, vol. 28, pp. 290–299, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[46] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th International Symposium on Microarchitecture (MICRO-27)*, (San Jose, CA), pp. 63–74, December 1994.

[47] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, 1981.

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

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

[50] F. Chow, "The priority-based coloring approach to register allocation," *ACM Transactions on Program Languages and Systems*, vol. 12, no. 4, pp. 501–536, October 1990.

[51] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon, "Coloring heuristics for register allocation," in *Conference on Programming Language Design and Implementation*, (Portland Oregon), pp. 275–284, SIGPLAN '89, June 1989.

[52] S. Beaty, "Register allocation and assignment in a retargetable microcode compiler using graph coloring," Master's thesis, Computer Science Department, Colorado State University, Fort Collins, CO, 1987.

[53] C. Norris and L. Pollock, "A scheduler-sensitive global register allocator," in *Proceedings of Supercomputing '93*, (Portland, OR), Nov. 1993.

[54] J. Goodman and W. Hsu, "Code scheduling and register allocation in large basic blocks," in *Proceedings of the ACM SIGPLAN 88 Conference on Program Language Design and Implementation*, 1988.

[55] D. Bradlee, S. Eggers, and R. Henry, "Integrating register allocation and instruction scheduling for RISCs," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, (Santa Clara, California), April 1991.

[56] S. M. Freudenberger and J. C. Ruttenberg, "Phase ordering of register allocation and instruction scheduling," in *Code Generation - Concepts, Tools, Techniques: Proceedings of the International Workshop on Code Generation* (R. Giegerich and S. L. Graham, eds.), (London), pp. 146–172, Springer-Verlag, May 1992.

[57] S. S. Pinter, "Register allocation with instruction scheduling: A new approach," *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 248–257, 1993.

[58] D. Wall, "Global register allocation at link-time," in *Symposium on Compiler Construction*, pp. 264–275, SIGPLAN '86, July 1986.

[59] J. Edmondson, P. Rubenfeld, and R. Preston, "Superscalar instruction execution in the 21164 alpha microprocessor," *IEEE Micro*, vol. 15, no. 2, pp. 33–43, 1995.

[60] IBM, *IBM Journal of Research and Development*, January 1990.

[61] D. A. Poplawski, "The unlimited resource machine (URM)," Tech. Rep. CS-95-01, Department of Computer Science, Michigan Technological University, Houghton, January 1995.

[62] C. Fraser and D. Hanson, "A code generation interface for ANSI C," *Software — Practice and Experience*, vol. 21, pp. 963–988, September, 1991.

[63] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon, "ParaScope: A parallel programming environment," in *Proceedings of the First International Conference on Supercomputing*, (Athens, Greece), June 1987.

[64] J. Reif and R. Tarjan, "Symbolic program analysis in almost-linear time," *SIAM Journal of Computing*, vol. 11, no. 1, pp. 81–93, February 1981.

[65] R. Mueller, M. Duda, P. Sweany, and J. Walicki, "Horizon: A retargetable compiler for horizontal micro-architectures," *IEEE Transactions on Software Engineering (Special Section on Microprogramming)*, vol. 14, no. 5, pp. 575–583, May 1988.

[66] V. Allan, "Data dependency graph bracing," in *Proceedings of the 21th Microprogramming Workshop (MICRO-21)*, (San Diego, CA), December 1988.

[67] M. Howland, R. Mueller, and P. Sweany, "Trace scheduling optimization in a retargetable microcode compiler," in *Proceedings of the 20th Microprogramming Workshop (MICRO-20)*, (Colorado Springs, CO), December 1987.

[68] B. Su, S. Ding, and J. Xia, "Microcode compaction with timing constraints," in *Proceedings of the 20th Microprogramming Workshop (MICRO-20)*, (Colorado Springs, CO), December 1987.

[69] G. Wood, "On the packing of micro-operations into micro-instruction words," in *Proceedings of the 9th Microprogramming Workshop (MICRO-9)*, December 1978.

[70] A. Gibbons, *Algorithmic Graph Theory*. Cambridge, England: Cambridge University Press, 1985.

[71] V. Allan and R. Mueller, "Microcode compaction with general synchronous timing," *IEEE Transactions on Software Engineering (Special Section on Microprogramming)*, vol. 14, no. 5, pp. 595–599, May 1988.

[72] J. Fisher, *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling*. PhD thesis, Courant Institute of Mathematical Sciences, New York University, New York, NY, October 1979.

[73] V. Allan and R. Mueller, "Compaction with General Synchronous Timing," *IEEE Transactions on Software Engineering*, vol. 14, pp. 595–599, May 1988.

[74] S. Beaty, *Instruction Scheduling Using Genetic Algorithms*. PhD thesis, Mechanical Engineering Department, Colorado State University, Fort Collins, Colorado, 1991.

[75] M. Howland, “Integration of a trace scheduling optimizer in a retargetable microcode compiler,” Master’s thesis, Computer Science Department, Colorado State University, Fort Collins, CO, 1987.

[76] D. Kuck, *The Structure of Computers and Computations Volume 1*. New York: John Wiley and Sons, 1978.

[77] G. Goff, K. Kennedy, and C.-W. Tseng, “Practical dependence testing,” *SIGPLAN Notices*, vol. 26, pp. 15–29, June 1991. *Proceedings of the ACM SIGPLAN ’91 Conference on Programming Language Design and Implementation*.

[78] T. Brasier, S. Beaty, S. Carr, and P. Sweany, “Craig: A practical framework for combining instruction scheduling and register assignment,” in *International Conference on Parallel Architectures and Compilation Techniques (PACT 95)*, (Limassol, Cyprus), pp. ?–?, June 1995.

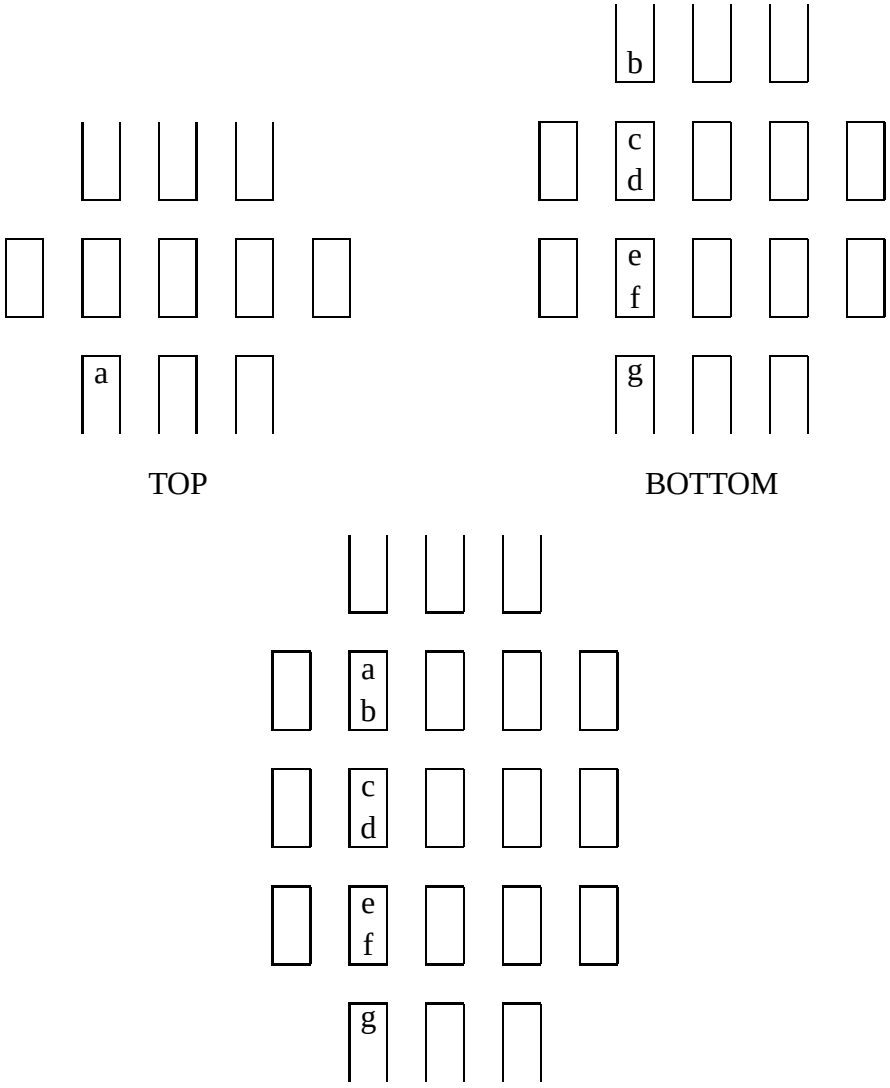

Fig. 5 A variable is used-before-defined in Bottom and Live_out in Top figure

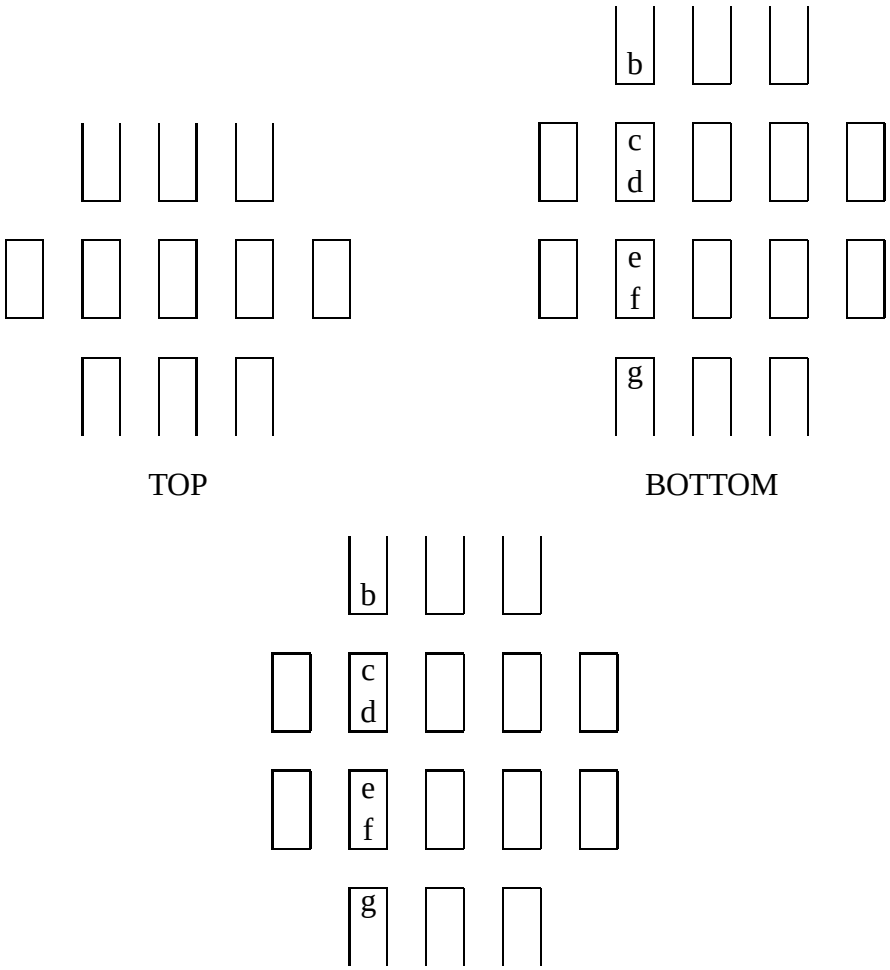

Fig. 6 A variable has no track in Top

figure

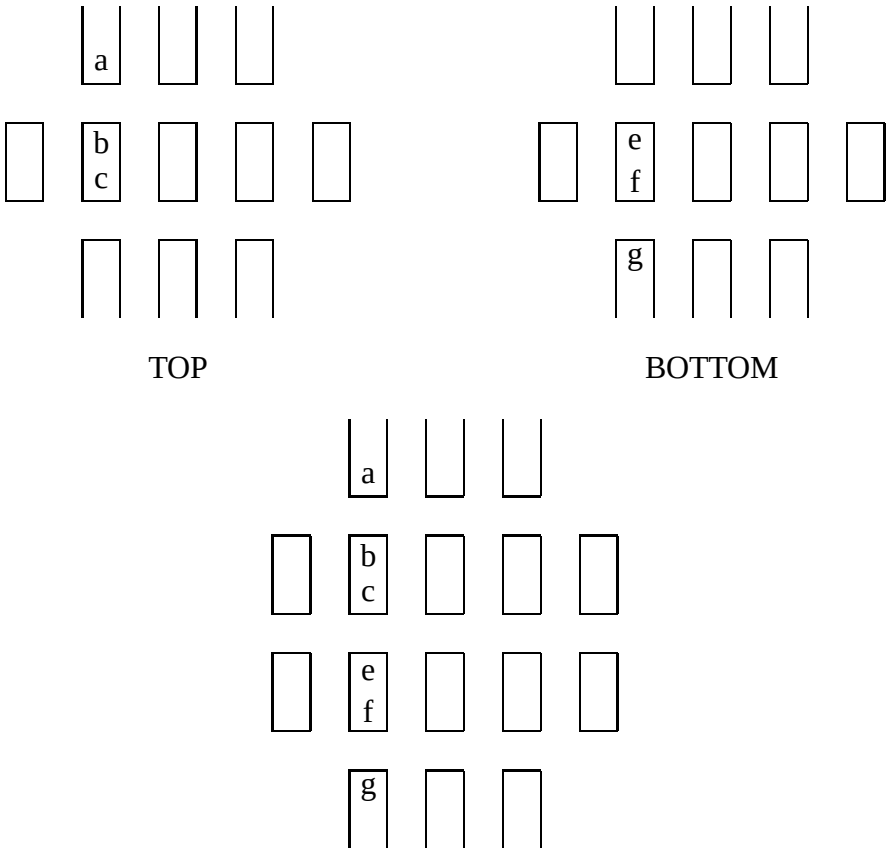

Fig. 7 The variable is not used-before-defined in β figure

Algorithm List Scheduling
Input:
Operation sequence OP = $op_1, op_2, \ldots, op_s$
Data Dependence DAG, DDD
Resource function, U
Output:
Instruction sequence I = $I_1, I_2, ..., I_k$
Algorithm:
Find the priority of each operation Priority(op_i)
C = 0
Scheduled = $\emptyset$
UnScheduled = OP
DRS = $\emptyset$
WHILE $\exists op$ such that $op \in UnScheduled$ DO
DRS = op_i such that $\forall j\ op_i \succeq op_j \Rightarrow op_j \in Scheduled$
C = C + 1
I_c = $\emptyset$
FOREACH x, $x \in$DRS, in ascending order of Priority(x) DO
IF resource compatible for all K: (as defined by
$\sum U(x, R_k) + U(x, R_k) \preceq 1$)
I_c = I_c + x
Scheduled = Scheduled + x
UnScheduled = UnScheduled - x
$\forall y$ such that $x \preceq y$
IF $\forall z$ such that $z \prec y \Rightarrow z \in Scheduled$
DRS = DRS + y
DRS = DRS - x
end IF
end FOREACH
end WHILE

Fig. 8 List Scheduling Algorithm

figure

Algorithm Dominator-Path Scheduling
Input:
Function Control Flow Graph
Dominator Tree
Post-Dominator Tree

Output:
Scheduled Instructions for the Function

Algorithm:
While at least one Basic Block is unscheduled
Heuristically choose a path $B1\ B2 \ldots B_n$
in the Dominator Tree that includes only
unscheduled Basic Blocks.

Perform Dominator Analysis to compute IDef and IUse sets

/* Build one DDD for the entire dominator path */
CombinedDDD = B1
For i = 2 to n
T = InitializeTransitionDDD(B_{i-1}, B_i)
CombinedDDD = Couple(CombinedDDD,T)
CombinedDDD = Couple(CombinedDDD,B_i)

Perform List Scheduling on CombinedDDD
Mark each block of DP Scheduled
Copy scheduled instructions to the Blocks of the path
(instructions between the BlockStart and BlockEnd
nodes for a Block are “written” to that Block)
End While

Fig. 9 Dominator-Path Scheduling Algorithm

figure

```
Algorithm InitializeTransitionDDD(B1, B2)
Input:
        Two Transition DDD templates, one with dummies and one without
        Two basic blocks, B1 and B2, that we wish to couple
        Dominator Tree
        Post-Dominator Tree
        The following dataflow information
                Def, Use, IDef, and IUse sets for B1 and B2
                Post-IDef, and Post-IUse sets for B1 and B2
                A basic block DDD for each of B1 and B2
Output:
        An initialized Transition DDD, T
Algorithm:
        /* Instantiate the Transition DDD */
        If B2 is the immediate flowgraph successor of B1
                T = SimpleTransitionDDD
        Else
                T = DummyTransitionDDD
        If B2 does not post-dominate B1
                Add B1's Use set to T's BlockEnd Def set
                Add B1's Def set to T's BlockEnd Use set
        Else
                Add B1's Post-IDef set to T's BlockEnd Def set
                Add B1's Post-IUse set to T's BlockEnd Use set
        Add B2's IDef set to T's BlockStart Def set
        Add B2's IUse set to T's BlockStart Use set
        return T
```

Fig. 10 Initialize Transition DDD Algorithm

figure