Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining

SOO-MOOK MOON
Seoul National University
and
KEMAL EBCIOĞLU
IBM T. J. Watson Research Center

Instruction-level parallelism (ILP) in nonnumerical code is regarded as scarce and hard to exploit due to its irregularity. In this article, we introduce a new code-scheduling technique for irregular ILP called "selective scheduling" which can be used as a component for superscalar and VLIW compilers. Selective scheduling can compute a wide set of independent operations across *all* execution paths based on renaming and forward-substitution and can compute available operations across loop iterations if combined with software pipelining. This scheduling approach has better heuristics for determining the usefulness of moving one operation versus moving another and can successfully find useful code motions without resorting to branch profiling. The compile-time overhead of selective scheduling is low due to its incremental computation technique and its controlled code duplication. We parallelized the SPEC integer benchmarks and five AIX utilities without using branch probabilities. The experiments indicate that a fivefold speedup is achievable on realistic resources with a reasonable overhead in compilation time and code expansion and that a solid speedup increase is also obtainable on machines with fewer resources. These results improve previously known characteristics of irregular ILP.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation*; *optimization*

General Terms: Algorithms, Experimentation, Languages

Additional Key Words and Phrases: Global instruction scheduling, instruction-level parallelism, software pipelining, speculative code motion, superscalar, VLIW

1. INTRODUCTION

In the last two decades, the performance of single-CPU microprocessors has increased enormously, mostly due to continuing improvements in computer architecture [Rau and Fisher 1993]. One major architectural breakthrough has been the RISC (Reduced Instruction Set Computer) technology [Patterson 1985] based on instruction pipelining which overlaps execution steps of different instructions,

Authors' addresses: S.-M. Moon, School of Electrical Engineering, Seoul National University, Seoul 151-742, Korea; email: smoon@altair.snu.ac.kr; K. Ebcioğlu, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598; email: kemal@us.ibm.com.

achieving an execution rate of close to one instruction per clock cycle. In order to increase processor performance beyond simple pipelining, modern architectures duplicate pipeline data paths and execute multiple instructions in parallel, attempting to reach execution rates of multiple instructions per clock cycle [Rau and Fisher 1993]. This technology, known as *instruction-level parallelism* or ILP, has rapidly become the leader in new processor technologies and has already been incorporated in many commercial machines [Colwell et al. 1988; IBM 1990; Rau 1989; Sites 1993].

ILP-machines require *independent instructions* in each cycle in order to make good use of duplicated pipelines. This entails scheduling instructions from the sequential code stream by analyzing data and control dependences. Depending on when scheduling is performed, two distinct approaches have been used: *superscalar* and *VLIW* (Very Long Instruction Word). Superscalar machines perform run-time scheduling and dependence analysis using hardware among the instructions within a window, and they issue independent instructions dynamically. On the other hand, VLIW machines rely on compile-time scheduling to determine independent instructions which can be issued concurrently at execution time.

The relative merits of these alternative approaches are still controversial, yet there is a growing consensus that an ILP compiler must rearrange code for better performance no matter which alternative is used [Rau and Fisher 1993]. That is, even the compilers for superscalar machines should group independent instructions together. Since the compiler has a larger (software) window and can utilize sophisticated analysis techniques that are too expensive to perform at run-time, such grouping helps the hardware find more parallelism.

One of the biggest challenges for an ILP compiler is to extract ILP in *nonnumerical code*, such as user application programs or system programs. It has been said that nonnumerical code leads to small speedup (as little as two) and that ILP in such code is difficult to exploit due to its irregularity [Jouppi and Wall 1989; Smith et al. 1989]. The following is a discussion of typical problems in exploiting irregular ILP and a brief overview of corresponding solutions proposed in this article.

Nonnumerical programs include a large number of conditional branches and control join points, resulting in basic blocks with few instructions. Such programs require *global scheduling* of instructions which includes two types of upward code motion: one ahead of conditional branches and one ahead of control join points. *Speculative code motion*, the code motion of instructions before preceding conditional branches in order to use otherwise idle resources, increases speedup if speculative execution turns out to be useful (i.e., the path where the speculative instructions came from is taken at execution time); otherwise the resources are wasted. Code motion ahead of control join points may cause instructions to be duplicated at incoming edges to the join points, yet it also helps in reducing the length of critical paths.

Static branch probabilities based on profiling have been widely used in numerical code, for speculative code motion on selected paths [Ellis 1985]. However, despite research advances [Gloy et al. 1995], static branch probability in nonnumerical code is not always reliable (it may not be the same for different data sets) or useful for code scheduling (a given branch may be taken half of the time). As an example, consider the outcome of a compare-and-branch used in sorting. In addition, it

```
if cc0 then {op1} else {op2};
if cc1 then {op3} else {op4};
...;
if ccn then {op2n−1} else {op2n};
op0;
```

(a)

```
op0
if cc0 then {op1} else {op2; op0};
if cc1 then {op3} else {op4; op0};
 ...;
if ccn then {op2n−1} else {op2n; op0};
```

(b)

Fig. 1. Speculative code motion and speculative bookkeeping code.

might be burdensome for the users or independent software vendors to obtain accurate profiling data for large applications, or profiling may be disallowed because of customer requirements. Therefore, it is desirable for a scheduling technique to do well in the absence of branch probability and use branch probability for obtaining additional performance when it is available [Fisher and Freudenberger 1992].

Mispredicted speculative code motion can also slow down program execution if code is moved past several control join points. For example, consider the consecutive `if-then-else-endif` constructs in Figure 1(a), where op_0 has no dependence on any op_{2k-1} $(k = 1, 2, .., n)$ (the `then` portions). All branches are predicted to take the true paths (the `then` portions), and op_0 is scheduled speculatively across the true paths, ahead of `if` cc_0. For semantic correctness, a bookkeeping copy of op_0 must be inserted at each incoming edge to the path of code motion, which is just after each op_{2k} $(k = 1, 2, .., n)$ (the `else` portions) as in Figure 1(b). Now, if all branches take the false paths (the `else` portions) during execution, op_0 is executed $n+1$ times, which could slow down program execution on machines with few resources, compared to the original code running on a sequential machine. The problem is that speculative code motion might generate excessive bookkeeping operations which are also speculative; so they are not always useful during execution, yet do take resources (in Figure 1(b), all bookkeeping copies of op_0 except for the last one are speculative).

In nonnumerical code, it is preferable to perform speculative code motion across *all* execution paths whenever resources permit, to cope with unpredictable branches.[1] At the same time, code motion across all paths should be performed, as a way of generating less speculative bookkeeping code. Both goals can be achieved if the compiler performs *useful* code motion, such as *nonspeculative* or *mildly speculative* code motion. For example, nonspeculative motion of an instruction across both targets of each branch will be useful regardless of which path is taken at execution time; similarly, code motion that does not pass too many branches will have a higher chance of being on the taken path and of generating less bookkeeping code. In the example above, if op_0 has no dependences on any op_{2k} $(k = 1, 2, .., n)$ as well, op_0 can be scheduled ahead of `if` cc_0 across both targets of each branch without generating any bookkeeping code. In case op_0 has data dependences on some instructions and cannot be scheduled across both targets of each branch, the scheduling technique should obtain an accurate estimate of the cost associated with moving up op_0 and should choose a more useful instruction to move if one is available. The job of the compiler is then to do as many useful code motions as possible,

[1]Loop control branches are generally taken (and therefore predictable), and we can utilize this branch behavior through software pipelining.

by using aggressive code motion techniques.

In order to enhance ILP beyond loop boundaries, the technique of software pipelining has been used for generating code where the execution of different iterations is overlapped in a pipelined fashion. Innermost loops of numerical code have often been software pipelined by computing the minimum initiation interval based on resource constraints and data dependence cycles; according to the initiation interval, the pipelined loop kernel, the startup code, and the drain code are scheduled [Lam 1988; Rau and Glaeser 1981]. These techniques are collectively called *modulo scheduling*. When the loop body includes conditional branches if any, modulo scheduling computes a conservative initiation interval across all paths, either by removing branches through *if-conversion* [Dehnert and Towle 1993] or by reducing the entire `if-then-else-endif` control structure to a single superinstruction through padding the shorter path with no-ops (*hierarchical reduction*) [Lam 1988].

These techniques with a *fixed* initiation interval are not appropriate for software pipelining of loops in nonnumerical code, where multiple execution paths have similar probabilities of being taken at execution time. A fixed initiation interval based on worst-case data dependences and resource requirements across all paths just loses loop parallelism. In the presence of conditionally executed subroutine calls or inner loops within a loop, the length of the worst-case dependence cycle may not even be precisely computable, since inner loops and subroutines take a variable amount of time. On the other hand, the idea of computing an initiation interval for each possible combination of loop execution paths does not provide an easy way of generating such a pipelined loop schedule. This requires a new method which combines global scheduling with software pipelining, so that speculative code motion is fully supported and so that different initiation intervals in different execution paths of a loop are allowed, thus obtaining a *variable* initiation interval [Ebcioğlu and Nakatani 1989].

The example in Figure 2 shows how a loop with a conditional branch can be software pipelined with a variable initiation interval. In Figure 2(a), each instruction in the loop is assumed to take a single cycle except for the multiplication, which takes two cycles. The loop can execute at a rate of one cycle/iteration when `cc` is false, and three cycles/iteration when `cc` is true. Figure 2(b) shows its software-pipelined loop with a variable initiation interval, where each shaded group includes independent instructions that can be executed in the same cycle. Each instruction includes its iteration number ([ ]), and a speculative instruction is marked with "`S.`". The parallel group L3 will be executed repeatedly in each cycle (i.e., at a rate of one cycle/iteration) except when `cc` is true; in this case, the execution follows the path L3-L4-L2 and goes back to L3 (at a rate of three cycles/iteration). The techniques of hierarchical reduction or if-conversion will reduce the "`if cc {x=x*y}`" to a single superinstruction that uses `x`, `y`, and `cc` and sets `x` as in Figure 2(c). There is a spurious dependence cycle, `x=x+x` → `cc=test(x)` → `x=super_op(cc,x,y)` → `x=x+x`, which precludes any hope of achieving one cycle per iteration, regardless of how many resources are available.

Previous compilers exploiting irregular ILP have been disappointing in terms of the performance of scheduled code and compilation efficiency. There have been two major approaches for global scheduling: those that allow code motion across only

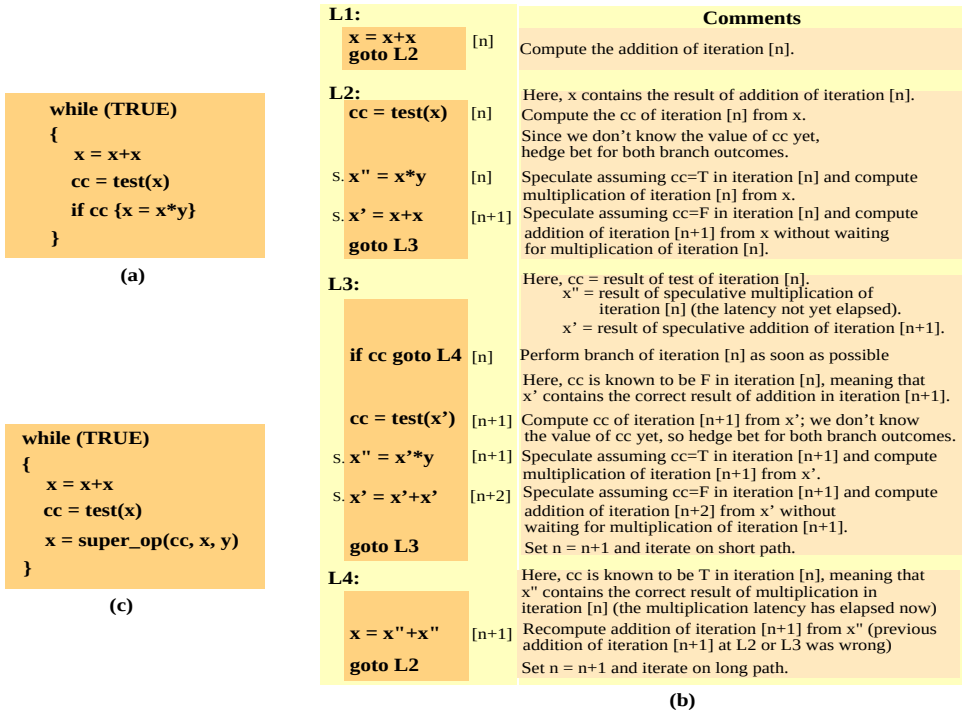


Fig. 2. Software pipelining with a variable initiation interval; three iterations ((n)th, $(n+1)$th, and $(n+2)$nd) are executed in parallel in L3; more details follow in Section 6.

one execution path that is chosen based on branch probability (*trace* based) and those that allow code motion across all execution paths of a DAG (DAG based).

Trace-based techniques [Ellis 1985] fundamentally suffer from loss of speedup during off-trace execution, which can be caused by speculative code motion and excessive bookkeeping code at the joins. In order to avoid the generation of bookkeeping code, there have been approaches to duplicate code below join points prior to scheduling [Hwu et al. 1993] or to perform nonspeculative code motion on a trace without generating redundant bookkeeping code [Freudenberger et al. 1994; Smith et al. 1992]. These techniques are limited in performing useful code motion in the sense that only the trace is examined for available parallelism, and they might generate unnecessary bookkeeping code.

DAG-Based techniques have advantages in performing useful code motion, since they have a more global view of the code than on a single trace. The control dependence graph [Ferrante et al. 1987] has been used in order to identify control-equivalent basic blocks or "regions" to perform nonspeculative code motion between them [Bernstein and Rodeh 1991; Gupta and Soffa 1990]. However, the control dependence graph alone is not precise in computing usefulness, as will be seen in Section 5.3. Greedy application of primitive code motion between adjacent nodes within a DAG can identify all nonspeculative code motions [Aiken and Nicolau 1988; Nakatani and Ebcioğlu 1993]. However, this approach generates a large amount of speculative instructions that cannot be accommodated

with finite resources, which might either slow down program execution or entail the unnecessary complexity of undoing code motions.

Among the above techniques, those that target machines with many resources [Aiken and Nicolau 1988; Ellis 1985] tend to suffer from inefficiency problems such as code explosion or long compilation time, which have made them difficult to use in practice. Other techniques that target machines with few resources [Bernstein and Rodeh 1991; Smith et al. 1992] are severely restricted; they employ neither renaming nor software pipelining, and code motion is constrained to occur only between basic blocks, so that neither creation of a new basic block nor destruction of an existing basic block is allowed during scheduling. These restrictions are to avoid the additional complexity of maintaining correct data flow information when aggressive code motion is performed and are based on the notion that a machine with few resources does not need aggressive code scheduling. We disagree with this notion because aggressive scheduling techniques help to find more of useful code motions, as well as to find more of available code motions.

In this article, we introduce a new DAG-Based global scheduling technique called *selective scheduling* and its compiler for ILP machines. Selective scheduling can extract a large amount of useful code motion across all paths and can be incorporated in software pipelining. Our optimizing compiler based on selective scheduling (thereafter referred to as *selective scheduling compiler*) can generate high-performance code for machines with many resources, as well as for machines with few resources, without resorting to branch profiling. For additional performance, branch probabilities can be used to choose among speculative operations to be moved. The compiler is reasonably efficient in terms of code expansion and compilation time.

The rest of this article is organized as follows. Section 2 briefly overviews the new features of the selective scheduling compiler. The intermediate representations of code that are manipulated by the compiler are described in Section 3, and Section 4 includes our techniques to overcome nontrue data dependences during code motion. Section 5 describes the details of the selective scheduling algorithm, and Section 6 includes our software pipelining techniques. Section 7 presents our experimental results to examine the performance and the scheduling efficiency of the compiler. Section 8 summarizes the main results of the article.

2. FEATURES OF SELECTIVE SCHEDULING

The selective scheduling compiler uses innovative techniques to exploit irregular ILP. We summarize its features in this section. The most important global scheduling problem is gathering a group of independent operations[2] at a point of a control flow graph as in Figure 3(a). Each group is eventually transformed into a VLIW instruction or is likely to be executed by superscalar hardware in the same cycle. Most global scheduling techniques [Bernstein and Rodeh 1991; Ellis 1985; Smith et al. 1992] solve this problem in the same way: first compute the set of all available operations that can move into the group (*availability set*); then, choose the best operations from the availability set and schedule them into the group.

[2]To avoid confusion, we use the term **operation** for a single primitive instruction, and **instruction** for the long instruction word with several operations.

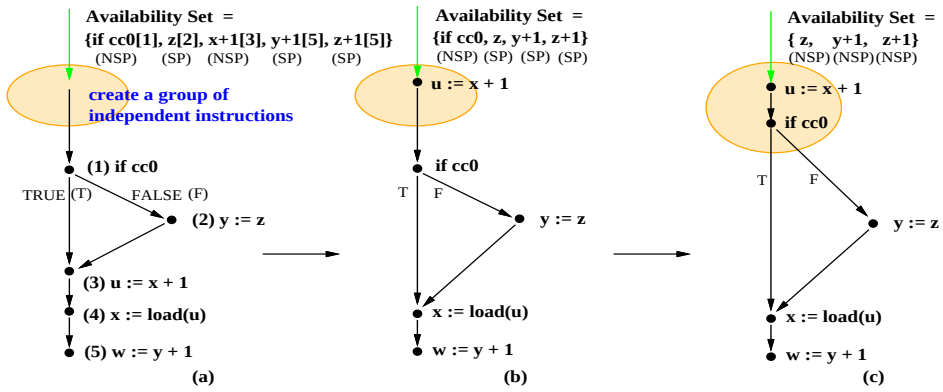


Fig. 3. Gathering a group of independent operations with resource constraints; SP: speculative, NSP: nonspeculative.

Selective scheduling also follows this two-step strategy.

Selective scheduling can compute a large availability set because

—it computes the set of available *right-hand sides* (RHS) of operations instead of computing the set of whole operations, since nontrue data dependences associated with target registers of operations can be overcome through renaming;

—it computes available RHS across all execution paths, allowing speculative code motion across both targets of each branch;

—it computes RHS across loop iteration boundaries combined with software pipelining;

—it *recomputes* the availability set after each code motion to obtain the correct availability set.

Figure 3(a) shows our availability set of RHS (the number shown in brackets after each RHS indicates the operation in the graph from which it is derived). Any of the RHS can move into the group with either its original target register or a new target register after renaming (for example, `y+1` can be scheduled after renaming). In addition, the two RHS in our availability set, `y+1` and `z+1`, are computed from the same operation `w:=y+1`, one across the true path of `if cc0` and the other one across the false path of `if cc0` after substituting `z` for `y`.[3] Consequently, when resources permit, selective scheduling can schedule operations on all paths as soon as their source operands are ready.

In order to choose useful operations from the availability set, the usefulness of each RHS is estimated during the computation through a simple comparison at the branches (see Section 5.1.1). In Figure 3(a), `x+1` is nonspeculative because it is available for moving up in both targets of the branch, whereas `y+1` or `z+1` are speculative because they are available in only one target. Figure 3(b) shows a nonspeculative code motion. The computation of RHS across all paths enhances our chances of finding more useful code motions, in addition to avoiding nontrue data dependences and increasing the number of available operations.

[3]Our techniques of renaming and substitution will be described in detail in Section 4.

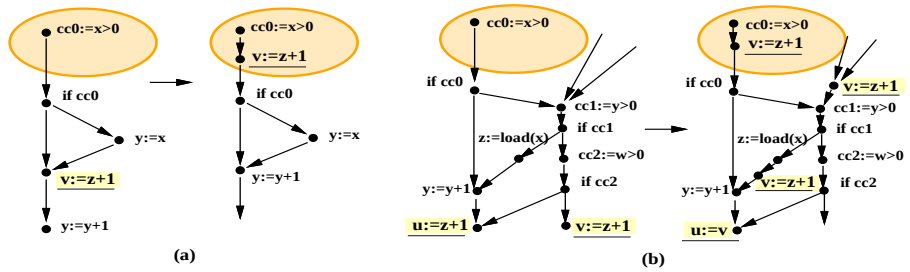
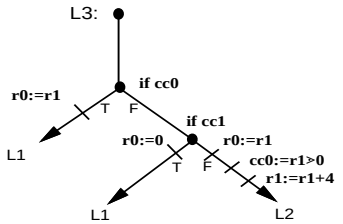


Fig. 4. (a) Unification in a simple `if-then-else-endif` construct; (b) in an unstructured code fragment.

Selective scheduling schedules conditional branch operations as well as data operations, as in Figure 3(c), to deal with the high frequency of branches in nonnumerical code. Although the code motion of a branch duplicates operations on its way, it helps in reducing unnecessary speculative code motions that may waste resources (i.e., the earlier we execute a branch and know its outcome, the less we need to speculate). In addition, by scheduling branches in a group with other data operations, we can utilize a more compact representation of concurrency where data and branch operations can be executed concurrently and where multiple branches can be executed in a single cycle. This representation is called a *tree* representation and is one component of our compiler (see Section 3.1). The tree representation helps to reduce the performance bottleneck of sequential branch execution in nonnumerical code [Moon and Ebcioğlu 1997].

Regarding the efficiency of selective scheduling, we should mention that the overhead of recomputation is low because it is done *incrementally*, only on the paths through which the chosen operation is actually moved up. In addition, selective scheduling reduces code expansion by *unifying* the same computation on different execution paths; multiple occurrences of the same computation are hoisted up as a single operation. A simple example of unification can be found in the `if-then-else-endif` construct shown in Figure 4(a); `v:=z+1` below the construct is moved up across both paths of the construct and is unified into one. If the operation were scheduled speculatively across only one target, a conservative scheduling technique would generate a bookkeeping copy at the join point, thus causing code expansion. In fact, unification includes the useful code motions past conditional branches, that were discussed in the introduction.

Selective scheduling can perform unification across multiple branches after generating bookkeeping code, even in an unstructured code fragment as the one depicted in Figure 4(b); two copies of `z+1` are unified into one at `if cc2`, which is then moved up across one target of `if cc1`, yet two copies of the operation are again unified into one at `if cc0`. Although these unstructured code fragments do not necessarily arise initially, aggressive code scheduling including branch code motion can render the code structure quite different from the initial program. It would be difficult for trace-based scheduling techniques to avoid the generation of speculative bookkeeping code and to perform useful code motion on these unstructured code fragments.



register values before executing L3

cc0	cc1	r0	r1
F	F	0	4

register values after executing L3

cc0	cc1	r0	r1
T	F	4	8

(b)

Fig. 5. A VLIW tree instruction.

3. INTERMEDIATE REPRESENTATION OF CODE

Our compiler generates parallelized code from sequential code using the features described in the previous section. In this section, we introduce the intermediate representations of code that are manipulated by the compiler and describe the scenario of code scheduling and code generation.

3.1 Tree Representation

In order to model uniprocessors that can execute multiple operations concurrently, we introduce the tree representation, a new intermediate representation of parallelized code. A VLIW implementation of the tree representation is called a *tree instruction* [Ebcioğlu 1988] which has the form of a binary decision tree.

The tree instruction allows the representation of a high degree of parallelism with respect to both data and branch operations. Figure 5(a) shows an example of a tree instruction `L3`. Each internal node of the tree corresponds to a test on a condition register (`if` cc_j), whereas terminal nodes have instruction labels indicating where this instruction can branch to. Each directed edge of the tree is annotated with zero or more data (ALU or memory) operations. All operations scheduled in a tree are guaranteed by the compiler to have no dependences on each other and can thus be executed simultaneously. For a specific implementation, there will be a finite limit on the number of *distinct* data operations and on the number of branch targets (which equals to the number of test nodes + 1) that can be contained in a tree instruction, and there may be other resource constraints as well. The instruction `L3` in Figure 5(a) has four distinct data operations even though there are five instances and does three-way branching, since there are two test nodes.

The execution of L3 consists of two steps. First, using the current values of the condition registers `cc0` and `cc1`, a unique taken path is determined by traversing the tree from the root to a leaf. The instruction label at the selected leaf becomes the next tree instruction to be executed. Then, the operations on the taken path are executed in parallel by performing all the operand "reads" first, and then all the operand "writes." If the initial values of both `cc0` and `cc1` are false (`F`), for example, `r0:=r1`, `cc0:=r1>0`, and `r1:=r1+4` are executed in parallel using the old values of `r1` that are available from previous instructions. In the next cycle, the updated values of `r0`, `cc0`, and `r1` will be observed by the instruction L2 as shown in Figure 5(b).

The abstract VLIW machine at this level is assumed to have multiple ALUs, all of which share a register file, and multiple condition registers that take binary

values (`T` or `F`). It is also assumed to have a branch unit capable of deciding the next target in a single cycle (*multiway branching* [Moon and Carson 1995]) and an execution mechanism to commit operations depending on the outcome of branching (*conditional execution* [Ebcioğlu 1988]). In order to reduce the critical path of a cycle time, multiway branching and conditional execution are performed concurrently [Ebcioğlu 1988], and this allows identical operations occurring in multiple paths on a tree (e.g., `r0:=r1` in `L3`) to be allocated to the same ALU, thus requiring four ALUs instead of five for the execution of `L3`. The tree instruction has a pipelined implementation (with bypass paths), so that it takes one short machine cycle. The tree instruction also does not incur any branch stalls. Ebcioğlu [1988] and Nakatani and Ebcioğlu [1993] describe methods for building a pipelined tree VLIW machine with a short cycle time.

It should be noted that neither multiway branching nor conditional execution is a requirement for applying our compilation techniques, although these architectural features are recommended for achieving better parallelism. The tree paradigm can be restricted at will to obtain the existing architectural paradigms. For those VLIW architectures that do not have the conditional execution features [Aiken and Nicolau 1988], all data operations will be placed at the root of the tree above any conditional branches. For those VLIW architectures that allow only two-way branching, our tree will have at most one test node. Superscalar resources can be represented by choosing an appropriate tree with maximum resources equal to the number of functional units in the superscalar (for example, one could allow a maximum of one integer operation, one conditional branch, and one floating-point operation in a tree for the IBM RS/6000 [Ebcioğlu et al. 1994]). Therefore, the resource constraints of many existing ILP machines can be accommodated by our algorithms in a flexible manner. In the context of the present work, our compiler generates parallelized code targeted for the VLIW tree instruction.

3.2 Sequential Representation

Although VLIW trees are generated as the final code, code scheduling itself is performed on a sequential representation of code. The input to the compiler is a control flow graph (CFG) whose nodes are primitive operations obtained from sequential code. This graph is called a *sequential program*. Based on specific resource constraints, the compiler generates a *parallelized program* which is still a sequential CFG, yet in which independently executable operations have been brought together and are located in adjacent nodes. This parallelized program is in fact called *superscalar code*, where each group of adjacent independent operations is likely to be executed in the same cycle by a superscalar machine. Finally, the parallelized program is converted into a *VLIW program* which is a CFG whose nodes are VLIW tree instructions. The VLIW program is assembled into binary VLIW code. Throughout the remaining of the article, we use the term "program" to denote the internal CFG form of code that is manipulated by our compiler, and we use the term *node* interchangeably with *operation* in the sequential program. Since each edge in the CFG directly represents the successor relationship, unconditional branches do not exist in the sequential program.

Figure 6(a) shows an example of a sequential program, and Figure 6(b) shows its corresponding parallelized program. A group of operations in the shaded area

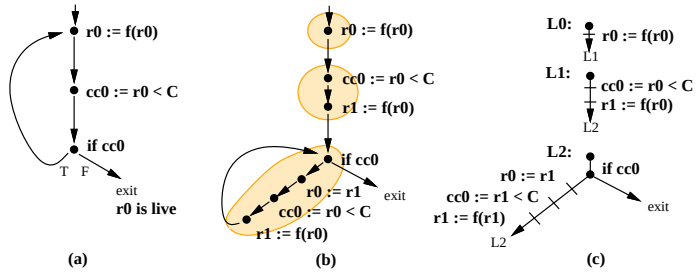


Fig. 6. (a) An example of a sequential program, (b) its parallelized version, and (c) the corresponding VLIW program.

indicates independently executable operations and is called a "*parallel group*." Each operation is assigned a sequence number (*seqno*), such that all operations in the same parallel group have the same *seqno*. Hence, the final transformation can easily identify each parallel group and can convert it to a VLIW tree instruction as in Figure 6(c).

Although there have been approaches to perform code scheduling directly on the VLIW program [Aiken and Nicolau 1988; Nakatani and Ebcioğlu 1993], code scheduling on the sequential program is more efficient and simpler. In order to generate parallel groups corresponding to final tree instructions, it is possible that a parallel group includes copy operations and other operations that are data dependent on the copies. For example, a parallel group can be formed from the operations [`r0:=r1`; `r1:=r2*r2`; `r3:=r0+1`], in this order. The operations cannot execute concurrently in this form, yet after forward substitution of the copy `r0:=r1` they can. The corresponding tree instruction will hold them as [`r0:=r1`; `r1:=r2*r2`; `r3:=r1+1`] (note that `r3:=r1+1` will read the old `r1` value, not the one written by `r1:=r2*r2` because of the execution semantics of the tree instruction, reads-first-then-writes). However, an operation such as `r4:=r1+r1` cannot be scheduled into the same parallel group, due to its data dependence on `r1:=r2*r2`. The point to note here is that the sequential form of an operation scheduled in a parallel group might be different from the VLIW form of the operation in the corresponding tree instruction. This distinction is made in our description of the algorithm in Section 5 by using different RHS notations (i.e., Sequential RHS *vs.* VLIW RHS). Basically, in the sequential program, the compiler schedules parallel groups, which correspond to the final VLIW tree instructions.

4. NONTRUE DATA DEPENDENCES

Code scheduling on the sequential program is frequently hampered by nontrue data dependences that are caused by storage conflicts arising from reusing registers. This section describes our techniques to avoid nontrue data dependences, which include both renaming registers and resolving false dependences related to copy operations. In order to keep our description of techniques manageable, we assume only register-to-register operations in the sequential program, yet in Section 6 we describe how we disambiguate memory operations in our real implementation. In addition, the sequential program is assumed to be already register allocated. Our compiler re-

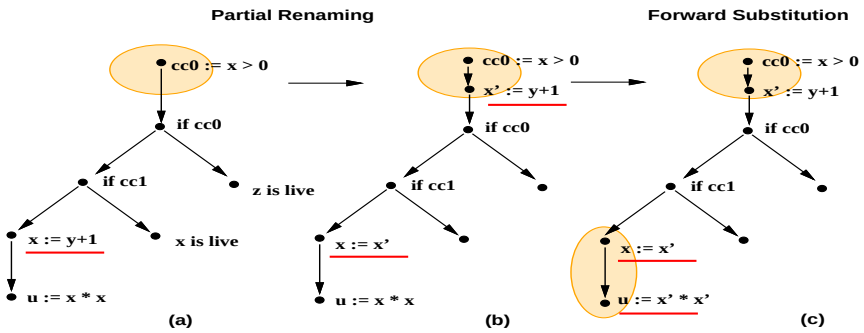


Fig. 7. Partial renaming and forward substitution techniques.

names registers directly using hardware registers during code scheduling, instead of first introducing symbolic registers, and then allocating hardware registers after scheduling.[4]

4.1 Partial Renaming and Forward Substitution

Previous renaming infrastructures such as live range renaming [Padua and Wolfe 1986] or static single assignment [Cytron et al. 1991] are not enough to avoid all nontrue data dependences that occur during code motion; for example, software pipelining often requires moving a definition of a register from iteration $n+1$ to iteration n, to a point where the same register of iteration n is still live at a loop exit, which cannot be achieved by these techniques alone. We introduce techniques that rename registers on an *as-needed* basis when parallelism opportunities arise.

In Figure 7(a), the operation `x:=y+1` cannot move into the parallel group because `x` is live at the other target of the branch `if cc1`. However, its right-hand side, `y+1`, can still move into the desired group with a new target register `x'` by substituting a copy operation for the original operation, as shown in Figure 7(b). This technique is called *partial* renaming, in order to distinguish it from previous techniques that rename whole live ranges.

Although the RHS is moved up, partial renaming alone cannot reduce the total time, since the copy operation still remains. However, the copy operation is relatively *cheap*, in the sense that it often has lower latencies than other operations, can be propagated and coalesced away at later stages, or can be executed concurrently with any operations that are data dependent on the copy operation. In Figure 7(b), there is a data dependence between `x:=x'` and `u:=x*x`, yet this is not a real true dependence; the operation `u:=x*x` can be executed concurrently with the copy `x:=x'` after replacing `x` by `x'` as in Figure 7(c). This technique of *forward substitution*[5] allows the scheduling of copies and data-dependent operations on the copies in the same parallel group as discussed in Section 3.2.

[4]Code scheduling performed before register allocation can use the same techniques in this section, yet should be careful not to exceed the hardware register resources, since register spill code can cause significant serialization on ILP machines.

[5]A more general technique called *combining* allows forward substitution with operations other than copies [Nakatani and Ebcioğlu 1989].

Forward substitution is also helpful in enhancing upward code motion. In Figure 7(b), `u:=x*x` can be scheduled before the copy `x:=x'` in the form of `u:=x'*x'`.[6] In fact, an operation can be transformed multiple times through substitution while it is being moved up if it passes through multiple copies. Since partial renaming results in the generation of many copy operations, upward code motion through substitution is important.

Selective scheduling incorporates both partial renaming and forward substitution techniques in its computation step: all available RHS that can be scheduled into the desired parallel group are computed, which

—do not violate any true dependences on the way up, yet can be substituted at copies,

—are not true dependent on any operations already scheduled in the group, yet may be dependent on copies in the group.

In order to describe this computation step precisely, we introduce a primitive scheduling function that shows the result of moving up a given RHS through an operation or through an execution path of operations.

4.2 The Functions `moveup()`

When we try to move up a given RHS Υ above an operation Θ, there are three possible outcomes as described in `moveup_rhs(`Υ, Θ`)` in Figure 8: `NULL` if Υ cannot be scheduled above Θ due to a nonsubstitutable true dependence, a substituted Υ' if Θ is a copy and if there is a substitutable dependence, and the original Υ if there is no dependence at all.

We can also extend the function `moveup_rhs()` across an execution path. Let $\Psi_{i,j}$ be an acyclic execution path from an operation i to an operation j in the flow graph of a sequential program. Moving up Υ across the path $\Psi_{i,j}$ (starting from j back to i following the arrows on the edges in reverse order) can be described by the function `moveup_path(`$\Upsilon, \Psi_{i,j}$`)` in Figure 8, which either returns the resulting Υ after it has been moved through the path or returns `NULL` indicating that the Υ could not be moved due to nonsubstitutable true dependences. Finally, the two functions are defined for a set of right-hand sides, Δ, as in `moveup_set_rhs(`Δ, Θ`)` and `moveup_set_path(`$\Delta, \Psi_{i,j}$`)` in Figure 8. For example, let Ψ_{x_1,x_4} be a given directed path $\overline{x_1x_2x_3x_4}$, where $oper(x_1) \equiv$ (`y:=z`), $oper(x_2) \equiv$ (`w:=w+1`), $oper(x_3) \equiv$ (`x:=y`), $oper(x_4) \equiv$ (`y:=w*2`). Then, `moveup_set_path(`$\{\mathtt{x}+1, \mathtt{y}+1, \mathtt{z}+1\}, \Psi_{x_1,x_4}$`)` $= \{$`z+1`$\}$; `x+1` is forward substituted at x_3 becoming `y+1`, then forward substituted at x_1 becoming `z+1`; `y+1` is deleted at x_4; `z+1` remains as it is, since it is not affected. The functions `moveup()` will be helpful in understanding the selective scheduling algorithm in the next section.

5. DESCRIPTION OF SELECTIVE SCHEDULING

In the next two sections, we describe the parallelization process in the selective scheduling compiler. The backbone of the parallelization is the software pipelining

[6]If `u:=x*x` was the last use of `x:=x'`, the copy operation can be deleted after `u:=x'*x'` is moved up above the copy because it now becomes a dead operation. This controls code expansion caused by partial renaming.

For an operation Θ in a sequential program,
$rhs(\Theta)$: the RHS of the operation $dest(\Theta)$: the target register
$opcode(\Theta)$: the operation code $sources(\Theta)$: the set of source registers in $rhs(\Theta)$

```
moveup_rhs(Υ, Θ)    (Υ: RHS or NULL, Θ: operation)
if (Υ ≠ NULL and dest(Θ) ∈ sources(Υ)) {
   if (opcode(Θ) ≡ copy (i.e., x:=y)) {
      Υ' = ReplaceEachOcurrenceBy(Υ, x, y) ; replace each occurrence of x in Υ by y
      return(Υ') ; move up through substitution
    }
    return(NULL) ; nonsubstitutable true data dependence
 }
 return(Υ) ; move up with no change
End moveup_rhs

moveup_path(Υ, Ψ_{i,j})    (Υ: RHS, Ψ_{i,j}: path)
 τ = Υ
 for (n=length(Ψ_{i,j}); n≥1; n=n-1); length(Ψ_{i,j})≡ number of operations in Ψ_{i,j}
    τ = moveup_rhs(τ, Ψ_{i,j}(n)) ; Ψ_{i,j}(n) ≡ nth operation on i→j, n = 1, 2, ..., length(Ψ_{i,j})
 return(τ)
End moveup_path

moveup_set_rhs(Δ, Θ) ≡ {τ|(∃Υ ∈ Δ)[τ = moveup_rhs(Υ, Θ) ∧ τ ≠ NULL]}  ;Δ: set of RHS

moveup_set_path(Δ, Ψ_{i,j}) ≡ {τ|(∃Υ ∈ Δ)[τ = moveup_path(Υ, Ψ_{i,j}) ∧ τ ≠ NULL]}
```

Fig. 8. The functions `moveup`.

of a loop as will be described in Section 6; however, at each *stage* of software pipelining, code motion through certain edges in the sequential program is inhibited such that the remaining edges where code motion is allowed form a DAG. The scheduling within this DAG is handled by the selective scheduling algorithm described in this section.

The input to the algorithm is a rooted DAG, $G_r = (V, E)$, obtained from a sequential program where there is an empty parallel group just before the root node, r. Figure 9(a) shows a rooted DAG, $G_{(3)}$, and an empty parallel group. The parallel group is associated with specific resource constraints, and our objective is to *"fill"* the group with independently executable operations. A filled parallel group will finally correspond to a VLIW tree instruction.

The following two actions are repeated until either the parallel group becomes full, that is, the resource constraints are met, or no more operations can be moved into the group due to data dependences.

—**Computation**: the set of all available right-hand sides (RHS) that can move into the parallel group is computed, and the best RHS is selected among them.
—**Code Motion**: the best RHS is actually scheduled into the group with a suitable target register

Our **av**ailability set[7] of RHS at the parallel group is described as `av(group)` in

[7]Availability set has been called "unifiable-ops" previously [Ebcioğlu and Nicolau 1989].

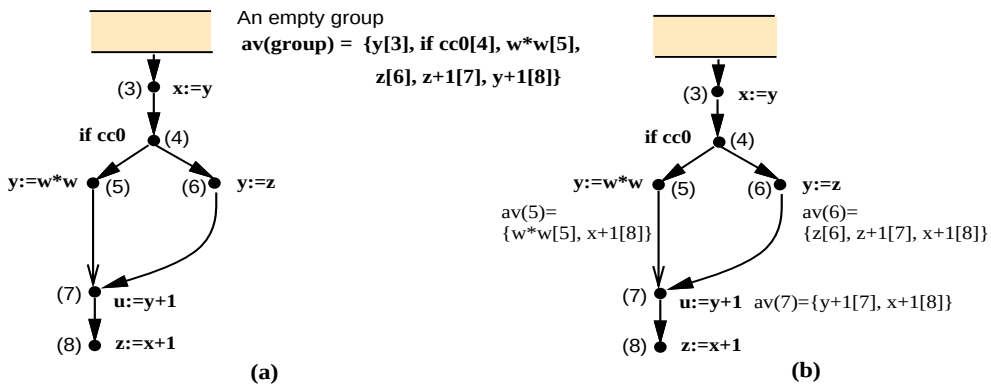


Fig. 9. A rooted DAG input to selective scheduling and the availability set; the number in brackets after each RHS in `av(group)` and in *av* sets indicates the original operation of the RHS.

Figure 9(a). Initially, the `av(group)` is computed by traversing the DAG from the bottom in reverse topological order[8] and by collecting all available RHS across all execution paths (This computation is actually performed by calling `compute_av()` routine, described in the Appendix, from the group boundary). If this computation is done from scratch after each code motion, it will introduce high overhead. To reduce the overhead, an intermediate availability set "*av*" is left at each *basic block header* (bb_header)[9] during the first computation, as shown in Figure 9(b). The data flow set, $av(n)$, is defined to be the set of all available RHS that can move up and *through* n from the "sub-DAG" rooted at n, $G'_n = (V', E')$, where $V' \subset V, E' \subset E$, and V' and E' are vertices and edges reachable from n in G.[10] Our idea is that after each code motion the `av(group)` can be recomputed *incrementally* based on *av* sets saved at the basic block headers closest to the parallel group. This incremental computation is possible because selective scheduling can maintain valid *av* sets efficiently during code motion due to the following two properties.

—During a code motion, the $av(n)$ at a bb_header n might become invalid only when the chosen operation is actually moved up through n; an $av(n')$ that is *not* located on the current path of code motion (which we call the *current moving path*) is not affected.

—The invalid $av(n)$ itself can also be recomputed incrementally based on the *av* sets saved at the bb_headers closest to n in G'_n.

During the code motion step of selective scheduling, $av(n)$ is recomputed on-the-fly only when n is located on the current moving path. Consequently, the portion of a given DAG which is walked through for the computation of the next `av(group)` is confined to the paths of the current code motion, rather than the whole DAG which is walked through during the first computation.

[8]For example, the DAG in Figure 9(a) is traversed in the order of `8, 7, 5, 6, 4`, and `3`.

[9]An operation is a basic block header if it has more than one predecessor, or is a target operation of a conditional branch.

[10]In the example of Figure 9(b), the sub-DAG rooted at `5` includes nodes `5, 7, 8`, and edges $\overline{57}$, $\overline{78}$.

We first describe the computation step using the `moveup()` functions discussed in Section 4 and then the code motion step. We include a brief comparison with other DAG-Based approaches.

5.1 Computation Step

The computation step consists of computing av sets and `av(group)`. Both computations are essentially the same, except that when computing `av(group)` the data dependences on the operations already scheduled in the group must also be considered according to the model of the tree instruction.

5.1.1 *Computing av Sets.* The (re)computation of $av(n)$ is performed incrementally, which means scanning the sub-DAG rooted at n (G'_n) by the depth-first search to identify bb_headers where valid av sets are found, followed by reverse topological-order traversal to collect available RHS (see `compute_av()` in the Appendix). When a valid av set is found at a bb_header m during the depth-first scanning, $av(m)$ is guaranteed to be valid because the overall (re)computation is performed in reverse topological order of the given DAG, G. Actually, we compute the av set for each operation, yet it is not saved unless the operation is a bb_header. Initially, the av set below the DAG is assumed to be empty. Then, the incremental computation of $av(n)$ of an operation n in the DAG is performed as follows:

—**When** $opcode(n) \neq$ `if`: Given the availability set av_0 below an operation n, $av(n)$ above n is computed as $[\texttt{moveup\_set\_rhs}(av_0, n) \cup \{rhs(n)\}]$.

—**When** $opcode(n) =$ `if`: Given that av_T and av_F have been computed on the `T` and `F` paths of the branch n whose operation is `if` cc_j, $av(n)$ becomes $[((av_T \cup av_F) - \text{AllCondBranches}^{11})) \cup \{\texttt{if}\ cc_j\}]$.

If n is a bb_header, then $av(n)$ is saved at n; otherwise it is not saved.

Each RHS Υ in $av(n)$ has an attribute "*spec*," the degree of speculativeness, which tells the maximum number of conditional branches through which this operation would have to be moved up from Υ's original location through n, in a manner that the operation is available at only one target of those branches. It shows the amount of "speculation" or "gambling" involved in moving Υ through n. During the above computation, the following apply:

—**When** $opcode(n) \neq$ `if`: The *spec* attribute of $rhs(n)$ is set to zero in $av(n)$. If a given RHS in av_0 is not true dependent on $oper(n)$, it is placed in $av(n)$ with its *spec* attribute unchanged. When $oper(n)$ is a copy, and if two distinct RHS in av_0 become the same RHS through substitution in $\texttt{moveup\_set\_rhs}(av_0, n)$, the maximum of their *spec* attributes is taken as the *spec* of the RHS.

—**When** $opcode(n) =$ `if`: The *spec* attribute of the conditional branch, (`if` cc_j), is set to zero. For other RHS in $av(n)$, if the RHS is present in both of av_T and av_F, its *spec* attribute is set to the maximum of the two; if the RHS is present in only one of av_T or av_F, its *spec* attribute is incremented by one.

[11] Conditional branches are not allowed to move above other conditional branches in the current implementation.

The above computation of $av(n)$ identifies all available RHS in the sub-DAG and estimates their degree of speculativeness. In Figure 9(b), there are three bb_headers 5, 6, and 7. The initial computation obtains $av(8)$, $av(7)$, $av(5)$, $av(6)$, $av(4)$, and $av(3)$ in this order, yet only $av(7)$, $av(5)$, and $av(6)$ are saved. $av(7)$ includes all RHS on its downward paths, $\mathtt{y}+\mathtt{1}\,(0)$ and $\mathtt{x}+\mathtt{1}\,(0)$, where the number in the parentheses shows the *spec* attribute. We say that `y+1` and `x+1` have come from their *original* operations `u:=y+1` and `z:=x+1`, respectively. $av(5)$ and $av(6)$ are computed as

$$av(5) = \mathtt{moveup\_set\_rhs}(av(7), \mathtt{y} := \mathtt{w} * \mathtt{w}) \cup \{\mathtt{w} * \mathtt{w}\} = \{\mathtt{w} * \mathtt{w}\,(0), \mathtt{x} + \mathtt{1}\,(0)\}$$

$$av(6) = \mathtt{moveup\_set\_rhs}(av(7), \mathtt{y} := \mathtt{z}) \cup \{\mathtt{z}\} = \{\mathtt{z}\,(0), \mathtt{z} + \mathtt{1}\,(0), \mathtt{x} + \mathtt{1}\,(0)\}.$$

$av(5)$ does not include `y+1`, since there is a true dependence on `y:=w*w`, whereas $av(6)$ includes `z+1` that is substituted at `y:=z`. Consequently, even though an element has come from its original operation n, its RHS may differ from the $rhs(n)$ due to substitution on the way up. When we compute $av(4)$ in Figure 9(b) as

$$av(4) = (av(5) \cup av(6)) \cup \{\mathtt{if\ cc0}\} = \{\mathtt{if\ cc0}\,(0), \mathtt{w} * \mathtt{w}\,(1), \mathtt{z}\,(1), \mathtt{z} + \mathtt{1}\,(1), \mathtt{x} + \mathtt{1}\,(0)\},$$

the *spec* of `x+1` is still zero (nonspeculative), since it is present at both targets of the branch, while the *spec* of others is incremented by one (speculative) because they are present at only one target. Figure 10 shows, in more detail, how $av(3)$ (= `av(group)`) in Figure 9(b) is computed, step by step (see procedure **compute_av** in the Appendix).

Through the above incremental computations performed on a given DAG, we can finally obtain the `av(group)` that includes all available RHS with estimated *spec* attached. The *spec* of an RHS in the `av(group)` indicates the maximum (over all paths p from the parallel group to an operation that derived RHS) of the number of conditional branches on the path p where the RHS corresponding to the operation at the end of the path is available in only one target. The parallel group is first filled by nonspeculative RHS whose *spec* is zero, and the remaining resources are filled by speculative RHS whose *spec* is small. Selective scheduling does not reorder branches to reduce code expansion, and thus all branch RHS are nonspeculative and are preferred for code motion.

Each bb_header n includes another data flow set, $lv(n)$, which is the set of registers that are live[12] at n. Before the parallelization starts, correct lv sets are initialized for all bb_headers and the procedure exit labels. Since the lv set also has properties similar to the av set mentioned above, invalid lv sets on the current moving path during a code motion are recomputed incrementally. Given the live variables lv_0 below an operation n, the live variables above n, $lv(n)$, are computed as $(lv_0 - \{dest(n)\}) \cup \{sources(n)\}$; similarly, given the live variables lv_T and lv_F at the targets of a conditional branch *if cc*, the live variables above the conditional branch are computed as $(lv_T \cup lv_F) \cup \{cc\}$. The lv set is used to determine an appropriate target register for the best RHS that is chosen for upward code motion.

[12] A register is live at n if there is a path starting from n in the flow graph where the register is used before being set.

```
---Visit 3.
---Visit 4.  4 has two successors, 5 and 6.  Try 5 first.
---Visit 5.
---Visit 7.
---Visit 8.  8 has no successors.  Compute from oper(8)= z:=x+1 :  av(8) = {
   x+1(0)[8] } (original operation node=8, spec=0).
---Backtrack to 7.  Compute from oper(7)= u:=y+1 and av(8):  av(7)= { y+1(0)[7],
   x+1(0)[8] }.  Save av(7) since 7 is a bb_header.
---Backtrack to 5.  Compute from oper(5)= y:=w*w and av(7):  av(5)= { w*w(0)[5],
   x+1(0)[8] }.  y+1 is not available because y:=w*w sets y.  Save av(5) since 5 is
   a bb_header.
---Backtrack to 4.  4 had another unvisited successor (i.e.  6) besides 5.
---Visit 6.
---Visit 7.  av(7) is already computed, do nothing.
---Backtrack to 6.  Compute from oper(6)= y:=z and av(7):  av(6)={ z(0)[6],
   z+1(0)[7], x+1(0)[8]}.  y+1 gets renamed as z+1, because of forward-substitution.
   Save av(6) since 6 is a bb_header.
---Backtrack to 4.  All successors of 4 are now visited.  Compute from oper(4)=
   if cc0, av(5), av(6):  av(4)= { if cc0(0)[4], w*w(1)[5], z(1)[6], z+1(1)[7],
   x+1(0)[8] }.  x+1 is available in both 5 and 6, so its spec value remains 0.  w*w
   is only available in 5 (and not 6), so its spec value is incremented.
---Backtrack to 3.  Compute from oper(3)= x:=y and av(4):  av(3)= { y(0)[3], if
   cc0(0)[4], w*w(1)[5], z(1)[6], z+1(1)[7], y+1(0)[8] }.  x+1 gets renamed as y+1
   because of forward substitution.
```

Fig. 10. Steps of the computation of av(3) by procedure **compute_av** of the Appendix.

5.1.2 *Computing* `av(group)`. Most of computation details in this section are due to the features of tree instruction, not the features of selective scheduling itself. In order to compute `av(group)`, *av* sets must be computed first at each *group boundary*. A group boundary is an edge in the DAG going from a node inside the parallel group to a node outside the parallel group. The computation of *av* at group boundaries is composed of two steps. First, *av* sets are computed at all nodes that are immediately outside the current group boundaries using the same method for computing $av(n)$ at the bb_headers. Then, the *av* set at each group boundary is further reduced to include only those RHS that are not true dependent on any operations already included in the parallel group, on the path from the boundary where they are computed to the root of the parallel group. The parallel group is maintained as a tree, so the path from a boundary to the root is unique. Substitutable (nontrue) dependences are again disregarded.

Initially, the empty group in a rooted DAG G_r is implemented as a dummy node, Ω; all predecessors of r are made to point to Ω, and Ω is made to be the sole predecessor of r. The dummy node will be deleted after the parallel group is filled. The first group boundary is located between Ω and r in which the first "best" operation will be scheduled. As operations including conditional branches move into the group, there can be more than one group boundary. For example, consider a parallel group that currently has n boundaries x_1, x_2, ..., x_n, and assume we are

about to compute the next `av(group)` for this group. Let x_i be a group boundary between operations p_i and s_i which are inside and outside of the group, respectively. $av(s_i)$ is computed, and it becomes $av(x_i)$. Since bb_headers hold correct av sets, this computation will be incremental, in the sense that it will search at most to the next bb_header on downward paths. Let us define $av'(x_i)$ from $av(x_i)$ as follows:

$$av'(x_i) \equiv \texttt{moveup\_set\_path}(av(x_i), \Psi_{\Omega,p_i}),$$

where Ψ_{Ω,p_i} is the unique path from Ω to p_i. This computation deletes those RHS in $av(x_i)$ that have true dependences on any operations on the path from x_i back to Ω, while all substitutable RHS are substituted. Then, $\bigcup_{1\le i\le n} av'(x_i)$ defines all VLIW RHS that are currently movable into the VLIW tree instruction corresponding to the group. Among them, one RHS Υ_{VLIW} is selected as the best operation, and the best operation will be moved up to every group boundary x_i whose $av'(x_i)$ includes Υ_{VLIW}. Since the scheduling is performed on a sequential program, not on a VLIW program, the best operation should be created at each x_i in the form of a sequential RHS rather than a VLIW RHS as discussed at the end of Section 3.2. $\Upsilon_{Seq}(x_i)$ is defined to be a set of RHS in $av(x_i)$ satisfying

$$\texttt{moveup\_set\_path}(\Upsilon_{Seq}(x_i), \Psi_{\Omega,p_i}) = \{\Upsilon_{VLIW}\}.$$

That is, $\Upsilon_{Seq}(x_i)$ is the set of (one or more) original RHS included in $av(x_i)$ which derived Υ_{VLIW}.

The empty parallel group in Figure 9 is filled with three operations in Figure 11(a). There are two group boundaries, `f1` and `f2`. av(`f1`) becomes the computed av at `u:=y+1`, $\{y+1, x+1\}$, while av(`f2`) is computed as $\{z+1, x+1, z\}$. The element $y+1$ is deleted in av'(`f1`), since it is flow dependent on the operation `y:=w*w`, while the element $x+1$ is substituted at `x:=y`, and the substituted RHS, $y+1$, is included in av'(`f1`). av'(`f2`) is similarly obtained. Assume that $\Upsilon_{VLIW} = y+1$ is selected as the best operation among the union of av'(`f1`) and av'(`f2`). Then, both Υ_{Seq}(`f1`) and Υ_{Seq}(`f2`) become $\{x+1\}$.

Each $\Upsilon_{Seq}(x_i)$ includes the operation that will be actually scheduled in each group boundary x_i although all of them correspond to the final VLIW operation Υ_{VLIW}. In Figure 11(a), for example, $x+1$ is scheduled at the boundary `f1`[13] and `f2` as in Figure 11(c), and they will be converted to $y+1$ in the corresponding VLIW tree instruction. That is, according to our method for scheduling VLIW operations in a group, the selection of the best RHS is based on the VLIW RHS, while the actual scheduling is based on the sequential RHS. It should be noted that even though the best operation has multiple copies in the parallel group (even in different sequential RHS forms), it becomes only one operation in the final tree instruction.

A software lookahead window can be used in the first computation of `av(group)` such that only the RHS of operations within a specific distance from the empty parallel group will show up in the av set at bb_headers. The purpose of this window is twofold. First, it reduces code expansion and the scheduling time. Second,

[13] If `z':=y+1` were created at `f1` instead of `z':=x+1`, it will read the wrong value of `y`, computed at `y:=w*w`. Whereas both forms can be scheduled in `f2`. To be conservative, the sequential RHS is scheduled instead of the VLIW RHS.

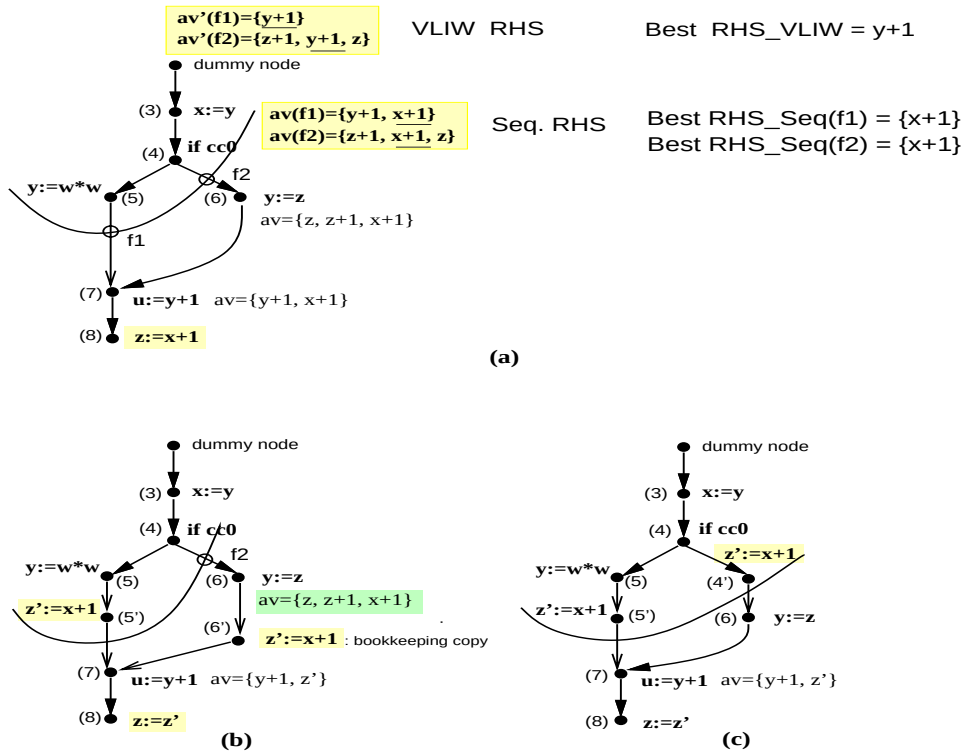


Fig. 11. A code motion into the parallel group.

it prevents code motion over long distances that can increase register pressure and cause the scheduler to run out of registers in subsequent stages. The current implementation uses a window size of 16 operations in all paths from the empty parallel group based on the results in Nakatani and Ebcioğlu [1993], which indicates that the window size of 16 is "enough" to extract most ILP in integer code.

5.2 Code Motion Step

For each boundary x_i where there is a nonempty set of RHS $\Upsilon_{Seq}(x_i)$ corresponding to the chosen Υ_{VLIW}, the algorithm traverses the sub-DAG rooted at x_i to look for the original operations matching $\Upsilon_{Seq}(x_i)$. The traversal is performed twice: first to compute the suitable target registers of the operation to be created at x_i, then to actually move up the original operations into x_i. During the code motion step, the generation of bookkeeping code and the recomputation of av and lv sets on the current moving path are performed on-the-fly.

5.2.1 *Search for Original Operations.* The traversal algorithm for finding the original operations that are being moved up is based on depth-first search. During the traversal on a path, the algorithm backtracks when an instance of the original operation is found. At a conditional branch, both paths of the branch are traversed to find the original operations before the algorithm backtracks. The algorithm also backtracks when the av set at a basic block header indicates that the operation is

not available after that point in the DAG. Due to the time and complexity concerns, only the first original operation on each path will be searched and moved up.

During the depth-first traversal from a boundary x_i, the simplest method to determine if a given operation n is the original operation is by checking if $rhs(n)$ is included in the given $\Upsilon_{Seq}(x_i)$. Since an operation in $\Upsilon_{Seq}(x_i)$ has possibly been substituted to a new form on the way up to x_i, it should be converted back during the traversal.

Let Σ be the set of original RHS that we are looking for during the traversal. In Figure 11(a), if $z+1$ were selected as the best Υ_{VLIW}, the traversal starts from `f2` with $\Sigma = \{z+1\}$. Σ is "unsubstituted" at `y:=z` as $\Sigma = \{y+1, z+1\}$, and both $y+1$ and $z+1$ will be searched for after passing `y:=z`, because either RHS can be possibly moved up above `y:=z` as $z+1$. Note that `moveup_rhs`($y+1$,`y:=z`) =`moveup_rhs`($z+1$,`y:=z`) $=z+1$. We have to look for more RHS than necessary, since there exists no single inverse for substitution. If Σ were $\{x+x\}$ when we pass `y:=x`, Σ becomes $\{x+x, x+y, y+x, y+y\}$. If nontrivial substitution [Nakatani and Ebcioğlu 1989] is allowed, the complexity of traversal might be increased more. Therefore, in this article we restricted the substitution in `moveup_rhs`(Υ, Θ) to happen only when the Θ is a copy operation.

Fortunately, bb_headers encountered on the traversal include the valid av sets, and Σ can be filtered correctly by intersecting with them. In Figure 11(a), Σ becomes $\{y+1\}$ after intersecting with av(`u := y + 1`), which is correct since $z+1$ does not exist below the bb_header. If the resulting Σ after intersecting with $av(n)$ is null, no original operation exists below the bb_header n, and the path below n is not traversed any more. Since the traversal is guided by av sets, the complexity of traversal is reduced, which is one of the reasons why we do recomputation to maintain valid av sets during code motion. The size of Σ is reduced further when any operation Θ which sets a register `r` is encountered during the traversal; all RHS using `r` are removed from Σ, since an RHS including `r` could not have been moved up above Θ (e.g., `moveup_rhs`($r+1$, `r := x * x`) = `NULL`, `moveup_rhs`($r+1$, `r := s`) $= s+1$). In Figure 11(a), the original operation is finally found at `z:=y+1` whose RHS is included in Σ=$\{y+1\}$.

Here is why we do not simply attach to each RHS in `av(group)` pointers to the original operations that derived this RHS and then look for operations matching these pointers in order to find the original operations to move up. We not only need to know which original operations correspond to an RHS, but also which specific paths to follow to get to them from the parallel group. There are cases where an operation that derived the chosen RHS in `av(group)` has to be moved up on one path to the parallel group, but should not be moved on another. Consider again the CFG shown earlier in Figure 3(a), where both `y+1` and `z+1` are computed from the same operation across `T` and `F` paths, respectively. If `z+1` is chosen as the best RHS, the traversal of the `T` path should fail, yet we cannot tell this only by comparing pointers. Our method of comparing RHS with unsubstitution correctly identifies original operations according to the path being traversed, without actually saving the path information in each RHS.

5.2.2 *Choosing the Target Register.* Based on above methods, the sub-DAG rooted at x_i is traversed to find suitable target registers for the operation to be

created at x_i, unless $\Upsilon_{Seq}(x_i)$ consists of a branch or a store operation. It should be a register that is

(1) not set or read on any path from x_i to an instance of the original operation,
(2) not among the live registers of the point immediately following the first original operation on a given downward path, except for the original target register of the operation, and
(3) not live on the other path of any conditional branch that is passed by the operation, in case original operations are not present on both paths of the conditional branch.

After the set of suitable target registers is determined for each boundary x_i, the algorithm takes the intersection of them and chooses the target register. We prefer a target register `d'` equal to the original target register `d` of one of the original operations, since this will eliminate the need for leaving a copy operation `d:=d'` in place of the original operation. If the original target registers cannot be used, and renaming is required, we look for a suitable target register. If a suitable target register cannot be found, the next best RHS is selected, and the process is repeated. Let us call the chosen target register τ. In the example of Figure 11(a), the original target register `z` for the selected best Υ_{VLIW}=`y+1` is used at `y:=z`; if `z:=x+1` were created at `f2`, `y:=z` would read the wrong value of `z`. Therefore, a new target register, τ (equal to `z'` in this case), must be introduced.

5.2.3 *Scheduling Best Operations.* In order to schedule an operation at the group boundary x_i, the sub-DAG rooted at n is traversed again. When an original operation n is found during the traversal, and n is not a conditional branch, its operation "$dest(n) := rhs(n)$" is replaced by "$dest(n) := \tau$" using the chosen target register τ. If $dest(n)$ and τ are the same (i.e., renaming is not required), this operation is deleted. The algorithm backtracks from n, and the traversal continues. Before backtracking from conditional branches, both paths are traversed to find and schedule original operations if they are available.

Bookkeeping copies are made on-the-fly for edges that are not on the current moving path, yet that are joining the current moving path from outside. Each bookkeeping copy may be of a different form depending on the copy operations that were passed on the way up. For the correct generation of bookkeeping copies, the algorithm returns the *current RHS* (C_RHS) when it backtracks, which has the current form of the RHS of the moving operation. When backtracking from an original operation n, the C_RHS becomes $rhs(n)$. When backtracking from a copy operation `x:=y`, `y` is substituted for `x` in C_RHS, if C_RHS uses `x` as a source register. After backtracking from both targets of a conditional branch, if the two targets returned two different C_RHS expressions, one among them is chosen arbitrarily (at this conditional branch, the two expressions must be equal). If there is a joining edge to n before backtracking from n, the algorithm generates a bookkeeping copy "$\tau := \Upsilon$" at the join point (where Υ is C_RHS), for semantic correctness of joining paths. If multiple edges are joining n from outside, only one bookkeeping copy is generated, and all edges are made to point it.

The algorithm will not visit the same node twice during code motion for the following reasons: bookkeeping copies of the original operation are placed at the

```
Moving Σ= { x+1 } to boundary f1:

---Visit 7:  u:=y+1, av(7)= { x+1,y+1 }, Σ= { x+1 }.  Set Σ=(Σ ∩ av(7)).  Σ is
   not null so do not backtrack:  an operation from Σ must be reachable from 7.
   u:=y+1 does not set a source register of a RHS in Σ (otherwise that RHS would be
   deleted).  So Σ is unchanged.
---Visit 8:  z:=x+1, Σ= { x+1 }.  RHS of z:=x+1 is included in Σ (found
   operation!).  Replace 8 by z:=z', where z' is the previously chosen target
   register.  Now backtrack:  8 is not a bb_header so do not recompute av(8), lv(8).
   Set C_RHS=x+1.
---Backtrack to 7:  u:=y+1, C_RHS=x+1.  No substitutions occur to C_RHS since u:=y+1
   is not a copy operation.  Node 7 has a second predecessor 6 not on the current
   path, so create the bookkeeping copy z':=x+1 (x+1= C_RHS) at a new node 6' on the
   edge 6->7.  7 is a bb_header through which we are moving code, so recompute av(7)
   as { y+1,z' }, and lv(7).  Backtrack with C_RHS = x+1.
---Backtrack to boundary f1:  C_RHS=x+1.  Create a new operation z':=x+1 (x+1=
   returned C_RHS) at boundary f1, at a new node 5'.

Moving Σ = { x+1 } to boundary f2:

---Visit 6:  y:=z, av(6)= { z,z+1,x+1 }, Σ= { x+1 }.  Set Σ=(Σ ∩ av(6)).  Σ is
   not null so do not backtrack:  an operation from Σ must be reachable from 6.
   y:=z does not set a source register of a RHS in Σ (otherwise that RHS would
   be deleted).  y:=z is a copy operation but it does not use the same source
   register as the source register of a RHS in Σ (otherwise that RHS would be
   "un-substituted").  So Σ is unchanged.
---Visit 6':  z':=x+1, Σ= { x+1 }.  RHS of 6' is included in Σ (found operation!).
   Replace 6' by z':=z' (which is later deleted).  Backtracking now:  6' is not a
   bb_header, so av(6'), lv(6') are not recomputed.  Set C_RHS=x+1.
---Backtrack to 6:  y:=z, C_RHS=x+1.  y:=z is a copy operation, but it does not set
   a source register in C_RHS=x+1 (otherwise C_RHS would be substituted).  Backtrack
   with C_RHS unchanged.
---Backtrack to boundary f2:  C_RHS=x+1.  In boundary f2, insert a new node 4' with
   operation z'=x+1 (where x+1=C_RHS).
```

Fig. 12. Steps of the **move_op** code motion algorithm.

edges that join the current path being traversed, and when the algorithm arrives at the join point for a second time from a different path, it will first find the original operation Υ in the bookkeeping copy "$\tau := \Upsilon$" that it left previously and will backtrack immediately.[14] This generation of bookkeeping code in our depth-first traversal gives the maximum opportunity for nonspeculative code motion by unifying multiple copies of the same operation below join points.

In Figure 11(a), the traversal from `f1` finds the original operation `z:=x+1` and replaces its operation by `z:=z'` as shown in Figure 11(b). The C_RHS, `x+1`, is moved up, and a bookkeeping copy `z':=x+1` is generated at the edge joining the current moving path. Finally, `z':=x+1` is created at `f1`. In Figure 11(b), the traversal from $f2$ also finds $x+1$ at the bookkeeping copy operation `z':=x+1` and replaces it by `z':=z'`, which is immediately deleted in Figure 11(c). `z':=x+1` is

[14]Even if the RHS Υ of this bookkeeping copy does not match any of the RHS that the algorithm is currently looking for, because of various unsubstitutions that occurred on the way, the algorithm will backtrack immediately anyway. An example of this subtle case is given in the Appendix.

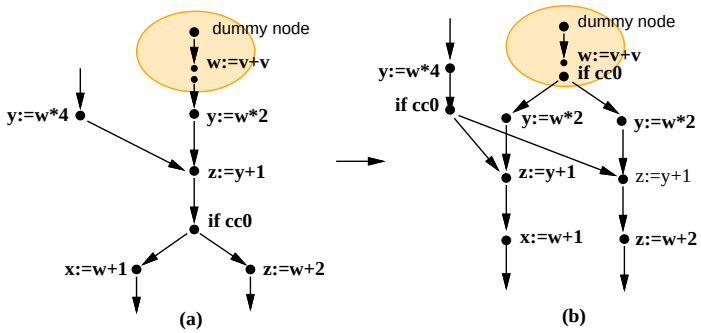


Fig. 13. Code motion of a conditional branch; one copy of `z:=y+1` in (b) can be deleted because it becomes dead after duplication; code expansion in branch code motion can be controlled in this way.

also created at $f2$. Figure 12 shows, in more detail, the step-by-step operation of the code motion algorithm **move_op** of the Appendix, on Figure 11.

A conditional branch `if` cc_j is moved up to a group boundary by making two copies of the operation stream between the group boundary and the original location of `if` cc_j. One copy ends at the `T` target of `if` cc_j, and the other ends at the `F` target of `if` cc_j. Therefore, the algorithm (see procedure `move_cj()` in the Appendix) returns *ins_T* and *ins_F* as well as C_RHS, which are the starting operations of these two operation streams. When `if` cc_j is included in the parallel group or is inserted as a bookkeeping copy, the `T` and `F` targets of it are connected to *ins_T* and *ins_F*, respectively. Figure 13 shows an example of branch code motion. A bookkeeping copy of `if cc0` has been made on the edge joining the moving path. Conditional branches are rarely on the critical path of execution, since the compiler performs speculative code motion and since the machine is capable of executing multiple branches per cycle. Nevertheless, moving them up is useful, because the sooner the branch is performed, the sooner the machine can stop executing speculative operations from untaken paths, thus conserving resources.

5.2.4 *Recomputation of av and lv Sets.* On the paths that were traversed by the move, the *av* and *lv* sets at bb_headers may be invalidated, since some operation on downward paths starting from them either disappeared or was replaced. However, the *av* and *lv* sets that are not located on the current moving path are unaffected. After the bookkeeping copy is generated in Figure 11(b), the *av* and *lv* sets at `y:=z` are still correct even though `x+1` on its downward path has moved up. First, {`z+1,x+1,z`} is still the correct *av* because `x+1` still comes from the bookkeeping copy, and the RHS of the new copy operation, `z'`, cannot be included due to the true dependence on the bookkeeping copy. Second, no new live variables are introduced at `y:=z`. Due to this property, we need to update the *av* and *lv* sets only on the current moving path as we move up operations, and the update is based on the incremental computation described previously. Note that during a code motion some bb_headers might disappear, and new bb_headers might be created; yet they are identified on-the-fly, and bb_headers always retain correct *av* and *lv* sets.

After the best RHS is scheduled, the next `av(group)` is computed incrementally

```
PROCEDURE fill_ins(oper)
/* gather a parallel group before "oper" on the rooted DAG, G_"oper" */
 word = make_empty_vliw_word();
root = create_dummy_root(oper);
av_VLIW = { }
WHILE (true) DO
     FOR each group boundary  x on  p s , where p is inside and s is outside of the group
          av(x) = compute_av(s, path=NULL, ws=0);  /* ws is lookahead window length  so far */
          av'(x) = moveup_set_path(av(x), path(root, p));
          av_VLIW = union(av_VLIW, av'(x));
     END FOR
     (RHS_VLIW, dest_reg) = find_best_RHS_and_dest_reg_that_fits(word, av_VLIW);
     /* If there are no more ready operations that meet resource requirements, return */
     IF (RHS_VLIW, dest_reg) is undefined THEN { delete_dummy_root(root); break;}
     FOR each group boundary x on p s whose av'(x) includes RHS_VLIW  DO
         RHS_Seq = (Elements r in av(x) satisfying moveup_path(r, path(root,p))=RHS_VLIW);
         IF (RHS_Seq == {if cc})
              (C_RHS, Ins_T, Ins_F) = move_cj(s, RHS_Seq, path = NULL);
              Schedule "if C_RHS goto Ins_T else goto Ins_F" at x
         ELSE
              C_RHS = move_op(s, RHS_Seq, dest_reg, path=NULL);
              Schedule "dest_reg = C_RHS" at x
         ENDIF
     END FOR
     word = add_to_vliw_word(word, RHS_VLIW);
  END WHILE
END PROCEDURE
```

Fig. 14. Algorithm to fill a parallel group.

based on *av* sets saved at the neighbor bb_headers of the parallel group; during the code motion, they have either remained untouched if they were not on the path of code motion; otherwise they must have already been updated. The routine `fill_ins()` in the Figure 14 describes the filling process, and the Appendix includes pseudocode for `compute_av()`, `move_op()`, and `move_cj()`. The correctness of the algorithm has been proved in Moon [1993a].

5.3 Comparison with Other DAG-Based Approaches

Our measure of speculativeness is different from that given in Bernstein and Rodeh [1991], which defines the degree of speculativeness of an operation as the number of conditional branches it is control dependent on. In fact, an operation can be control dependent on zero conditional branches yet still be speculative, as depicted in Figure 15. The operation `z:=x+y` can be scheduled speculatively before `if cc0` across the `F` path after generating a bookkeeping copy at the `T` path. Selective scheduling correctly computes the *spec* of `x+y` to be one above the branch as shown in the *av*(`if cc0`) in Figure 15, even though this operation is control dependent on no branches. A control dependence graph alone is not precise enough for estimating speculativeness.

Another DAG-Based approach in Nakatani and Ebcioğlu [1993] uses *greedy scheduling* which, unlike the conventional two-step (computation and code motion) platform, performs local code motion repetitively starting from the bottom of a DAG toward the root; it moves all movable operations from the bottom tree instruction to the one before that, then all movable operations from that instruction to the

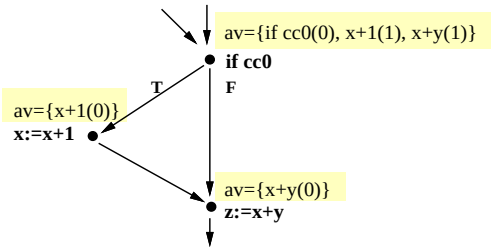


Fig. 15. A speculative RHS that is not control dependent on any branch.

one before that, and so on in reverse topological order of the DAG. When the algorithm arrives at a conditional branch, maximal upward code motion has already been performed at both targets of the branch, and thus the same computations on both targets can be easily unified and scheduled before the branch as a single computation. Consequently, all nonspeculative code motions in the DAG can be performed before the root tree instruction is scheduled.

Unfortunately, greedy scheduling is not appropriate for finite-resource scheduling, because it generates a large number of speculative operations and speculative bookkeeping codes that cannot be accommodated with finite resources; the speculative operations either slow down program execution if their original paths are not taken, or their code motions might be undone in later stages [Nakatani and Ebcioğlu 1993]. Selective scheduling is an approach to enhance greedy scheduling on the efficient two-step platform. Instead of greedy and local code motions, selective scheduling computes available operations first. Unlike other two-step scheduling techniques, our computation is greedy, in that all candidate operations across all paths are computed and that the computation is repeated after each code motion for attaining a precise knowledge of available operations. In addition, the usefulness of each operation is estimated during computation to find the most useful one. The code motion for the selected operation is performed globally without resorting to any intermediate steps as in Nakatani and Ebcioğlu [1993] and Aiken and Nicolau [1988].

6. SOFTWARE PIPELINING AND MEMORY DISAMBIGUATION

The previous section described selective scheduling that schedules a parallel group at the root of a DAG. This section shows how we software pipeline a loop by repetitively applying selective scheduling. Our compiler uses a modified version of the *enhanced pipeline scheduling* algorithm[15] [Ebcioğlu and Nakatani 1989]. We first describe the software pipelining process for innermost loops and then show how nested loops for general procedures are software pipelined. Our techniques for handling memory disambiguation and multilatency operations are also briefly described. Finally, the generation of tree instructions from the parallelized program is discussed.

[15]A special case of this algorithm was independently developed later for pipelining of innermost loops with no branches [Jain 1991].

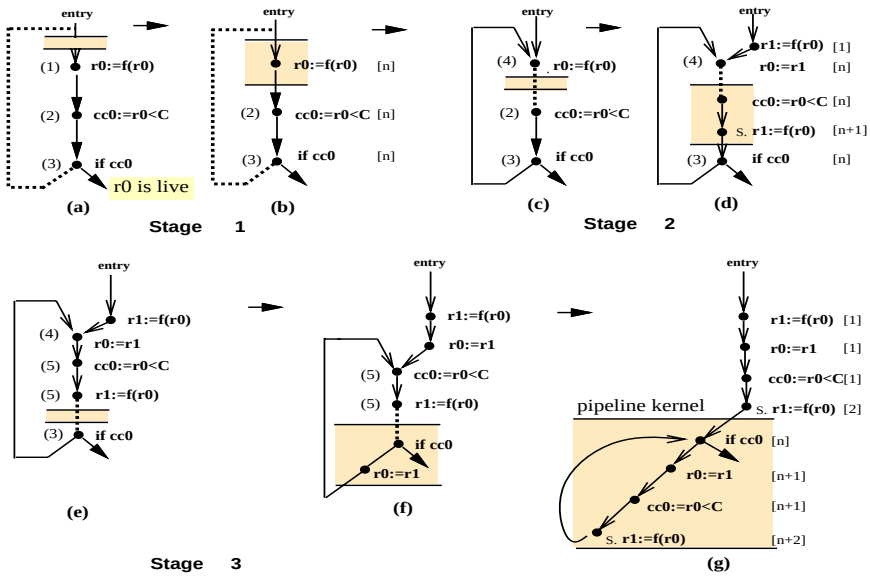


Fig. 16. Software pipelining process for an example loop; brackets after each operation in (b), (d), and (g) show the iteration number, and speculative operation is marked with "S.".

6.1 Software Pipelining of the Innermost Loop

Our software pipelining process is illustrated using an example in Figure 16 where the loop described in Figure 6 is software pipelined. To keep our description manageable, we used a loop with a single execution path as an example, yet our technique is applicable to general loops with multiple execution paths.

Basically, the algorithm repetitively applies two actions: first, cut some edges of the given cyclic graph of a loop to yield an acyclic graph; then, schedule the acyclic graph and generate a parallel group on each edge that was cut. This process is repeated in such a way that parallel groups generated later become increasingly compact, since they can absorb parallel groups generated earlier, achieving tighter schedules for the pipeline kernel. An attractive feature of this algorithm is that the problem of software pipelining is reduced to the simpler problem of global scheduling, so that the repetitive scheduling of DAGs automatically generates the schedules of the pipelined loop kernel, the startup code, and the drain code. More importantly, loops with conditional branches can be pipelined with a variable initiation interval so that a variable cycles/iteration rate can be achieved depending on which path through the loop is followed at execution time (see the example in Figure 2).

As shown in Figure 16, three *stages* are required to pipeline the given innermost loop; at each stage, the cyclic graph is cut at certain edges to define a DAG. Those edges that are cut at each stage are called *backward edges* whereas other edges are called *forward edges*. Backward edges are depicted as dotted lines in Figure 16. Our algorithm creates an empty parallel group just before the target node of each backward edge (which we call a *fence* group) and schedules the parallel group using selective scheduling.

The distinction between backward and forward edges in each stage is made by manipulating the sequence number (*seqno*) of each operation, which is initially set to the depth-first ordering number (topologically sorted) of the operation assigned from the loop entry [Aho et al. 1986, p. 660]. The edge from a higher to a lower *seqno* in each stage becomes a backward edge; in Figure 16(a), the edge from the *seqno* (3) to (1) becomes the first backward edge on which an empty fence group is created. The fence group is scheduled by code motion across paths of forward edges whose *seqno* are nondecreasing. After the parallel group is scheduled, all scheduled operations in the group are assigned a new *seqno* which is unique and greater than the maximum *seqno* used so far. Consequently, previous backward edges become forward edges and are available for code motion, and new backward edges are defined starting a new stage.

In Figure 16(b), the fence group is filled with `r0:=f(r0)` and is assigned a new *seqno*, 4 (see Figure 16(c)); the successor edge of `r0:=f(r0)` becomes a new backward edge on which a new fence group is created. During the scheduling of the fence group in Figure 16(d), the RHS `f(r0)` moves into the group speculatively ahead of the loop exit branch, after renaming its target register with `r1` since `r0` is live at the exit. This operation `r1:=f(r0)` is scheduled as a next ($(n+1)$st) iteration operation and is scheduled concurrently with the current (nth) iteration operation `cc0:=r0<C`, thus allowing software pipelining. During the code motion, a bookkeeping copy is "popped out" at the loop entry which belongs to the first iteration and comprises the pipeline startup code. In the third stage, `r1:=f(r0)` is again scheduled into the fence group as an $(n+2)$nd iteration operation, after popping out its copy of the second iteration at the loop entry (see Figure 16(g)). Consequently, an operation scheduled in the previous stage as an mth iteration operation is available for code motion in the current stage as an $(m+1)$st iteration operation. Finally, a pipelined loop kernel performing at the rate of one cycle per iteration is obtained in Figure 16(g), where the executions of nth, $(n+1)$st, and $(n+2)$nd iterations are overlapped (see the corresponding tree instruction in Figure 6).

In Figure 16, parts (b), (d), and (g), each operation is assigned its iteration number (within brackets) and is marked with "`s.`" if it is speculative. Since the pipelined code is obtained by legal code motions, it should produce the same execution results as the original code. If the original loop in Figure 16(b) iterates C times (i.e., $1 \leq n \leq C$), the operation `if cc0` in the pipelined kernel in Figure 16(g) also executes exactly C times because it is nonspeculative. Consequently, all operations in the kernel excluding `if cc0` execute C-1 times. Now, if we sum up how many times each RHS in the original code is executed in the pipelined code, we can find that all RHS are executed C times excepting `f(r0)`, which is executed $C+1$ times (i.e., two before entering the kernel and C-1 times in the kernel), yet the last execution result is discarded (actually, no further operation uses the result, since `r1` is not live at the loop exit). This is due to the speculative execution involved in `r1:=f(r0)` that was scheduled ahead of the loop exit branch, thus being useful only when the following loop exit branch does not exit and the copy `r0:=r1` is executed. The speculative code motion ahead of loop exit branches is the fundamental means of achieving software pipelining in our compiler.

If the loop body includes multiple conditional branches, there may exist more

than one fence group at a stage. Each fence group is scheduled separately by selective scheduling and is assigned a unique *seqno*. All selective scheduling processes in the same stage can share the *av* sets at the bb_headers because all of them are performed on the same DAG. Therefore, valid *av* sets must be saved at the bb_headers after the scheduling of each parallel group, which is another reason for our recomputation of *av* sets. However, the *av* sets are computed from scratch when a new stage starts because a new DAG is defined.[16]

The pipeline startup code is generated automatically as bookkeeping code when operations are scheduled before the join point at the loop entry. Similarly, the pipeline drain code is generated automatically if the loop exit branch is moved up and if operations on its way are duplicated; those duplicates located in the exit comprise the drain code. The particular example in Figure 16 does not generate any drain code.

The procedure `pipeline()` in Figure 17 describes the driver routine for software pipelining, which repetitively calls selective scheduling to schedule fence groups in each stage. The pipelining process repeats until all operations in the current iteration are scheduled. In order to make `pipeline()` finish, selective scheduling prefers operations that belong to earlier iterations. The current implementation relies on three heuristic attributes of a RHS to prioritize `av(group)`: iteration number, degree of speculativeness (*spec*), and original depth-first ordering number in the original loop, with their importances in this order. To encourage software pipelining, the *spec* attribute of a RHS is not incremented when it is computed ahead of a loop exit branch, yet in the final code speculative operations are marked precisely for correct exception handling.

The tree VLIW architecture can mimic the exact exception behavior of the original user program, since it has hardware support for speculation; see Ebcioğlu [1988], Silberman and Ebcioğlu [1993], and Ebcioğlu and Groves [1990]. The discussion of the machine hardware and the compilation support for precise speculative exception handling is beyond the scope of this article.

6.2 Software Pipelining of Nested Loops

Our software-pipelining method described above extends directly to nested loops. The nested loop hierarchy of a procedure is first determined using an interval analysis technique [Schwartz and Sharir 1979]. After the inner loop is software pipelined, the operations that were "popped-out" from the strongly connected part of the inner loop (i.e., the startup and drain code of the software-pipelined loop) are merged with the outer loop, and the outer loop is software pipelined. The strongly connected part of the inner loop is treated specially, as if it were a superinstruction that reads and writes multiple registers and memory locations and that branches to multiple targets. The *av* and *lv* sets are computed in a similar way as in the innermost loops, and operations under an inner loop can be scheduled above the inner loop if dependences permit. In this manner, all loops in the procedure are software pipelined, starting from innermost loops and working toward outer loops in the nested loop hierarchy in reverse topological order of the loop hierarchy tree.

[16] In order to invalidate *av* sets saved in the previous stage, the pseudocode in the Appendix uses `global_level` variable that is incremented after each stage (see also Figure 17).

```
PROCEDURE pipeline(loop_entry_oper)
FOR each operation n in the loop DO
    seqno(n) = depth_first_ordering_number(n);
orig_max_seqno = max({seqno(n)| n is an operation in the loop});
fence = {loop_entry_oper};   global_level = 0;
WHILE (fence is not empty) DO
    let max_f and min_f be the maximum and minimum seqno used in the current fence
    FOR each operation   n   in fence (in descending seqno)  DO
        fill_ins(n);   /* Create an empty group R before n with seqno(R)= seqno(n) − max_f −1   */
                       /*  Operations are moved up into R across edges with non−decreasing seqnos */
                       /*  All operations in the group R now have the same sequence number, seqno(R)  */
    END FOR
    new_fence = { s |  ∃ a predecessor p of s such that seqno(p) is the same as the seqno(R) of
                       one of filled fence groups R  AND  0 < seqno(s) <= orig_max_seqno}
    FOR each filled fence group R DO
        increase the seqno of all operations in R by highest_seqno_in_use + 1 + (max_f − min_f + 1)
    END FOR
    fence = new_fence; /* Start a new stage */
    global_level++; /* Invalidate av sets saved in the previous stages (see Appendix B) */
  END WHILE
END PROCEDURE
```

Fig. 17. Algorithm for software pipelining of the innermost loop.

In the outermost interval of the entire procedure, only DAG compaction occurs, unless the entire procedure is itself a loop.

6.3 Memory Disambiguation

We briefly describe how dependences due to memory references are determined and resolved in the context of software pipelining. A memory load/store operation includes an address derivation (*aderiv*) attribute describing the memory location accessed, which is a symbolic linear expression possibly including the induction variables of the enclosing nested loops, other variables, constant base addresses of external data structures, and integer constants. During the computation of `moveup_rhs(`Υ, Θ`)`, store-after-load, load-after-store, and store-after-store dependences are checked if Υ and Θ are memory operations. To decide if Υ and Θ can refer to overlapping locations, our memory disambiguation technique symbolically subtracts their *aderiv* expressions and tests whether the subtraction result can be zero within the given data sizes. If it is guaranteed not to be zero, two memory accesses are independent, and `moveup_rhs(`Υ, Θ`)` returns Υ; otherwise it returns `NULL`. Whenever a memory operation in a filled fence group is made to belong to the next iteration, its *aderiv* equation is appropriately modified to perform the correct memory disambiguation between iterations. Our technique contains improvements over the one used in the Bulldog compiler [Ellis 1985], in the sense that we rediscover the induction variables from the assembly code where the original induction variables have been eliminated and that we update *aderiv*s to deal with the code motions of software pipelining. More details about our memory disambiguation algorithms can be found in Moon and Ebcioğlu [1993].

6.4 Multilatency Operations

Pipelined operations such as memory loads or divide operations are scheduled by the compiler such that their target registers are not set or used for at least their latencies, n (> 1). This can be accomplished by adding n-1 `delay` operations, `x:=delay(x)`, after each pipelined operation `x:=`*oper* in the sequential program, and the code scheduling is performed as usual. Delay pseudo-operations have the same semantics as copy operations, yet do not take resources and are not subjected to any optimizations of copies such as forward substitution or copy propagation. Renamed delay operations during software pipelining are removed through coalescing after unrolling the loop kernel [Moon et al. 1997]. The final VLIW code does not include any delay operations, and if a VLIW instruction contains only delay operations it is converted into a no-op VLIW. If the machine detects stall conditions automatically, the no-op VLIWs can be removed.

The above technique for dealing with multilatency operations with `delay` is sufficient for pipelined functional units that can accept a new operation every cycle, yet where each operation, once issued, takes n cycles to complete without further contention for resources. This is the case for most operations on modern microprocessors such as RS/6000 [IBM 1990]. That is, when a load or a floating-point operation in RS/6000 whose latency is l is issued in cycle n, it will complete without encountering any resource constraints, and its result register is usable by other operations in cycle $n+l$ and later. This is guaranteed since each functional unit has its own result bus and register write port. The Intel i860 also has pipeline constraints similar to the RS/6000 for load-use and compare-branch delays [Atkins 1991], and the same scheduling method could be applied.

Some older VLIW architectures may involve further contention for resources beyond the issue cycle (e.g., there may be a result bus shared by both an adder and a multiplier). This scheduling problem has been solved using *reservation tables* in the context of scheduling loops with a single basic block [Dehnert and Towle 1993; Lam 1988]. Extending the concept of reservation tables to general loops with conditional branches and control join points is also possible in the context of our scheduling algorithms, yet it is the beyond the scope of this article.

6.5 Generation of VLIW Tree Instructions

The VLIW program is obtained from the parallelized program through local transformation. Since a parallel group is a maximal connected subgraph having the same *seqno*, a VLIW tree instruction can be generated by collecting operations starting from each entrance of a parallel group. If a path in a parallel group includes a copy operation and an operation that is data dependent on the copy, the operation is substituted to obtain Υ_{VLIW} from each $\Upsilon_{Seq}(x)$. Similarly, if a path in a group includes an operation "$\tau := \Upsilon$", and a copy "$dest(n) := \tau$," the copy is merged into "$dest(n) := \Upsilon$" to obtain the original operation, when the $C\_RHS = \Upsilon$ was moved out of a scheduled fence group and generated a bookkeeping copy at the group entrance in Section 5.2.3 (we assign the bookkeeping copy the same *seqno* of the group). We check data dependences and resource constraints conservatively when collecting operations in a group and putting them in a tree instruction.

A *peephole compaction* technique is performed to merge adjacent VLIW instruc-

tions, if the resource constraints are not exceeded and if there are no data dependences between their operations. This is implemented in the code transformation as follows: operations are collected into an empty VLIW tree instruction starting from an entrance of a parallel group; when a current group boundary x is met,[17] additional operations after x are conditionally included such that if the whole next group starting from x can be accommodated in the current VLIW instruction, they are included; otherwise none of them are included, and a new VLIW instruction starts from x. This conditional inclusion guarantees not to increase the number of VLIW instructions in the final code. Also, the merging process allows putting operations both inside and outside of a loop into the same VLIW, thus exposing further parallelism not exposed during the parallelization phase where the strongly connected part of an inner loop is not touched once it is software pipelined.

7. EXPERIMENTAL RESULTS

The previous sections provided the details of our compilation techniques. In this section, we describe our experimental results performed on selected integer benchmarks using the selective scheduling compiler. The objective of this experiment is to examine the performance and the efficiency of the compiler for its practical applicability to ILP scheduling; it is not intended to evaluate the effectiveness of our scheduling algorithms compared to others. Since most ILP compilation techniques are quite sophisticated, it is generally difficult to implement other scheduling techniques completely and to make a fair and detailed comparison in the same framework. The experimental results in this section give an abstract measure of the value of our compilation techniques.

7.1 VLIW Environment

The experiments have been performed as follows. The input C code is first compiled using the PL.8 optimizing compiler [Warren et al. 1986], which performs the traditional code optimizations including register allocation and generates our version of RISC assembly code (basically similar to the 801 instruction set, but with only simple addressing modes such as absolute and register indirect); each operation is assumed to take a single cycle.[18] This sequential code is then parallelized into VLIW code. Finally, the VLIW code is executed on a VLIW simulator, producing outputs and execution statistics.

The benchmarks used are the four SPEC 89 integer benchmarks and five AIX utilities (listed in Table I). For the SPEC 89 integer benchmarks, the official input files were used. For AIX utilities, we used our own test files. In all cases, programs were simulated to completion.

Our model of the tree VLIW machine is based on the parametric resource constraints of n-ALUs and n-way branching. Each ALU can perform only one operation

[17] A boundary is detected when the *seqno* of x differs from the *seqno* of the entrance or when x has multiple predecessors.

[18] Most operations in integer code take a single cycle. Since only a register-indirect mode is used, no address addition is involved in memory load/store; for example, our memory load is performed through [`r1:=r2+4; r3:=load(r1)`], which is equivalent to `r3:=load(4, r2)` with a two-cycle latency as in IBM RS/6000, in terms of the length of critical paths.

Table I. Benchmarks Used in the Experiment

Benchmarks		Number of Instructions		Input Data Files
		Static	Dynamic	
SPEC Integer	eqntott	7,665	1,378,107,572	int_pri_3.eqn
	espresso	61,204	2,972,579,201	bca.in,ti.in,tial.in,cps.in
	li	25,314	8,634,011,086	9 queens
	gcc	192,116	1,473,787,410	19 C files
AIX Utility	sort	2,818	3,005,512	phone directory
	yacc	9,350	893,737	grammar of calculator
	sed	5,197	1,185,023	phone directory
	compress	2,481	173,049	a small test file
	fgrep	1,233	1,045,600	phone directory

Table II. The Resource Constraints of the Four VLIW Machines

Machine	ALU_ops	mem_ops	ALU_ops + mem_ops	n-way branching
16-ALU	16	8	16	16
8-ALU	8	4	8	8
4-ALU	4	2	4	4
2-ALU	2	1	2	2

in each cycle. All ALUs can perform ALU operations, and half of them can perform memory operations. (This model corresponds to the integer unit of RS/6000 [IBM 1990], which can perform either an integer operation or a memory operation in a cycle.) Multiway branching is assumed to be handled by a dedicated branch unit without using ALUs. Therefore, a VLIW tree instruction on an n-ALU and n-way branching machine should satisfy the following resource constraints.

$$\texttt{num\_ALU\_ops} \le n \quad \texttt{num\_memory\_ops} \le \frac{n}{2} \quad \texttt{num\_test\_nodes\_in\_the\_tree} < n$$
$$\texttt{num\_ALU\_ops} + \texttt{num\_memory\_ops} \le n$$

We have evaluated four machines, with $n = 2$, 4, and 8, 16; all these machines have 128 general-purpose registers[19] and 16 condition registers, whereas the object code generated by PL.8 is based on 32 general-purpose registers and one condition register. The four machines are summarized in Table II, and thereafter are referred to as `16-ALU` machine, `8-ALU` machine, `4-ALU` machine, and `2-ALU` machine, respectively.

During our experiments, we have simulated several billion VLIW instructions. We have obtained correct results on all benchmarks in all machines. A *compiled simulation technique* is used to reduce the simulation time on these benchmarks. Each tree instruction generated by the parallelizer is translated into dedicated C code. The code first updates statistics and histogram tables according to the properties of the tree instruction. Then, it executes the specific arithmetic operations, load/stores and tests required by the tree instruction, and updates the machine registers and memory. Finally, it branches to the label of another similar piece of dedicated code for the next tree instruction. I/O is performed within all of these benchmarks in a memory-mapped fashion, by detecting loads and stores to certain

[19]The final VLIW code rarely includes a register that is in the range of between `r64` and `r127`. This is due to the routine for the selection of a target register in `fill_ins()`, which searches a suitable one starting from `r0` for conserving reuse.

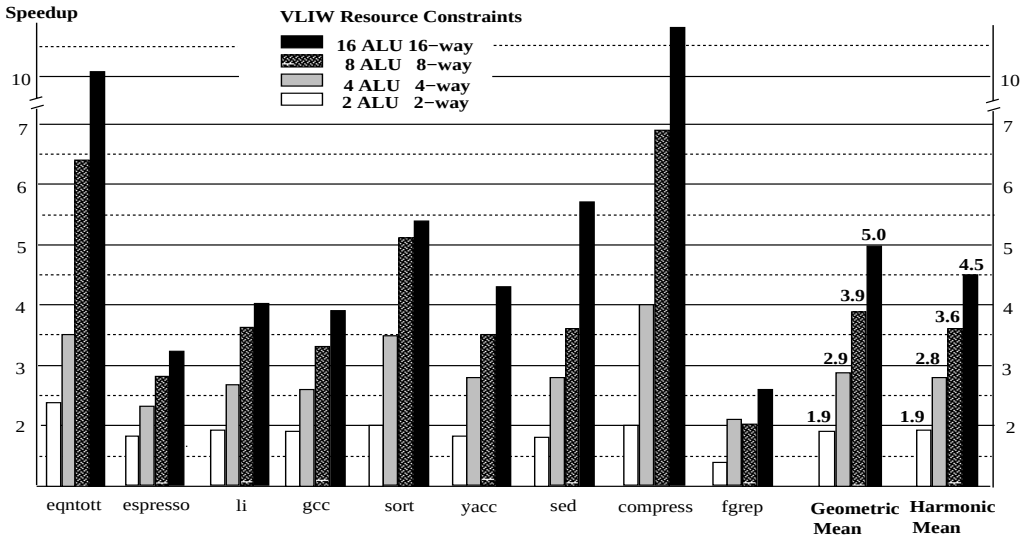


Fig. 18. VLIW performance of four machines over the sequential RISC on the benchmarks.

fixed addresses in the simulator and by interpreting the I/O request in the host file system.

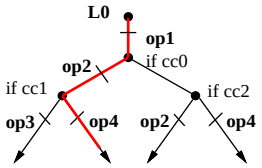
7.2 Experimental Results

We parallelized each benchmark, based on the different resource constraints listed in Table II, and compared the VLIW code with the sequential code generated by the PL.8 compiler to compute the VLIW speedup. Both codes have been simulated on the same VLIW simulator. For sequential simulation, each instruction in the original sequential code became a tree instruction with either a single data operation or a single branch. We assumed perfect caches in both simulations.[20] The utilization of ALUs and the parallelization overhead were also examined.

7.2.1 *Performance.* Figure 18 shows the speedup of VLIW code over sequential code. The speedup of each benchmark is computed by dividing the sequential execution count (the number of single-operation tree instructions in the sequential execution trace) by the VLIW execution count (the number of tree instructions in the VLIW execution trace) of the benchmark. The speedup in this experiment is computed by comparing only the execution counts in order to measure the static performance advantage of code scheduling, independent of dynamic factors such as cache misses or memory traffic.

Figure 18 includes both the geometric mean and the harmonic mean of speedups. The `2-ALU` machine, which can execute a maximum of two data operations and one conditional branch in a single cycle, obtains a geometric mean of 1.9x speedup. The speedup increases almost linearly when the number of ALUs and the number of test nodes are doubled in each step, thus showing a logarithmic speedup increase. The

[20] For some cache studies with VLIW tree instructions, the reader is referred to Moon [1997].



Specified Ops :	**op1 op2 op3 op4**	if cc0 if cc1 if cc2
Performed Ops :	**op1 op2 op4**	if cc0 if cc1
Useful Ops :	**op1 op4**	if cc0 if cc1

Fig. 19. Classification of operations in a dynamic tree instruction.

`16-ALU` machine obtains a geometric mean of 5x speedup.[21] Our results show that the selective scheduling compiler is useful for the code optimization of machines with many resources as well as machines with few resources[22]

7.2.2 *ALU Utilization.* We also measured how the given ALUs are utilized in each cycle to achieve the speedup. The operations in a tree instruction are classified into three categories. The entire set of operations in a tree instruction is called *specified* operations, whereas those on the taken path of the tree are called *performed* operations. Among the performed operations in each cycle, the execution of some operations does not contribute to the final performance because they are either speculative operations from untaken paths or copy operations that have not been eliminated after partial renaming. Those operations that are used to actually increase the speedup are referred to as *useful* operations.

Figure 19 describes the specified and performed operations when the path of $\texttt{cc0} \cdot \overline{\texttt{cc1}}$ is taken in the tree instruction `L0`. Let us assume that `op2` is a speculative operation from some untaken path. Then, the useful operations do not include `op2`. It should be noted that all performed conditional branches are useful, because no speculative conditional branches are generated by the selective scheduling compiler.

Figure 20 shows the ALU utilization of the experiments in Figure 18. There are four bars in each benchmark that correspond to the four machines. Each bar includes the average number of specified, performed, and useful ALU or memory operations executed per cycle. The average number of specified and performed operations has been obtained by averaging the number of operations present in each cycle in the dynamic execution trace. The average number of useful operations is computed approximately from the speedup and from the number of performed operations. That is, the difference between the speedup and the average number of total performed operations (ALU+memory+branch) in each cycle is the average number of useless operations executed in each cycle. Since all performed branch operations should contribute to the final performance, all the useless operations come from the performed ALU or memory operations. If this difference is sub-

[21] `eqntott` shows a higher speedup due to a highly compacted tree instruction in the subroutine "cmppt"; the critical inner loop executes at a rate of one cycle/iteration. The higher speedup in `compress` is due to a loop in the inlined subroutine "cl_hash" which is frequently executed. The loop is software pipelined into a highly compacted tree instruction. `fgrep` has the lowest speedup. One of the reasons is that its frequent execution path is an outer loop where the enclosed inner loop acts as a coarse-grain serializing operation, and its pipeline startup and drain codes slow down the execution of the outer loop.

[22] A recent implementation of the compiler for the SPARC instruction set has achieved similar speedup curves [Park et al. 1997].

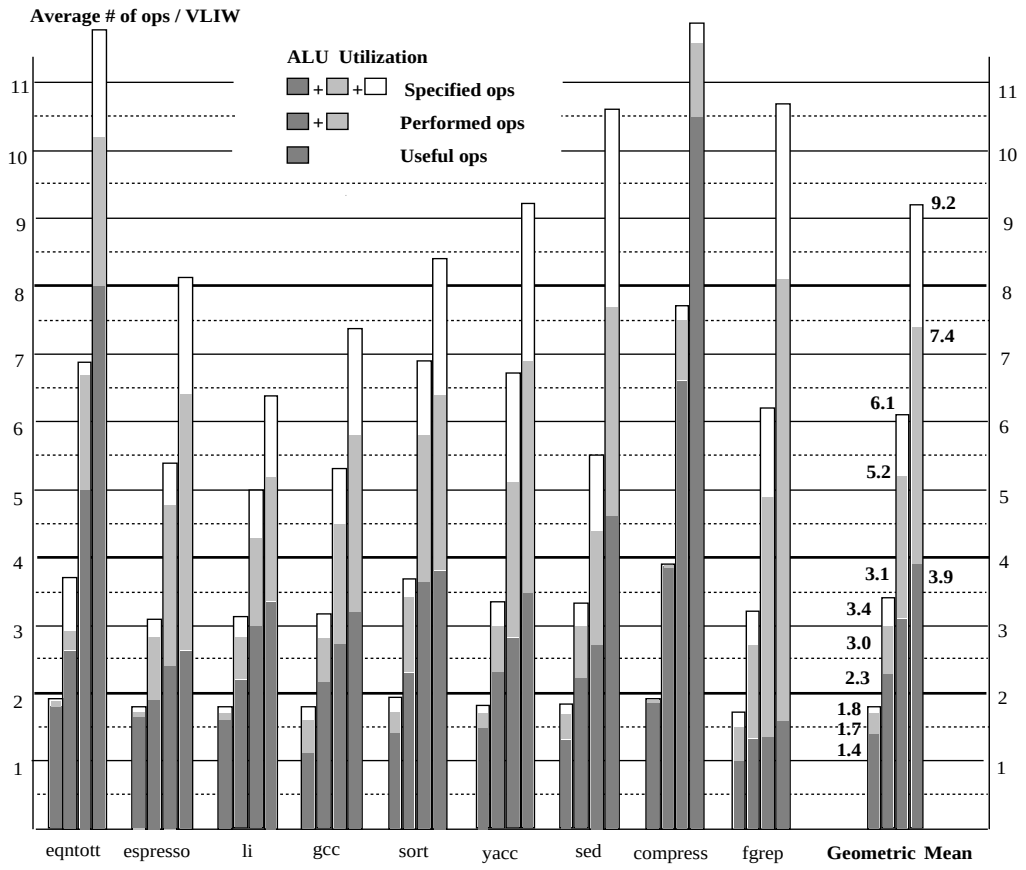


Fig. 20. ALU utilization of the benchmarks in the four machines.

tracted from the average number of performed operations on ALUs, we can obtain the average number of useful operations in each cycle.

In Figure 20, the `16-ALU` machine specifies a geometric mean of 9.2 operations in each cycle, yet only 7.4 operations among them are actually performed. Among the 7.4 operations, only 3.9 operations actually increase the speedup. This can be rephrased as follows:

—9.2 ALUs out of 16 start execution at the beginning of the cycle;

—7.4 ALUs out of 9.2 commit their execution results; and

—3.9 ALUs out of 7.4 are actually used to increase speedup.

The machines with fewer ALUs obtain better utilization and waste fewer ALUs. The reason we achieve better performance when there are more ALUs is that the number of useful ALUs still increases, even though the number of wasted ALUs also increases.[23] The first waste factor among specified and performed ALUs comes from

[23]The higher speedups in `eqntott` and `compress` are associated with good utilization of ALUs. In

the artifact of conditional execution, which helps to reduce the cycle time. The second waste factor between performed and useful ALUs mainly comes from scheduling speculative operations on untaken paths. It should be noted that these results were obtained without using any branch probabilities during selective scheduling. If branch probabilities were used to distinguish the priority of speculative code motion, this waste factor would be reduced, and the speedup would increase.

7.2.3 *Code Expansion.* The average code expansion ratios of VLIW code over the original sequential code in the four machines are as follows: 1.0 (`2-ALU`), 1.3 (`4-ALU`), 1.8 (`8-ALU`), 2.1 (`16-ALU`). The code expansion ratio is computed by comparing the number of operations in the original unparallelized code and the number of operations in the final VLIW trees. Multiple occurrences of the same operation in a given VLIW tree are counted as a single operation, and no-ops are not counted. (This is a reasonable estimate for a compacted representation of VLIW code in memory. Additional no-ops would have to be recreated when fetching instructions back into the instruction cache to fill the unused slots in each VLIW word [Colwell et al. 1988].)

The average code expansion ratio decreases in machines with fewer ALUs because the number of code motion decreases, thus requiring fewer bookkeeping codes. There is some code reduction when the sequential code is converted into the sequential program, since the unconditional branch instructions disappear in the latter representation. In addition, speculative code motion along with unification and substitution may result in opportunities of common-subexpression elimination or copy propagation that the conservative sequential optimizations missed. This reduces the code expansion somewhat in the benchmarks, and the `2-ALU` machine incurs almost no code expansion even though there still is code duplication during code scheduling.

7.2.4 *Parallelization Time.* Table III shows the parallelization time (from the sequential object code to the VLIW code) of the benchmarks in the four VLIW machines. We compare the parallelization time with the original C compilation time of the PL.8 compiler (from C source code to the object code) to determine the parallelization overhead. The computer used in the compilation is the IBM 9021.

The parallelization time decreases when the amount of resources decreases due to fewer number of code motions. All the machines have been parallelized with the software lookahead window size equal to 16 on all execution paths. If we use a smaller lookahead window size when the number of ALUs is small, the parallelization time would be reduced without affecting the performance, since most of the scheduled operations will come from the close neighborhood of a parallel group.

Our parallelization time is consistently smaller than the C compilation time of the PL.8 compiler on all benchmarks, even for the most relaxed `16-ALU` machine. Based on this result, we claim that one can expect to obtain the speedup depicted in Figure 18 by increasing the original compilation time at most twice (and in probably less time with further tuning).

`fgrep`, the average number of specified and performed ALUs in each cycle is not significantly less than that of other benchmarks, yet it has the smallest speedup since most of them are wasted.

Table III. Parallelization Time and PL.8 C Compilation Time of Benchmarks on the IBM 9021 in the Four Machines

Benchmarks	PL.8 Time	Parallelization Time (seconds)			
		16-ALU	8-ALU	4-ALU	2-ALU
eqntott	50.29	11.47	8.59	5.46	4.14
espresso	168.96	107.44	81.72	49.55	33.74
li	94.75	26.47	21.64	16.36	13.19
gcc	435.87	344.88	238.54	151.53	113.94
sort	7.59	5.84	4.14	2.75	1.81
yacc	21.18	17.26	13.15	7.93	5.56
sed	12.57	7.23	5.77	3.92	2.67
compress	6.21	3.02	2.33	1.71	1.35
fgrep	4.21	1.86	1.31	0.99	0.70
Total	801.63	525.47	377.19	240.20	177.10

8. SUMMARY

This article has introduced selective scheduling to exploit irregular ILP in nonnumerical code and has examined the performance and efficiency of statically scheduled machines using the selective scheduling compiler. Selective scheduling is based on three concepts: (1) greedy computation based on renaming, substitution, and all-path speculation, (2) selective global code motion based on estimated usefulness rather than branch profiling, and (3) scheduling of conditional branches. Selective scheduling combined with software pipelining can generate high-performance code with a reasonable amount of scheduling overhead. Consequently, the approach of compile-time scheduling is shown to improve previously known characteristics of irregular ILP.

APPENDIX

A. A PARALLELIZATION EXAMPLE IN THE 16-ALU MACHINE

A.1 Input C Code

```
/* procedure to search through a linked list for a matching item x */
struct recrd *srch(x,l)
struct recrd *l;
{
register struct recrd *p;
for(p=l;p!=NULL && p->data!=x;p=p->link);
return p;
}
```

A.2 Sequential Code (with Comments)

```
      ;Most operations have the form (opcode src1 ... srcn dest)
      ;Offset of link field=0, offset of data field=4
      ;in struct recrd
      (PROC srch)                          ;x is r2, l is r3
      (EQ $r3 0 cc$4)                      ;cc4=(l==NULL)
      (IF cc$4 (GOTO $$L9))                ;if (l==NULL) return l
      (ADD $r3 4 $r0)                      ;r0= &(l->data)
      (LOAD _M$MEMORY_data_1365 $r0 $r0)   ;r0=(l->data)
      (EQ $r0 $r2 cc$4)                    ;cc4=(l->data==x)
      (IF cc$4 (GOTO $$L9))                ;if (l->data==x) return l
$$L7  (LOAD _M$MEMORY_link_1364 $r3 $r3)   ;r3=(l->link)
```

```
      (EQ $r3 0 cc$4)                          ;cc4=((l->link)==NULL)
      (IF cc$4 (GOTO $$L9))                    ;if ((l->link)==NULL)
                                               ;return (l->link)
      (ADD $r3 4 $r0)                          ;r0= &((l->link)->data)
      (LOAD _M$MEMORY_data_1365 $r0 $r0)       ;r0=((l->link)->data)
      (EQ $r0 $r2 cc$4)                        ;cc4=((l->link)->data==x)
      (IF (NOT cc$4) (GOTO $$L7))              ;if ((l->link)->data!=x)
                                               ;set l:=(l->link), loop back
                                               ;else return (l->link)
$$L9  (COPY $r3 $r2)                           ;r2=return value register
      (GOTO $$ENDLABEL)                        ;means return to the caller
      (PEND srch)
```

A.3 VLIW Code Generated by Our Compiler (with Comments)

```
(PROC srch)                              ;x is r2, l is r3
(srch
 ((EQ $r3 0 cc$4)                        ;cc4=(l==NULL)
  (ADD $r3 4 $r0)                    S.;r0=&(l->data)
  (LOAD _M$MEMORY_link_1364 $r3 $r1) S.;r1=(l->link)
  (GOTO srch_$$4) ))                     ;goto next VLIW
(srch_$$4
 ((IF cc$4                                    ;if (l==NULL)
      ((COPY $r3 $r2)                         ;return l
       (GOTO $$ENDLABEL))                     ;r2 = return value register
   ELSE                                       ;if (l!=NULL)
      ((LOAD _M$MEMORY_data_1365 $r0 $r0)     ;r0=(l->data)
       (EQ $r1 0 cc$0)                   S.;cc0=((l->link)==NULL)
       (ADD $r1 4 $r4)                   S.;r4=&((l->link)->data)
       (LOAD _M$MEMORY_link_1364 $r1 $r5) S.;r5=(l->link->link)
       (GOTO srch_$$3) )) ))                 ;goto next VLIW
(srch_$$3
 ((EQ $r0 $r2 cc$4)                       ;cc4=((l->data)==x)
  (LOAD _M$MEMORY_data_1365 $r4 $r0) S.;r0=((l->link)->data)
  (EQ $r5 0 cc$1)                    S.;cc1=((l->link->link)==NULL)
  (ADD $r5 4 $r4)                    S.;r4=&((l->link->link)->data)
  (LOAD _M$MEMORY_link_1364 $r5 $r6) S.;r6=(l->link->link->link)
  (GOTO srch_$$2) ))                     ;goto next VLIW
(srch_$$2
 ((IF cc$4                          ;if ((l->data)==x)
      ((COPY $r3 $r2)               ;return l
       (GOTO $$ENDLABEL) )          ;r2 = return value register
   ELSE                             ;if ((l->data)!=x)
      ((COPY $r1 $r3)               ;r3=l->link
       (IF cc$0                     ;if ((l->link)==NULL)
           ((COPY $r1 $r2)          ;return (l->link)
            (GOTO $$ENDLABEL) )
        ELSE                        ;if ((l->link!=NULL)
           ((EQ $r0 $r2 cc$4)       ;cc4=(((l->link)->data)==x)
            (COPY $r5 $r1)      S.;r1=(l->link->link)
            (COPY cc$1 cc$0)    S.;cc0=((l->link->link)==NULL)
            (LOAD _M$MEMORY_data_1365 $r4 $r0)
                                S.;r0=((l->link->link)->data)
            (COPY $r6 $r5)      S.;r5=(l->link->link->link)
            (EQ $r6 0 cc$1)     S.;cc1=((l->link->link->link)==NULL)
            (ADD $r6 4 $r4)     S.;r4=&((l->link->link->link)->data)
            (LOAD _M$MEMORY_link_1364 $r6 $r6)
```

```
                                S.;r6=(l->link->link->link->link)
            (GOTO srch_$$1) )) )) )) ;goto next VLIW
;srch_$$1 is the steady state of the software pipelined loop,
;executes original loop at a rate of 1 cycle/iteration
(srch_$$1
 ((IF (NOT cc$4)                ;if ((l->link)->data!=x)
      ((COPY $r1 $r3)           ;r3=(l->link->link)
       (IF cc$0                 ;if (l->link->link==NULL)
           ((COPY $r1 $r2)      ;return r2=(l->link->link)
            (GOTO $$ENDLABEL) )
        ELSE                    ;if  (l->link->link!=NULL)
           ((EQ $r0 $r2 cc$4)   ;cc4=((l->link->link)->data)==x)
            (COPY $r5 $r1)    S.;r1=(l->link->link->link)
            (COPY cc$1 cc$0)  S.;cc0=((l->link->link->link)==NULL)
            (LOAD _M$MEMORY_data_1365 $r4 $r0)
                              S.;r0=((l->link->link->link)->data)
            (COPY $r6 $r5)    S.;r5=(l->link->link->link->link)
            (EQ $r6 0 cc$1)   S.;cc1=((l->link->link->link->link)==NULL)
            (ADD $r6 4 $r4)   S.;r4=&((l->link->link->link->link)->data)
            (LOAD _M$MEMORY_link_1364 $r6 $r6)
                              S.;r6=(l->link->link->link->link->link)
            (GOTO srch_$$1) )) ) ;set l:=(l->link) and loop back
   ELSE                         ;if (l->link)->data==x
      ((COPY $r3 $r2)           ;return (l->link)
       (GOTO $$ENDLABEL) )) ))
(PEND srch)
```

B. PARALLELIZATION SUBROUTINES

For an operation Θ in a sequential program,
$\overline{oper(\Theta)}$: operation of Θ $\overline{dest(\Theta)}$: target register of Θ when $opcode(\Theta) \notin$ {`if`,`store`}
$\overline{s(\Theta,T)}$: successor operation of Θ; TRUE successor when $opcode(\Theta) \equiv$ `if`
$\overline{s(\Theta,F)}$: the test-FALSE successor operation of Θ when $opcode(\Theta) \equiv$ `if`
$\overline{av(\Theta)}$: the av set saved in Θ $\overline{av\_level(\Theta)}$: an integer showing the validity of $av(\Theta)$
/* $\overline{av}(\Theta)$ is valid only when $\overline{av\_level(\Theta)} \equiv$ `global_level`. When a new stage of enhanced pipeline scheduling starts, a new `global_level` is defined by incrementing it (see Figure 17) to invalidate $av(\Theta)$ saved in the previous stages. */
$\overline{lv(\Theta)}$: the lv set saved in Θ $\overline{num\_preds\_gt\_1(\Theta)}$: TRUE if Θ is a control join point
$\overline{preds(\Theta)}$: the set of predecessor edges, $\overline{\{(\theta,\gamma)|s(\theta,\gamma) \equiv \Theta\}}$, where θ is an operation and $\gamma \in \{T,F\}$

compute_av(Θ, Ψ, ws)
Θ: the current operation Ψ: the current path, which is list of edges traversed so far
ws: the length of the path Ψ (software lookahead window size)
::: Incrementally compute the $av(\Theta)$ and leaves it at Θ if bb_header(Θ) is true.
::: Return the $av(\Theta)$
/* Return NULL if Θ is not on the legitimate downward path */
`IF (is_ineligible_successor(`Θ, Ψ`)) return(NULL);`
/* If a valid $av(\Theta)$ is already at Θ, just return $av(\Theta)$ */
`IF (av_level(instr)==global_level) return(`$av(\Theta)$`);`
/* If the window size exceeds at Θ during the first computation of `av(group)`, leave a window boundary mark at Θ, so further `update_data_sets` calls do not compute past Θ.*/
`IF (`$ws \geq$ `MAX_WS) {` $av(\Theta) =$ `NULL;` $av\_level(\Theta)$ `= global_level; return(NULL);}`
/* First, recursively compute av under Θ */

$av1$ = **compute_av**($s(\Theta, T)$, add_path($(\Theta, T), \Psi$), $ws + 1$);
IF ($opcode(\Theta) \equiv$ if) $av1 = [av1 \cup$ **compute_av**($s(\Theta, F)$, add_path($(\Theta, F), \Psi$), $ws+1$))] - $All\_Conditional\_Branches$;

/* Then, compute $av1$ above Θ */
$av1$ = **moveup_set_rhs**($av1, \Theta$) $\cup$ $\{rhs(\Theta)\}$;

/* If Θ is a bb_header, leave a copy of $av1$ here; otherwise erase leftovers ($av\_level \equiv 0$ means it is not valid) */
IF (bb_header(Θ)) { $av(\Theta) = av1$; $av\_level(\Theta)$ = global_level; }
ELSE {$av(\Theta)$ = NULL; $av\_level(\Theta)$ = 0;}
return ($av1$);
End **compute_av**

compute_live(Θ)
::: Compute the set of all live registers at the point before Θ and leave it at Θ if bb_header(Θ) is true
/* The ignore_first flag is employed to compute lv correctly during a code motion. In the following loop, (L: z:=z'; ...; if cc goto L ELSE goto exit), z:=x+1 is just replaced by z:=z' in move_op() and the new lv(z := z') must be recomputed using lv(z:=x+1) that was valid before the move_op(). The ignore_first flag is set to TRUE at move_op() before recomputation to ignore the lv when encountered first */
IF ($lv(\Theta) \neq$ NULL && !ignore_first) return($lv(\Theta)$);
ignore_first = FALSE;
/* First, recursively compute lv under the Θ */
lv1 = **compute_live**($s(\Theta, T)$);
IF ($opcode(\Theta) \equiv$ if) $lv1 = lv1 \cup$ **compute_live**($s(\Theta, F)$);
/* Then, compute lv above the Θ */
IF (has_dest(Θ)) $lv1 = lv1 - \{dest(\Theta)\}$; /* Subtract the target register of Θ */
$lv1 = lv1 \cup sources(\Theta)$; /* Include the source registers of Θ */
IF (bb_header(Θ)) $lv(\Theta) = lv1$; /* If Θ is a bb_header, leave a copy of $lv1$ here */
return($lv1$);
End **compute_live**

move_op($\Theta, \Sigma, \tau, \Psi$)
Θ: the current operation Σ: the set of original RHS that we are looking for
τ: the chosen target register Ψ: the current path (list of edges traversed so far)
::: Traverse the DAG starting from Θ and move up the RHS of original operations of Σ if they exist. Generate a bookkeeping copy if Θ is a join point. Return the C_RHS
IF ($\Sigma \equiv \phi$ || is_ineligible_successor(Θ, Ψ)) return(NULL);
/* No original operations exist below Θ */
IF ($av\_level(\Theta) \equiv$ global_level) {
  /* If $av(\Theta)$ is valid, Σ is filtered by intersecting with $av(\Theta)$*/
  IF ($\Sigma \cap av(\Theta) \equiv \phi$) return(NULL); /*No original ops below Θ*/
  ELSE Σ = {choose_one($\Sigma \cap av(\Theta)$)}; /* For correct traversal, choose one */
} /* if there is more than one RHS after the intersection */
IF ($rhs(\Theta) \in \Sigma$) { /* Found an original operation at Θ; replace $rhs(\Theta)$ by τ */
  C_RHS = $rhs(\Theta)$; $oper(\Theta)$ = "$dest(\Theta) = \tau$";
  IF ($dest(\Theta) \equiv \tau$) /* No renaming is required; remember to delete Θ later */
    {mark_for_deletion(Θ)=TRUE; $oper(\Theta)$ = NULL;}
}

```
ELSE { /* Need to search deeper recursively after unsubstituting the Σ if necessary */
   IF (has_dest(Θ) && τ ≡ dest(Θ)) Σ = φ;/* do not pass bookkeeping code[24] */
   ELSE Σ = un_substitute(Σ, Θ);
   /* unsubstitution removes those elements of Σ whose sources are set by Θ; and then,
   if Θ is a copy x:=y and if there is a RHS r in Σ using y, it adds all variants of r to Σ
   obtained by replacing one or more occurrences of y by x*/
   C_RHS = move_op(s(Θ,T), Σ, τ, add_path((Θ,T), Ψ));
   IF (opcode(Θ) ≡ if) {
      C_RHS' = move_op(s(Θ,F), Σ, τ, add_path((Θ,F), Ψ));
      IF (C_RHS ≡ NULL) C_RHS = C_RHS'; /* Merge C_RHS and C_RHS' into one */
   }
   C_RHS = substitute(C_RHS,Θ);/* if Θ is a copy x:=y, substitute y for x in C_RHS */
} /* END IF (rhs(Θ) ∈ Σ) */
IF (C_RHS ≡ NULL) return(NULL);
/* generate a bookkeeping copy of C_RHS with τ if Θ is a join point */
IF (num_preds_gt_1(Θ)) {
   Θ' = make_new_operation(); /* Create a new operation whose fields are NULL */
   (Θ_prev, γ_prev)= last_element(Ψ);
   /* (Θ_prev, γ_prev) is the edge where we came from to Θ */
   FOR each (θ,γ) ∈ preds(Θ) such that (θ,γ) ≠ (Θ_prev, γ_prev)
     s(θ,γ) = Θ'; /* Connect all predecessor edges except for (Θ_prev, γ_prev) to Θ'*/
   End FOR
   s(Θ', T) = Θ; /* The successor edge of Θ' is connected to Θ */
   oper(Θ') = "τ = C_RHS"; /* The operation of the bookkeeping copy */
   /* If Θ' is a bb_header, it is safe to transfer the flow sets of Θ to Θ' */
   IF (bb_header(Θ')) { av(Θ') = av(Θ); lv(Θ') = lv(Θ);}
} /* End IF (num_preds_gt_1(Θ)) */
IF (bb_header(Θ)) update_data_sets(Θ); /* Need to update av and lv sets */
IF (mark_for_deletion(Θ) { /* We need to delete Θ */
   IF (bb_header(Θ) && !bb_header(s(Θ,T)) {av(s(Θ,T))=av(Θ);lv(s(Θ,T))=lv(Θ)}
   /* Transfer data flow sets before deletion; otherwise might delete the only sets left*/
   delete_operation(Θ);
} /* END IF (mark_for_deletion) */
```

[24] For example, consider the following code fragment and try to move the RHS `x+1` to the point just before `if (cc0)`, in order to see why this check is needed:

```
[ if (cc0) {y:=x+x;} else {y:=x; x:=load(u);} z=x+1; w:=y+1; ]
```

Although `x+1` is derived from both `z:=x+1` and `w:=y+1`, one application of `move_op` will move only `z:=x+1` due to this check, since the bookkeeping `z':=x+1` will block further traversal to `w:=y+1`:

```
[ z':=x+1; if (cc0) {y:=x+x;} else {y:=x; x:=load(u); z':=x+1;} z:=z'; w:=y+1; ]
```

A second application of `move_op` can reach `w:=y+1` and complete the move of the RHS `x+1`:

```
[ z':=x+1; w':=x+1 /* equalt to z'*/; if (cc0) {y:=x+x; w':=y+1;} else {y:=x /* dead
*/; x:=load(u); z':=x+1;} z:=z';w:=w'; ]
```

If the first `move_op` did not stop at the bookkeeping `z':=x+1` (which was generated by the first `move_op` itself), it would yield the wrong result:

```
[ z'=x+1; if (cc0) {y:=x+x;z'=y+1;} else {y:=x; x:=load(u); z':=x+1;} z:=z'; w:=z'] ,
```

(this is wrong because `w`, `z` can have different values at the exit in the original code, yet are equal at the exit in the transformed code.) The reason is that, when a RHS to be moved is derived from different operations, a single target register `z'` cannot be used to represent the value of the RHS (in the context of `z:=z'; w:=z'`) for both operations, since even though RHS of different operations are equal and identical when moved to the parallel group, they might not be equal in their original locations.

```
return(C_RHS);
End move_op
```

is_ineligible_successor(Θ, Ψ)

::: Return TRUE if Θ is not a downward continuation of the given path Ψ in the current stage

```
IF (Ψ ≡ NULL) return(FALSE);  IF (Θ is outside of DAG) return(TRUE);
(Θ_prev, n) = last_element(Ψ);
IF (seqno(Θ) < seqno(Θ_prev) ||  /* a backward edge */
seqno(Θ) ≡ seqno(Θ_prev) && in_path(Θ, Ψ) ||  /* Θ is already visited */
seqno(Θ) > seqno(Θ_prev) && in_current_fence(Θ)) return (TRUE); ELSE return(FALSE);
End is_ineligible_successor
```

move_cj(Θ, Σ, Ψ)

Θ: the current operation Σ: the set of original RHS that we are looking for
Ψ: the current path, which is the list of edges traversed so far

::: Traverse the DAG from Θ and move up the cond. branch in Σ. Create two copies of the sequence of nonbranch operations seen until the branch is found, one (ins_T) which leads to the T target of the original branch, and another (ins_F) which leads to the F target of it. Return (C_RHS, ins_T, ins_F)

```
IF (oper(Θ) ≡ "if cc") {  /* Found the original operation */
  /* Since reordering of branch is not allowed, "if cc" must be in Σ */
  (Θ_prev, T) = last_element(Ψ);  /* (Θ_prev, T) is the edge where we came from to Θ */
  s(Θ_prev, T) = s(Θ, T);
  /* Connect the successor edge of Θ_prev to the TRUE successor of the branch Θ */
  IF (has_no_preds(Θ)) delete_operation(Θ);  /* if no other predecessors exist */
  return ("IF cc", s(Θ, T), s(Θ, F));
}
ELSE {  /* search deeper for the branch after unsubstituting if necessary */
  Σ = un_substitute(Σ, Θ);
  (C_RHS, ins_T, ins_F)= move_cj(s(Θ, T), Σ, add_path((Θ, T), Ψ));
  /* substitute C_RHS if Θ is a copy and there is a substitutable dependence. */
  C_RHS = substitute(C_RHS, Θ);
  /* create a copy Θ" of Θ and add it to the stream ins_F that leads to the FALSE target
  of the original branch. Θ itself is already added to the stream ins_T that leads to the
  TRUE target. */
   Θ" = make_new_operation(); oper(Θ") = oper(Θ);
   s(Θ",T) = ins_F;  /* The successor edge of Θ" is connected to ins_F */
   /* Two operation streams, ins_T and ins_F, now start from Θ and Θ", respectively*/
   ins_T= Θ; ins_F= Θ";
   /* If Θ is a control join point, generate a bookkeeping copy */
   IF (num_preds_gt_1(Θ)) {
      Θ'=make_new_operation(); oper(Θ')=C_RHS;
      s(Θ',T)=ins_T; s(Θ',F)=ins_F;
       /* The TRUE and FALSE targets of Θ' become ins_T and ins_F, respectively. */
      (Θ_prev, T) = last_element(Ψ);  /* It is the edge where we came from to Θ */
      FOR each (θ, γ) ∈ pred(Θ) such that (θ, γ) ≠ (Θ_prev, T)
          s(θ, γ) = Θ';  /*Connect all predecessor edges except for (Θ_prev, T) to Θ' */
      End FOR
      IF (bb_header(Θ')) { lv(Θ')= lv(Θ); av(Θ')= av(Θ);}
   } /* End IF (num_preds_gt_1(Θ)) */
```

```
} /* av and lv at ins_T and ins_F need to be updated. */
IF (bb_header(ins_T)) update_data_sets(ins_T);
IF (bb_header(ins_F)) update_data_sets(ins_F);
return(C_RHS, ins_T, ins_F);
End move_cj
```

update_data_sets(Θ)
::: Θ is a bb_header and its data sets have just become incorrect since an operation was moved through it. Incrementally recompute the $av(\Theta)$ and $lv(\Theta)$ using the correct sets at other bb_headers

```
ignore_first = TRUE; /*If oper(Θ) ≡ NULL, the recomputation correctly disregards Θ*/
compute_live(Θ);
compute_av(Θ, NULL, 0);
/* compute_av correctly stops at the original window boundary b where av(b) ≡ NULL and
av_level(b) ≡ global_level */
End update_data_sets
```

ACKNOWLEDGMENTS

The memory disambiguation, interval analysis, live-variable analysis, and final code-printing stages of the compiler were implemented by Manoj Franklin. The VLIW simulator and the PL.8 front-ends were implemented by Arkady Polyak and Toshio Nakatani, respectively. Jaime Moreno carefully reviewed the initial draft of this article, and Mike Schlansker provided helpful comments on the article. We are also grateful to Gabby Silberman and Ashok Agrawala for helpful discussions. We thank three anonymous referees whose detailed comments were very helpful in improving the paper.

REFERENCES

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

Aiken, A. and Nicolau, A. 1988. A development environment for horizontal microcode. *IEEE Trans. Softw. Eng. 14,* 5 (May), 584–594.

Atkins, M. 1991. Intel i860 processor. *IEEE Micro 11*, 24–28.

Bernstein, D. and Rodeh, M. 1991. Global instruction scheduling for superscalar machines. In *Proceedings of the SIGPLAN 1991 Conference on Programming Language Design and Implementation.* ACM Press, New York, 241–255.

Colwell, R., Nix, R., O'Donnel, J., Papworth, D., and Rodman, P. 1988. A VLIW architecture for a trace scheduling compiler. *IEEE Trans. Comput. 37,* 8 (Aug.), 967–979.

Cytron, R., Ferrante, J., Rosen, B., Wegman, M., and Zadeck, F. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13,* 4 (Jan.), 451–490.

Dehnert, J. and Towle, R. 1993. Compiling for cydra 5. *J. Supercomput. 7,* 1/2, 181–228.

Ebcioğlu, K. 1988. Some design ideas for a VLIW architecture for sequential natured software. In *Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing).* North Holland, Amsterdam, 3–21.

Ebcioğlu, K. and Groves, R. 1990. Some global compilation optimizations and architectural features for improving performance of superscalars. Res. Rep. RC-16145, IBM T. J. Watson Research Center, Yorktown Heights, N.Y.

Ebcioğlu, K., Groves, R., Kim, K.-C., Silberman, G., and Ziv, I. 1994. VLIW compilation techniques in a superscalar environment. In *Proceedings of the SIGPLAN 1994 conference on Programming Language Design and Implementation.* ACM Press, New York, 36–48.

Ebcioğlu, K. and Nakatani, T. 1989. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture. In *Languages and Compilers for Parallel Computing*. MIT Press, Cambridge, Mass., 213–229.

Ebcioğlu, K. and Nicolau, A. 1989. Percolation scheduling with resource constraints. Tech. Rep. 89-31, Univ. of California, Irvine, Calif.

Ellis, J. 1985. Bulldog: A compiler for VLIW architecture. Ph.D. thesis, Yale Univ., New Haven, Conn.

Ferrante, J., Ottenstein, K., and Warren, J. 1987. The program dependency graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9,* 3, 319–349.

Fisher, J. and Freudenberger, S. 1992. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York, 85–95.

Freudenberger, S. M., Gross, T. R., and Lowney, P. 1994. Avoidance and suppression of compension code in a trace scheduling compiler. *ACM Trans. Program. Lang. Syst. 16,* 4, 1156–1214.

Gloy, N., Smith, M., and Young, C. 1995. Performance issues in correlated branch schemes. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*. IEEE Computer Society Press, Los Alamitos, Calif., 3–14.

Gupta, R. and Soffa, M. 1990. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Trans. Softw. Eng. 16,* 4 (Apr.), 421–431.

Hwu, W.-M., Mahlke, S., Chen, W., Chang, P., Warter, N., Bringmann, R., Ouellette, R., Hank, R., Kiyohara, T., Haab, G., Holm, J., and Lavery, D. 1993. The superblock: An effective technique for VLIW and superscalar compilation. *J. Supercompt. 7,* 1/2, 229–248.

IBM. 1990. A special issue on IBM RISC System/6000. *IBM J. Res. Devel. 34,* 1 (Jan.).

Jain, S. 1991. Circular scheduling: A new technique to perform software pipelining. In *Proceedings of the SIGPLAN 1991 Conference on Programming Language Design and Implementation*. ACM Press, New York, 219–228.

Jouppi, N. and Wall, D. 1989. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York, 272–282.

Lam, M. 1988. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*. ACM Press, New York, 318–328.

Moon, S.-M. 1993. Compile-time parallelization of non-numerical code; VLIW and superscalar. Ph.D. thesis, Univ. of Maryland, College Park, Md.

Moon, S.-M. 1997. Increasing cache bandwidth using multiport caches for exploiting ILP in non-numerical codes. *IEEE Proceedings - Computers and Digital Techniques 144,* 5 (Sept.), 295–303.

Moon, S.-M. and Carson, S. 1995. Generalized multiway branch unit for VLIW microprocessors. *IEEE Trans. Parall. Distrib. Syst. 6,* 8 (Aug.), 850–862.

Moon, S.-M. and Ebcioğlu, K. 1993. A study on the number of memory ports in multiple instruction issue machines. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*. IEEE, New York, 49–58.

Moon, S.-M. and Ebcioğlu, K. 1997. Performance analysis of tree VLIW architecture for exploiting branch ILP in non-numerical code. In *Proceedings of the 1997 International Conference on Supercomputing*. ACM, New York, 301–308.

Moon, S.-M., Kim, S., Park, J., and Ebcioğlu, K. 1997. Unrolling-based copy coalescing. Tech. Rep. SNU-EE-TR-1997-7, Seoul National Univ., Seoul, Korea.

Nakatani, T. and Ebcioğlu, K. 1989. Combining as a compilation technique for a VLIW architecture. In *Proceedings of the 22nd Annual Workshop on Microprogramming*. IEEE, New York, 43–55.

Nakatani, T. and Ebcioğlu, K. 1993. Making compaction based parallelization affodable. *IEEE Trans. Parall. Distrib. Syst. 4,* 9 (Sept.), 1014–1529.

Padua, D. and Wolfe, M. 1986. Advanced compiler optimizations for supercomputers. *Commun. ACM 29,* 12 (Dec.), 1184–1201.

Park, S., Shim, S., and Moon, S.-M. 1997. Evaluation of scheduling techniques on a SPARC-based VLIW testbed. In *Proceedings of the 30th Annual International Symposium on Microarchitecture.* IEEE, New York.

Patterson, D. 1985. Reduced instruction set computers. *Commun. ACM 28,* 1 (Jan.), 8–21.

Rau, B. 1989. The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *IEEE Comput. 22,* 1 (Jan.), 12–34.

Rau, B. and Fisher, J. 1993. Instruction-level parallel processing: History, overview, and perspective. *J. Supercomput. 7,* 1/2, 9–50.

Rau, B. and Glaeser, C. 1981. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming.* IEEE, New York, 183–198.

Schwartz, J. and Sharir, M. 1979. A design for optimizations of the bit vectoring class. Tech. Rep. 17, Courant Inst. of Computer Science, New York Univ., New York.

Silberman, G. and Ebcioğlu, K. 1993. An architectural framework for supporting heterogeneous instruction set architectures. *IEEE Comput. 26,* 6 (June), 39–56.

Sites, R. 1993. Alpha AXP architecture. *Commun. ACM 36,* 2 (Feb.), 33–44.

Smith, M., Horowitz, M., and Lam, M. 1992. Efficient superscalar performance through boosting. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM Press, New York, 248–259.

Smith, M., Johnson, M., and Horowitz, M. 1989. Limits on multiple instruction issue. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM Press, New York, 290–302.

Warren, H., Auslander, M., Chaitin, G., Chibib, A., Hopkins, M., and MacKay, A. Jun 1986. Final code generation in the PL.8 compiler. Res. Rep. RC 11974, IBM T.J. Watson Research Center, Yorktown Heights, N.Y.

Received June 1994; revised August 1995; accepted October 1995