VLIW Compilation Techniques in a Superscalar Environment

Kemal Ebcioglu*, Randy D. Groves**, Ki-Chang Kim*, Gabriel M. Silberman* and Isaac Ziv*

* IBM T.J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598

** IBM Risc System/6000 Division
11400 Burnet Road, Austin, TX 78758

Abstract

We describe techniques for converting the intermediate code representation of a given program, as generated by a modern compiler, to another representation which produces the same run-time results, but can run faster on a superscalar machine. The algorithms, based on novel parallelization techniques for *Very Long Instruction Word (VLIW)* architectures, find and place together independently executable operations that may be far apart in the original code, i.e., they may be separated by many conditional branches or belong to different iterations of a loop. As a result, the functional units in the superscalar are presented with more work that can proceed in parallel, thus achieving higher performance than the approach of using hardware instruction dispatch techniques alone.

While general scheduling techniques improve performance by removing idle pipeline cycles, to further improve performance on a superscalar with only a few functional units requires a reduction in the pathlength. We have designed a set of new algorithms for reducing pathlength and removing stalls due to branches, namely speculative load-store motion out of loops, unspeculation, limited combining, basic block expansion, and prolog tailoring. These algorithms were implemented in a prototype version of the IBM RS/6000 xlc compiler and have shown significant improvement in SPEC integer benchmarks on the IBM POWER machines.

Also, we describe a new technique to obtain profiling information with low overhead, and some applications of profiling directed feedback, including scheduling heuristics, code reordering and branch reversal.

Keywords: Global scheduling, software pipelining, VLIW, superscalars, compiler optimizations, profiling directed feedback

INTRODUCTION

A great amount of attention is presently being paid to improving the performance of RISC superscalar processors, i.e., uniprocessors that can achieve a peak execution rate of more than one instruction per cycle. Such architectures execute a standard sequential instruction set, such as one normally executed by RISC uniprocessors, but are able to fetch and dispatch to their multiple functional units a peak of two or more instructions in each cycle.

Most speedup measurements on superscalar machines, tested on code generated by existing compilers, have been disappointing. For example, Smith et al. [23] indicate that practical speedups over an existing RISC processor would be limited to a factor of about 2, even with aggressive superscalar configurations. The purpose of this paper is to describe some new global compiler optimization techniques to help overcome the obstacles to speedup on superscalar architectures.

One reason for poor performance of superscalar uniprocessors on existing code, is the small lookahead window of the hardware, which limits the parallelism that can be extracted. Another reason is the unpredictability of branches, and the expense and difficulty of maintaining the execution state on all possible paths in hardware, in case one tried to avoid branch prediction, and execute operations on all paths instead. A third reason is the difficulty of maintaining a *sustained* execution rate of several conditional branches per cycle, to achieve a high degree of performance in commercial applications or system code: branches are very frequent in these types of codes, so high branch throughput seems mandatory. The scheduling techniques described here should help to solve the first problem and part of the second problem. To solve all three problems, and increase the issue rate without complicating the hardware, requires (in our view) a VLIW hardware implementation.

The techniques presented in this paper are inspired by the new compilation techniques and architectural features that have been incorporated in the compiler and architecture for the IBM VLIW machine project at the IBM T.J. Watson Research Center [7,9,20,10,22,18,21]. This project consists of the design of a parallelizing compiler and an architecture (an 8-ALU hardware prototype is currently operational) for extracting parallelism from extremely sequential, non-numerical code. Our compiler techniques can bring together, into the same VLIW instruction, independently executable operations and tests that may be separated by many conditional branches in the original code, or that may belong to different iterations of a loop. (There have been some publications that have approached inter-basic block scheduling for superscalars as a new problem, e.g. [16,14]. However, these authors do not appear to have done a thorough literature search on previously published VLIW scheduling techniques.) The resulting code can execute operations on all paths as soon as their operands are ready if resources permit; register renaming to maintain execution on multiple paths is managed at compile time. Resources are conserved by stopping execution of the remaining operations on a path, as soon as it is known that the path will not be taken. Also, the compiler merges redundant computations on multiple paths into a single computation, to further conserve resources. Advanced memory disambiguation techniques (enhancements of those used in the Bulldog compiler [11]) are used for determining if two memory references can access the same location. The same scheduling techniques can be applied to superscalars as well, by imagining a VLIW with the same resources as the superscalar, scheduling for that VLIW, but leaving the resulting code in superscalar format, rather than in VLIW format.

The program dependence graph (PDG) [13] has been considered in scheduling superscalar code for extracting fine-grain parallelism. A PDG for a procedure has two parts, a control dependence graph and a data dependence graph. The control dependence graph portion explicitly indicates which tests in the program matter in whether a given statement in the program is executed or not, and what values these tests should have (true of false) to cause this statement to be executed (these are the tests on which the statement is control dependent). As a typical application of the program dependence graph described in [13], in the context of standard optimizations, statements (including complex ones, such as loops or diamonds) that are control dependent on an identical set of test results can be re-ordered, if data dependences permit. In the context of scheduling, this re-ordering corresponds to code motion of non-speculative operations that do not require duplication (non-speculative, because if one statement among a set of statements with identical control dependences is executed, then all will be executed).

Bernstein and Rodeh [4] have described an application of the program dependence graph to superscalar scheduling, that includes re-ordering of non-branch instructions with identical control dependence conditions within a given loop body, as well as a limited speculative code motion technique that allows an instruction to be moved above *one* conditional branch. Gupta and Soffa [15] have applied the program dependence graph for somewhat more aggressive code motions. Their technique allows code motion of operations above join points with duplication, motion of a conditional branch above an operation, speculative motion of operations above conditional branches, and loop peeling and unrolling (but no software pipelining). However, in the Gupta-Soffa approach, the unit of code duplication is an entire region; thus, moving conditional branches above join points requires more duplication than necessary. Also, the technique is applied only as an intermediate step for increasing parallelism of basic blocks and large regions containing sub-regions, rather than for the actual generation of machine code. Thus, it is difficult to define it as a true finite-resource scheduling technique. The parallel regions executed dynamically in a single cycle by superscalars, and VLIWs with multi-way branching, tend to cross basic block and structured region boundaries in rather arbitrary ways.

Scheduling *per se* improves performance of a superscalar by removing idle slots in the pipeline, or by keeping the functional units busier. For superscalars with a small number of resources, further performance gains can be achieved by using compiler techniques beyond classical optimizations, aimed at reducing *pathlength.* We have developed a number of original pathlength reduction techniques, including speculative load-store motion out of loops, unspeculation, limited combining, expansion of basic blocks, and prolog tailoring, which result in good performance improvements, which we present in this paper.

In the following we present some of the VLIW-inspired scheduling and optimization techniques (we shall refer to these collectively as "VLIW scheduling techniques") and their application to code generation for superscalars. Examples are given which illustrate their use. Some of these techniques have been incorporated into an experimental version of the RS/6000 xlc compiler, contributing to significant performance enhancements on RS/6000. The same compiler is used to generate code for the PowerPC 601 and Power2 processors, with similar performance gains. Experimental results for the SPEC integer benchmarks, measured on RS/6000 hardware, are presented.

Our VLIW scheduling techniques do not depend on branch probabilities to generate efficient code, as opposed to trace scheduling and its derivatives ([11,6]). Nevertheless, profiling information should be used where it is available, since it allows approaching traditional compiler optimizations in a totally new light, making further performance improvements possible. We describe our current work on profiling directed feedback (PDF), including a new technique for obtaining profiling information with low overhead, and some applications of PDF, such as scheduling heuristics, basic block re-ordering, and branch reversal.

VLIW SCHEDULING TECHNIQUES

In the following we outline some of the novel components of VLIW scheduling, which fit within the back-end of a modern compiler, usually after classical optimizations have been applied, but before register allocation is performed. We ignore here the compiler front-end portion (which parses the high level language input program and generates intermediate code), since VLIW scheduling does not affect its function.

Each component by itself contributes a small portion of the overall performance improvement. But, the synergy among them results in significant gains, as measured on several benchmarks, including SPECint92.

We include here only those components which are original work, These techniques include *speculative load/store motion out of loops, unspeculation, scheduling, limited combining*, and *basic block expansion*. Also mentioned is the *prolog tailoring* technique. Other techniques, which are adaptations of previously published work are not presented in this paper.

Speculative Load/Store Motion Out of Loops

This technique is aimed at avoiding memory accesses from within loops, specially loads from memory. These instructions tend to introduce delays into the operation of the processor, specially if the requested data does not reside in the cache. In essence, this is a generalization of the idea to move loop-invariant instructions out of loops [1], with the additional capability of moving loads and stores which are *conditionally executed* (i.e., part of *if* statements), if they are considered "safe." Although non-memory operations have ordinarily been moved speculatively out of loops by previous techniques, the motion of memory operations has been very conservative in the past. The code motion is done as follows:

A group of load and/or stores is a candidate to move out of a loop if *all* the following conditions hold:

1. Each load and/or store in the group: **(a)** uses the same base register, **(b)** has the same displacement from this base, and **(c)** identical operand length and data type.
2. The base register in question is not written in the loop.
3. The location accessed is not declared volatile (shared variables or memory-mapped I/O) in the source language.
4. There is no possibility for the operands in this group of loads and/or stores to overlap with *any* other memory references (loads, stores or calls) within the same loop, or inside any inner loops contained in this loop.
5. On every path to the loop entry point coming from *outside* the loop (i.e., not counting any back edges from inside the loop), there is either **(a)** a load of the address constant of an external variable of sufficient size, into the base register (the *Table of Contents (TOC)* in RS/6000 software conventions is an area that holds such address constants), or **(b)** a load or store to the same location (base+displacement). This last set of conditions ensures that this load/store group is always "safe" to perform, i.e., it will never cause an exception, even if it is only *sometimes* executed in the original loop.

A simple example of load/store motion out of loops follows. The original code shows:

```
    L    r4=.a(r2,0)   // load from TOC (address of
                       //    a), sizeof(a)>=16
    ...
  CL.0:                // beginning of loop
    ...
    BT   CL.1          // conditional inside loop
    L    r3=a(r4,12)   // candidate for motion
    AI   r3=r3,1
    ST   a(r4,12)=r3   // candidate for motion
  CL.1:
    ...
    BCT  CL.0          // end of loop
```

which becomes:

```
   L r4=.a(r2,0)
   ...
   L r10=a(r4,12)    // r10 is a "register-
                     //  cached" copy of a(r4,12)
 CL.0:
   ...
   BT CL.1
   LR  r3=r10        // replaces   L  r3=a(r4,12)
   AI  r3=r3,1
   LR  r10=r3        // replaces   ST a(r4,12)=r3
 CL.1:
   ...
   BCT CL.0
   ST  a(r4,12)=r10 // needed at all exits
                    //  from the loop
   ...
```

Notice that in the above example, both `LR` operations inside the loop will eventually be eliminated by a later coalescing or limited combining stage, leaving only an `AI r10=r10,1` instruction in the loop. Clearly, the new loop has fewer instructions, resulting in higher performance. But, sometimes parts of an inner loop containing loads and stores are infrequently executed, and thus moving these loads and stores out of the loop may potentially slow down an outer loop. Therefore, *execution profiles* may be very helpful in deciding when this type of optimization should be applied (see PDF below).

As a special case of load/store motion out of loops, for certain I/O library procedures with known properties (e.g., storage modifications confined to parameters), loads and stores are moved out of the loop even if there are calls to such subroutines in the loop (this is an exception to point **4** above). Before each of these I/O procedure calls, those memory locations for which loads and/or stores were moved out of the loop (which the called procedure may use) are updated from their "register-cached" copies before the call, and reloaded into registers after the call. This seems justified because I/O procedures are either executed infrequently or do a lot of work anyway, so that the storing and reloading overhead will not cause much degradation. This strategy can be extended to general procedures, using an *inter-procedural analysis* tool (that has access to library routines as well) to extract the relevant information about accesses to memory locations.

Unspeculation

Speculative operations (operations whose results do not always contribute to the program's final result) can appear in the intermediate code due to various code hoisting techniques that have been applied, or due to the fact that such operations were present in the original program. A sequence of operations becomes speculative if it is moved above a conditional branch which determines whether they should be executed to achieve the desired result of the program. Thus, in those cases when the conditional branch leads away from the path where the instructions were originally located, their execution may cause performance degradation. The objective of *unspeculation* is to discover and reduce the number of speculative operations, thus avoiding the potential performance degradation, by "pushing down" (groups of) speculative operations onto one of the two targets of a conditional branch, making them non-speculative there. (Groups of instructions refer to possibly a number of basic blocks with a single entry and exit. Single exit loops and nested *if-then-else-endif* statements are examples of such groups.)

To perform unspeculation on a speculative (group of) instruction(s) `I`, in the context

```
I
other instructions or groups of instructions
a conditional branch
```

the following conditions must be met:

1. The destination registers of `I` are *all* dead in one of the targets of the conditional branch, but not on the other branch.
2. Any of the instructions between `I` and the conditional branch *must not*
   a. *set* any source *or* destination registers for `I`,
   b. *use* any destination registers for `I`, or
   c. contain instructions with side effects on any memory location(s) loaded from in `I`.
3. None of the instructions in `I` has side effects (e.g., stores to memory, access to volatile variables).

If all of the above conditions are met, then `I` can be deleted from its original place and moved to the branch target edge where its destinations are live.

The instruction scheduler can later choose to make single operations speculative again, when it can determine that there is an otherwise idle resource to execute such operations. This is in contrast to unspeculation, which moves groups of instructions to avoid unnecessary execution of speculative instructions.

Unspeculation can also remove inefficiencies the programmer may have introduced for simplifying coding. Here is a common example, using a C program:

```
flag=1; /* result not always used */
if (cond) {..., flag=0;}
```

becomes the following after unspeculation:

```
if (cond) {..., flag=0;}
else {flag=1;}
```

The unspeculation algorithm proceeds with the following steps:

1. The basic blocks are physically re-ordered using a reverse post-order enumeration of the flow graph nodes. When two basic blocks were consecutive in the original ordering, but are not consecutive in the new ordering, a label is introduced at the beginning of the second basic block if needed, and an unconditional branch to this label is introduced at the end of the first basic block, to retain the original program semantics. The reverse post-order enumeration places all basic blocks of a group of instructions (e.g., a nested *if-then-else-endif* construct) consecutively in the program, so the construct can be easily moved as a whole.
2. The hierarchy of single-entry-single-exit groups of instructions is identified in each loop and in the entire procedure. Local data flow information (the registers which are used and set in this group, etc.) is collected for each group.
3. For each conditional branch in the program (following their order of occurrence in the program), each instruction or group of instructions that precedes it is examined in reverse program order, deciding if the instruction or group of instructions stays in place, goes to the left target edge of the conditional branch, or goes to the right target edge of the conditional branch, according to the conditions described above. The backward traversal stops when a join point or another conditional branch is encountered at the same level in the group hierarchy. A speculative (group of) instruction(s) can thus be pushed repeatedly under successive conditional branches. Live variables and other data flow information are incrementally updated, as new instructions are pushed under conditional jumps. Code is never pushed into loops from the outside. Speculative code inside a loop can, however, be pushed out of exits.

A similar technique, called *revival transformation*, was independently discovered and reported in [12]. In contrast with the revival transformation, our version of unspeculation handles loops, and pushing speculative code (including complex constructs) out of loops. Also, our algorithm is applied to the control flow graph, rather than the control dependence graph, thus legal serial code is guaranteed after each code motion step. Furthermore, we move all movable constructs down at once; pushing down one speculative construct may enable further speculative constructs above it to be pushed down as well.

Unrolling, Renaming, Global Scheduling, Software Pipelining

The regions of the program are compacted through the combination of *global scheduling* [10] and *enhanced pipeline scheduling* [7], starting from the innermost regions (loops) and ending with the outermost region (the whole procedure). Global scheduling consists of choosing the best operation from the set of operations that can move to a point, moving all instances of the operation to that point, making bookkeeping copies for edges that join the paths of code motion but are not on them, and updating the data flow attributes of basic blocks only on the paths that were traversed by the instances of the moved operations. To compact the loops, not just within an iteration, but also across iterations, enhanced pipeline scheduling puts a fence at the current scheduling point, and lets global scheduling search for the best operation on all paths which can possibly cross the loop back edges, *but not* the fence. The loops are unrolled prior to scheduling and live range renaming is performed, to increase scheduling opportunities.

The VLIW compiler techniques developed by our group are beyond the scope of this paper and have been described in detail in [10,18,9,21]. They are different from other VLIW and superscalar scheduling techniques, in features that include the following:

1. They operate on an entire program with arbitrary control structure, rather than on a single most probable path, as in trace scheduling. The code motion technique is very general: whenever there is a path from a point A to a point B in the program, where there are no true data dependences for an operation currently at B, that operation can be moved from B to A on that path, even if there are other paths from A to B where there are dependences. Multiple instances of the same operation on different paths are hoisted as one operation.
2. Conditional branches can be moved up above other operations when dependences permit such an action. While conditional branches are rarely on the critical path, executing them early reduces the guesswork for determining which speculative operations to execute.
3. Our software pipelining technique (enhanced pipeline scheduling [9]) applies to loops with arbitrary flow control, and allows generating schedules with a variable iteration issue rate, depending on which path is followed at run time (unlike Lam's hierarchical reduction technique and its derivatives, that respect the worst case data dependence cycle across all paths in the loop [17]).

The particular flavor of our VLIW scheduling techniques used in the context of the xlc compiler will be described in detail in [8]. We try to provide the gist of the techniques with an example here (SPEC li "xlygetvalue" subroutine). In the context of the RS/6000 code below, the objective is to put one independent instruction between a load and the use of its destination register, and three independent instructions between a compare and a dependent conditional branch. This is conceptually the same as scheduling for a VLIW that can simultaneously accommodate a maximum of one integer operation and one branch per VLIW instruction, while respecting pipeline latencies for compare and load operations.

```
r8=address of first list element
r3=item to match against
"L r1=(r8,4)" extracts the "car" field of list
              element pointed to by r8
"L r2=(r8,8)" extracts the "cdr" field of list
              element pointed to by r8
No. of cycles are shown next to each instr.

loop:
L      r4=(r8,4)    1                  (BB #1)
L      r5=(r4,4)    2
C      cr0=r5,r3    2
BT     found,cr0.eq 0  if (car(car(r8))==r3)
                                    goto found
L      r8=(r8,8)    1  r8=cdr(r8)      (BB #2)
C      cr1=r8,0     2
BF     loop,cr1.eq  3  if (r8!=NULL) goto loop
endofchain:
  ...
found:     #   r4 is live here

SPEC li benchmark inner loop.  Executes at
11 cycles per iteration.
```

Here is the loop after unrolling and live range renaming. For each register r that is live at an edge that leaves the loop, a "non-coalesceable" register copy operation LR r=r is inserted at that exit edge before live range renaming. This technique increases scheduling opportunities.

```
loop:
L      r4'=(r8,4)
L      r5'=(r4',4)
C      cr0'=r5',r3
BT     found',cr0'.eq
L      r8'=(r8,8)
C      cr1'=r8',0
BT     endofchain,cr1'.eq
L      r4=(r8',4)
L      r5=(r4,4)
C      cr0=r5,r3
BT     found,cr0.eq
L      r8=(r8',8)
C      cr1=r8,0
BF     loop,cr1.eq
endofchain:
  ...
found':     #non-coalesceable
LR  r4=r4'  #copy operation for detaching
            #live ranges of r4 and r4'
            #(can be copy-propagated later)
found:      #   r4 is live here
```

In what follows we can see a version of the loop which executes at 14 cycles for 2 iterations, obtained by code motion within a loop body (global scheduling with no software pipelining). When we combine global scheduling with software pipelining, the loop executes at 10 cycles for 2 iterations, as shown in the next page.

```
                                                     (unrolled original loop)
loop:                                                    loop:
L      r4'=(r8,4)         1                              L   r4'=(r8,4)
L      r8'=(r8,8)         1  (moved)<------------------|
L      r5'=(r4',4)        1                            | L   r5'=(r4',4)
C      cr1'=r8',0         1  (moved)<----------------| |
C      cr0'=r5',r3        1                          | | C   cr0'=r5',r3
L      r4=(r8',4)         1  (moved)<--------------| | |
L      r8=(r8',8)         1  (moved)<------------| | | |
L      r5=(r4,4)          1  (moved)<----------| | | | |
BT     found',cr0'.eq     0                    | | | | | BT  found',cr0'.eq
                                               | | | | -< L   r8'=(r8,8)
                                               | | | |--< C   cr1'=r8',0
C      cr1=r8,0           1  (moved)<--------| | | |
BT     endofchain,cr1'.eq 0                  | | | |     BT  endofchain,cr1'.eq
                                             | | | |----< L   r4=(r8',4)
                                             | |-|------< L   r5=(r4,4)
C      cr0=r5,r3          1                  |   |       C   cr0=r5,r3
BT     found,cr0.eq       3                  |   |       BT  found,cr0.eq
                                             |   |------< L   r8=(r8',8)
                                             |----------< C   cr1=r8,0
BF     loop,cr1.eq        1                              BF  loop,cr1.eq
endofchain:
  ...
found':
LR  r4=r4'
found:     #   r4 is live here
```

```
                                                 (loop from the previous figure)
loop:                                             loop:
L      r4'=(r8,4)                       |--------< L      r4'=(r8,4)
L      r8'=(r8,8)                     |-|--------< L      r8'=(r8,8)
L      r5'=(r4',4)                  |-|-|--------< L      r5'=(r4',4)
C      cr1'=r8',0                 |-|-|-|--------< C      cr1'=r8',0
loop1: (new loop starts here)     | | | |
C      cr0'=r5',r3         1      | | | |          C      cr0'=r5',r3
L      r4=(r8',4)          1      | | | |          L      r4=(r8',4)
L      r8=(r8',8)          1      | | | |          L      r8=(r8',8)
L      r5=(r4,4)           1      | | | |          L      r5=(r4,4)
BT     found',cr0'.eq      0      | | | |          BT     found',cr0'.eq
C      cr1=r8,0            1      | | | |          C      cr1=r8,0
BT     endofchain,cr1'.eq  0      | | | |          BT     endofchain,cr1'.eq
C      cr0=r5,r3           1      | | | |          C      cr0=r5,r3
L      r4'=(r8,4)          1  <---|-|-|-|
L      r8'=(r8,8)          1  <---|-|-|
L      r5'=(r4',4)         1  <---|-|
BT     found,cr0.eq        0      |                BT     found,cr0.eq
C      cr1'=r8',0          1  <---|
BF     loop1,cr1.eq        0                       BF     loop,cr1.eq
endofchain:
  ...
found':
LR  r4=r4'
found:     #   r4 is live here
```

Note that we have assumed that the first few bytes of page zero contains zeros in this example, as suggested in [5]. This allows operations such as taking car(car(NIL)) to proceed without causing an exception.

Limited Combining

This technique scans the code looking for opportunities to reduce path length by combining collapsible operations such as `LR r4=r5 ; A r6=r4,r7` where `r4` is used for the last time by the second operation, into `A r6=r5,r7`. This is normally achieved by the value numbering optimization [1], when both operations are inside the same basic block. The limited combining technique can span a number of basic blocks looking for collapsible operations, possibly including join points, through code duplication.

This transformation will search a sequence of instructions, starting from a *load immediate to register* or *register copy* instruction ("register" means fixed point, floating point, or condition code register), through *unconditional branches*, until a "last use" of the destination register for the starting instruction is found. Neither the source nor the destination registers for this instruction should be set by any instruction on the way to the last use. If the search succeeds, the whole sequence of instructions (beginning from the instruction following the starting one, and ending at the last use instruction) replaces the original starting instruction. All occurrences of the destination register for the starting instruction in this sequence are replaced by the source (literal or register) for the starting instruction, and an unconditional branch to the instruction following the last use is added. Any unreachable code left from this transformation may be deleted by common "unreachable code elimination" techniques. Combining as a dynamic technique used on-the fly during scheduling is further discussed in [20].

For example:

```
LR    r5=r4
 ...             // instruction group (1)
B     L3

 ...

L3:              // other paths may join here
 ...             // instruction group (2)
L     r3=4(r5)
 ...             // instruction group (3)
B     L4

L4:
 ...             // instruction group (4)
L     r7=8(r5)   // this is the last use of r5
 ...             // operation following last use
```

where neither `r5` nor `r4` are set in instruction groups (1), (2), (3) or (4), becomes:

```
                    // LR r5=r4 deleted!!
   ...              // instruction group (1)
   ...              // instruction group (2)
L      r3=4(r4)     // uses r4 instead of r5
   ...              // instruction group (3)
   ...              // instruction group (4)
L      r7=8(r4)     // uses r4 instead of r5
B      L10          // new branch closes duplicated
                    //  sequence

L3:                 // original sequence kept for
                    // other paths joining here
   ...              // instruction group (2)
L      r3=4(r5)
   ...              // instruction group (3)
B      L4

L4:
   ...              // instruction group (4)
L      r7=8(r5)     // end of original sequence

L10:                // join from duplicated code
   ...              // operation following last
                    //  use of r5
```

Basic Block Expansion

Some of the optimizations, such as limited combining above, create small basic blocks that end with an unconditional branch. Moreover, the original program itself may also naturally contain tiny basic blocks that end with an unconditional branch. These branches are not considered as resources in a VLIW environment, but actually consume resources (and possibly cycles) in a superscalar machine. For example, the RS/6000 can be significantly slowed down if an untaken conditional branch is followed immediately by a (taken) unconditional branch. Since *conditional* branches cannot be avoided, the basic block expansion technique tries to minimize the occurrence of *unconditional* branches on the execution trace, by copying code from the target of a branch.

For a given unconditional branch, basic block expansion determines how many cycles worth of non-branch instructions we need to copy, by examining the code immediately preceding the branch, using machine specific rules. For example the RS/6000 requires 4-5 non-branch instructions (cycles) between an integer compare, a dependent (untaken) conditional branch, and an unconditional branch, in order to avoid a stall. Then the code at the target of the unconditional branch is examined, to determine the *stopping point* for the copying process (i.e., how many instructions to copy). The search for a stopping point can go past conditional branches, CALLs, or unconditional branches. For the latter case, searching continues with the target of the unconditional branch. Labels encountered on the way are not copied. As the search passes a conditional branch or CALL, the objective number of consecutive non-branch instructions to copy, is re-calculated.

The search stops when enough consecutive non-branch instructions have been gathered, or when an instruction (inside a loop) is revisited. If a return from a procedure is encountered (or a "branch on count" in the RS/6000, which ends inner loops), the search stops as well. Obviously, there is a limit to the number of instructions scanned in this process (the "window size"), and exceeding this number aborts the search. In this case, the stopping point that gives the minimal machine stall, among the ones encountered so far, is chosen. Good candidates for stopping points are instructions immediately preceding (hopefully untaken) conditional branches.

Once a stopping point is chosen, code is copied, starting from the target of the unconditional branch until the stopping point. The original unconditional branch is deleted and a new unconditional branch is inserted after the copied code, branching to the instruction immediately following the stopping point.

The net effect of basic block expansion, when successful, is to remove an unconditional branch altogether from the execution trace, push it outside an inner loop, or put sufficiently many non-branch operations before it, so that it can be executed concurrently with other operations on the RS/6000, Power2 or PowerPC hardware.

Basic block expansion on the following code:

```
C      cr0=...      // set condition register 0
BF     L1,cr0       // branch conditional on cr0
op1
B      L2           // need 4 instructions here

 ...
L2:                 // other paths may join here
C      cr1=...      // set condition register 1
BF     L3,cr1       // reset objective to 5 inst.
op2
op3                 // potential stopping point
BF     L5
op5
op6
BCT    loop         // stopping point

L3:                 // code continues here
```

produces:

```
C      cr0=...   // set condition register 0
BF     L1,cr0    // branch conditional on cr0
op1
C      cr1=...   // set condition register 1
BF     L3,cr1    //
op2
op3              // potential stopping point
BF     L5
op5
op6
BCT    loop      // stopping point
B      L3        // rejoin code
```

Note that the original code starting at L2 will be deleted as "unreachable code" if no other instructions branch to it.

A milder form of basic block expansion was described in [14], where the process of adding instructions would stop when any branch was encountered. A more aggressive experimental code replication technique appears in [19], which replicates *all* operations on the shortest path from each unconditional branch to the end of the procedure, in order to eliminate unconditional branches almost totally. This technique entails a high degree of code expansion, and can have an uncertain effect on the machine stalls of superscalars like the RS/6000, due to the conditional branches that are reversed to make the replicated path a straight line. Our technique uses a window size constraint to keep the code expansion reasonable, and makes intelligent decisions on where to stop copying to minimize the machine stall; and thus it seems more appropriate for a practical commercial environment.

Prolog Tailoring

In the RS/6000 linkage conventions, a register belonging to a particular subset of the machine registers must be saved upon entry and restored upon exit in a procedure, if that register is killed (overwritten) inside the procedure. A prolog is the portion of code at the beginning of a procedure which stores the values of registers that are killed in the procedure, whereas the epilog restores these registers at the end of the procedure. Our *prolog tailoring* technique delays the execution of store instructions for killed-registers as late as possible into the procedure, so that each execution path therein contains a reduced number of such store instructions. However, register save operations are never pushed inside loops.

Some high level languages, like those in the PL/I family, require correctly *unwinding* the stack after an interrupt, in order to return to an exception handler established by a procedure high up in the call chain. For example, assume procedure A establishes an exception handler; then A calls B, which saves some registers on the stack and calls C, which saves some registers on the stack and then causes a trap. The trap handler code must correctly unwind the stack by first restoring the registers saved by C, then those saved by B. Control must finally be transferred to the user's exception handler in A. In the existing software conventions, the set of registers to restore is found by the exception handler in a *traceback table* at the end of a procedure, and is fixed for a given procedure. If prolog tailoring is applied, the set of registers to restore is no longer constant, varying depending on the point of execution in the procedure. This requires special handling when a program interrupt occurs, to ensure proper exception behavior of the program.

In order to ensure correct unwinding of the stack after program interrupts, we enforce the rule that at any point in the procedure, all paths reaching this point from the start of the procedure have the same set of saved registers. Thus, the registers to be restored on an exception can be found by back-tracing any path from the point where the exception occurred to the start of the procedure (a flow graph must first be constructed from the binary code by the exception handler for this purpose [22]), without having to know what particular path was followed to reach the point of exception.

In the following page we show an example of the application of prolog tailoring.

The prolog tailoring algorithm has the following stages:

1. Identify outermost loops first and collapse them to single nodes. Make a copy of each basic block that ends with a *return*, for each edge that goes to it. Straighten the flow graph, and do this once more, if code size constraints allow. Then identify *bi-connected components* in the undirected version of the flow graph using Tarjan's algorithm [2]. For example, an outermost *if – then – else – endif* statement constitutes a bi-connected component in the undirected version of the flow graph. Create a tree from these bi-connected components where the root is the component containing the special procedure start node, and its children are the components that share a node with the root. The children of a non-root component *c* are the components other than the parent of *c*, that share a node with *c*. For the purpose of computing the registers killed in a bi-connected component in the next step, the basic block shared between a parent component and a child component in the tree is considered to be inside the parent and not the child.
2. Determine a data flow attribute at each node of the tree, called the *MustKill* registers. These are the registers that will be definitely killed (written into) starting at this tree node, regardless of which path is

```
WITHOUT TAILORED PROLOG                   WITH TAILORED PROLOG
(saves all registers that are killed
anywhere in the procedure)

     PROC sub                                 PROC sub
     save r28,r29,r30,r31
     ...                                      ...          (restore nothing on exception)
     BT   L1                                  BT   L1
                                              save r29,r31

     L    r29=                                L    r29=    (restore r29,r31 on
     L    r31=                                L    r31=        exception)
     ...                                      ...
     restore r29,r31                          restore r29,r31
     return                                   return

L1:                                       L1:
                                              save r28
     L r28=                                   L r28=
     ...                                      ...          (restore r28 on exception)
     BT  L2                                   BT   L2
                                              save r30
     L   r30=                                 L    r30=    (restore r28,r30 on exception)
     L   r28=                                 L    r28=
                                              restore r30
L2:                                       L2:
     (r30 not used or set
          after here)                                      (restore r28 on exception)
     ...                                      ...
     restore r28,r30                          restore r28
     return                                   return
```

taken in the tree. The *MustKill* attribute of a tree node n is computed as:

$$MustKill(n) = \left[\bigcap_{s \in succ(n)} MustKill(s) \right] \cup KilledBy(n)$$

where $succ(n)$ is the set of the children of n, and $KilledBy(n)$ is the set of registers that are killed by any basic block inside n. This data flow attribute can be determined in one backward pass over the tree, in reverse topological order.

3. For each node following a forward topological order of the tree, if a register will definitely be killed starting at this tree node, but has not yet been saved on the path from the root, generate code to save that register on every actual flow graph edge that connects this node and its parent in the tree (multiple actual flow graph edges may connect two nodes in the tree).

Consider maximal sequences entered only at the top, consisting of non-branch instructions, *if-then-else-endif* constructs and single entry, single exit loops (a branchless basic block is such a sequence). If a register is set for the first time since the beginning of the program in such a sequence, and is never used or set after the sequence, that register can be saved on entry to the sequence, restored on exit from the sequence, and then the entire sequence can be treated as if it did not kill this register. This analysis and insertion of the save and restore code for the register should be done before stage 2 of the algorithm above.

Experimental Results

The performance of code generated by an experimental version of the RS/6000 xlc compiler, using VLIW scheduling, has been extensively tested on Power (RS/6000), Power2 and PowerPC 601 hardware. Following is a snapshot of these measurements, using the 6 SPEC '92 integer benchmarks. (These *do not* represent any official measurement by IBM.) In this case the resulting improvement is about 13%. Compared to the -O option of xlc, there was an average compile time increase of 36%, and an average code size increase of 8% using static binding. The most time consuming transformation is VLIW scheduling. Aggressive compiler techniques such as the ones described here are thus appropriate for the -O3 option of the xlc compiler, which is intended for optimizations that require longer compile time.

In the following table we present performance measurements done on an RS/6000 model 580 machine, using xlc version 1.2 and the KAP C Preprocessor from Kuck & Associates. The columns labeled "xlc" show the baseline measurement, with VLIW optimizations disabled, while the "VLIW" columns show performance with the new scheduling and optimization techniques, including the effects of some AIX library routines that were rewritten by the authors for better performance. The VLIW measurements were done on an RS/6000 model 980, whose performance on SPECint92 is equivalent to that of a model 580.

	SPECint92 Measurements			
Benchmark	*xlc Time*	*xlc SPECmark*	*VLIW Time*	*VLIW SPECmark*
espresso	41.70	54.44	38.30	59.27
li	99.00	62.66	81.90	75.82
eqntott	13.60	80.88	10.70	102.80
compress	53.90	51.39	48.10	57.59
sc	69.20	65.46	62.40	72.60
gcc	91.40	59.61	90.20	60.53
SPECint92		61.73		69.93

PROFILING DIRECTED FEEDBACK

Profiling Directed Feedback (PDF) has been used in the past for VLIW approaches based on trace scheduling and its derivatives [11,6]. Whereas PDF is a fundamental requirement in these techniques, our VLIW scheduling techniques do not depend on the availability of branch probabilities, and already generate good schedules when there is no profiling directed feedback. Nevertheless, profiling directed feedback provides the compiler with new information, allowing traditional optimizations to be approached in a fashion heretofore considered too risky, because of potential performance degradation, violation of program semantics, or both. In the following, we describe our current work for using profiling directed feedback in superscalar optimizations. The optimizations described below (consisting of scheduling heuristics, basic block re-ordering, and branch reversal) have been implemented and result in a 4-5% additional improvement on SPECint92 (using the short SPEC inputs for generating profiling data, where available).

Low Overhead Profiling Directed Feedback

Regardless of what profiling directed feedback is used for, the collection of the profile information, in the form of execution counts in the flow graph, should have low overhead in order to make profiling more usable in a commercial compiler environment. Therefore, we gather exact counts for basic block execution, and derive from them the edge counts which are used in the optimizations. Furthermore, we have devised techniques to reduce the number of basic block counts required, to further decrease the execution overhead of profiling.

For generating and using profiling code, we compile a program twice, and run each version of it separately. During the first pass of PDF, the compiler inserts run-time counting code in a subset of the basic blocks. When the program thus compiled is executed, it creates a file that indicates the number of times each basic block that contained counting code was executed. Counts from multiple runs of the same program can be accumulated. During the second pass of PDF, the compiler reads back from this file the execution counts, *at the same place in the compiler where counting code was inserted in the first pass*, computes the complete set of edge counts and basic block counts from the partial set of basic block counts that were read back, and then uses these edge counts for aggressive optimizations. The flow graph edge counts are maintained as compiler transformations occur, so subsequent stages of the compiler can also use the profiling feedback. Note that it is sometimes necessary to create new ("dummy") basic blocks during PDF, so that the edge counts can be uniquely determined from the basic block counts. In this case, the flow graph is modified in the same way on both passes, so the state of the program at the point the counting code is inserted in the first pass, and at the point the counts are read back during the second pass, is identical.

As we observed above, not all basic blocks need to have runtime counting code for uniquely computing all the edge counts. It is also possible to reduce the amount of counting instructions in inner loops, and thus reduce the profiling overhead, by moving out invariant loads and stores from the loop, in a manner similar to our speculative load-store motion described above. Following is an example on the inner loop of the SPEC eqntott benchmark (the original version, without the KAP/C preprocessor). Only the basic blocks *BB1, BB2*, and *BB4* inside the loop, and *BB7* and *BB8* outside the loop, have been augmented by (profiling) counting code. The reader can verify that all the remaining basic block and edge counts for the flow graph of this loop can be uniquely determined from the basic block counts of the selected subset. Outside the inner loop (e.g., in *BB8*), the counting code overhead is three instructions per block (load the previous count for this basic block, add one, and store back the updated count). The load and store instructions related to counting code have been moved out of the inner loop, since their operand addresses are loop invariant, and thus the counting code overhead is one instruction per basic block inside the loop. The profiling code is inserted before scheduling, register allocation, and other optimizations, so further code improvements are possible. These are not shown in the example, to preserve its structure.
We use a *constraint-propagation* algorithm, inspired by Artificial Intelligence techniques applied to electrical networks (e.g., [24]), for finding (and possibly creating) the basic blocks for counting code insertion. The idea is to have "just enough" counts, so that all the remaining edge and basic block counts in the flow graph can be uniquely determined from the gathered counts. The details of this algorithm are beyond the scope of this paper.

```
/* r31= initialized to address of
   global basic block counts table */

/* moved loop invariant loads to preheader */
L       r11=bbtable(r31,4004)   counting code
L       r12=bbtable(r31,4008)     .
L       r13=bbtable(r31,4012)     .
(BB 1)
CL.0:
AI      r11=r11,1          counting for BB 1
LHAU    r4,r3=a(r3,2)      load hlfwrd / incr r3
LHAU    r6,r5=b(r5,2)
C       cr0=r4,2
BF      CL.1,cr0.eq
(BB 2)
AI      r12=r12,1          counting for BB 2
LI      r4=0
(BB 3)
CL.1:
C       cr1=r6,2
BF      CL.2,cr1.eq
(BB 4)
AI      r13=r13,1          counting for BB 4
LI      r6=0
(BB 5)
CL.2:
C       cr0=r4,r6
BT      CL.10,cr0.eq
(BB 6)
BCT     CL.0
(BB 7)
/* invariant stores pushed to loop exit */
ST      bbtable(r31,4004)=r11  counting code
ST      bbtable(r31,4008)=r12        .
ST      bbtable(r31,4012)=r13        .
L       r0=bbtable(r31,4016)   counting code
AI      r0=r0,1                  for
ST      bbtable(r31,4016)=r0       BB 7
  (other code  for BB 7)
   ...
(BB 8)
CL.10:
/* invariant stores pushed to loop exit */
ST      bbtable(r31,4004)=r11  counting code
ST      bbtable(r31,4008)=r12     .
ST      bbtable(r31,4012)=r13     .
L       r0=bbtable(r31,4020)    counting code
AI      r0=r0,1                   for
ST      bbtable(r31,4020)=r0        BB 8
  (other code  for BB 8)
```

Ball and Larus [3] have used a profiling technique that inserts counting code on a subset of the *edges* in the flow graph. They apply their technique to a final executable program, whereas in our case profiling code is inserted in the compiler backend, allowing many sophisticated optimizations, such as motion of profiling overhead code out of loops and scheduling, to be applied thereafter. Also, the backend's infrastructure provides us with better static prediction knowledge, so that counting code can be placed in less frequently executed locations in the program. Further, we prefer placing counting code in existing basic blocks where possible, rather than generating new unconditional branches and extra tiny basic blocks that result from placing counting code on edges. This facilitates the application of subsequent optimization phases.

Scheduling Heuristics, Basic Block Re-ordering, Branch Reversal

The Profile directed feedback feature is currently used for supplying branch probabilities to the VLIW scheduler. This alters the view of the scheduler in terms of which operations are speculative and which others are non-speculative (if an operation is present only on a less frequently executed path it is considered speculative). Non-speculative operations are preferred over speculative ones as usual, and there are no major modifications to the scheduling otherwise. The true (not from profiling) speculativeness of operations is still computed for determining the safeness of loads. Similarly, scheduling with duplication can be used on the most frequent path only, but now without fear of slowing down the other paths.

Also, just before final code generation, the basic blocks are physically reordered following a depth-first enumeration of the flow graph, using the basic block ordering procedure that is already available in the context of unspeculation, described above. Then standard code straightening optimizations of the xlc compiler are applied to eliminate any awkward branching that may have resulted from the re-ordering.

During the depth-first enumeration, the flow graph edges that are executed most frequently are followed first, unless the target of the edge is already visited. That is, the enumeration algorithm performs the following steps: First it assigns the next number to the current node (basic block), and marks it visited. Next it recursively visits the most probable successor of the current node, if it is not already visited. Lastly, it recursively visits each remaining successor of the current node, if it has not been visited. This causes the most frequently executed path to occur first in the enumeration, and therefore be arranged in a straight line, where almost all branches fall through.

By itself, this re-ordering technique will not cause all conditional branches to fall through most of the time. For example, the basic block that ends a critical loop may occur last in the enumeration, and therefore the code may end with a conditional branch that is taken most of the time. Nevertheless, most Power and PowerPC hardware work better when conditional branches fall through most of the time, and additional performance benefits may be obtained if unconditional branches are eliminated. To achieve the latter goals, any conditional branches that are taken most of the time are reversed, so they are not taken most of the time, as follows:

```
BT CL.1,cr1.eq  taken most of the time
```

becomes

```
BF CL.2,cr1.eq  untaken most of the time
B CL.1
CL.2:             a new label
```

and then code is copied from the basic block(s) at the target label CL.1, using the basic block expansion technique described above.

Profiling directed feedback seems to offer many more optimizations for superscalars beyond the ones described above, either in the form of new transformations, or traditional optimizations re-cast in a new light. This fertile area is the current focus of our work in progress.

Acknowledgements

We are grateful to Bob Blainey, Sohel Saiyed, and Marty Hopkins for many helpful discussions.

Bibliography

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

[2] A.V. Aho, J.D. Ullman, and J.E. Hopcroft, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[3] T. Ball and J.R. Larus, "Optimally Profiling and Tracing Programs," *Technical Report no. 1031*, Computer Sciences Dept., U. of Wisconsin-Madison, 1991. Also in POPL '92.

[4] D. Bernstein and M. Rodeh, "Global Instruction Scheduling for Superscalar Machines," *Proc. SIGPLAN '91 Symp. on Programming Language Design and Implementation* (published as *SIGPLAN Notices*, Vol. 26, No. 6), pp. 241-255, June 1991.

[5] D. Bernstein, M. Rodeh, and S. Sagiv, "Proving Safety of Speculative Load Instructions at Compile Time," *Lecture Notes in Computer Science (Proc. 4th European Symposium on Programming)*, No. 582, B. Krieg-Brueckner (Ed.), Springer-Verlag, February 1992.

[6] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Warter, and W.W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction Issue Processors," *Proc. 18th International Symposium on Computer Architecture*, pp. 266-275, May 1991.

[7] K. Ebcioglu, "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps," *Proc. of the 20th Annual Workshop on Microprogramming*, pp. 69-79, ACM Press, 1987.

[8] K. Ebcioglu, "Global Scheduling," section of a book on compiler optimizations, F. Allen, K. Zadeck, B.K. Rosen (eds.), to appear.

[9] K. Ebcioglu and T. Nakatani, "A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture," in *Lang. and Comp. for Parallel Comp.*, D. Gelernter, A. Nicolau, and D. Padua (eds.), Research Monographs in Parallel and Distributed Computing, pp. 213-229, MIT Press, 1990.

[10] K. Ebcioglu and A. Nicolau, "A Global Resource Constrained Parallelization Technique," *Proc. of 1989 International Conference on Supercomputing*, pp. 154-163, Crete, Greece, 1989.

[11] J. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, 1986.

[12] L. Feigen, D. Klappholz, R. Casazza, and X. Xue, "The Revival Transformation," *Proc. POPL '94*, pp. 421-434, ACM Press, January 1994.

[13] J. Ferrante, K.J. and Ottenstein, and J.D. Warren, "The Program Dependence Graph and its Use in Optimization," *ACM Trans. on Programming Languages and Systems*, Vol. 9, No. 3, pp. 319-349, July 1987.

[14] M.C. Golumbic, and V. Rainish, "Instruction Scheduling Beyond Basic Blocks," *IBM J. Research and Development*, 34, 1, pp. 93-97, 1990.

[15] R. Gupta and M. Soffa, "Region Scheduling: An Approach for Detecting and Redistributing Parallelism," *IEEE Trans. on Software Engineering*, Vol. 16, No. 4, pp. 421-431, 1990.

[16] S. Jain, "Circular Scheduling: A new technique to perform software pipelining," *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991.

[17] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *Proc. ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 318-327, June 1988.

[18] S.-M. Moon and K. Ebcioglu, "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors," *Proc. 25th Annual International Symposium on Microarchitecture*, pp. 55-71, ACM, 1992.

[19] F. Mueller and D.B. Whalley, "Avoiding Unconditional Jumps by Code Replication," *Proc. of PLDI '92*, pp. 322-330, ACM, 1992.

[20] T. Nakatani and K. Ebcioglu, "Combining as a Compilation Technique for VLIW Architectures," in *Proc. 22nd Workshop on Microprogramming and Microarchitecture*, Dublin, ACM, pp. 43-57, 1989.

[21] T. Nakatani and K. Ebcioglu, "Making Compaction Based Parallelization Affordable," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 9, pp. 1014-1029, September 1993.

[22] G.M. Silberman and K. Ebcioglu, "An Architectural Framework for Migration from CISC to Higher Performance Platforms," *Proc. 1992 International Conference on Supercomputing,* pp. 198-215, ACM Press, 1992.

[23] D. Smith, M. Johnson, and M. Horowitz, "Limits on Multiple Instruction Issue," *Proc. of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pp. 290-302, ACM and IEEE, Boston Massachusetts, 1989.

[24] G.J. Sussman and G.L. Steele, "Constraints-- A Language for Expressing Almost Hierarchical Descriptions," *Artificial Intelligence*, Vol. 14, pp. 1-39, 1980.