RC 16145 (#71759) 10/2/90
Computer Science 13 pages

Research Report

Some Global Compiler Optimizations and Architectural Features for Improving Performance of Superscalars

Kemal Ebcioglu

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

Randy Groves

IBM Advanced Workstations Division
11400 Burnet Road
Austin, Texas 78758

Some Global Compiler Optimizations and Architectural Features for Improving Performance of Superscalars*

Kemal Ebcioğlu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598

Randy Groves
IBM Advanced Workstations Division
11400 Burnet Road
Austin, Texas 78758

Abstract

We describe a method for converting a given program in the assembly language of a superscalar machine to another program written in the same assembly language, such that the resulting program produces the same final results as to the original one, and can run significantly faster. The method, inspired by new global parallelization techniques for VLIW architectures, finds and places together independently executable operations that may be far apart in the original code (i.e., that may be separated by many conditional branches or that may belong to different iterations of a loop.) As a result, the pipelined functional units in the machine, which could not be kept busy because of the limited size of the execution lookahead window in the hardware, are given more work to do, and higher performance is achieved. We discuss some new architectural features and software support required for *speculative* operations, that result from moving operations above conditional jumps as part of the techniques. As a preliminary demonstration of the value of the techniques, an inner loop of the sequential natured SPEC Lisp Interpreter benchmark is automatically parallelized; the result indicates a potential performance of 3.5 RISC instructions/cycle in this inner loop.

Some Global Compiler Optimizations and Architectural Features for Improving Performance of Superscalars*

Kemal Ebcioğlu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598

Randy Groves
IBM Advanced Workstations Division
11400 Burnet Road
Austin, Texas 78758

Abstract

We describe a method for converting a given program in the assembly language of a superscalar machine to another program written in the same assembly language, such that the resulting program produces the same final results as to the original one, and can run significantly faster. The method, inspired by new global parallelization techniques for VLIW architectures, finds and places together independently executable operations that may be far apart in the original code (i.e., that may be separated by many conditional branches or that may belong to different iterations of a loop.) As a result, the pipelined functional units in the machine, which could not be kept busy because of the limited size of the execution lookahead window in the hardware, are given more work to do, and higher performance is achieved. We discuss some new architectural features and software support required for *speculative* operations, that result from moving operations above conditional jumps as part of the techniques. As a preliminary demonstration of the value of the techniques, an inner loop of the sequential natured SPEC Lisp Interpreter benchmark is automatically parallelized; the result indicates a potential performance of 3.5 RISC instructions/cycle in this inner loop.

1 Introduction

A great amount of attention is presently being paid to improving the performance of RISC processors. The *superscalar* architecture is a recent name coined to 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 (e.g. the Intel i860[8]).

Speedup measurements on superscalar machines tested on existing code generated by existing compilers have so far been disappointing. For example M.D. Smith et al. [10] indicate that practical speedups over an existing RISC processor would be limited to a factor of about 2 even with aggressive superscalar configurations. Smith's paper nevertheless mentions the possibility that further speedup might be achieved if a compiler were able to significantly rearrange the instructions, but does not elaborate on the techniques of how to do this. The purpose of our present paper is to describe some new global compiler optimization techniques and architectural features to help overcome the obstacles to speedup on superscalar architectures.

One potential reason for poor performance of 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 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 system code; in system code branches are very frequent, so high branch throughput seems mandatory. The software and hardware techniques described here should help to solve some of the problems listed above.

The techniques outlined 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 ([2, 1, 3]). This project consists of the design of a parallelizing compiler and an architecture (whose prototype is being built), for extracting parallelism from extremely sequential, non-numerical code. Our compiler techniques can bring together in the same VLIW instruction independently executable operations and tests that may be separated by

*This paper was accepted to, and presented at the *International Conference on Computer Design* (ICCD-1990), held in Cambridge, Massachusetts, in September 1990. We were late in submitting this final paper, so it does not appear in the ICCD-1990 Proceedings.

many conditional branches in the original code, or that may belong to different iterations of a loop. 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 conserve resources further. Advanced memory disambiguation techniques (enhancements of those used in the Bulldog compiler [5]) are used for determining if two memory references can access the same location.

The thrust of the present paper is not a new compilation technique for superscalars, but a mapping from VLIW instructions to groups of independently executable RISC/superscalar instructions, that, albeit simple, make these very aggressive parallelization techniques that we heretofore thought to be applicable mainly to VLIW's, applicable to plain superscalar code as well, offering the possibility of high speedups on superscalars. We will describe the features of our project related to superscalar machines below.

2 Overview of our VLIW Architecture

The superscalar architecture we are assuming is a machine where groups of regular RISC instructions are used for abbreviating a VLIW instruction. To understand the superscalar model, it is first necessary to explain the instruction semantics for our VLIW machine.

Our VLIW machine has multiple functional units all of which share a register file and multiple condition code registers that assume binary values (true or false). It supports multiway branching and conditional execution.

The instructions of the machine have the form of a decision tree (please refer to the example tree labeled L8 at its root, given below). The tree is encoded in binary form in the instruction word. At the terminal nodes of the tree there are labels, which this instruction can branch to. At each non-terminal node of the tree, there is a test on a condition code register (the machine has multiple condition code registers). On each directed edge of the tree there can be zero or more three-register arithmetic operations, or memory loads/stores. An instruction is executed in a single machine cycle.[1] Conceptually, there are two phases in the machine cycle: the path selection phase and the execution phase.

At the path selection phase, the machine determines a unique path from the root of the tree to a tip node of the tree, based on the *old* values of the condition code registers that were set in the previous instructions, in a decision tree like fashion. If the test on a given node is true, the taken path branches to the left, otherwise the taken path branches to the right. (This conceptually sequential process is really done with very fast, parallel hardware).

At the execution phase, only the three-register arithmetic operations and loads/stores that are on the selected path are executed, using the *old* values of the registers available from the previous instruction as operands or storage addresses. (Even if one operation such as $x\ op\ y \rightarrow z$ sets the source register z of another operation $z\ op\ w \rightarrow u$, simultaneous execution is possible since $z\ op\ w$ will use the old value of z available from the previous instruction). If more than one operation sets the same destination register on the selected path, or stores into the same memory byte, the operation closest to the tip node determines the final value in the destination register or memory byte. A load on the selected path will read the memory before any stores are performed to the same memory location.

Finally the results of the operations are written into the register file, and control branches to the instruction whose label is indicated at the tip node of the taken branch of the current instruction. The next instruction will observe the updated values of the registers, condition codes and storage locations.

For the specific implementation, there will be a finite limit on the number of *distinct* arithmetic operations, loads/stores, and the number of branch target addresses in a particular instruction. Otherwise, the shape of the tree and the placement of operations on its edges can be arbitrary. For a more detailed discussion on the architecture, and on how the seemingly complex instruction semantics described here is implemented with a fast cycle time, readers should refer to [2].

Now to exemplify how the tree instructions in our VLIW paradigm can speed up sequential natured code, consider the following inner loop taken from the *xlygetvalue* subroutine of SPEC Lisp Interpreter benchmark. We write the sequential code in the assembly language of a "generic" RISC machine with several general purpose registers and several boolean condition code registers. This RISC machine can execute three register arithmetic operations, register-immediate arithmetic operations, compares, loads and stores using the base+index or base+displacement addressing modes or the autoincrement variants of these addressing modes, and conditional and unconditional branches. We assume that, to prevent a pipeline stall, there must be at least one cycle delay between a load and any operation using the result register of that load, and at least one cycle delay between a compare and a conditional branch using the condition code set by the compare, regardless of how many instructions can be issued per cycle. The Intel i860(tm) has such one cycle load-use and compare-branch delays, for example. We will add these delays as explicit instructions to the original sequential code as shown below, to make it fit the single cycle paradigm of our VLIW compiler.

[1] If OP is an n cycle pipelined operation, it can be can be represented in this single cycle paradigm as n single cycle operations, one "r1 OP r2 $\rightarrow$ r3", followed by $n-1$ "delay r3 $\rightarrow$ r3" operations. Delay operations do not take resources, and have the semantics of a copy operation.

```
SPEC Lisp Interpreter benchmark xlygetvalue
subroutine inner loop

Explanation of registers at entry to loop:
r8 = address of 1st element of linked list
8 = offset of cdr field
4 = offset of car field
r3 = address of item to match against

Purpose of loop: look for an element x of the list
with car(car(x)) equal to r3, and if found exit to
"found" with r9=car(x). Otherwise, if the end of
the list is reached, exit to "endofchain"

loop:
load  8(r8)->r8  ;r8= cdr(1st) = address of 2nd
delay  r8->r8    ;to represent load-use delay
(r8==0) -> cc0   ;cc0 = (2nd == NULL)
delay cc0-> cc0  ;compare-branch delay
if cc0 goto endofchain ;if 2nd=NULL go endofchain
load 4(r8)-> r9        ;r9 = car(2nd)
delay r9-> r9         ;load-use delay
load  4(r9)->r10      ;r10 = car(car(2nd))
delay r10->r10        ;load-use delay
(r3==r10) -> cc1      ;cc1 = (r3 == car(car(2nd)))
delay cc1-> cc1       ;compare-branch delay
if not cc1 goto loop  ;if (r3== car(car(2nd)))
                      ;exit at "found"
                      ;else goto loop with 2nd now
                      ;replaced with 3rd etc.
found:
  r9 is live here
  ...
endofchain:
     no registers set in loop live here
```

We now show a few tree-instructions in our VLIW architecture paradigm, which are the parallelized version of the above loop. This VLIW code is a commented version of the result that was obtained automatically from the sequential code above, by our VLIW parallelizing compiler. To understand how the VLIW machine instructions work, and get a sense of what sort of parallelization opportunities exist, we would suggest that the reader verify that the parallel VLIW program (2 VLIW cycles/iteration = 3.5 useful RISC instructions/cycle), is semantically equivalent to the original sequential program. This program fragment is probably one of the more sequential natured fragments in the SPEC suite.

```
loop
|load 8(r8)->r8  ;(1) r8= cdr(1st) = address of 2nd
L1               ;"(1)" is the iteration number

L1
|delay r8->r8   ; idle for one cycle
L2

L2
| (r8==0)->cc0    ;(1) cc0= (2nd==NULL)
| load 4(r8)->r9  ;(1) r9 = car(2nd)
| load 8(r8)->r8  ;(2) r8=cdr(2nd)=addr of 3rd
L3

L3
| delay cc0->cc0 ; idle for one cycle
| delay r9->r9
| delay r8->r8
L4

         L4
         |
       if cc0 ;(1) if 2nd==NULL goto endofchain
       /  \
      /    \ load 4(r9)->r10 ;(1) r10=caar(2nd)
     /      \ (r8==0)->cc0  ;(2) cc0=(3rd==NULL)
    /        \ load 4(r8)->r9' ;(2) r9'=car(3rd)
   /          \  load 8(r8)->r8 ;(3) r8=cdr(3rd)
  /            \                ;= addr of 4th
endofchain      L5

L5
|  delay r10->r10  ;idle for one cycle
|  delay cc0->cc0
|  delay r9'->r9'
|  delay r8->r8
L6

         L6
          | (r3==r10)->cc1 ;(1) cc1=(r3==caar(2nd))
     if  cc0  ;(2) if 3rd==NULL goto endofchain''
        / \
       /   \ load 4(r9')->r10 ;(2) r10=caar(3rd)
      /     \ (r8==0)->cc0 ;(3)cc0=(4th==NULL)
     /       \ load 4(r8)->r9'' ;(3) r9''=car(4th)
    /         \ load 8(r8)->r8  ;(4) r8=cdr(4th)
   /           \                ; = addr of 5th
endofchain''    L7

L7
| delay cc1->cc1  ;idle for one cycle
| delay r10->r10
| delay cc0->cc0
| delay r9''->r9''
| delay r8->r8
L8

/*
  Initially n=0
  loop invariants here, for n=0,1,2,...
  cc1 =  (r3 == car(car(n+2nd)) )
  r10  = car(car(n+3rd))
  cc0  = (address of n+4th elem == NULL)
  r9   = car(n+2nd)
  r9' =  car(n+3rd)
  r9'' = car(n+4th)
  r8  =  address of n+5th element
  r3 =  parameter (sym) to match against
*/
```

```
         L8
         |
        if cc1 ;(1) if r3=caar 2nd exit
        / \
       /   \ (r3==r10)->cc1
      /     \  ;(2) cc1= (r3==caar 3rd)
found        if cc0  ;(3) if 4th=NULL exit
             /   \
    r9'->r9 /     \  r9'-> r9 ;(overhead)
           /       \  ;(2) r9=car(n+3rd)
          /         \  r9''->r9' ;(overhead)
      endofchain''   \ ;(3) r9'=car (n+4th)
                      \ load 4(r9'')->r10
                       \  ;(3)r10=caar(n+4th))
                        \ (r8==0)->cc0
                         \ ;(4) cc0=(n+5th==NULL)
                          \  load  4(r8)->r9''
                           \ ;(4) r9''=car(n+5th)
                            \  load 8(r8)->r8
                             \ ;(5) r8=cdr(n+5th)
                              L7     ;=addr(n+6th)

endofchain''          ;e.g. when coming here from L8,
  | delay cc1->cc1    ;addr(n+4th) is known to be nil,
endofchain'           ;but we still need to check

endofchain'          ;if (caar(n+3rd)==r3)
    |                ;is true, and if so
   if cc1            ;go to found with r9=car(n+3rd)
  /   \
 /     \
found    endofchain
```

Different degrees of parallelization opportunities exist for different sequential natured fragments of the SPEC benchmarks, for example the inner loop of subroutine *cmppt* of the SPEC *eqntott* benchmark (13 instructions with the generic RISC we defined above) can be executed at 1 cycle/iteration on a VLIW with sufficient resources. More extensive experiments on the SPEC benchmarks and other applications will provide a more definitive answer regarding the degree of the available parallelism, although these preliminary results are very promising.

3 A superscalar machine based on our VLIW architecture

We will now describe a model of a hypothetical superscalar machine derived from our VLIW architecture.

The proposed superscalar machine executes a group of adjacent, independent RISC instructions every cycle. The RISC's instruction set is the instruction set of the hypothetical sequential machine fed into our VLIW compiler. The group of RISC instructions corresponding to a VLIW instruction is obtained by breaking up the VLIW instruction into a sequence of serial RISC instructions that perform an equivalent transformation on memory and registers. We call such a group of independent RISC instructions, that are the translation of a VLIW instruction, a "VLIW group."

Sometimes it is possible to convert the operations and tests in a VLIW tree to the corresponding group of independent RISC instructions in a straightforward way, emitting the RISC instructions in the order they occur in a pre-order traversal of the tree starting from the root, and get the same effect as the VLIW instruction would; this case is illustrated by the VLIW instructions in the SPEC Lisp Interpreter example. But such a translation is not correct in general. Since a path on the VLIW instruction tree may have operations such as x+4→x, followed by x+8→ y (one operation uses the old value of a register set by the other) these operations have to be reordered in the serial code, by placing an operation or conditional jump that uses the old value of a register before one that sets the same register to a new value.

One simple translation method from a VLIW tree to RISC code, is to choose an operation or test from the tree that, in RISC code, can be executed before all the other operations and tests in the tree without violating the original VLIW semantics, emit the RISC instruction corresponding to this operation or test, delete all of its occurrences from the tree, and then recursively emit code in the same way for the resulting tree(s). In the case of a test node, deleting it from the tree results in two trees, one which behaves as if the test were true, and another which behaves as if the test were false; The "branch taken" target of the RISC conditional branch corresponding to the test node will be the RISC code generated by the former tree, and the "branch untaken" target will be the RISC code generated by the latter tree. For empty trees that just branch to the next VLIW tree (encountered when all operations and tests are deleted), a RISC unconditional branch will be emitted, unless the branch target is the immediately following RISC instruction. Assume the tree has first been re-arranged, so that multiple assignments to a register on the same tree path have been eliminated. Then, an operation or test *op* in the tree can be executed in RISC code before all other operations and tests in the tree without violating the original VLIW tree semantics, if and only if the following are all satisfied: if *op* has a destination register, then this register is not used as a source by any other operation or test in the tree, and on every path of the tree from the root to a tip node, this register is either set on the path, or is not live at the target VLIW instruction of the path; if *op* is a store, it occurs as the first store on every path of the tree (we respect order of stores on each tree path), and there are no loads in the tree that could access the same location. In some cases only circularly dependent operations such as x:=y and y:=x, which exchange the values of x and y in the VLIW code, will remain in a tree; there is *no* possible order to execute these operations in RISC code, and get the same effect as the VLIW instruction; so no operation can be chosen by the above criterion. In this case, a new temporary destination register t has to be introduced in the corresponding serial RISC code to break the dependence cycle (giving the serial code t:=y, y:=x, x:=t, in the above example). If temporaries were thus introduced, it may be useful to do some further local compaction to attempt to recover the lost parallelism (otherwise, e.g. in the above example, t:=y and x:=t could have to be executed sequentially by the superscalar hardware, depending on the design).

For a superscalar with small resources, say with 1-2 functional units and 1 branch unit, the technique suggested above may be easy to implement as the natural hardware dispatch mechanism. That is, the superscalar may merely execute the next group of adjacent independent RISC instructions every cycle, or at least we can usefully model it in the compiler as if it did so, even if the superscalar has a more complex dispatch mechanism that can do more. To benefit from the compiler capabilities, the original assembly code of the RISC can be first converted to "sequential" VLIW code with appropriate representation of the pipeline delays, with one operation per VLIW tree instruction; then parallelized by VLIW techniques according to the finite resource constraints of the machine; and then turned back to serial assembly code of the original superscalar (the *delay* $r \rightarrow r$ operations are deleted at this stage). Since pipeline delay cycles due to load-use, compare-branch and other dependences have been explicitly represented to the compiler as VLIW instructions to fill, operations will be moved across basic block and loop iteration boundaries to occupy these previously idle delay cycles, thus yielding better performance than the original sequential code. This global scheduling approach is better than some previous approaches that examined mainly a single basic block at a time for reordering instructions, on pipelined uniprocessors (e.g. [7]).

The hardware implementation methods that could be used to execute a larger group of independent operations and conditional jumps per cycle on a superscalar, are beyond the scope of this paper; we hope to discuss these in a future paper.

For the sake of showing how the translation from the VLIW tree-instructions to RISC code is done, we give here the serialized RISC versions of each VLIW instruction in the Lisp Interpreter example above. The delay operations have been deleted. Horizontal lines indicate VLIW group boundaries.

```
Loop:
load 8(r8)->r8  ;(1) r8=addr of 2nd
------------------
(r8==0)->cc0  ;(1)  cc0= (2nd==NULL)
load 4(r8)->r9  ;(1)  r9 = car(2nd)
load 8(r8)->r8  ;(2)  r8 = cdr(2nd) = addr of 3rd
------------------
if cc0 goto endofchain ;(1) if 2nd==NULL exit
load 4(r9)->r10 ;(1) r10 = car(car(2nd))
(r8==0)->cc0   ;(2) cc0 = (3rd == NULL)
load 4(r8)->r9' ;(2) r9'= car(3rd)
load 8(r8)->r8 ;(3) r8 = cdr(3rd) = addr of 4th
-------------------
(r3==r10)->cc1 ;(1)  cc1= (r3 == car(car(2nd)))
if cc0 goto endofchain';(2) if 3rd==NULL exit
load 4(r9')->r10 ;(2)  r10 = car(car(3rd))
(r8==0)->cc0 ;(3)  cc0 = (addr of 4th== NULL)
load 4(r8)->r9'' ;(3)  r9''= car(4th)
load 8(r8)->r8  ;(4)  r8=cdr(4th)=addr of 5th
--------------------
L8:
if cc1  goto found
(r3==r10)->cc1
if not cc0 goto L8.1
r9'->r9
goto endofchain'
L8.1:
r9'-> r9  ;(2) (overhead) r9=car(n+3rd)
r9''->r9' ;(3) (overhead)  r9'=car (n+4th)
load 4(r9'')->r10 ;(3)  r10=car(car(n+4th))
(r8==0)->cc0 ;(4)  cc0=(addr of n+5th == NULL)
load  4(r8)->r9'' ;(4)r9''=car(n+5th)
load 8(r8)->r8 ;(5) r8=cdr(n+5th)
goto L8
---------------------
found: ... ;r9 live here
endofchain':       ;when coming here from L8,
                   ;n+4th is nil but still need
 if cc1 goto found ;to check if r3==car(n+3rd)
endofchain: ...
```

4 The parallelization algorithm

The task of parallelizing for the superscalar machine (with a matching VLIW machine) is then to start with the sequential code, parallelize it for the corresponding VLIW with the same number of resources, and turn the resulting compacted VLIW trees back into sequential RISC instructions. There are several techniques for compacting VLIW code, including [1, 3, 4, 9] and [5]. We will describe a version of the *enhanced pipeline scheduling* technique ([4]) used in our VLIW compiler briefly, details may be found in other papers mentioned above.

The input to the enhanced pipeline scheduling algorithm is a list of the tree-instructions of a loop, sorted in depth first order.[2] The input trees contain only one arithmetic operation or one load/store or one conditional branch, i.e. the input is sequential code. The algorithm proceeds by constructing the software pipelined schedule stage by stage. At each stage, a data structure called "fence" contains the set of VLIW instructions that will be filled with operations during this stage. A synopsis of the algorithm is as follows (the term "instruction" refers to VLIW instruction here):

```
place the entry instruction of the loop
in "fence". Mark each operation and test in
the loop as belonging to iteration 1.

while there are still operations and tests from
 iteration 1, and the fence is not empty do

 for each instruction n in the fence:
   Move operations  and conditional jumps that are
   below n in the list to n (subject to finite
   resource constraints).
   mark n "filled"
 end for
```

[2]The algorithm is applied to innermost loops first, and then to the next level of outer loops, and eventually to the entire program, after the strongly connected part of each pipelined inner loop is represented as an atomic VLIW instruction at the next level outer loop.

```
  newfence := the successors of members n of the
  fence that are below n in the list and that
  are not yet "filled"

  Remove fence instructions from the list and
  append to the bottom of the list. Increment
  the iteration numbers of operations and tests
  in the fence by 1.

  fence := newfence
end while
```

The actual upward motion of the operations can be made with a variety of acyclic code compaction techniques, e.g. ([3]). The above algorithm will terminate if operations in iteration n are chosen before operations belonging to a later iteration $n+1$, $n+2$,..., since that way eventually no operations will remain from iteration 1. The reader is encouraged to attempt to reconstruct the parallelized version of the Lisp Interpreter loop starting from the algorithm sketch given here.

5 On exceptions caused by speculative operations

When, in the parallelized version of a program, an operation is executed earlier than a conditional branch on which it was control dependent ([6]) in the original code, we call it a *speculative* operation. A VLIW compiler may aggressively move an operation above a conditional jump which preceded it in the original code, and determined whether it would really be executed; thus, in the parallelized code, this speculative operation may execute many cycles before the conditional jump that originally preceded it. The intent here is to execute operations before even knowing the outcome of conditional jumps that precede them, in order to gain some speedup just in case the program decides to take a path that includes these operations. Some operations that may cause fatal errors (such as accessing a very high virtual memory address not available to the program, or causing an arithmetic overflow) will also be moved by the compiler above conditional jumps that originally preceded them. The problem in this case is what the machine and operating system should do, when such a speculative operation causes an exception. The program should not necessarily be aborted, since it is not even known whether the path containing the operation would really be taken in the original, un-parallelized program. Speculative code motion can be very important, for example no improvement results in the Lisp Interpreter code above, unless loads are moved above conditional jumps. Note that stores are never speculative.

For existing machines where the instruction set cannot be changed, there is not much to do except to fully debug the program in the unoptimized mode, and then, in the parallelized code, try to prevent or ignore any fatal errors due to speculative operations belonging to untaken paths, but process any nonfatal exceptions, such as page faults, as usual. Fatal memory exceptions due to speculative loads on untaken paths (that could occur, if for example a Lisp atom variable is treated as a list and its "car" field is loaded by a speculative operation before even executing the conditional jump that tests if it is an atom, accessing a random virtual address beyond the allowable range), can be treated by either providing an extra read only page at the location requested by the load, to hopefully prevent occurrence of the same error again in a loop, or by efficiently ignoring the fatal error at the first level interrupt handler. Note that a load will be executed speculatively in the parallelized code and will cause overhead by accessing an extra page not referenced by the original program, only if all of the following are true: 1-computing the base register of the load in the parallel schedule takes shorter than computing the condition code which determines whether the load will really be executed in the original code; 2-there are sufficiently large machine resources, or a sufficiently small degree of parallelism, for the compiler to choose this speculative load over other operations, during those cycles where the load address is ready, but the condition code is not yet ready (nonspeculative operations will be preferred to speculative ones by the compiler when there is a scheduling choice); 3-the original program takes a path where this load is not executed; 4-the original program never references the page addressed by the speculative load (so the page is either out of bounds, or bringing this page in is definitely useless). The solution then could be to allow a possibly larger working set for an optimized program, along with the special handling of fatal errors. Measurements of the various factors on an actual implementation will determine the tradeoffs.

Code with authority to do memory mapped I/O loads with side effects, cannot make use of these speculative code motions, since speculative loads coming from untaken paths of the original program may in general access unpredictable addresses. Thus, although our proposal for using the VLIW compiler techniques for an existing machine has the potential to improve performance of the majority of operating system utilities, user applications and kernel code; in order to remain safe regarding software compatibility, such aggressive compiler optimizations would best be implemented as a new, optional high optimization level in the compilers, and the operating system would have to be alerted to handle the fatal exceptions in a way that differs from the default, by a system call at the beginning of each program with any speculative operations.

If a superscalar machine is being designed from scratch, a fairly comprehensive support for speculative operations can be achieved by mimicking what is in the IBM VLIW machine ([2]). The techniques of the IBM VLIW machine, described below, are very inexpensive to implement, and provide a usable degree of error checking.

In our VLIW architecture, every operation has a speculative and a nonspeculative version (encoded by an extra bit in the opcodes). The compiler can determine which operations become speculative, when it moves an operation above a conditional jump. The registers have a 33rd bit called the exception tag (saved and restored accross process switches). When a speculative operation causes an error in the "fatal error" class, the machine does not take an interrupt, only the 33rd bit of the result register is set. When this result is used as an operand in subsequent speculative operations the 33rd bit is propagated to the results of these operations. (The address of the original exception-causing operation can also be saved in the remaining 32 bits of the result, and propagated like the 33rd bit; but when both operands of a speculative operation have the 33rd bit set, address information in only one operand will be propagated.) If one of the original paths which contained the speculative operation is finally taken, and the speculative operation result with the 33rd bit=1 is finally used by a nonspeculative operation, an interrupt will occur. The information in the operand which had the exception tag bit set, can identify the original operation which caused the interrupt, and hence at least one possible line in the source program where this operation could have come from, assuming compiler tables were maintained. If the path that originally contained the speculative operation is not taken, nothing happens. The speculative operation then serves merely as an unnecessary computation afforded by the large resources of the machine, that does not affect the final outcome of the program.

Some exceptions (like translation exceptions, or operand alignment interrupts, or IEEE floating point exceptions) are not necessarily fatal. These cannot be ignored even on exceptions due to speculative operations. A simple technique to use with the exception tag and speculative opcode architectural features, is to process the interrupt as usual anyway if it is nonfatal (e.g. give the program the requested page on a page fault, handle the alignment error), and to resume the program with a result with the 33rd bit set, if the exception is found to be fatal, but the operation causing it is speculative. The program can be aborted as usual, on a fatal exception caused by a nonspeculative operation. Some extra exceptions, that never occurred in the original sequential program, may also need to be handled with this approach, when a number of conditions (like those listed above for memory exceptions) are simultaneously true.

Since speculative loads on untaken paths may access unpredictable addresses in general, speculative loads that could cause side effects by touching I/O space accidentally must be ignored by the hardware (hardware can recognize that a load is speculative from the opcode). I/O space loads with possible side effects can only be nonspeculative, assuming the I/O space variables are declared in a special way in the high level language program, and the compiler inhibits the optimization of the instructions that access these. This approach allows a machine with the exception tag/speculative opcodes feature to perform uninhibited memory mapped I/O as well.

6 Conclusions

We have described a method to apply the compiler algorithms for achieving paralellism on VLIW machines, to superscalar architectures. We have described the architectural and operating system support, namely the support for speculative loads, to make maximal use of the resources of a superscalar machine. Although a more extensive study is needed, our approach appears promising for making better use of superscalar architectures.

References

[1] Ebcioğlu, K. [1987]. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming*, pp. 69-79, ACM Press.

[2] 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)*, M. Cosnard et al. (eds.), pp. 3-21, North Holland.

[3] Ebcioğlu, K. and Nicolau, A. [1989]. A Global Resource Constrained Parallelization Technique. *Proceedings of 1989 International Conference on Supercomputing*, Crete, Greece, pp. 154-163.

[4] Ebcioğlu, K. and Nakatani, T. [1989]. *A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture*, Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing, University of Illinois at Urbana-Champaign.

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

[6] Ferrante, J., Ottenstein, K., and Warren, J. [1987]. *The Program Dependence Graph and Its Use in Optimization*, ACM Transactions on Programming Languages and Systems, 9:3, pp. 319-349.

[7] Gibbons, P.B. and Muchnick, S.S. [1986]. "Efficient Instruction Scheduling for Pipelined Processors" *Proc. SIGPLAN '86 Symposium on Compiler Construction*, ACM Press, pp. 11-16.

[8] Intel Corp. [1989]. *i860(tm) 64-bit Microprocessor Programmer's Reference Manual*, Santa Clara, California.

[9] Nakatani, T. and Ebcioğlu K. [1989]. "Combining" as a Compilation Technique for VLIW Architectures. *Proceedings of the 22nd Workshop on Microprogramming and Microarchitecture*, ACM and IEEE, Dublin, pp. 43-55.

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

APPENDIX: An application of our VLIW compiler techniques on the IBM RS/6000

(taken from the foils we presented at ICCD-1990)
Search a linked list for an element matching given key.

Original code:
r3= address of list
r4= key to be compared against
r3= return value (null or address of matching record)

```
   loop:
      l    r0,x(r3)        1
      cmp  cr0,r0,r4       2
      beq  cr0,ret         0
      l    r3,link(r3)     1
      cmpi cr1,r3,0        2
      bne  cr1,loop        3
ret:  return r3
```

Performance: 9 cycles/iteration

RS/6000 pipeline delays (IBM J. of R&D, January 1990):

★ load into *r* - use *r*, delay is 1 cycle
★ compare-taken cond. branch delay is 3 cycles (3 ∸ *k* cycles, if there are *k* instr. between compare and branch)
★ compare-untaken cond. branch delay is 0 cycles (but an unconditional branch following the untaken branch may then incur extra delay).

A previous technique for reducing branch delays on RS/6000

(Golumbic and Rainish, IBM J. of R&D, January 1990)

Moves a few instructions from the top of the loop to the bottom, to cover compare-taken branch delay.

```
   loop:
      l    r0,x(r3)
      cmp  cr0,r0,r4
   loop1:
      beq  cr0,ret           0
      l    r3,link(r3)       1
      cmpi cr1,r3,0          2
      beq  cr1,ret           0
      l    r0,x(r3)          1
      cmp  cr0,r0,r4         2
      b    loop1             0-1
ret:  return r3

6-7 cycles/iteration
```

Load-use delays are still not covered

Compilation Stages for VLIW approach

- ★ Unroll/unwind inner loops
- ★ Rename registers
- ★ Insert dummy delay operations to represent pipeline delays
- ★ Convert to VLIW code
- ★ VLIW compaction and software pipelining (e.g. enhanced pipeline scheduling, Ebcioglu-Nakatani 1989)
- ★ Re-allocate registers
- ★ Convert back to serial code

Compilation Stages for VLIW approach

Code after unrolling, renaming, delay insertion:
(State just before global "VLIW" scheduling)
Each cycle is a VLIW instruction to fill

```
loop: l    r0',x(r3)
      <1 delay on r0'>   /* E.g: l r3',link(r3) moves here */
      cmp  cr0',r0',r4
      <3 delays on cr0'>
      beq  cr0',ret
      l    r3',link(r3)
      <1 delay on r3'>
      cmpi cr1',r3',0
      <3 delays on cr1'>
      beq  cr1',ret$
      l    r0'',x(r3')
      <delay>
      cmp  cr0'',r0'',r4
      <3 delays>
      beq  cr0'',ret$
      l    r3'',link(r3')
      <delay>
      cmpi cr1'',r3'',0
      <3 delays>
      beq  cr1'',ret$$
```

Compilation Stages for VLIW approach

```
      l    r0,x(r3'')
      <delay>
      cmp  cr0,r0,r4
      <3 delays>
      beq  cr0,ret$$
      l    r3,link(r3'')
      <delay>
      cmpi cr1,r3,0
      <3 delays>
      bne  cr1,loop
ret:   return r3
ret$:  lr   r3,r3'
       b ret
ret$$: lr  r3,r3''
       b ret
```

Final result of VLIW compilation techniques

```
loop:  l      r0,x(r3)
       l      r6,link(r3)
       cmp    cr0,r0,r4
loop1: cmpi   cr1,r6,0        1
       l      r0,x(r6)        1
       l      r5,link(r6)     1
       beq    cr0,ret         0
       cmp    cr0,r0,r4       1
       beq    cr1,ret$        0
       cmpi   cr1,r5,0        1
       l      r0,x(r5)        1
       l      r3,link(r5)     1
       beq    cr0,ret$        0
       cmp    cr0,r0,r4       1
       beq    cr1,ret$$       0
       cmpi   cr1,r3,0        1
       l      r0,x(r3)        1
       l      r6,link(r3)     1
       beq    cr0,ret$$       0
       cmp    cr0,r0,r4       1
       bne    cr1,loop1       0
ret:   return r3
ret$:  lr r3,r6
       b  ret
ret$$: lr   r3,r5
       b ret
```

(12 cycles/3 iter. = 2.25X better than original version.
At least 1.5X better than Golumbic-Rainish version.)