Chapter 3

INCREASING INSTRUCTION LEVEL PARALLELISM

IA-64's instruction format is designed to specify instructions which can execute concurrently. Often, the straightforward means of generating code would not provide much instruction level parallelism. To increase the number of instructions which can execute concurrently, the programmer (or compiler) can attempt to execute instructions speculatively to fill instruction slots which otherwise would go unused. The difficulty in such code rearrangement is that premature instruction execution may cause exceptions which would not occur had a conventional execution order been chosen. For example, code preceding an access to memory may test that the pointer is valid. Executing the load before the test would make the data from the load operation available sooner, but at the risk of an exception which would not have happened in normal instruction sequencing. IA-64 provides means for executing instructions speculatively but suppressing exceptions until it is determined that the instruction should have been executed.

Branching can be eliminated in some cases and be replaced by predicated code. Instead of branching, instructions on both the taken and fallthrough paths can be interleaved and predicated by the result of the compare instruction, which would otherwise determine the outcome of the branch. In this way, two or more instruction streams can be interpreted concurrently, but only the chosen paths execute. Programs which are restructured in this manner are said to be *if-converted*.

In the case of loops, a loop iteration may be started before the previous one is completed. By having several loop iterations in progress concurrently, there are more instructions available for concurrent execution. The performance improvements can be dramatic when long latency instructions are involved in a loop. Such instructions include memory accesses and floating point computations. Many of the algorithms presented in Part II are designed to be software pipelined.

3.1 BRANCHING

Frequent branching tends to slow down high speed computers. The fundamental reason is that it is difficult to prefetch instructions along the execution path. Attempting to prefetch along both possible directions of a conditional branch would generate much instruction fetch traffic, much of it useless, and it could also waste instruction cache space or destroy useful instruction cache lines. IA-64 ameliorates this situation by eliminating branching altogether in some situations, by means of predicated (conditional) execution, and by allowing branch prediction hints to be given about branches to help the hardware prefetch correctly the instructions which will be required in the near future.

3.1.1 Predication

Predication is a means of conditionally executing instructions. In conventional architectures, conditional branches cause control to flow to those instructions which should be executed under circumstances which triggered the conditional branch. Since mispredicted branches cause disruption in the execution pipeline, means to avoid conditional branches can improve processor performance. Each IA-64 syllable contains a 6-bit predicate field, whose contents reference a *predicate register*, called the *qualifying predicate*. Each predicate register is 1 bit wide. When a qualifying predicate is true (= 1), the instruction is executed. With very few exceptions, when the qualifying predicate is false, the instruction is ignored.

Predicate registers are written as the result of compare instructions. Integer compare instructions take the form[1]

(qp) `cmp.crel.ctype p1,p2=r2,r3`

in which the completer `crel` represents a relation as shown in Table 3.1. `ctype` can either be absent, or take on the values `or`, `and`, `or.andcm`, or `unc`. Except when `ctype` is `unc`, the `cmp` instruction behaves normally: it is executed only when its qualifying predicate `qp` is true (= 1). When `ctype` is absent, `p1` is set to the value of the relationship `r2 crel r3`, and `p2` is set to the complement of `p1`. When `ctype` is `unc`, the instruction executes normally if `qp` is true; otherwise both `p1` and `p2` are set to false (0).

The remaining three values of `ctype` are called parallel compares, and the possible relationships are restricted, as shown in Table 3.2. When `ctype` is `or` (and `qp` is true), both `p1` and `p2` are set to true if the relationship `r2 crel r3` is true, and not changed otherwise. When `ctype` is `and`, both `p1` and `p2` are set to false if the relationship `r2 crel r3` is false, and not changed otherwise. Finally, when `ctype` is `or.andcm` when the relationship `r2 crel r3` is true, `p1` is set to true, and `p2` is set to false.

Although the IA-64 assembler will accept any of the 10 relations shown in Table 3.1 for normal or `unc` compares, the only ones really available in the hardware are

[1] A similar instruction, `fcmp` is used for floating point comparisons and is treated in Section 4.5.

crel	Compare Relation (a rel b)		Pseudo op		
eq	a == b				
ne	a != b		eq		p1 ↔ p2
lt	a < b				
le	a ≤ b	signed	lt	a ↔ b	p1 ↔ p2
gt	a > b		lt	a ↔ b	
ge	a ≥ b		lt		p1 ↔ p2
ltu	a < b				
leu	a ≤ b	unsigned	ltu	a ↔ b	p1 ↔ p2
gtu	a > b		ltu	a ↔ b	
geu	a ≥ b		ltu		p1 ↔ p2

Table 3.1. Relations for Normal and UNC Compares

`eq`, `lt`, ant `ltu`; the other seven are pseudo-ops, and their effects are obtained by permuting the result predicates or the inputs. For example, an instruction written as `cmp.le p1,p2=r2,r3` is treated by the assembler as `cmp.lt p2,p1=r3,r2`. The operand `r2` may be a general purpose register or an immediate field of 8 bits. Similarly, for parallel compares, if one of the comparands is 0, it must be r2; if it is specified as r3, the assembler treats it as a pseudo-op as shown in Table 3.2.

crel	Compare Relation (a rel b)		Pseudo op	
eq	a == b			
ne	a != b			
lt	0 < b			
lt	a < 0	unsigned	gt	a ↔ b
le	0 ≤ b			
le	a ≤ 0		ge	a ↔ b
gt	0 > b			
gt	a > b		lt	a ↔ b
ge	0 ≥ b			
ge	a ≥ 0		le	a ↔ b

Table 3.2. Relations for Parallel Compares

Thus some sequences can be programmed without the need for any branching. For example, the C code fragment

```
if (a<b) i++;
else i--;
```

can be executed with only two instruction groups, assuming that registers r8, r9,

and r10 contain the values of a, b, and i, respectively, as follows:

```
      cmp.lt  p2, p3 = r8, r9;;
(p2)  add     r10 = 1, 10
(p3)  add     r10 = -1, r10;;
```

The qualifying predicates are shown in parentheses. If no qualifying predicate is shown, the instruction references predicate register 0, whose contents are always 1. The first instruction sets `p2` to `true` (1) if $a < b$, and sets `p2` to `false`(0) if $a \geq b$. Predicate `p3` is set to the complement of `p2`. The two semicolons indicate the end of an instruction group. The remaining two syllables both reference the predicates set by the first. Only one of the two instructions will be executed, depending on the outcome of the comparison instruction.

In this little example, one of two execution units receiving add instructions in the second instruction group will not produce a result. This shows another way to exploit the wide instruction word. Essentially several alternate computations may be encoded in the same instruction group, but only some will execute. If the computations in each clause of the `if` statement were sufficiently long, a branch on one of the predicates should be taken.

Short predicated sequences are generally preferable to conditional branches, but excessively long predicated sequences may have the effect of losing useful, concurrent execution because roughly half of the syllables considered for execution will be predicated *off*.

Several related instructions test general purpose registers, and set predicate registers in a manner similar to `cmp`. The instruction

(qp) `cmp4.crel.ctype p1,p2=r2,r3`

behaves just like the cmp instruction, but only considers the low 32 bits of `r2` and `r3`, to give better performance to the 32-bit data types. A single bit in a register can be tested by the instruction

(qp) `tbit.tz.ctype p1,p2=r2,`$\texttt{pos}_6$

The bit in register r2 specified by the constant $\texttt{pos}_6$ is tested, and the predicates are set as indicated in Table 3.3. Likewise the NaT bit associated with a register can be tested by the instruction

(qp) `tnat.tz.ctype p1,p2=r2`

3.1.2 Branch Instructions

Predicates are also used to control conditional branches. Table 3.4 shows some of the branch instruction types that are available. They are coded in one of the following two forms:

(*qp*) br.brtype.brhint target
(*qp*) br.brtype.brhint `b1`=target

tz	Condition
z	$\texttt{r2}_{pos_6} = 0$
nz	$\texttt{r2}_{pos_6} = 1$

Table 3.3. Bit Testing Relations

The second form is used for call-type branches, in which `b1` specifies a branch register into which the processor writes the program counter of the next sequential instruction group. The branch register `b1` will be used as the target of a br.ret instruction to return from the callee. The target is either a 25-bit immediate field specifying the target relative to the instruction pointer, or a branch register which contains the target branch address.

brtype	Function	Branch Condition
none	unconditional branch	
cond	conditional branch	qualifying predicate
call	conditional call	qualifying predicate
ret	conditional return	qualifying predicate
cloop	counted loop branch	loop count
ctop, cexit	Software pipelined counted loop	loop count epilog count
wtop, wexit	Software pipelined while loop	qualifying predicate epilog count

Table 3.4. Branch Types

Conditional branches are simply branches which are predicated. In IA-64, instructions which set predicates, that is to say, instructions which calculate whether or not to branch, are separate from the branch instruction itself.[2] In IA-64, call instructions and return instructions are predicated. If the qualifying predicate is p0, they become unconditional, but the power to make them conditional is an architectural feature.

IA-64 relaxes the read-after-write restrictions, and permits a compare instruction to be in the same instruction group as a branch whose qualifying predicate is computed by the branch. In the case of one cycle latency instructions (such as cmp or cmp4), there would be no delays caused by combining the compare and dependent branches in the same instruction group. Longer latency compares, such as `fcmp` (floating point compare), may also be in the same instruction group as the branch which consumes its result, but long delays may occur.

Branch hints are designed to permit the coder (human or compiler) to specify the likelihood that a branch will be successful (that is, that it will cause execution

[2] In both IA-32 and PA-RISC architectures, the parents of IA-64, a single instruction calculated a conditional expression, and branched on certain outcomes.

to resume at the target, rather than to fall through to the next instruction). The branch hint consists of three components: `brhint = bwh.ph.dh`, given in Tables 3.5, 3.6, and 3.7. The hints do not affect the functional behavior of a program, but an implementation of IA-64 is entitled to use the hints to aid in the prefetching of instructions. Caution: Improperly predicted branches can cause substantial performance penalties.

bwh Completer	Branch Whether Hint
spnt	Static Not-Taken
sptk	Static Taken
dpnt	Dynamic Not-Taken
dptk	Dynamic Taken

Table 3.5. Branch Whether Hint (bwh)

The branch-whether hints allow the coder to statically predict whether the branch will be taken. The prediction can be deferred to dynamic branch prediction hardware. The second branch hint allows the coder to specify how many instruction cache lines to prefetch from the predicted target. The deallocation hint specifies whether to keep or clear recent dynamic branch history.

ph Completer	Prefetch Hint
few or *none*	Prefetch few lines
many	Prefetch many lines

Table 3.6. Sequential Prefetch Hint

dh Completer	Deallocation Hint
none	Do not deallocate
clr	Deallocate branch information

Table 3.7. Branch Cache Deallocation Hint

3.1.2.1 Example

Consider the following C program, which is used as part of a multiprecision arithmetic package. Its purpose is to add two 64-bit integers and a possible carry (of 1), producing a sum and a carry.

```
typedef struct {
     unsigned long int sum;
     unsigned long int carry;
     } fullsum;
```

```
fullsum fulladd(unsigned long int a,
                unsigned long int b,
                unsigned long int carry)
{
     fullsum result;
     if (carry) {
          result.sum = a + b + 1;
          if (result.sum <= a) result.carry = 1;
          else result.carry = 0;
     } else {
          result.sum = a + b;
          if (result.sum < a) result.carry = 1;
          else result.carry = 0;
     }
     return result
}
```

The code utilizes the fact that a carry can occur if and only if the sum (modulo word size) is less than either of its operands. In the case where the sum includes an input carry of 1, an output carry can occur if and only if the sum is no larger than either operand. The compiler generated assembly language for the routine `fullsum` makes use of if-conversion. Consequently, none of the code in the if statements results in branches.

```
          .proc          fulladd
fulladd::
          add            r8=r32,r33      //r32 = a, r33 = b
          cmp.eq         p6,p8=r0,r34;;  //r34 = carry
   (p8)   adds           r8=1,r8
   (p6)   cmp.ltu.unc    p6,p7=r8,r32;;  //if (result.sum < a)
   (p8)   cmp.ltu.unc    p8,p9=r32,r8    //if (result.sum <= a)
   (p6)   mov            r9=1
   (p7)   mov            r9=0;;
   (p9)   mov            r9=1
   (p8)   mov            r9=0
          br.ret.sptk.few rp;;
          .endp          fulladd
```

As a result of the first `cmp` instruction, `p6` is true for the case where the parameter `carry` is zero, and `p8` is true when there is an input carry. In the second instruction group, in the carry case, the new sum is completed into return register `r8`. Otherwise, the new sum is compared to an input, and `p6` is true if an output carry is to be generated, and `p7` is true if no output carry is generated. Because

the `unc` form of compare was used, if `p6` was false, both `p6` and `p7` are set to false (so that instructions in subsequent bundles, predicated by `p6` or `p7`, would not be executed).

In the third instruction group, there are three instructions, each with a different qualifying predicate. Only one of the predicates will be true. When p8 is true, the code is determining whether a new carry is generated when there was an input carry. Notice how the operands and results are reversed (because the `leu` compare type is a pseudo-op; see Table 3.1). Other than the return in the final instruction group, there are no branches. The entire subroutine executes in four cycles, and includes two additions, three comparisons, and some data moves.

3.1.3 Software Pipelining

The remaining branch instruction types are designed to enable software pipelining. Instruction level parallelism is sometimes enhanced by starting an iteration of a loop before the previous iteration has finished; this technique is called *software pipelining*. It is often achieved by unrolling a loop and then scheduling the resultant, unrolled loop body. If the loop is very long, or has an indeterminate number of iterations, then the loop is only unrolled a small, fixed number of times, the new loop body consists of the unrolled loop body, and the new loop is executed $\lfloor n/r \rfloor$ times, where n is the original number of loop iterations, and r is the unrolling factor. The start and end of the unrolled sequence are likely to exhibit less instruction level parallelism than the interior of the unrolled loop, and additional code may be needed if it cannot be established that r divides n.

An IA-64 architectural feature, *register renaming*, combined with predication and special loop terminating instructions, enables software pipelining to be achieved without unrolling, and without loss of instruction level parallelism. Floating point registers `fr32` through `fr127`, predicate registers `pr16` through `pr63`, and a subset of the general purpose registers starting with `gr32`, are subject to renaming whenever a loop-closing branch instruction is executed. For all renamable registers, the actual register name is the virtual name (register number appearing in the instruction) plus a register relocation factor.

There are separate relocation factors for the general purpose, floating point, and predicate registers. A side effect of the loop closing branches is to decrement the register relocation factors by 1, modulo the number of registers available for rotation. Thus, if the relocation factor starts at 0, referencing `fr32` will access `fr32`; on the next loop iteration, this same reference to `fr32` will access what was formerly `fr127`. To access what had been in `fr32` on the first iteration, `fr33` must be referenced. Thus quantities left in a rotating register in one iteration appear in successively higher virtual registers in subsequent iterations. The effect of the rotation is to allow the identical code to be used in all iterations, without the need to issue additional register move instructions at the end of each iteration.

IA-64 architecture provides five special loop closing instructions: `cloop` to close simple counted loops, `ctop` and `cexit` for software pipelined counted loops, and

`wtop` and `wexit` for software pipelined while-loops. `ctop` and `wtop` are for loops which terminate with the last instruction in the loop; `cexit` and `wexit` are for situations where control leaves the loop at other than the last instruction; in the case of `cexit`, it can be employed when combining loop unrolling with software pipelining, or to accommodate zero-trip loops.

The branch types `cloop`, `ctop`, and `cexit` do not use the qualifying predicate field. They use the *loop count* register (and `ctop` and `cexit` also use the *epilog count* register), to determine when to exit a loop.

At the beginning of a software pipelined loop, predicate register 16 is set to true, and the remaining predicate registers are initialized to false. The loop count register is initialized to one less than the number of iterations desired. Instructions meant to execute in the first trip of the loop are predicated by `pr16`. Another side effect of a loop-closing branch is to set the (new) `pr16` to true. Thus, in the second trip through the loop, a new computation is begun by executing all instructions predicated by `pr16`, while the initial computation continues by executing all instructions predicated by `pr17`. The `br.ctop` instruction will decrement the loop count register whenever it is nonzero. If the loop count register is zero, then, if the epilog counter is also zero, the loop terminates. Otherwise, the epilog counter is decremented by one, the registers are rotated, but `pr16` is set to false. If the epilog counter remains non-zero after being decremented, the loop is repeated. Because pr16 is set to zero whenever the loop counter is zero, no new iterations are started, but iterations which were started before the loop counter became zero are permitted to finish. [The `br.cexit` instruction is used in the interior of a loop; it causes a branch (loop exit) whenever a `br.ctop` would not have branched to the loop beginning, and would fall through to the next instruction (with the loop counter or epilog counter decreased by one) if the br.ctop would have caused another loop iteration. An illustration using the `br.cexit` instruction can be found in Section 5.6.2.]

To illustrate software pipelining, let us extend the fulladd routine of Section 3.1.2.1 to a full multiprecision addition. In performance-critical code, a blank line will separate instruction groups, to make them more readily visible.

```
void multiadd(unsigned long int a[],
              unsigned long int b[],
              unsigned long int c[],
              int n)
{
   int i;
   int carry = 0;              //no carry initially
   for (i = 0; i < n; i++) {
      if (carry) {
         c[i] = a[i] + b[i] + 1;
         if (c[i] <= a[i]) carry = 1; else carry = 0;
      } else {
         c[i] = a[i] + b[i];
```

```
        if (c[i] < a[i]) carry = 1; else carry = 0;
      }
   }
}
```

The translation of this routine uses a software pipelined loop, scheduled for an IA-64 implementation in which the L0 data cache has a 1 cycle latency:

```
          .proc multiadd
multiadd:
          alloc         r31=ar.pfs,0,11,0,8 //8 rotating registers
          mov           r42=pr
          addl          r2=0,r1
          cmp4.lt.unc   p0,p6=r0,r35;;  //p6=(n>0)
          mov.i         r41=ar.lc
          mov           r10=r32
          mov           r11=r33
   (p6)   br.ret.spnt.few rp;;
          mov           pr.rot=0        //zero all rotating prs
          adds          r8=-1,r35;;     //vector length - 1
          mov           r14=r34         //move result pointer
          mov           ar.lc=r8;;      //set up loop counter
          mov           r9=0            //set carry=0 outside loop
          cmp.ne        p16,p0=1,r0     //force p16 to be 1
          mov.i         ar.ec=2;;       //set up epilog counter
loop::
   (p16)  ld8           r32=[r10],8     //fetch a[i]
   (p16)  ld8           r34=[r11],8     //fetch b[i]
   (p20)  cmp.ltu.unc   p21,p22=r36,r33 //c[i]<a[i]
   (p19)  cmp.ltu.unc   p23,p24=r35,r38 //c[i]<=a[i]
   (p20)  st8           [r14]=r36,8;;   //store c[i] (no carry case)

   (p19)  st8           [r14]=r38,8     //store c[i] (carry case)
   (p16)  add           r35=r32,r34     //a[i]+b[i]
   (p22)  mov           r9=0            //new carry when carry was 0
   (p21)  mov           r9=1            //new carry when carry was 0
   (p24)  mov           r9=1            //new carry when carry was !0
   (p23)  mov           r9=0;;          //new carry when carry was !0

   (p16)  cmp.eq.unc    p19,p18=r0,r9   //p18 = carry, p19 = no carry
   (p16)  adds          r37=1,r35       //a[i]+b[i]+1
          br.ctop.sptk.few  loop;;

          mov.i         ar.lc=r41;;
```

```
mov           pr=r42,0x1fffe
br.ret.sptk.few  rp;;
.endp         multiadd
```

Looking at the code in the software pipelined loop, there are only three instruction groups! However, it takes two loop traversals to complete the calculation of one component of the result vector, and to determine the carry for the next iteration. The algorithm is said to require two *stages*. In the software pipelined case, there are the additional tasks within the loop to fetch data from memory and store a result. The data loading takes place in the first cycle of the first stage (when `p16` is 1). In a software pipelined loop, `p17` is normally the qualifying predicate during the second stage. But the algorithm which is being software pipelined is itself heavily predicated. At the end of the first stage, predicates `p18` and `p19` are live. Due to rotation, these become `p19` and `p20` during the second stage, of which one is true and the other false. Together, they behave as `p17` would, to control the second stage. Because the compares in the first instruction group are both unconditional, their targets both become false when their qualifying predicates are false. Thus, during the second stage, exactly one of `p21`, `p22`, `p23`, and `p24` are true at the second instruction group, where the carry is calculated into `r9`.

The last operand of the `ld8` and `st8` instructions indicates that the base register is postincremented after the memory access. During the very first trip through the loop, `p19` and `p20` are both false, and the pointer to the vector `c` is not postincremented. During subsequent loop traversals, as we have already observed, only one of `p19` and `p20` is true, and so the pointer to the vector `c` is updated only once.

Notice that the first time the loop is executed, only p16 is initially true. Thus the four instructions which modify `r9` (`r9` represents the C variable `carry`) are all ignored, and the value of `r9` used by the compare in the third instruction group is the value 0, which was set into `r9` before the loop (corresponding to the C statement: `int carry = 0;`).

The closed subroutine in Section 3.1.2.1 required four cycles to complete a single addition. Here, once the data is available, in the second cycle of the first stage, four cycles are also needed: the remaining cycles in the first stage, and the first two instruction groups during the second stage. However, the closed subroutine utilized only 10 syllables of a potential 24 which are available in four instruction groups. The vector version requires an additional four syllables, due to retrieving and storing of data. The total of 14 syllables have been scheduled into three instruction groups, which is the minimum number of instruction groups possible on an implementation which supports instruction groups of six syllables. We should note that the minimum number of instruction groups cannot always be realized.

3.2 SPECULATION

Speculation is the execution of an instruction before it is certain that its result will be needed. In code where there is not much instruction level parallelism, the other-

wise idle execution units can execute instructions which are normally not required until later in the code. There are two major kinds of speculative execution: control speculation and data speculation.

3.2.1 Control Speculation

In control speculation, instructions are issued before the conditional branch which determines whether or not they are required. While this enables results to become available sooner, there is the danger that premature execution will cause exceptions which the conditional branch was designed to avoid.

Speculative load instructions suppress exceptions should they occur. If the speculative load instruction would have caused an exception, it returns a specially tagged result to indicate that the load did not produce a result. In the general purpose registers, the special tag is called the `NaT` (Not a Thing) bit. If the load is successful, the `NaT` bit associated with the register is set to `false` (0), and otherwise it is set to `true` (1). `NaT` bits propagate. If any operand to an instruction has the `NaT` bit set to 1 (or, more colloquially, if an operand is a `NaT`), the result of the operation, if it is a general purpose register, is a `NaT`. In the case of compares, if either comparand is a NaT, both result predicates are set to `false` when the `ctype` specifier is `none`, `unc`, `and`, or `andcm`. For `or` or `or.andcm` specifiers, the result predicates are left unchanged. If an attempt is made to store a `NaT`, a speculation exception occurs. For the purpose of context switches or register saves and restores at subroutine boundaries, the completer `spill` applied to store instructions will store a general purpose register and its `NaT` bit without causing an exception. Likewise, the `fill` completer on a load instruction will restore a general purpose register with its `NaT` bit.

Similarly, a load to a floating point register produces a special value (`NaTVal`) in the floating point register whose role is similar to that of the `NaT` bit. `NaTVal`s propagate similarly to `NaT`s. Instructions that move data between general purpose and floating point registers will produce a `NaT` or `NaTVal` if their input is `NaTVal` or `NaT`, respectively. A floating point store instruction with a `spill` completer stores the entire 82-bit contents of a floating point register into a 16-byte field, without causing an exception. Likewise, a floating point load instruction with a `fill` completer loads an 82-bit subfield from a 16-byte field into a floating point register.

Eventually, a test must be made to determine whether a speculative sequence would have caused an exception. The check instruction branches to a specified label if its register argument is `NaT` or `NaTVal`.

For example, suppose that two registers are loaded speculatively in an off-critical path. Since loads from memory are relatively slow operations, it is advantageous to perform them as much in advance as possible.

```
if (a<b) t = mem1;
else u = mem2;
```

The machine language program might look something like:

```
        ld.s    t=[mem1]    //off-critical path
        ld.s    u=[mem2]    ;;
        ...
        cmp.le  p2,p3=a,b   ;;
        ...
(p2)    chk.s   t,fix_t     //fixup for loading t
(p3)    chk.s   u,fix_u     ;;//fixup for loading u
```

In this code, `ld.s` is the control speculative form of the load instruction, and `chk.s` is the check instruction which determines whether its operand contains a NaT. `chk.s` is also used to check a floating point register for a `NaTVal`. The speculative load instructions begin the memory access earlier than the C code fragment implies. One cycle is used after the compare to insure that the appropriate `ld.s` instruction was successful. One of the `ld.s` may well have failed to execute, but it may be the one which the compare instruction was designed to avoid.

3.2.2 Data Speculation

Data speculation occurs when a load from memory is issued prematurely in advance of a store, which may modify the same address that the load addresses. In the event that a programmer (or compiler) cannot determine with mathematical certainty that such an addressing collision cannot occur, but a collision is deemed highly unlikely, it may be good strategy to start a long latency load instruction prematurely. Data-speculative load instructions, also called advanced loads, are supported by an advanced load address table (ALAT). The effect of an advanced load instruction is to place an entry into the ALAT describing the targeted memory. Instructions which modify memory (such as stores) cause ALAT entries which describe areas overlapping the modified memory to be cleared.

At the point where the load instruction logically should have occurred, one of two types of check instructions may be issued. The first `chk.a` determines if there is an ALAT entry corresponding to its input register. It is similar to the chk.s instruction, which we have seen in the previous section. If the `chk.a` does not find an ALAT entry, a branch to fixup code is taken. Alternatively, `ld.c`, which is a check advanced load instruction, similarly determines if an ALAT entry corresponding to its input argument exists. If it does, then the previously issued speculative load was valid, and no additional action is taken. Otherwise, the load is reissued.

An example where an advanced load may be useful is shown in the following C segment:

$$\mathtt{a[i]} = \mathtt{p} * \mathtt{q};$$
$$\mathtt{c} = \mathtt{a[j]};$$

If the compiler cannot establish that $i \neq j$, but has reason to believe that $i \neq j$ is likely to be true, it can exploit the latency of the floating point multiplication with

an advanced load:

```
fmpy   t=p,q
ldf.a  c=[rj];;   //assume rj points to a[j]
stf    [ri]=t;;   //assume ri points to a[i]
ldf.c  c=[rj];;   //executes only if ri = rj
```

If the advanced load succeeded, the `ldf.c` will complete in one cycle, and `c` can be used in the following instruction. The effective latency of the `lfd.a` instruction has been reduced by the latency of the floating point multiplication. The `stf` and `ldf.c` cannot be in the same instruction group, since there may be a read-after-write dependency.

The architecture also provides for an advanced, control-speculative load.

3.3 PROBLEMS

1. In the example in Section 3.1.2.1, a predicate can be used to represent the variable `carry`, since it represents data which only takes on the values 0 and 1. Recode the example to exploit this observation.

2. IA-64 contains a special form of integer addition:

   (qp) `add.1  r1=r2, r3`

   in which the value stored in `r1` is the sum of the contents of `r2` and `r3`, incremented by 1. Recode the example of Section 3.1.2.1 to exploit this instruction as well as the observation in Problem 1.

3. Recode the software pipelined loop of Section 3.1.3 using the observations of Problems 1 and 2.

4. Recode the software pipelined loop of Section 3.1.3 under the assumption that the latency of a load instruction is 8 cycles.

5. Recode the example in Section 3.2.2 using predication instead of the advanced load. Under what circumstances is the predication approach less useful?