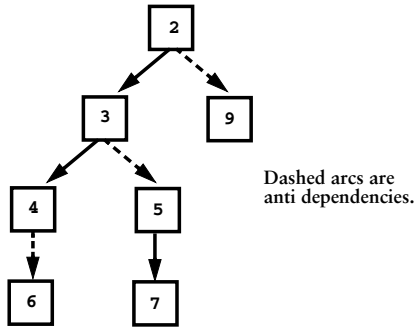
Example

Consider the following simple function which adds an array index to each element of an array and copies the results into a second array:

```
void f (int a[],int b[]) {
  t1 = &a[0];
  t2 = &b[0];
  for (i=0;i<1000;i++,t1++,t2++)
     *t1 = *t2 + i;
}
```

The code for `f` (compiled as a leaf procedure) is:


```
1.   f:   mov     0, %g3
2.   L:   ld      [%o1], %g2
3.        add     %g3, %g2, %g4
4.        st      %g4, [%o0]
5.        add     %g3, 1, %g3
6.        add     %o0, 4, %o0
7.        cmp     %g3, 999
8.        ble     L
9.        add     %o1, 4, %o1
10.       retl
11.       nop
```



We'll software pipeline the loop body, excluding the conditional branch (which is placed after the loop kernel is formed).

This loop body contains 2 loads/ stores, 5 arithmetic and logical operations (including the compare) and one conditional branch.

Let's assume the processor we are compiling for has 1 load/store unit, 3 arithmetic/logic units, and 1 branch unit. That means the processor can (ideally) issue and execute simultaneously 1 load or store, 3 arithmetic and logic instructions, and 1 branch. Thus its maximum issue width is 5. (Current superscalars have roughly this capability.)

Considering resource requirements, we will need at least two cycles to process the contents of the loop body. There are no loop-carried dependencies.

Thus we will estimate this loop's best possible Initiation Interval to be 2.

Since the only instruction that can stall is the root of the dependency dag, we'll schedule using estimated critical path length, which is just the node's height in the tree. Hence we'll schedule the nodes in the order: 2,3,4,5,6,7,9.

We'll schedule all instructions in a legal execution order (respecting dependencies), and we'll try to choose as many instructions as possible to execute in the same cycle.

Starting with the root, instruction 2, we schedule it at cycles 1, 3 (=1+ll), 5 (=1+2*ll):

```
cycle      instruction
1.          ld    [%o1], %g2
2.
3.          ld    [%o1], %g2
4.
5.          ld    [%o1], %g2
```

No conflicts so far, since each of the loads starts an independent iteration.

We'll schedule instruction 3 next. It must be placed at cycles 3, 5 and 7 since it uses the result of the load.

```
cycle      instruction
1.          ld    [%o1], %g2
2.
3.          add   %g3, %g2, %g4
3.          ld    [%o1], %g2
4.
5.          add   %g3, %g2, %g4
5.          ld    [%o1], %g2
6.
7.          add   %g3, %g2, %g4
```

Note that in cycles 3 and 5 we use the current value of **%g2** *and* initiate a load into **%g2**.

Instruction 4 is next. It uses the result of the add we just scheduled, so it is placed at cycles 4 and 6.

```
cycle      instruction
1.          ld    [%o1], %g2
2.
3.          add   %g3, %g2, %g4
3.          ld    [%o1], %g2
4.          st    %g4, [%o0]
5.          add   %g3, %g2, %g4
5.          ld    [%o1], %g2
6.          st    %g4, [%o0]
7.          add   %g3, %g2, %g4
```


Instruction 5 is next. It is anti dependent on instruction 3, so we can place it in the same cycles that 3 uses (3, 5 and 7).

```
cycle      instruction
1.          ld    [%o1], %g2
2.
3.          add   %g3, %g2, %g4
3.          ld    [%o1], %g2
3.          add   %g3, 1, %g3
4.          st    %g4, [%o0]
5.          add   %g3, %g2, %g4
5.          ld    [%o1], %g2
5.          add   %g3, 1, %g3
6.          st    %g4, [%o0]
7.          add   %g3, %g2, %g4
7.          add   %g3, 1, %g3
```

Instruction 6 is next. It is anti dependent on instruction 4, so we can place it in the same cycles that 4 uses (4 and 6).

```
cycle      instruction
1.          ld    [%o1], %g2
2.
3.          add   %g3, %g2, %g4
3.          ld    [%o1], %g2
3.          add   %g3, 1, %g3
4.          st    %g4, [%o0]
4.          add   %o0, 4, %o0
5.          add   %g3, %g2, %g4
5.          ld    [%o1], %g2
5.          add   %g3, 1, %g3
6.          st    %g4, [%o0]
6.          add   %o0, 4, %o0
7.          add   %g3, %g2, %g4
7.          add   %g3, 1, %g3
```


Next we place instruction 7. It uses the result of instruction 5 (%g3), so it is placed in the cycles following instruction 5 (4 and 6).

```
cycle      instruction
1.          ld    [%o1], %g2
2.
3.          add   %g3, %g2, %g4
3.          ld    [%o1], %g2
3.          add   %g3, 1, %g3
4.          st    %g4, [%o0]
4.          add   %o0, 4, %o0
4.          cmp   %g3, 999
5.          add   %g3, %g2, %g4
5.          ld    [%o1], %g2
5.          add   %g3, 1, %g3
6.          st    %g4, [%o0]
6.          add   %o0, 4, %o0
6.          cmp   %g3, 999
7.          add   %g3, %g2, %g4
7.          add   %g3, 1, %g3
```


Finally we place instruction 9. It is anti dependent on instruction 2 so it is placed in the same cycles as instruction 2 (1, 3 and 5).

```
cycle      instruction
1.          ld    [%o1], %g2
1.          add   %o1, 4, %o1
3.          add   %g3, %g2, %g4
3.          ld    [%o1], %g2
3.          add   %o1, 4, %o1
3.          add   %g3, 1, %g3
4.          st    %g4, [%o0]
4.          add   %o0, 4, %o0
4.          cmp   %g3, 999
5.          add   %g3, %g2, %g4
5.          ld    [%o1], %g2
5.          add   %o1, 4, %o1
5.          add   %g3, 1, %g3
6.          st    %g4, [%o0]
6.          add   %o0, 4, %o0
6.          cmp   %g3, 999
7.          add   %g3, %g2, %g4
7.          add   %g3, 1, %g3
```


We look for a 2 cycles kernel that contains all 7 instructions of the loop body that we have scheduled. We also want a kernel that sets the condition code (via the cmp) during its first cycle so that it can be tested during its second (and final) cycle. Cycles 4 and 5 meet these criteria, and will form our kernel.

We place the conditional branch just before the last instruction in cycle 5 (to give the conditional branch a useful instruction for its delay slot).

We now have:

```
cycle        instruction
1.            ld    [%o1], %g2
1.            add   %o1, 4, %o1
3.            add   %g3, %g2, %g4
3.            ld    [%o1], %g2
3.            add   %o1, 4, %o1
3.            add   %g3, 1, %g3
4.       L:   st    %g4, [%o0]
4.            add   %o0, 4, %o0
4.            cmp   %g3, 999
5.            add   %g3, %g2, %g4
5.            ld    [%o1], %g2
5.            add   %o1, 4, %o1
5.            ble   L
5.            add   %g3, 1, %g3
6.            st    %g4, [%o0]
6.            add   %o0, 4, %o0
6.            cmp   %g3, 999
7.            add   %g3, %g2, %g4
7.            add   %g3, 1, %g3
```


A couple of final issues must be dealt with:

- Does the iteration count need to be changed?
  In this case no, since the final valid value of `i`, 999, is used to compute `%g4` in cycle 5, before the loop exits.
- What instructions do we keep as the loop's epilogue?
  None! Instructions past the kernel aren't needed since they are part of future iterations (past `i==999`)which aren't needed or wanted.
- Note that `b[1000]` and `b[1001]` are "touched" even though they are never used. This is probably OK as long as arrays aren't placed at the very end of a page or segment.

Our final loop is:

```
cycle        instruction
1.            ld    [%o1], %g2      !N0
1.            add   %o1, 4, %o1     !N0
3.            add   %g3, %g2, %g4   !N0
3.            ld    [%o1], %g2      !N1
3.            add   %o1, 4, %o1     !N1
3.            add   %g3, 1, %g3     !N0
4.       L:   st    %g4, [%o0]      !N0
4.            add   %o0, 4, %o0     !N0
4.            cmp   %g3, 999        !N0
5.            add   %g3, %g2, %g4   !N1
5.            ld    [%o1], %g2      !N2
5.            add   %o1, 4, %o1     !N2
5.            ble   L               !N0
5.            add   %g3, 1, %g3     !N1
```

This is very efficient code—we use the full parallelism of the processor, executing 5 instructions in cycle 5 and 8 instructions in just 2 cycles. All resource limitations are respected.

False Dependencies & Loop Unrolling

A limiting factor in how "tightly" we can software pipeline a loop is reuse of registers and the false dependencies reuse induces.

Consider the following simple function that copies array elements:

```
void f (int a[],int b[], int lim) {
  for (i=0;i<lim;i++)
     a[i]=b[i];
}
```

The loop that is generated takes 3 cycles:

```
cycle        instruction
1.       L:   ld    [%g3+%o1], %g2
1.            addcc %o2, -1, %o2
3.            st    %g2, [%g3+%o0]
3.            bne   L
3.            add   %g3, 4, %g3
```

We'd like to tighten the iteration interval to 2 or less. One cycle is unlikely, since doing a load and a store in the same cycle is problematic (due to a possible dependence through memory).

If we try to use modulo scheduling, we can't put a second copy of the load in cycle 2 because it would overwrite the contents of the first load. A load in cycle 3 will clash with the store.

The solution is to unroll the loop into two copies, using different registers to hold the contents of the load and the current offset into the arrays.

The use of a "count down" register to test for loop termination is helpful,

since it allows an easy exit from the middle of the loop.

With the renaming of the registers used in the two expanded iterations, scheduling to "tighten" the loop is effective.

After expansion we have:

```
cycle       instruction
1.     L:    ld    [%g3+%o1], %g2
1.           addcc %o2, -1, %o2
3.           st    %g2, [%g3+%o0]
3.           beq   L2
3.           add   %g3, 4, %g4
4.           ld    [%g4+%o1], %g5
4.           addcc %o2, -1, %o2
6.           st    %g5, [%g4+%o0]
6.           bne   L
6.           add   %g4, 4, %g3
       L2:
```

We still have 3 cycles per iteration, because we haven't scheduled yet.

Now we can move the increment of `%g3` (into `%g4`) above other uses of `%g3`. Moreover, we can move the load into `%g5` *above* the store from `%g2` (*if the load and store are independent*):

```
cycle       instruction
1.     L:    ld    [%g3+%o1], %g2
1.           addcc %o2, -1, %o2
1.           add   %g3, 4, %g4
2.           ld    [%g4+%o1], %g5
3.           st    %g2, [%g3+%o0]
3.           beq   L2
3.           addcc %o2, -1, %o2
4.           st    %g5, [%g4+%o0]
4.           bne   L
4.           add   %g4, 4, %g3
       L2:
```

We can normally test whether `%g4+%o1` and `%g3+%o0` can be equal at compile-time, by looking at the actual array parameters.
(Can `&a[0] == &b[1]`?)

Predication

We have seen that conditional execution complicates code scheduling by creating small basic blocks and limiting code movement across conditional branches.

However, the problems conditionals introduce are even more fundamental.

Consider the following code fragment:

```
if (a<b)
     a++;
else b++;
if (c<d)
     c++;
else d++;
```

The two conditionals are completely independent, but they can't be evaluated concurrently in a single thread.

Why?

Look at the Sparc code generated:

```
  cmp     %o0, %g1
  bge,a   L1
  add     %g1, 1, %g1
  add     %o0, 1, %o0
L1:
  cmp     %o5, %o4
  bge,a   L2
  add     %o4, 1, %o4
  add     %o5, 1, %o5
L2:
```


The two compares can't be executed concurrently (because there is only one condition code register).

We can't do two conditional branches to two different places simultaneously.

And we must select the correct combination of two of the four adds to execute.

We could restructure this code into a four-way switch, but this far beyond what a code scheduler is expected to do.

The problem is that while values can easily be computed in parallel, flow of control can't.

The solution?

Convert flow of control computations into value computations.

Our first step is to generalize a single condition code register into a set of predicate registers. The Itanium, for example, includes 64 predicate registers that hold a single boolean value. For our purposes, let's denote a predicate register as `%p0` to `%p63`.

Predicate registers are set by doing compare or test instructions.

Thus

```
  cmpeq   %o0, %g1,%p1
```

sets `%p1` true if the two operands are equal and false otherwise.

The real power of predication is that most instructions can be controlled (predicated) by a predicate register.

Thus

```
add(%p1) %r1,%r2,%r3
```

does an ordinary add but only commits the result (into `%r3`) if `%p1` is true.

A negated form is often included too:

```
add(~%p1) %r1,%r2,%r3
```

In this form, the add is completed only if `%p1` is false.

Using predication, we can eliminate many conditional branches. Now both legs of a conditional can be evaluated, with only one leg allowed to commit.

Returning to our earlier example,

```
if (a<b)
     a++;
else b++;
if (c<d)
     c++;
else d++;
```

we now generate

```
 1. cmplt      %o0, %g1, %p1
 1. cmplt      %o5, %o4, %p2
 2. add(%p1)   %g1, 1, %g1
 2. add(~p1)   %o0, 1, %o0
 2. add(%p2)   %o4, 1, %o4
 2. add(%~p2)  %o5, 1, %o5
```

This entire code fragment can now execute in two cycles, since the two compares and four adds are independent of each other.

Predication Enhances Software Pipelining

Conditionals in a loop body greatly complicate software pipelining since we usually won't know exactly what instructions future iterations will execute.

Consider this minor variant of our earlier example:

```
void f (int a[],int b[]) {
  t1 = &a[0];
  t2 = &b[0];
  for (i=0;i<1000;i++,t1++,t2++)
     if (i%2)
           *t1 = *t2 + i;
     else  *t1 = *t2 - i;

}
```



```
1.   f:   mov     0, %g3
2.   L:   andcc   %g3, 1, %g0
3.        bne     L1
4.        ld      [%o1], %g2
5.        b       L2
6.        sub     %g3, %g2, %g4
7.  L1:   add     %g3, %g2, %g4
8.  L2:   st      %g4, [%o0]
9.        add     %g3, 1, %g3
10.       add     %o0, 4, %o0
11.       cmp     %g3, 999
12.       ble     L
13.       add     %o1, 4, %o1
14.       retl
15.       nop
```

We've added an `andcc` (to do the `i%2` computation) as well as a conditional and unconditional branch. Each iteration will do an add or a subtract.

A two cycle per iteration schedule seems most unlikely.

But predication helps immensely!

The generated code becomes much cleaner:

```
1.   f:   mov        0, %g3
2.   L:   and        %g3, 1, %p1
3.        ld         [%o1], %g2
4.        sub(~%p1)  %g3, %g2, %g4
5.        add(%p1)   %g3, %g2, %g4
6.        st         %g4, [%o0]
7.        add        %g3, 1, %g3
8.        add        %o0, 4, %o0
9.        cmp        %g3, 999
10.       ble        L
11.       add        %o1, 4, %o1
12.       retl
13.       nop
```

And guess what? We can still software pipeline this into 2 cycles per iteration:

```
cycle      instruction
1.          ld    [%o1], %g2
1.          add   %o1, 4, %o1
2.           and   %g3, 1, %p1
3.          add(%p1)  %g3, %g2, %g4
3.          sub(~%p1) %g3, %g2, %g4
3.          ld    [%o1], %g2
3.          add   %o1, 4, %o1
3.          add   %g3, 1, %g3
4.       L: st    %g4, [%o0]
4.          add   %o0, 4, %o0
4.           and   %g3, 1, %p1
4.          cmp   %g3, 999
5.          add(%p1)  %g3, %g2, %g4
5.          sub(~%p1) %g3, %g2, %g4
5.          ld    [%o1], %g2
5.          add   %o1, 4, %o1
5.          ble   L
5.          add   %g3, 1, %g3
```

We now do need to be able to issue four ALU operations per cycle (since we issue both the add and subtract in the same cycle).

Automatic Instruction Selection

Besides register allocation and code scheduling, a code generator must also do *Instruction Selection*.

For CISC (Complex Instruction Set Computer) Architectures, like the Intel x86, DEC Vax, and many special purpose processors (like Digital Signal Processors), instruction selection is often *challenging* because so many choices exist.

In the Vax, for example, one, two and three address instructions exist. Each address may be a register, memory location (with or without indexing), or an immediate operand.

For RISC (Reduced Instruction Set Computer) Processors, instruction formats and addressing modes are far more limited.

Still, it is necessary to handle immediate operands, commutative operands and special case null operands (add of 0 or multiply of 1).

Moreover, automatic instruction selection supports *automatic retargeting* of a compiler to a new or extended instruction set.

Tree-Structured Intermediate Representations

For purposes of automatic code generation, it is convenient to translate a source program into a *Low-level, Tree-Structured IR*.

This representation exposes translation details (how locals are accessed, how conditionals are translated, etc.) without assuming a particular instruction set.

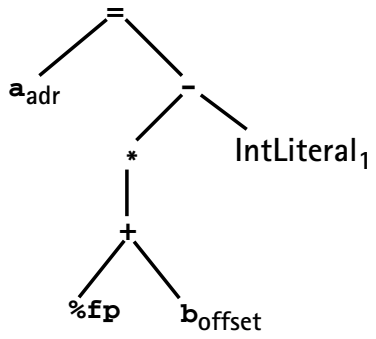
In a low-level, tree-structured IR, leaves are registers or bit-patterns and internal nodes are machine-level primitives, like load, store, add, etc.

Example

Let's look at how

```
a = b - 1;
```

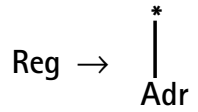
is represented, where `a` is a global integer variable and `b` is a local (frame allocated) integer variable.



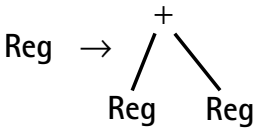
Representation of Instructions

Individual instructions can be represented as trees, rooted by the operation they implement.

For example:

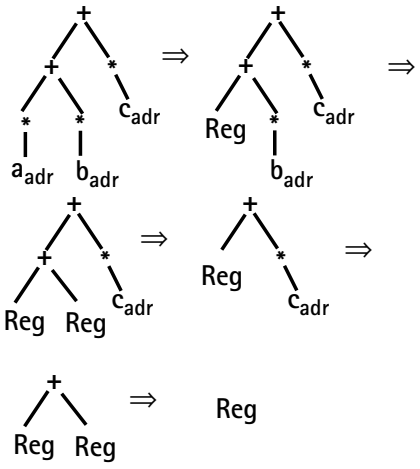


This is an instruction that loads a register with the value at an absolute address.



This is an instruction that adds the contents of two registers and stores the sum into a third register.

Using the above pair of instruction definitions, we can repeatedly match instructions in the following program IR:



Each match of an instruction pattern can have the side-effect of generating an instruction:

```
ld   [a],%R1
ld   [b],%R2
add  %R1,%R2,%R3
ld   [c],%R4
add  %R3,%R4,%R5
```

Registers can be allocated on-the-fly as Instructions are generated or instructions can be generated using pseudo-registers, with a subsequent register allocation phase.

Using this view of instruction selection, choosing instructions involves finding a *cover* for an IR tree using Instruction Patterns.

Any cover is a *valid* translation.

Tree Parsing vs. String Parsing

This process of selecting instructions by matching instruction patterns is very similar to how strings are parsed using Context-free Grammars.

We repeatedly identify a sub-tree that corresponds to an instruction, and simplify the IR-tree by replacing the instruction sub-tree with a nonterminal symbol. The process is repeated until the IR-tree is reduced to a single nonterminal.

The theory of reducing an IR-tree using rewrite rules has been studied as part of BURS (Bottom-Up Rewrite Systems) Theory by Pelegri-Llopart and Graham.

Automatic Instruction Selection Tools

Just as tools like Yacc and Bison automatically generate a string parser from a specification of a Context-free Grammar, there exist tools that will automatically generate a tree-parser from a specification of tree productions.

Two such tools are BURG (Bottom Up Rewrite Generator) and IBURG (Interpreted BURG). Both automatically generate parsers for tree grammars using BURS theory.

Least-Cost Tree Parsing

BURG (and IBURG) *guarantee* to find a cover for an input tree (if one exists).

But tree grammars are usually *very* ambiguous.

Why?—Because there is usually more than one code sequence that can correctly implement a given IR-tree.

To deal with ambiguity, BURG and IBURG allow each instruction pattern (tree production) to have a *cost*.

This cost is typically the size or execution time for the corresponding target-machine instructions.

Using costs, BURG (and IBURG) not only guarantee to find a cover, but also a *least-cost cover*.

This means that when a generated tree-parser is used to cover (and thereby translate) an IR-Tree, the *best possible* code sequence is guaranteed.

If more than one least-cost cover exists, an arbitrary choice is made.