
Compiler Optimizations for Modern VLIW/EPIC Architectures

Benjamin Goldberg
New York University

Introduction

New architectures have hardware features for supporting a range of compiler optimizations

- we'll concentrate on VLIW/EPIC architectures
 - Intel IA64 (Itanium), HP Lab's HPL-PD
 - Also several processors for embedded systems
 - e.g. Sharc DSP processor
- Optimizations include software pipelining, speculative execution, explicit cache management, advanced instruction scheduling

VLIW/EPIC Architectures

Very Long Instruction Word (VLIW)

- processor can initiate multiple operations per cycle

<code>r1 = L r4</code>	<code>r2 = Add r1,M</code>	<code>f1 = Mul f1,f2</code>	<code>r5 = Add r5,4</code>
------------------------	----------------------------	-----------------------------	----------------------------

- specified completely by the compiler (unlike superscalar machines)

Explicitly Parallel Instruction Computing (EPIC)

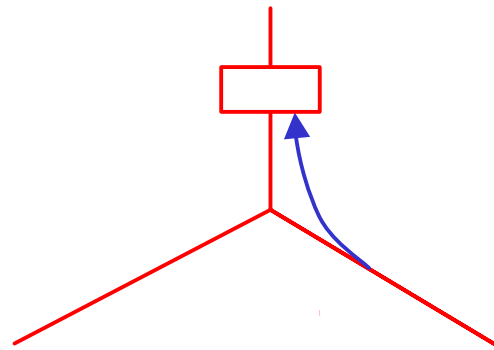
- VLIW + New Features
 - predication, rotating registers, speculations, etc.

This talk will use the instruction syntax of the HP Labs' HPL-PD. The features of the Intel IA-64 are similar.

Control Speculation Support

Control speculation is the execution of instructions that may not have been executed in unoptimized code.

- Generally occurs due to code motion across conditional branches



- these instructions are speculative
- safe if the effect of the speculative instruction can be ignored or undone if the other branch is taken
- What about exceptions?

Speculative Operations

Speculative operations are written identically to their non-speculative counterparts, but with an “E” appended to the operation name.

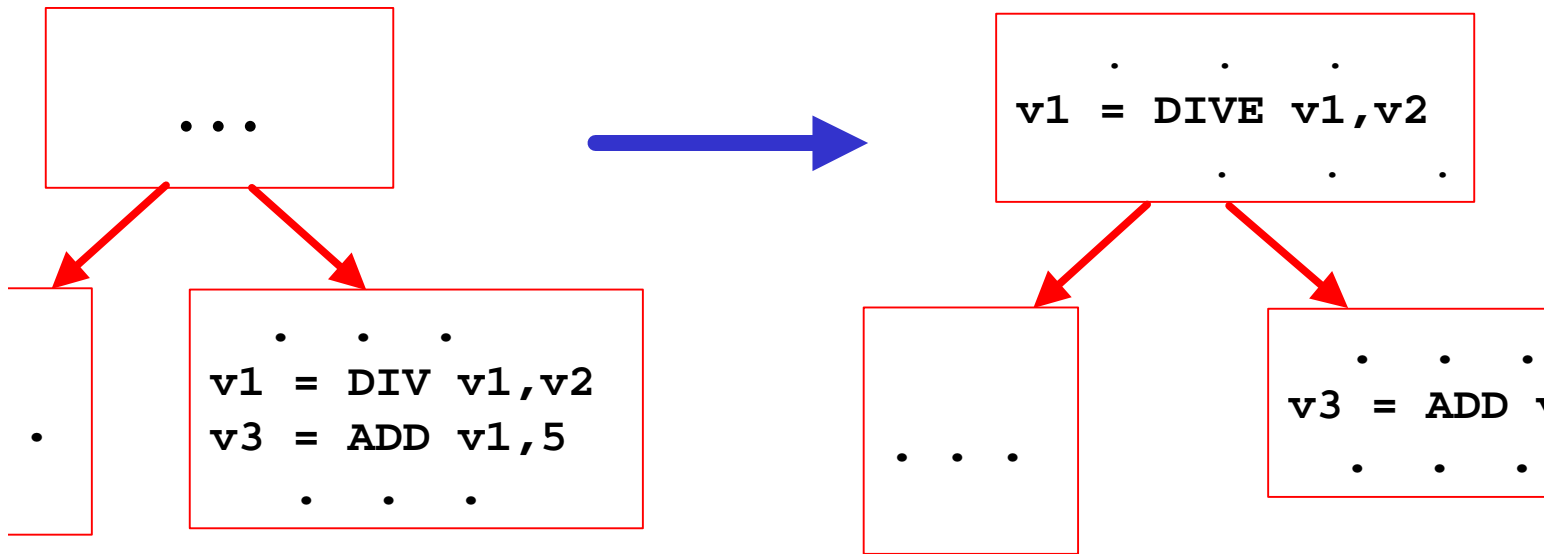
- e.g. **DIVE ADDE PBRRE**

If an exceptional condition occurs during a speculative operation, the exception is not raised.

- A bit is set in the result register to indicate that such a condition occurred.
- Speculative bits are simply propagated by speculative instructions
- When a non-speculative operation encounters a register with the speculative bit set, an exception is raised.

Speculative Operations (example)

Here is an optimization that uses speculative instructions:



- The effect of the DIV latency is reduced.
- If a divide-by-zero occurs, an exception will be raised by ADD.

Predication in HPL-PD

In HPL-PD, most operations can be predicated

- they can have an extra operand that is a one-bit predicate register.

r2 = ADD r1,r3 if p2

- If the predicate register contains 0, the operation is not performed
- The values of predicate registers are typically set by “compare-to-predicate” operations

p1 = CMPP<= r4,r5

Uses of Predication

Predication, in its simplest form, is used with

- if-conversion

A use of predication is to aid code motion by instruction scheduler.

- e.g. hyperblocks

With more complex compare-to-predicate operations, we get

- height reduction of control dependences

If-conversion

If-conversion replaces conditional branches with predicated operations.

For example, the code generated for:

```
if (a < b)
  c = a;
else
  c = b;
if (d < e)
  f = d;
else
  f = e;
```

might be the two EPIC instructions:

CMPP.< a,b	P2 = CMPP.>= a,b	P3 = CMPP.< d,e	P4 = CMPP.>
a if p1	c = b if p2	f = d if p3	f = e if p4

Compare-to-predicate instructions

In previous slide, there were two pairs of almost identical instructions

- just computing complement of each other

HPL-PD provides two-output CMPP instructions

- $p1, p2 = \text{CMPP.W.<.UN.UC } r1, r2$ 

- U means unconditional, N means normal, C means complement
- There are other possibilities (conditional, or, and)

If-conversion, revisited

Thus, using two-output CMPP instructions, the code generated for:

```
if (a < b)
  c = a;
else
  c = b;
if (d < e)
  f = d;
else
  f = e;
```

Only two CMPP operations, occupying less of the EPIC instruction.

might be instead be:

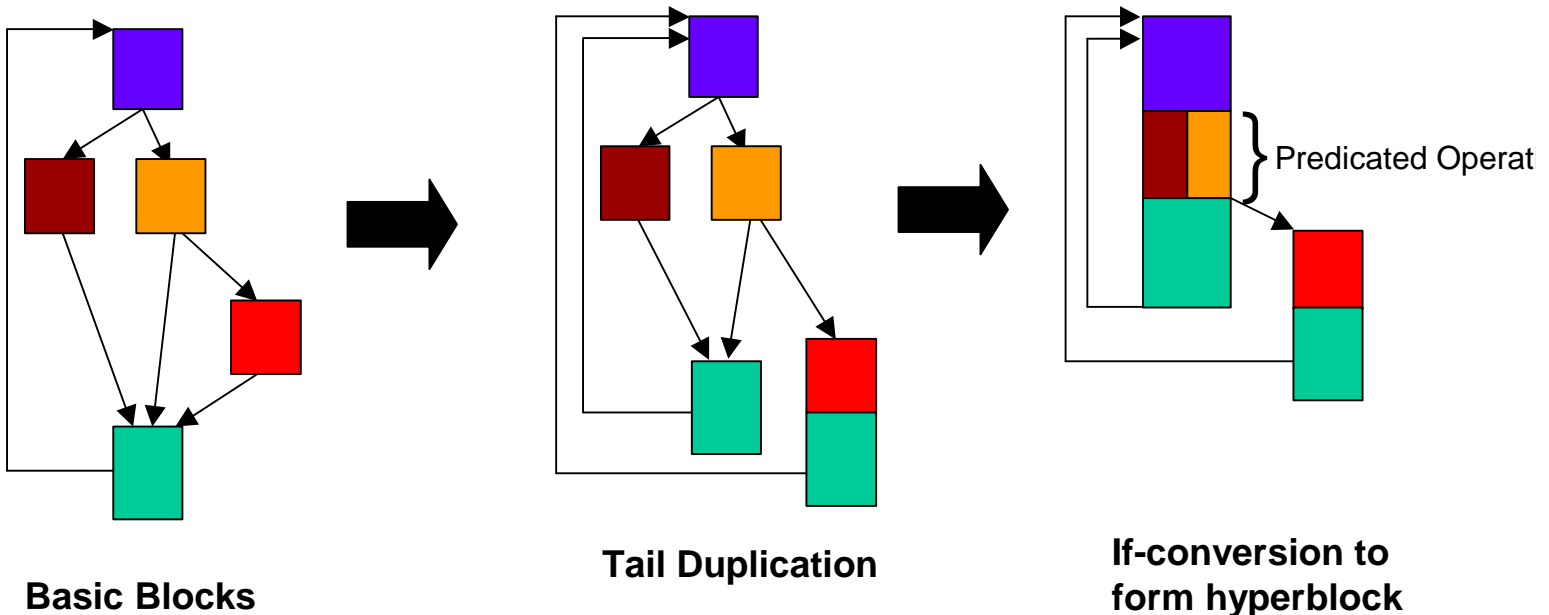
<code>,p2 = CMPP.W.<.UN.UC a,b</code>	<code>p3,p4 = CMPP.W.<.UN.UC d,e</code>
--	--

<code>= a</code>	<code>if p1</code>	<code>c = b</code>	<code>if p2</code>	<code>f = d</code>	<code>if p3</code>	<code>f = e</code>	<code>if p4</code>
------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------	--------------------

Hyperblock Formation

In hyperblock formation, if-conversion is used to form larger blocks of operations than the usual basic blocks

- tail duplication used to remove some incoming edges in middle of block
- if-conversion applied after tail duplication
- larger blocks provide a greater opportunity for code motion to increase ILP.



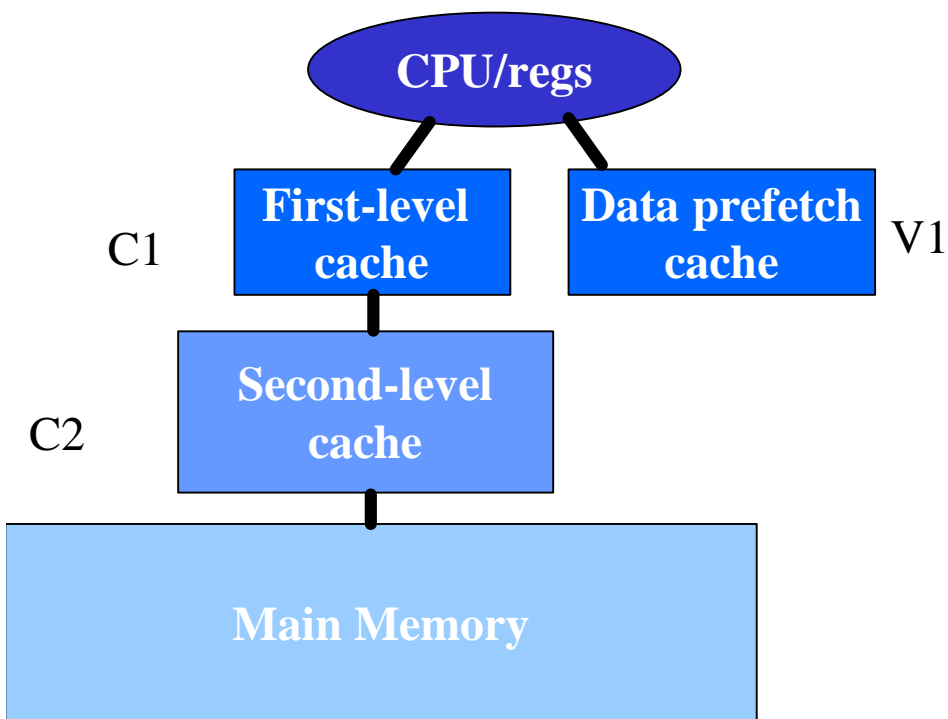
The HPL-PD Memory Hierarchy

IPL-PD's memory hierarchy is unusual in that it is visible to the compiler.

- In store instructions, compiler can specify in which cache the data should be placed.
- In load instructions, the compiler can specify in which cache the data is expected to be found and in which cache the data should be left.

This supports static scheduling of load/store operations with reasonable expectations that the assumed latencies will be correct.

Memory Hierarchy



data-prefetch cache

- Independent of the first-level cache
- Used to store large amounts of cache-polluting data
- Doesn't require sophisticated cache-replacement mechanism

Load/Store Instructions

Load Instruction:

$r1 = L.W.C2.V1 r2$

Source Cache

Target Cache

Operand register
(contains address)

Store Instruction:

$S.W.C1 r2, r3$

Target Cache

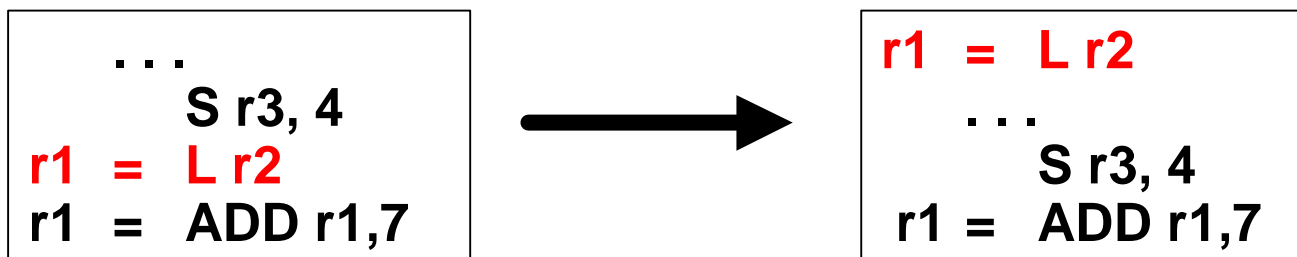
value to be stored

Contains address

What if source cache specifier is wrong?

Run-time Memory Disambiguation

Here's a desirable optimization (due to long load latencies):



However, this optimization is not valid if the load and store reference the same location

- i.e. if r2 and r3 contain the same address.
- this cannot be determined at compile time

IPL-PD solves this by providing *run-time memory disambiguation*.

Run-time Memory Disambiguation (cont)

IPL-PD provides two special instructions that can replace a single load instruction:

r1 = LDS r2 ; speculative load

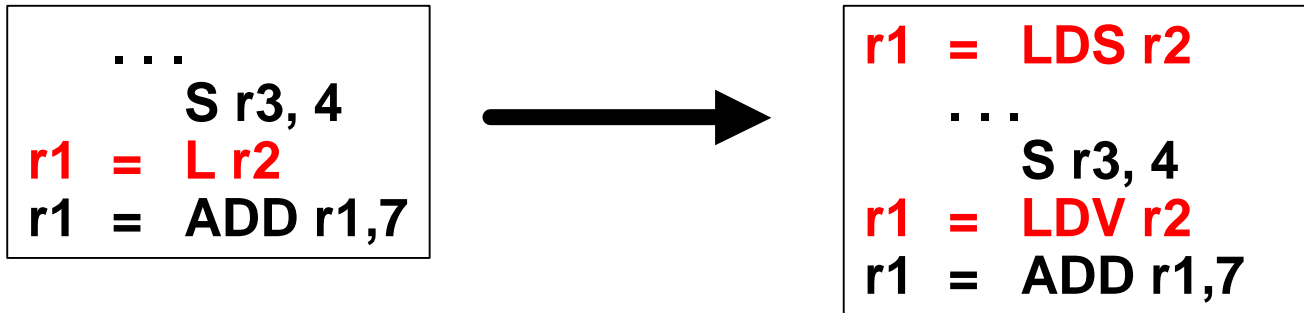
- initiates a load like a normal load instruction. A log entry can be made in a table to store the memory location

r1 = LDV r2 ; load verify

- checks to see if a store to the memory location has occurred since the LDS.
- if so, the new load is issued and the pipeline stalls. Otherwise, it's a no-op.

Run-time Memory Disambiguation (cont)

The previous optimization becomes



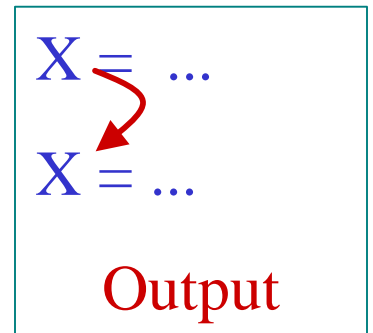
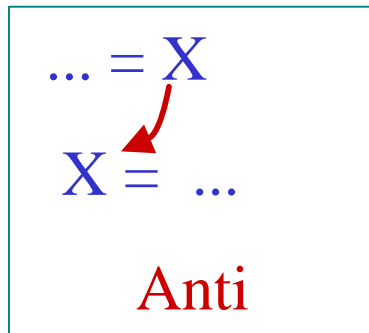
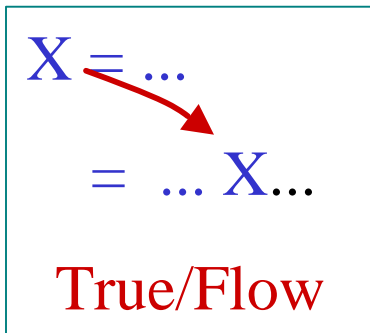
There is also a BRDV (branch-on-data-verify) for branching to compensation code if a store has occurred since the LDS to the same memory location.

Dependence Analysis

Foundation of instruction reordering optimizations, including software pipelining, loop optimizations, parallelization.

Determines if the relative order of two operations in the original (sequential) program must be preserved in the optimized version.

Three types of dependence:



Dependence Analysis (cont)


Dependences can be loop independent

- dependence is either not within a loop or is within the same iteration of a loop

or loop carried

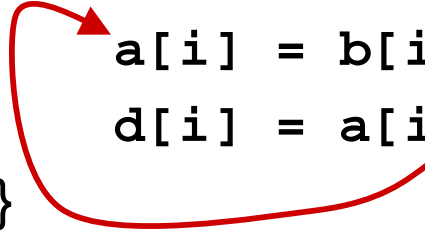
- dependence spans multiple iterations of a loop

```
or(i=0;i<n;i++) {  
    a[i] = b[i] + c;  
    d[i] = a[i] * 2;  
}
```



Loop Independent

```
for(i=0;i<n;i++) {  
    a[i] = b[i] + c;  
    d[i] = a[i+1] * 2;  
}
```

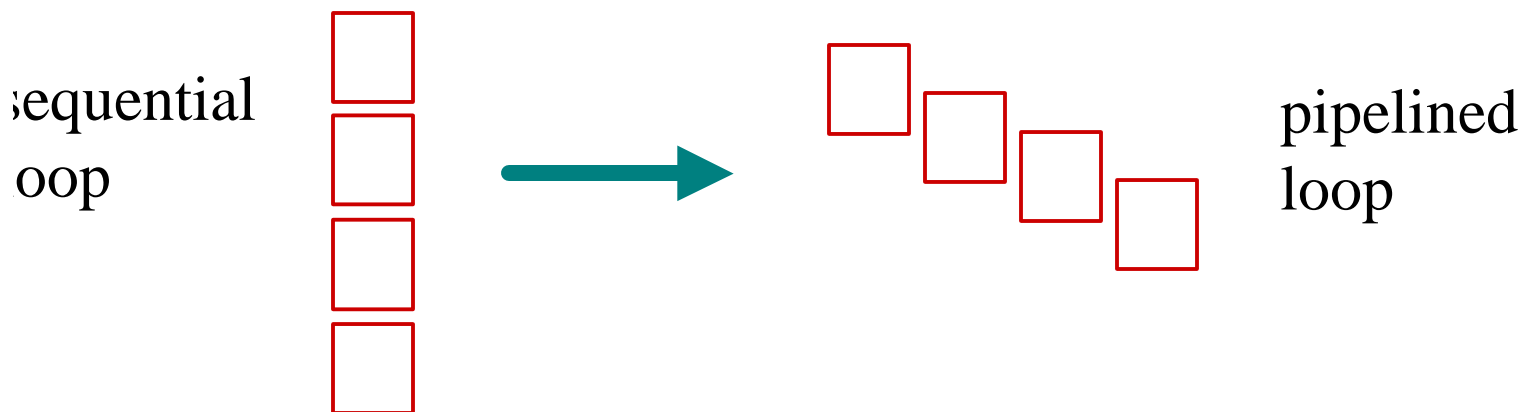


Loop Carried

Software Pipelining

Software Pipelining is the technique of scheduling instructions across several iterations of a loop.

- reduces pipeline stalls on sequential pipelined machine
- exploits instruction level parallelism on superscalar and VLIW machines
- intuitively, iterations are overlaid so that an iteration starts before the previous iteration have completed



Software Pipelining Example

Source code:

```
for(i=0;i<n;i++) sum += a[i]
```

Loop body in assembly:

```
r1 = L r0  
--- ;stall  
r2 = Add r2,r1  
r0 = add r0,4
```

Unroll loop &
allocate registers



```
r1 = L r0  
--- ;stall  
r2 = Add r2,r1  
r0 = Add r0,1  
r4 = L r3  
--- ;stall  
r2 = Add r2,r1  
r3 = add r3,1  
r7 = L r6  
--- ;stall  
r2 = Add r2,r1  
r6 = add r6,1  
r10 = L r9  
--- ;stall  
r2 = Add r2,r1  
r9 = add r9,1
```

Software Pipelining Example (cont)

schedule Unrolled Instructions, exploiting VLIW (or not)

1 = L r0

4 = L r3

2 = Add r2,r1 r7 = L r6

0 = Add r0,12 r2 = Add r2,r4 r10 = L r9

3 = add r3,12 r2 = Add r2,r7 r1 = L r0

6 = add r6,12 r2 = Add r2,r10 r4 = L r3

9 = add r9,12 r2 = Add r2,r1 r7 = L r6

0 = Add r0,12 r2 = Add r2,r4 r10 = L r9

3 = add r3,12 r2 = Add r2,r7 r1 = L r0

6 = add r6,12 r2 = Add r2,r10 r4 = L r3

9 = add r9,12 r2 = Add r2,r1 r7 = L r6

. . .

0 = Add r0,12 r2 = Add r2,r4 r10 = L r9

3 = add r3,12 r2 = Add r2,r7

6 = add r6,12 Add r2,r10

9 = add r9,12

Identif
repeati
patter
(kerne

Software Pipelining Example (cont)

loop becomes:

```
1 = L r0
4 = L r3
2 = Add r2,r1 r7 = L r6
```

← prolog

```
0 = Add r0,12 r2 = Add r2,r4 r10 = L r9
3 = Add r3,12 r2 = Add r2,r7 r1 = L r0
6 = Add r6,12 r2 = Add r2,r10 r4 = L r3
9 = Add r9,12 r2 = Add r2,r1 r7 = L r6
```

← kernel

```
0 = Add r0,12 r2 = Add r2,r4 r10 = L r9
3 = Add r3,12 r2 = Add r2,r7
6 = Add r6,12 Add r2,r10
9 = Add r9,12
```

← epilog

Register Usage in Software Pipelining

In the previous example, the kernel contained many instructions

- due to replication of the original loop body for register allocation
- this can have an adverse impact on instruction cache performance

The HPL-PD and IA64 support rotating registers to reduce the code size of the kernel

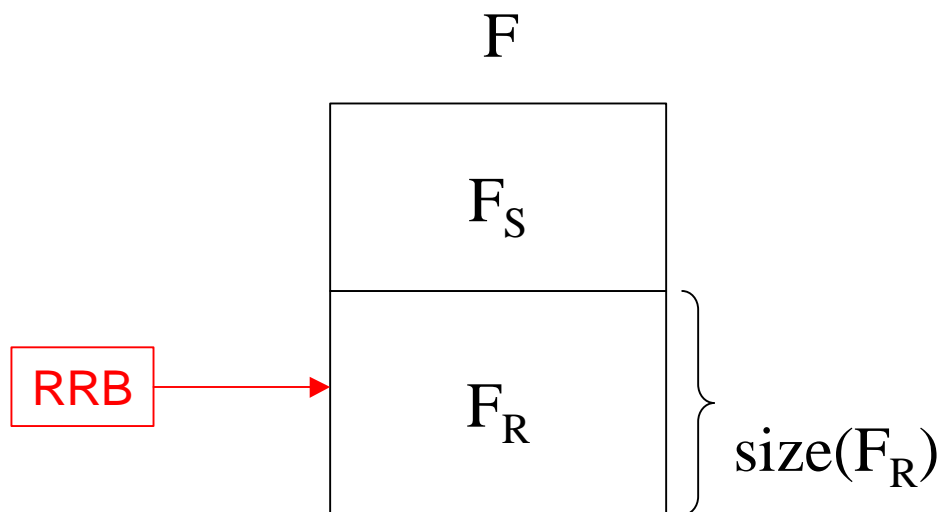
Rotating Registers

Each register file may have a static and a rotating portion

In HPL-PD, the i th static register in file F is named F_i

The i th rotating register in file F is named $F[i]$.

- Indexed off the RRB, the rotating register base register.



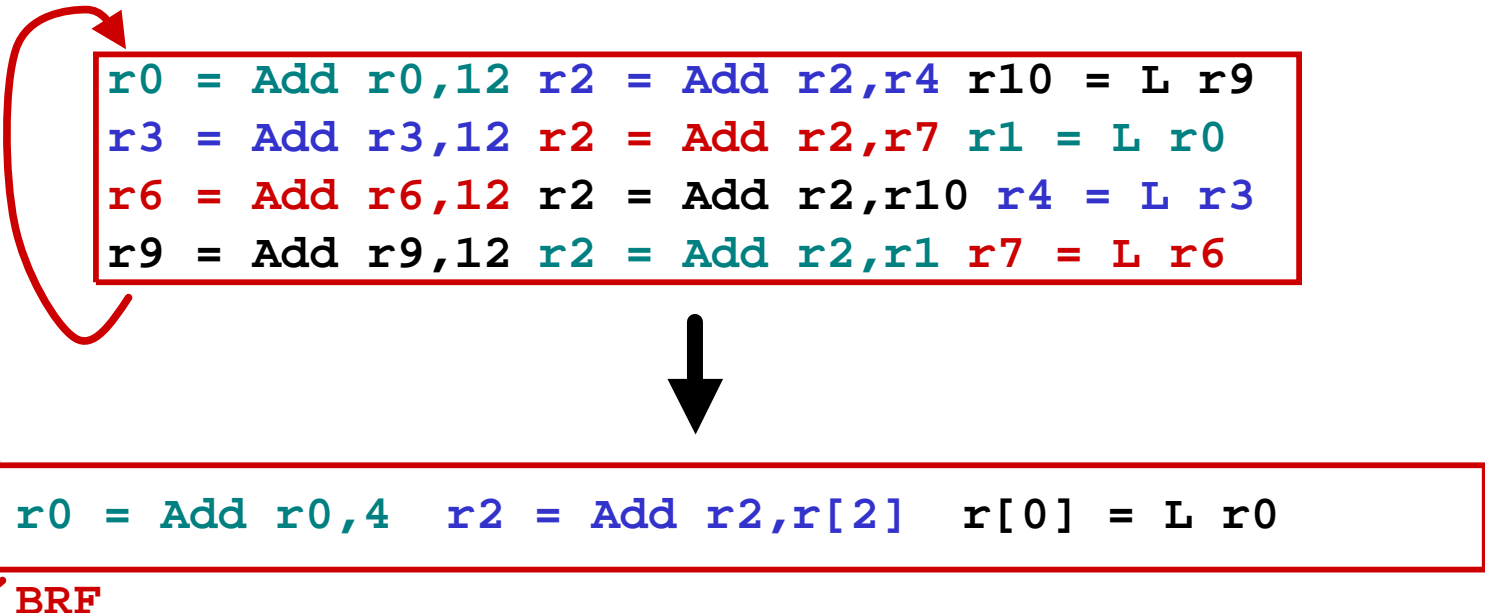
$$F[i] ? FR [(RRB + i) \% size(FR)]$$

Rotating Registers (cont)

In HPL-PD, there are branch instructions, e.g. **BRF**, that decrement the RRB

After the **BRF** instruction, the register that was referred to as **r[i]** is now referred to as **r[i+1]**

Note how the kernel can be transformed:



```
r0 = Add r0,12 r2 = Add r2,r4 r10 = L r9
r3 = Add r3,12 r2 = Add r2,r7 r1 = L r0
r6 = Add r6,12 r2 = Add r2,r10 r4 = L r3
r9 = Add r9,12 r2 = Add r2,r1 r7 = L r6
```

```
r0 = Add r0,4 r2 = Add r2,r[2] r[0] = L r0
```

BRF

Rotating Predicate Registers

There are also rotating predicate registers

- referred to as p[0], p[1], etc.

BRF causes them to rotate

- after BRF, p[1] has the value that p[0] had

Thirty-two predicate registers can be used as a 32-bit aggregate register

```
r1 = mov 110110110b
```

```
PR = mov r1
```



32-bit register consisting
of 32 1-bit predicate registers

Constraints on Software Pipelining

The instruction-level parallelism in a software pipeline is limited by

- Resource Constraints
 - VLIW instruction width, functional units, bus conflicts, etc.
- Dependence Constraints
 - particularly loop carried dependences between iterations
 - arise when
 - the same register is used across several iterations
 - the same memory location is used across several iterations



Memory Aliasing

Aliasing-based Loop Dependences

Source code:

```
for(i=2; i<n;i++)
    a[i] = a[i-3] + c;
```

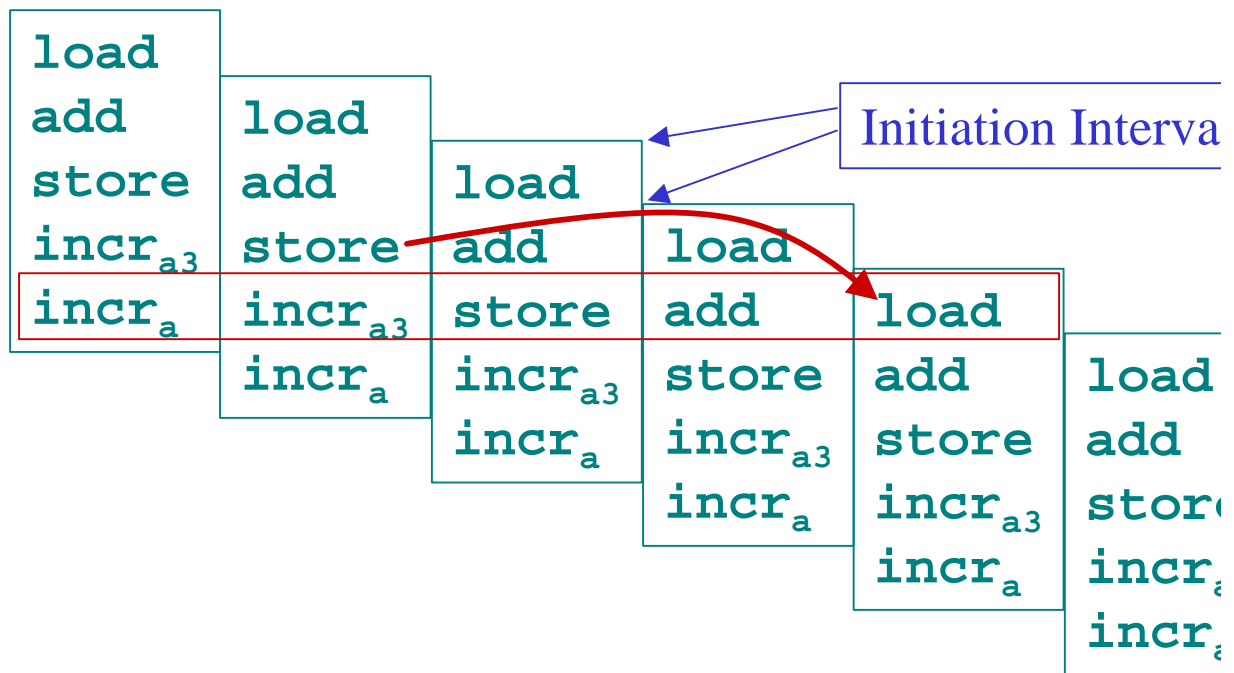
• Assembly:

```
loada
add
store
incra3
incra
```

dependence spans
3 iterations
distance = 3"

pipeline

kernel
1 cycle



Aliasing-based Loop Dependences

Source code:

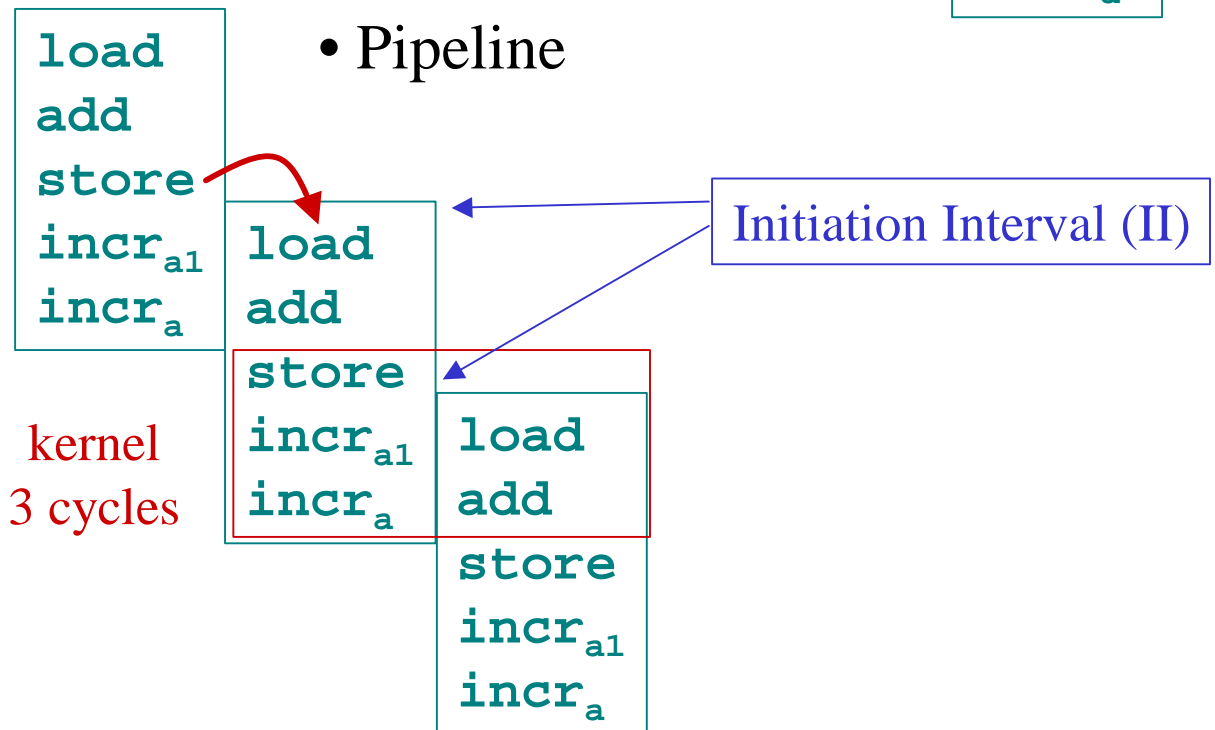
```
for(i=2; i<n;i++)  
  a[i] = a[i-1] + c;
```

ance = 1

• Assembly:

```
loada  
add  
store  
incra1  
incra
```

• Pipeline



Dynamic Memory Aliasing

What if the code were:

```
for(i=k;i<n;i++)  
    a[i] = a[i-k] + c;
```

where k is unknown at compile time?

- the dependence distance is the value of k
 - “dynamic” aliasing

The possibilities are:

- $k = 0$ no loop carried dependence
- $k > 0$ loop carried true dependence with distance k
- $k < 0$ loop carried anti-dependence with distance $|k|$

The worst case is $k = 1$ (as on previous slide)

The compiler has to assume the worst, and generate the most pessimistic pipelined schedule

Pipelining Despite Aliasing

This situation arises quite frequently:

```
void copy(char *a, char *b, int size)
{ for(int i=0;i<n;i++)
  a[i] = b[i];
}
```

- Distance = (b - a)

What can the compiler do?

- Generate different versions of the software pipeline for different distances
 - branch to the appropriate version at run-time
 - possible code explosion, cost of branch

Another alternative: Software Bubbling

- a new technique for Software Pipelining in the presence of dynamic aliasing

Software Bubbling

Compiler generates the most optimistic pipeline

- constrained only by resource constraints
 - perhaps also by static dependences in the loop

All operations in the pipeline kernel are predicated

- rotating predicate registers are especially useful, but not necessary

The predication pattern determines if the operations in a given iteration “slot” are executed

The predication pattern is assigned dynamically, based on the dependence distance at run time.

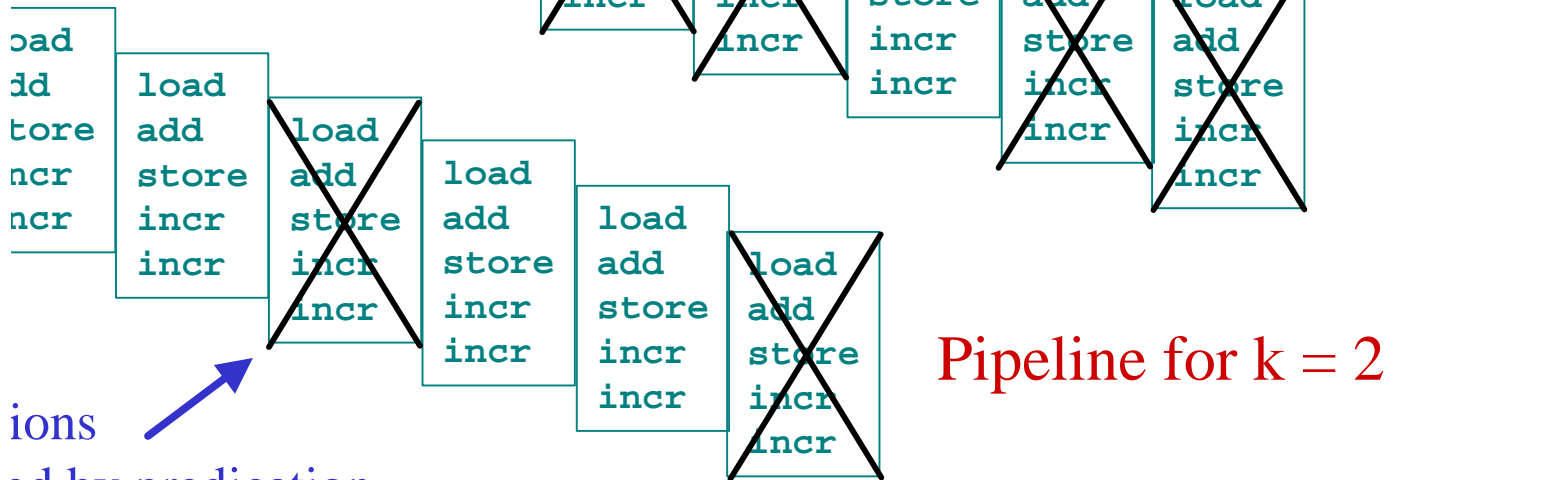
Continue to use simple example:

```
for(i=k;i<n;i++)a[i] = a[i-k] + c;
```

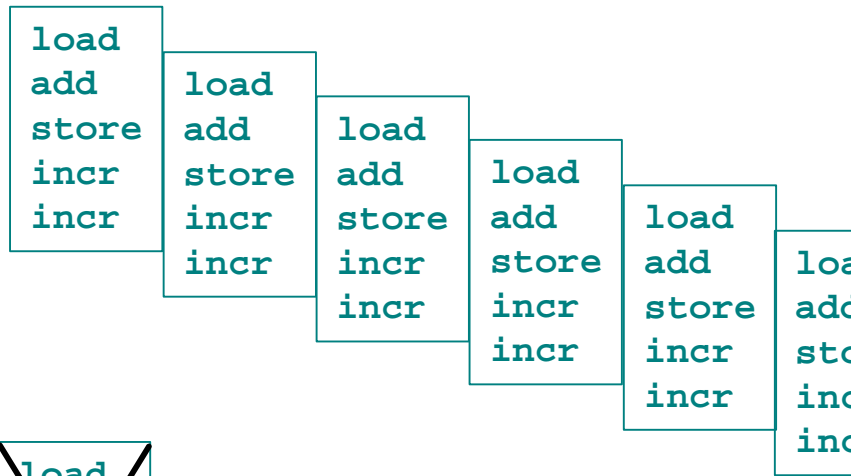
Software Bubbling

Optimistic Pipeline
for $k > 2$ or $k < 0$

Pipeline for $k = 1$



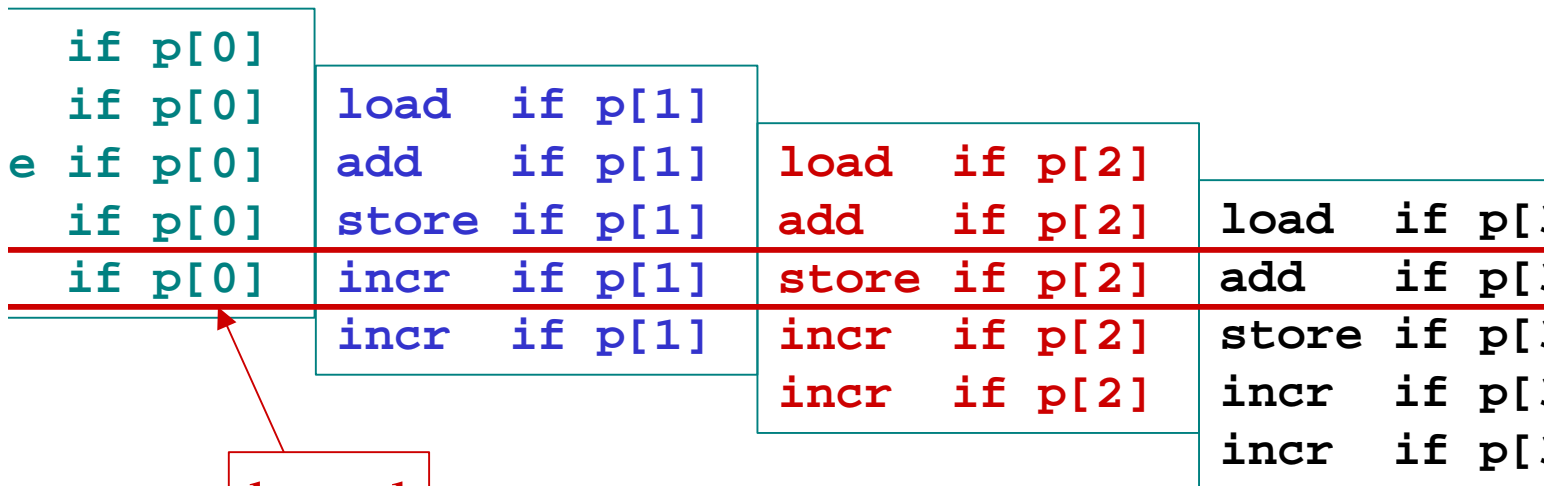
Pipeline for $k = 2$



The Predication Pattern

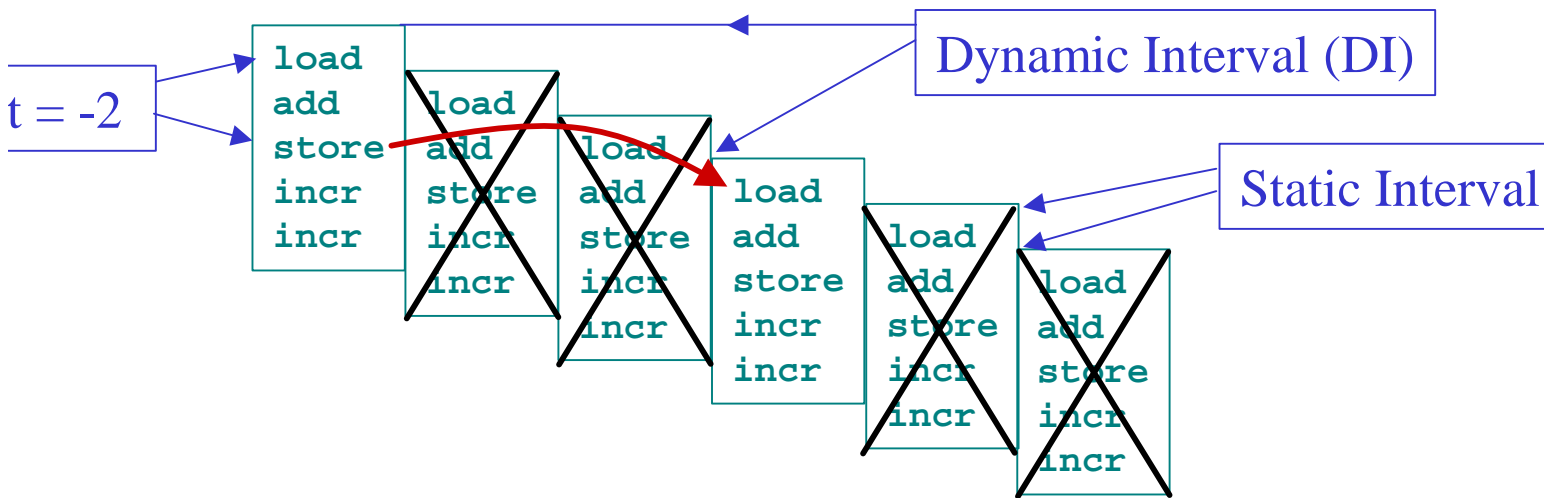
Each iteration slot is predicated upon a different predicate register

- all operations within the slot are predicated on the same predicate register



ad if p[0] add if p[1] store if p[2] incr if p[3] incr if p[3]

Computing the predication pattern



$$L = \lceil \text{latency}(\text{store}) - \text{offset}(\text{store}, \text{load}) \rceil / \Pi$$

= 3, the factor by which the Π would have to be increased, assuming the dependence spanned one iteration

$$\text{DI} = L/d * \Pi$$

= 3, where $d = 1$ is the dependence distance

The predication pattern should insure that only d out of L iterations slots are enabled. In this case, 1 out of 3 slots.

Computing the Predication Pattern (cont)

To enable d out of L iteration slots, we simply create a bit pattern of length L whose first d bits are 1 and the rest are 0.

$$= 2^d - 1.$$

Before entering the loop, we initialize the aggregate predicate register (PR) by executing

```
PR = shl 1, rd
```

```
PR = sub PR, 1
```

where r_d contains the value of d (run-time value)

The predicate register rotation occurs automatically using **BRF** and adding the instruction

```
p[0] = mov p[L]
```

within the loop, where L is a compile-time constant

Generalized Software Bubbling

So far, we've seen Simple Bubbling

- d is constant throughout the loop

If d changes as the loop progresses, then software bubbling can still be performed.

- The predication pattern changes as well
- This is called Generalized Bubbling
 - test occurs within the loop
 - iteration slot is only enabled if less than d iteration slots out of the previous L slots have been enabled.
- Examples of code requiring generalized bubbling appear quite often.
 - Alvin Spec Benchmark, Lawrence Livermore Loops Code

Experimental Results

Experiments were performed using the Trimaran Compiler Research Infrastructure

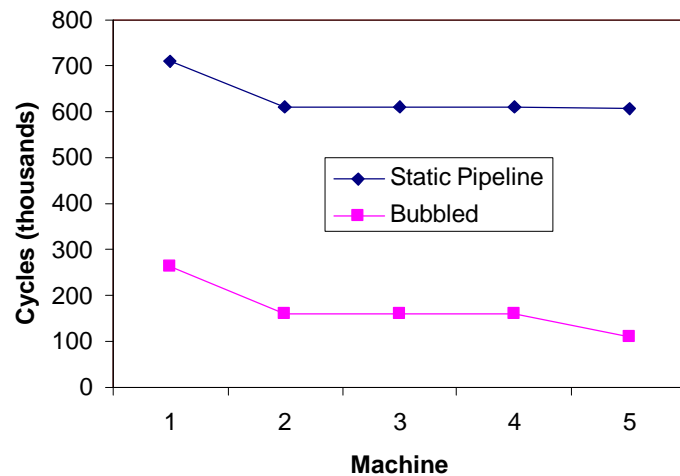
- www.trimaran.org
 - produced by a consortium of HP Labs, UIUC, NYU, and Georgia Tech
- Provides an highly optimizing EPIC compiler
- Configurable HPL-PD cycle-by-cycle simulator
- Visualization tools for displaying IR, performance, etc.

Benchmarks from the literature were identified as being amenable for software bubbling

Simple Bubbled Loops

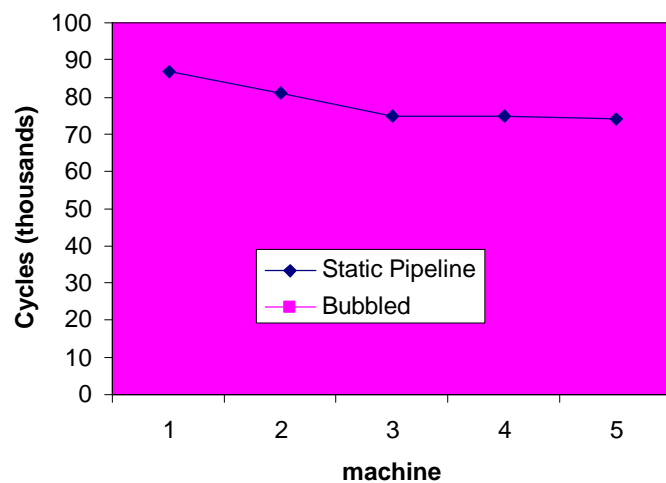
Illahan-Dongerra-Levine
52 Loop Benchmark

S152 Total Execution Time



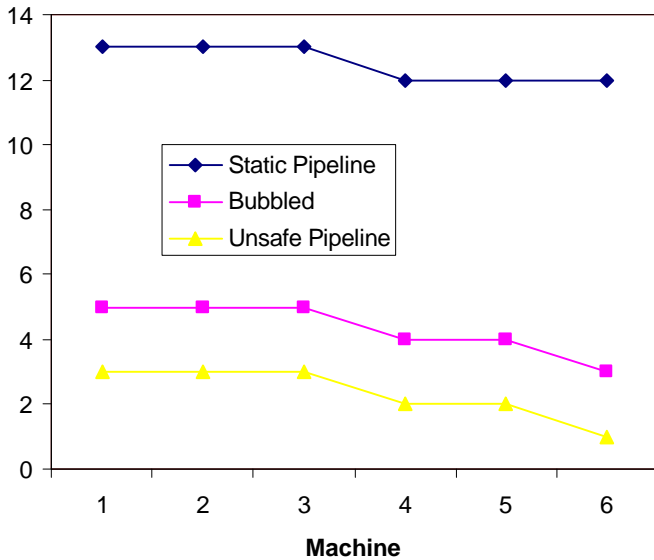
Matrix Addition

Matrix Add Total Execution Time

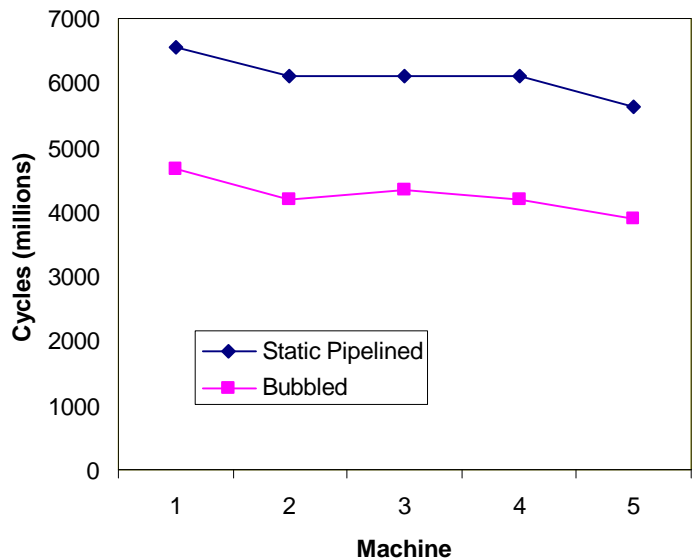


Generalized Bubbled Loops

Alvinn Cycles per Pipelined Loop



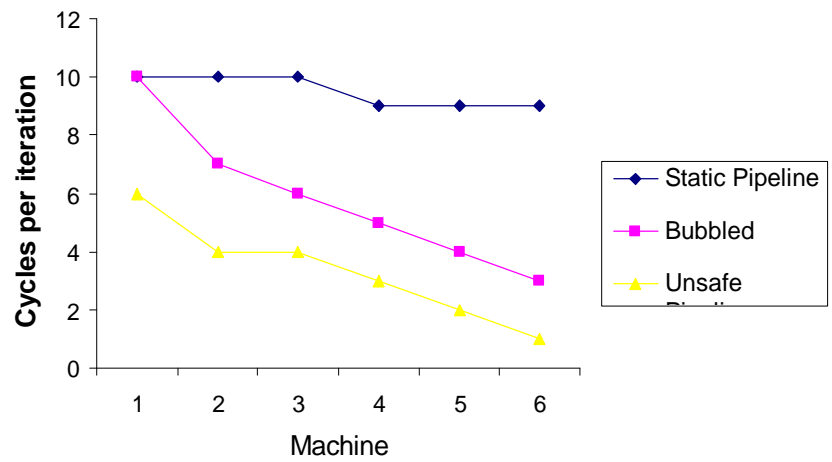
Alvinn Total Execution Time



Alvinn SPEC Benchmark

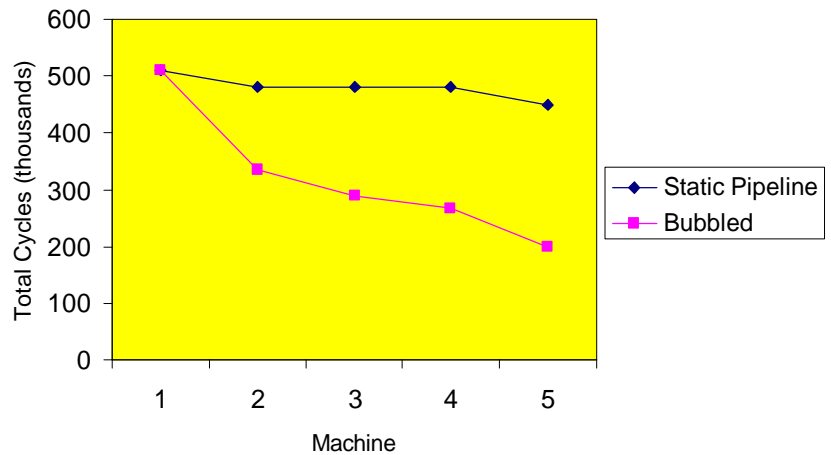
Generalized Bubbled Loops (cont)

Cycles per Loop



reference Livermore Loops
Kernel 2 Benchmark

Total Execution Time



Conclusions

Modern VLIW/EPIC architectures provide ample opportunity, and need, for sophisticated optimizations

Predication is a very powerful feature of these machines

Dynamic memory aliasing doesn't have to prevent optimization