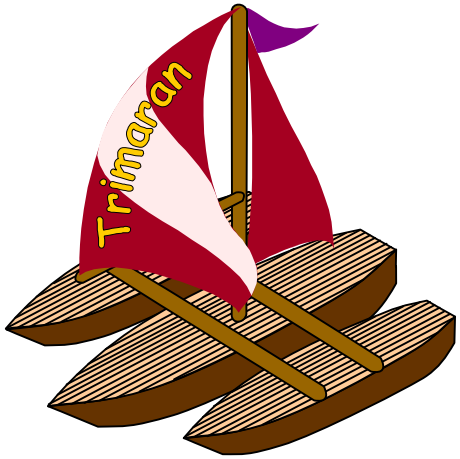


HPL-PD
A Parameterized Research Architecture

HPL-PD

HPL-PD is a parameterized ILP architecture

- It serves as a vehicle for processor architecture and compiler optimization research.
- It admits both EPIC and superscalar implementations
- The HPL-PD parameter space includes:
  - number and types of functional units
  - number and types of registers (in register files)
  - width of the instruction word (for EPIC)
  - instruction latencies

Novel Features of HPL-PD

HPL-PD has a number of interesting architectural features, including:

- Support for speculative execution
  - data speculation (run-time address disambiguation)
  - control speculation (eager execution)
- Predicated (guarded) execution
  - conditionally enable/disable instructions
- Memory system
  - compiler-visible cache hierarchy
  - serial behavior of parallel reads/writes

Novel Features (cont)

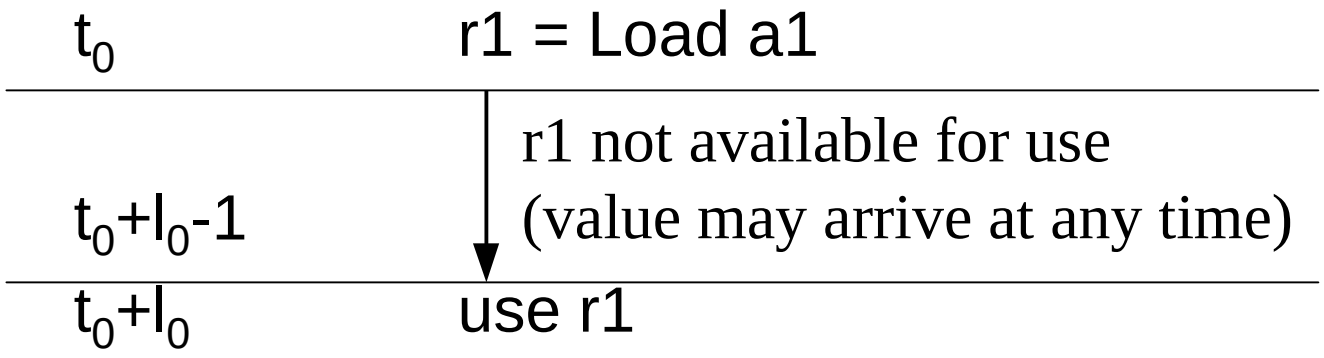
- Branch architecture
  - architecturally visible separation of fetch and execute of branch target
- Unusual simultaneous write semantics
  - hardware allows multiple simultaneous writes to registers
- Efficient boolean reduction support
  - parallel evaluation of multi-term conjunction/disjunction
- Software loop pipelining support
  - rotating registers for efficient software pipelining of tight inner loops
  - branch instructions with loop support (shifting the rotating register window, etc)

The Scheduling Model

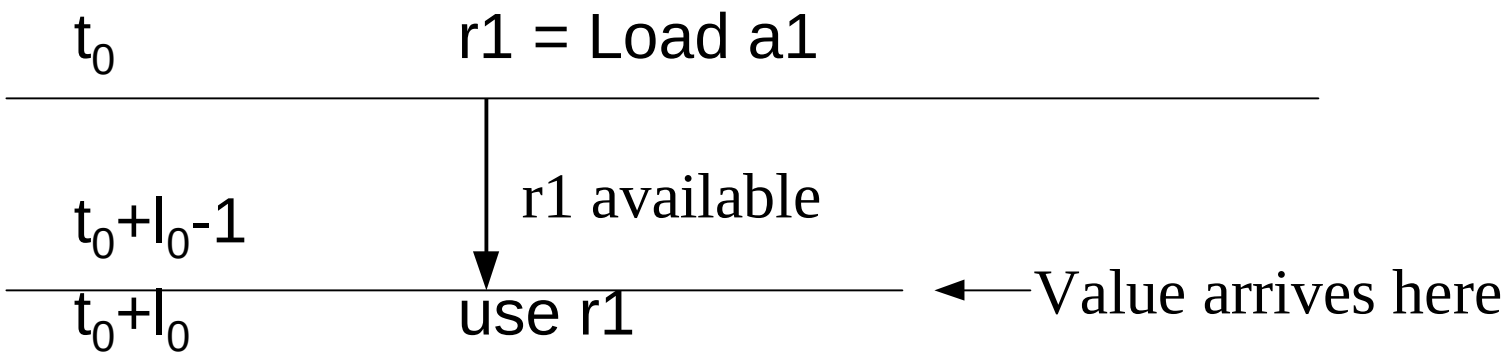
- HPL-PD supports either the Equals or Less Than or Equals model.
- Less than or Equals Model (LTE)
  - Destination register of an operation is reserved from the start of the operation until the value is delivered.
    - Model used by all current machines
- Equals model (EQ)
  - Value is delivered to the destination register at exactly the time determined by the instruction latency (according to the architectural specification).
  - Prior to that, the register can be used by other operations.
  - Reduces register pressure.

Scheduling Models

- Less Than or Equals (LTE) model



- Equals (EQ) model



LTE Vs EQ Model (example)

Before Register Allocation

```
v1  =   L v0          ; latency = 4
v2  =   DIV v3, 15    ; latency = 2
v4  =   ADD v5, 5     ; latency = 0
        S v4
        S v2
v1  =   ADD v1, 7
```

ADD result

DIV result

Load result

EQ Model

```
r1  =   L r0
r1  =   DIV r2, 15
r1  =   ADD r3, 5
        S r1
        S r1
r1  =   ADD r1, 7
```

LTE Model

```
r1  =   L r0
r2  =   DIV r3, 15
r4  =   ADD r3, 5
        S r4
        S r2
r1  =   ADD r1, 7
```

HPL-PD Register Files

The following classes of register files can currently be specified in the HPL-PD architecture

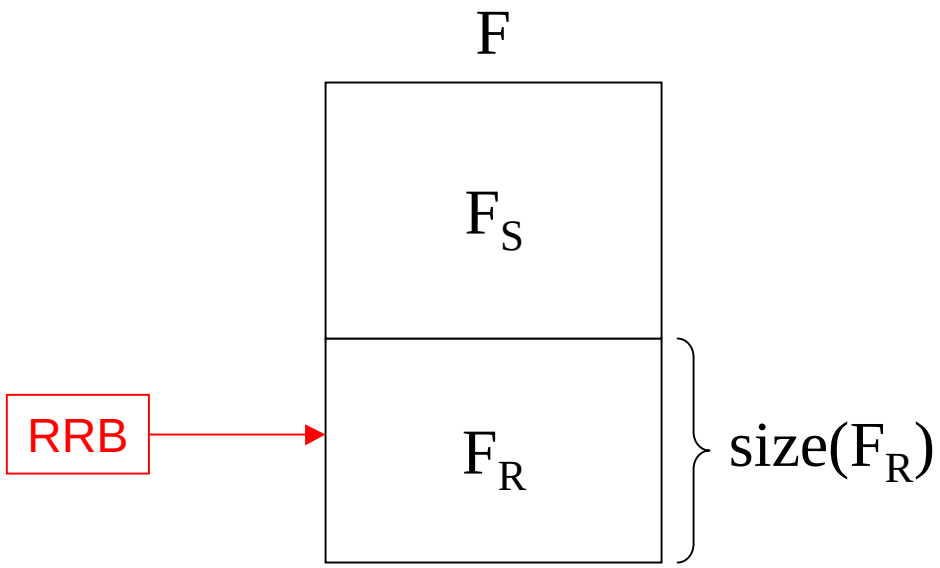
- General purpose (GPR)
  - 32 bits + 1-bit speculative tag
  - 32-bit signed and unsigned integers
- Floating point (FPR)
  - 64 bits + 1-bit speculative tag
  - IEEE compliant 32-bit single precision or 64-bit double precision floating point numbers
- Predicate (PR)
  - 1 bits + 1-bit speculative tag
  - 1-bit boolean values, used for predicated execution
- Branch target (BTR)
  - 32-bit address + 1-bit static prediction + 1-bit speculative tag

Register Files (cont)

- Control Registers (CR)
  - 32 bits
  - PC - program counter
  - PSW - program status word
  - RRB - rotating register base
  - LC - loop counter
  - ESC - epilog stage counter (for software pipelined loops)
  - PV(i,j) - 32 1-bit predicate register values (i=file, j=group of 32)
  - IT(i,j)
FT(i,j)
PT(i,j) - 32 1-bit speculative tags
(I=integer, F=floating point, P=predicate)
(i=file, j=group of 32)
  - BTRL(i,j)
BTRH(i,j) - high and low portions of a branch target register
(i=file, j=register number)

Register Files (cont)

- Each register file may have a static and a rotating portion
- The ith static register in file F is named Fi
- The ith rotating register in file F is named F[i].
  - Indexed off the RRB, the rotating register base register.

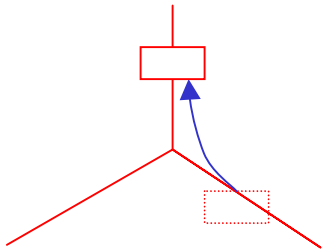


- $F[i] \equiv F_R[(RRB + i) \% size(F_R)]$

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
  - e.g. an instruction in one branch is moved above the conditional jump.



- these instructions are said to be speculative
- this transformation is generally safe if the effect of the speculative instruction can be ignored or undone if the other branch is taken
- However, if a speculative instruction causes an exception, the exception should not be raised if the other branch is taken.
  - HPL-PD provides hardware support for this.

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.
  - More information (e.g. type of condition, IP of instruction) is stored.
    - not currently specified how or where.

Speculative Operations (cont)

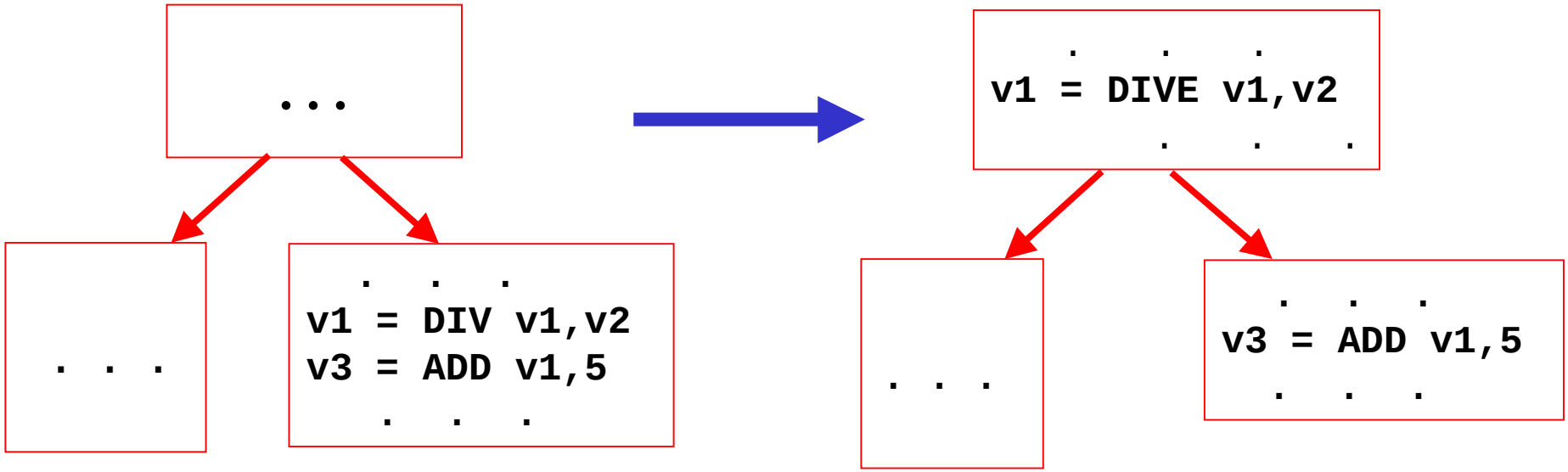
The behavior of speculative operations is as follows:

- if none of the operand registers have their speculative bits set, the operation proceeds normally. If an exceptional condition occurs, the speculative bit in the result register is set, but no exception is raised.
- if the speculative bit of any operand register is set, then the operation simply sets the speculative bit of the result register.

If a non-speculative operation has an operand with its speculative bit set, or if an exceptional condition occurs during the operation, 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.W 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.

- Those instructions in branches not taken are disabled by predication.

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:

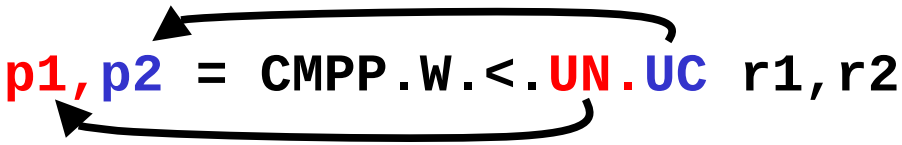
P1 = CMPP.< a,b	P2 = CMPP.>= a,b	P3 = CMPP.< d,e	P4 = CMPP.>= d,e
c = a if p1	c = b if p2	F = d if p3	F = e if p4

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



- 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;
```

might be instead be:

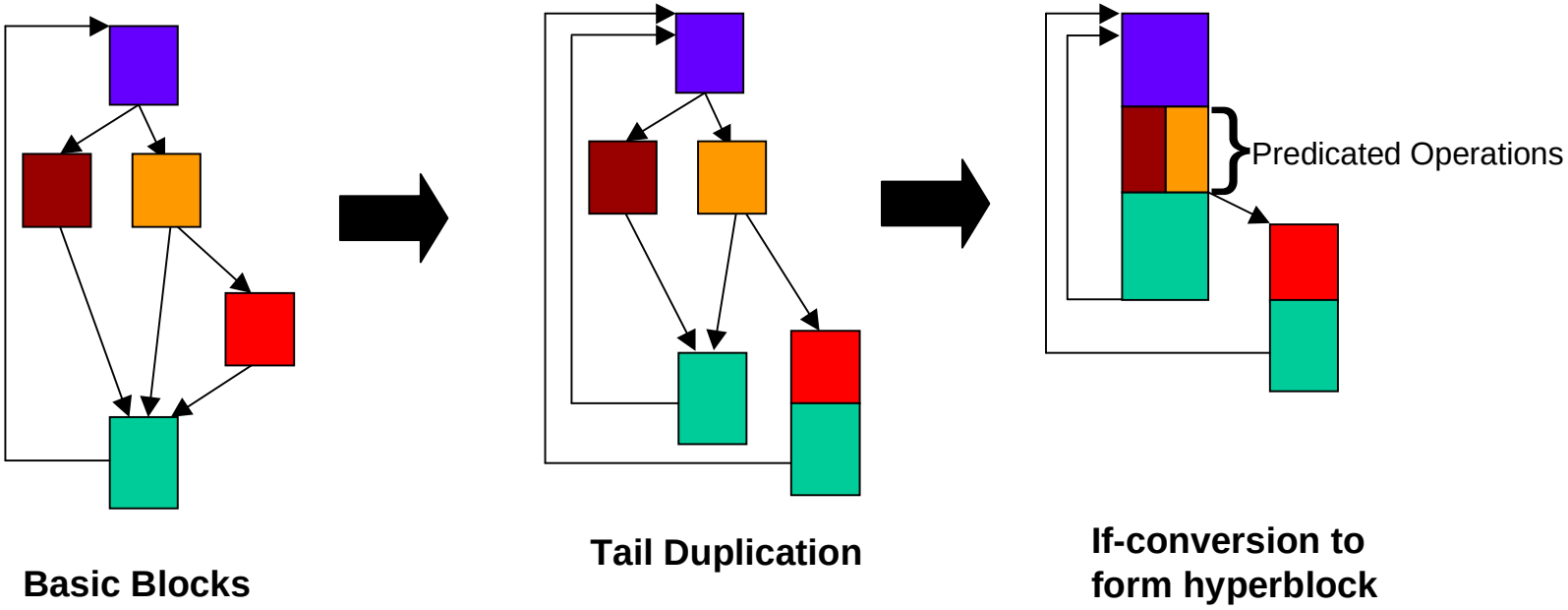
Only two CMPP operations, occupying less of the EPIC instruction.

`p1,p2 = CMPP.W.<.UN.UC a,b`	`p3,p4 = CMPP.W.<.UN.UC d,e`

`c = a if p1`	`c = b if p2`	`F = d if p3`	`F = e if p4`

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.

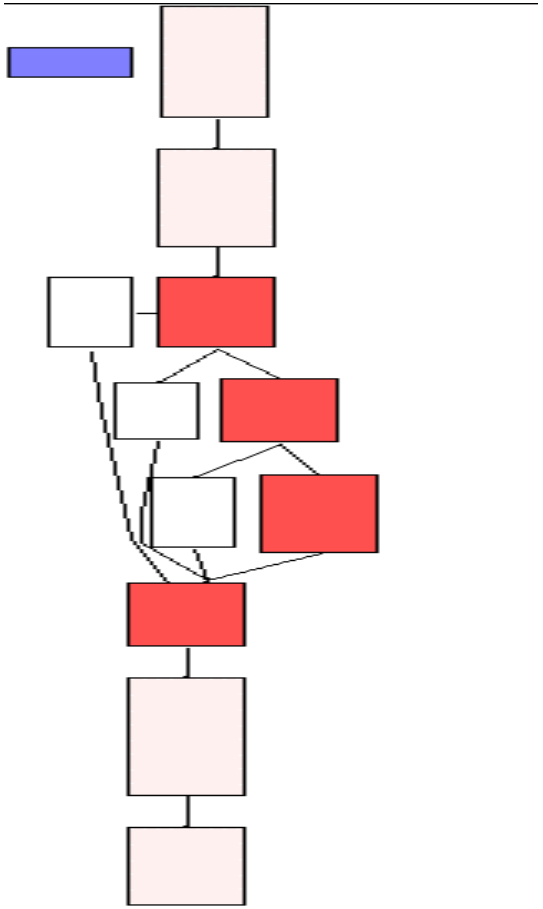


Hyperblock Example (hyper.c)

```
int main()
{
  int i, a, b, c;
  a = b = c = 0;
  for (i=0; i<200; i++)
  {
     a+=1;
     if (i%10==0) continue;
     b+=2;
     if (i%10==5) continue;
     c+=3;
  }
  printf ("a:%d b:%d c:%d\n", a, b, c);
  exit (0);
}
```

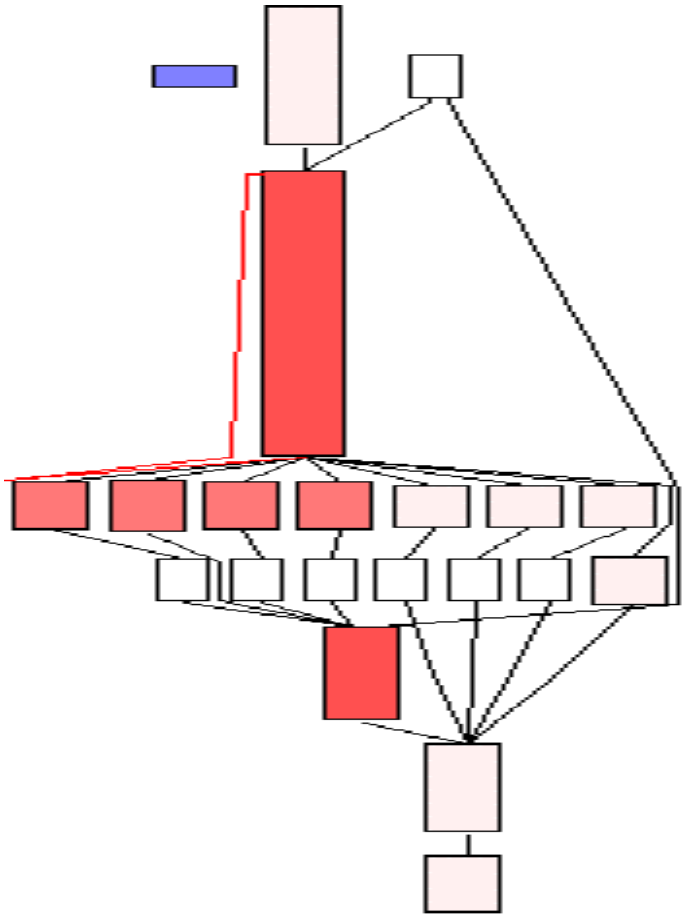
Hyper.c Control Flow (w/o Hyperblocks)

Image generated by Trimaran GUI



Hyper.c Control Flow (w/ Hyperblocks)

Image generated by Trimaran GUI



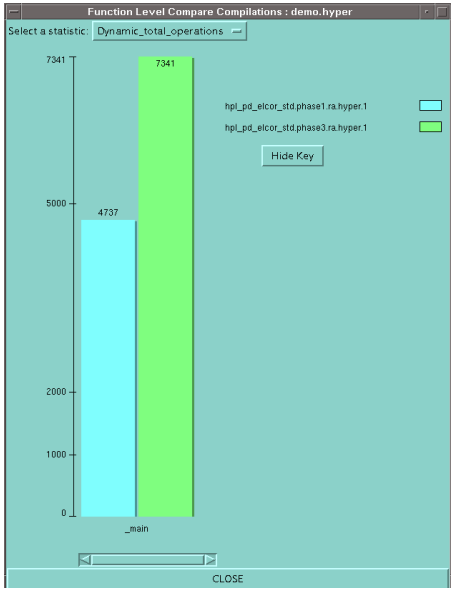
Hyperblock Performance Comparison

- Although the total number of operations executed increases, so does the parallelism.

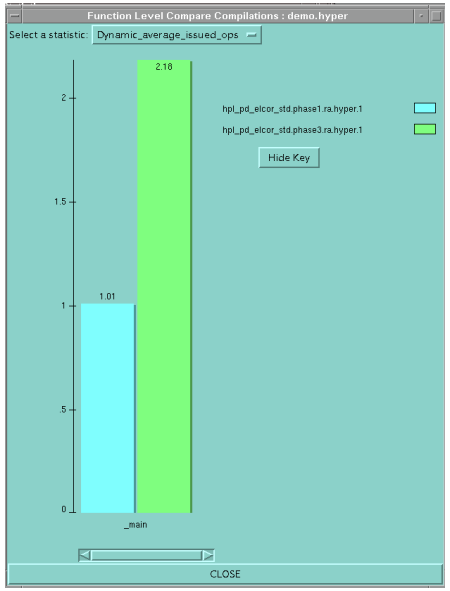
without hyperblock formation

with hyperblock formation

Total number of operations executed



Average number of operations executed per cycle

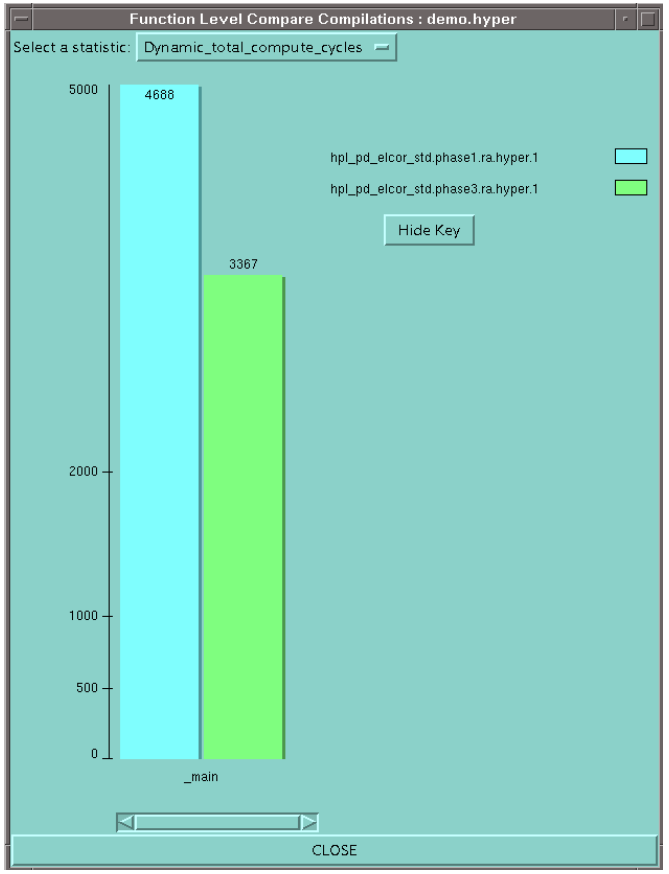


Hyperblock Performance Comparison (cont)

- Execution time is reduced

without hyperblock formation

with hyperblock formation



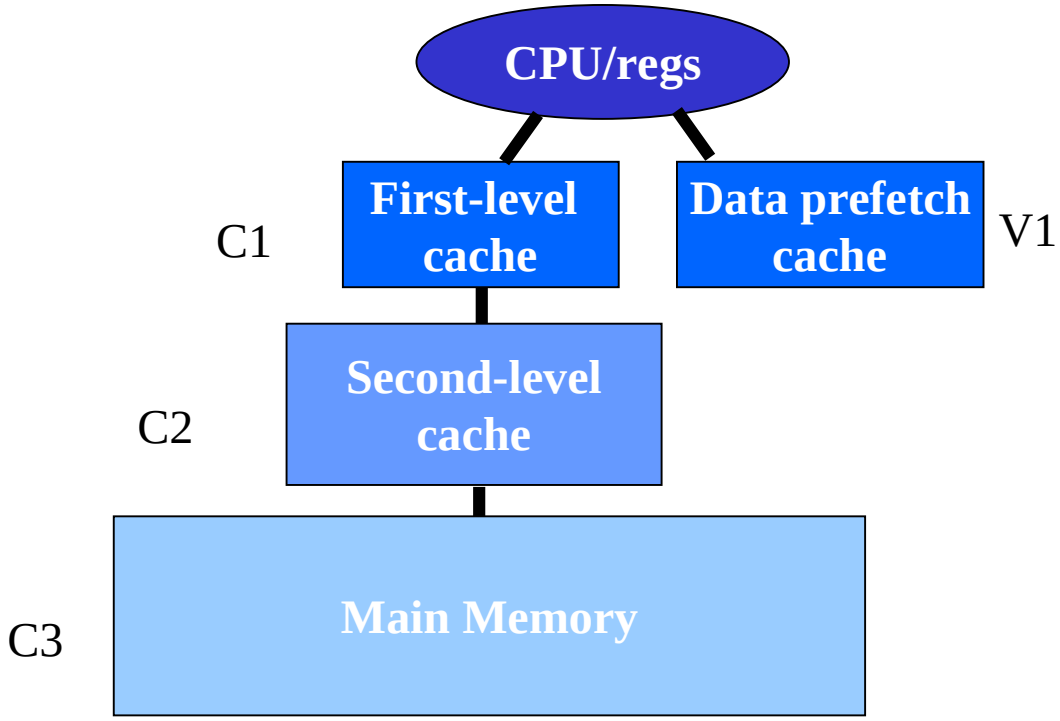
The HPL-PD Memory Hierarchy

HPL-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

Sample Load Instruction

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

Source Cache

Target Cache

Operand register (contains address)

Sample Store Instruction

```
S.W.C1 r2,r3
```

Target Cache

Contains address

Contains value to be stored

Source cache specifiers

On a load, the data might not be in the source cache specified in the instruction.

- Actual latency might be greater or less than expected.

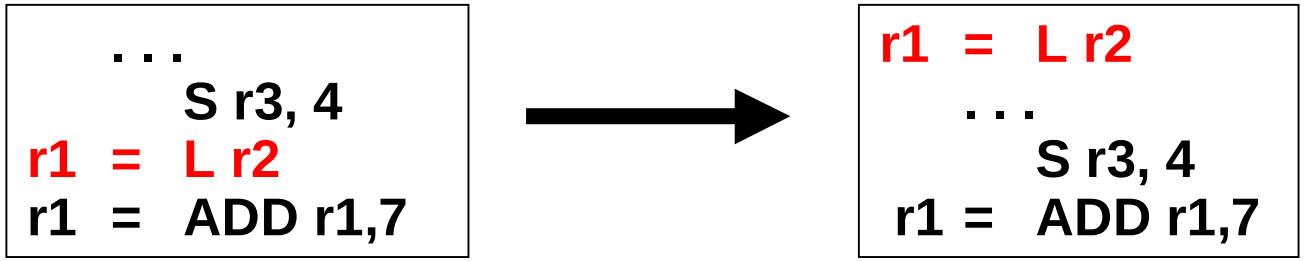
If data is available sooner than expected, its arrival is delayed until the expected time.

If the latency is greater than expected, the entire pipeline is stalled.

- Every functional unit sits idle.

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

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

Run-time Memory Disambiguation (cont)

HPL-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 made in a table to store the memory location

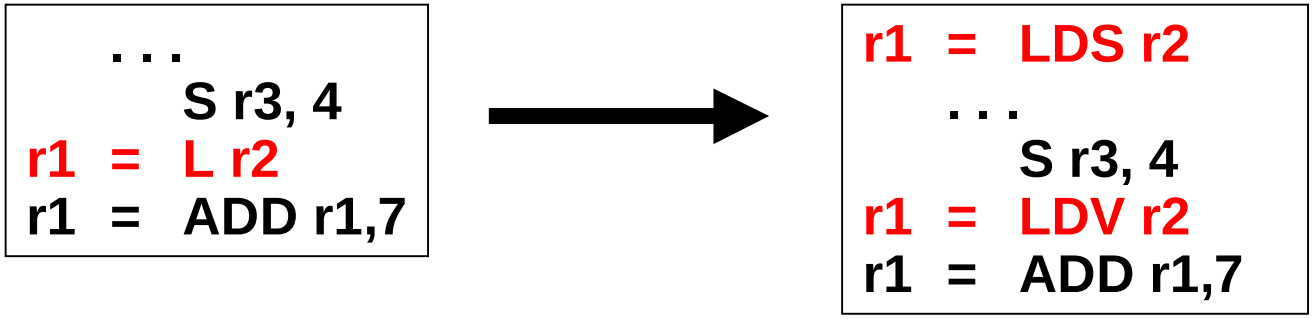
`r1 = LDV r2` ; load verify

– 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.

The HPL-PD Branch Architecture

HPL-PD replaces conventional branch operations with two operations:

- Prepare-to-Branch operations (PBRR, etc)
  - loads target address into a branch target register
  - initiates prefetch of the branch target instruction to minimize branch delay
  - contains field specifying whether the branch is likely to be taken.
  - must precede any branch instruction
- Branch operations (BRU, etc)
  - branches to address contained in a branch target register
  - there are branch instructions for function calls, loops, and software pipelining.

Branches (example)

```
    LC = Mov N
    b1 = PBRR Loop,1
Loop:
    r = L s
    r = ADD r,M
          S s,r
    s = Add s,4
    BRLC b1
```

Store address of Loop in b1. The second operand is a hint that the branch will be taken.

If LC > 0, decrement LC and jump to address in b1.

Other Branch Instructions

BRU b3 if p3

- unconditional branch

BRCT b4,p2 if p4

- branch on condition (in predicate register) true

b2 = BRL b3

- branch to address in b3, save return address in b2

BRF and BRW

- branch instructions to support software pipelining...

Software Pipelining Support

Software Pipelining is a technique for exploiting parallelism across iterations of a loop.

- Iterations are overlaid

HPL-PD's rotating registers support a form of software pipelining called Modulo Scheduling

- Rotating registers provide automatic register renaming across iterations
- The rotating base register, RRB, is decremented by the BRLC instruction.
  - Thus, r[i] in one iteration is referenced as r[i+1] in the next iteration.

Modulo Scheduling (example)

Initial C code:

```
for(i = 0; i < N; i++)
  a[i] += M;
```

Non-pipelined code (r and s are GPR registers)

```
    LC =  MOV N-1
    s  =  MOV a
    b1 =  PBRR Loop,1
Loop:
    r  =  L s
    r  =  ADD r,M
          S s,r
    s  =  Add s,4
          BRLC b1
```

Modulo Scheduling (cont)

With rotating registers, we can overlay iterations of the loop.

- e.g. r[j] in one iteration was r[j-1] in the previous iteration, r[j-2] in the iteration before that, and so on.
- thus a single EPIC instruction could conceivably contain an operation from each of the n previous iterations.
  - where n is the size of the rotating portion of a register file

Modulo Scheduling (cont)

We can overlay the iterations:

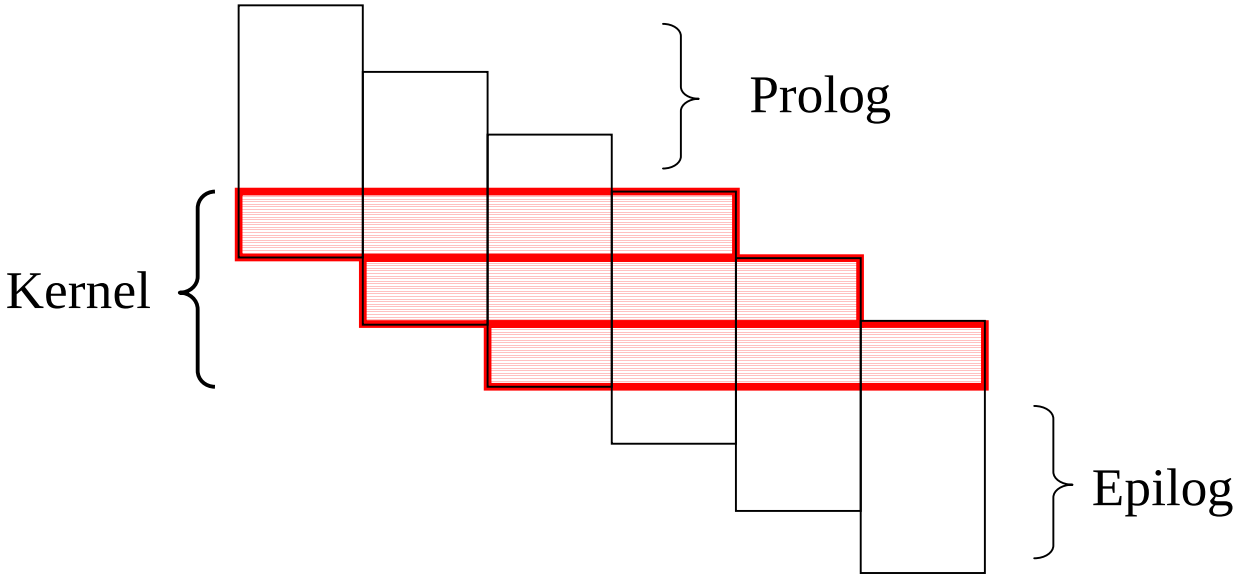
```
r = L s
----                L r,s
r = Add r,M         ----              L r,s
    S s,r         r = Add r,M         ----              L r,s
s = Add s,4           S s,r         r = Add r,M         ----              L r,s
                  s = Add s,4           S s,r         r = Add r,M         ----
                                    s = Add s,4           S s,r         r4 = Add r,M
                                                      s = Add s,4           S s,r
                                                                        s = Add s,4
```

and take a slice to be executed as a single EPIC instruction:

```
s[0] =     Add s[1],4       ; increment i
           S s[4],r[3]      ; store a[i-3]
r[2] =     Add r[2],M       ; a[i-2]= a[i-2]+M
r[0] =     L s[1]           ; load a[i]
```

Loop Prolog and Epilog

Consider a graphical view of the overlay of iterations:



Only the shaded part, the loop kernel, involves executing the full width of the EPIC instruction.

- The loop prolog and epilog contain only a subset of the instructions.
  - "ramp up" and "ramp down" of the parallelism.

Prolog and Epilog (cont)

The prolog can be generated as code outside the loop by the compiler:

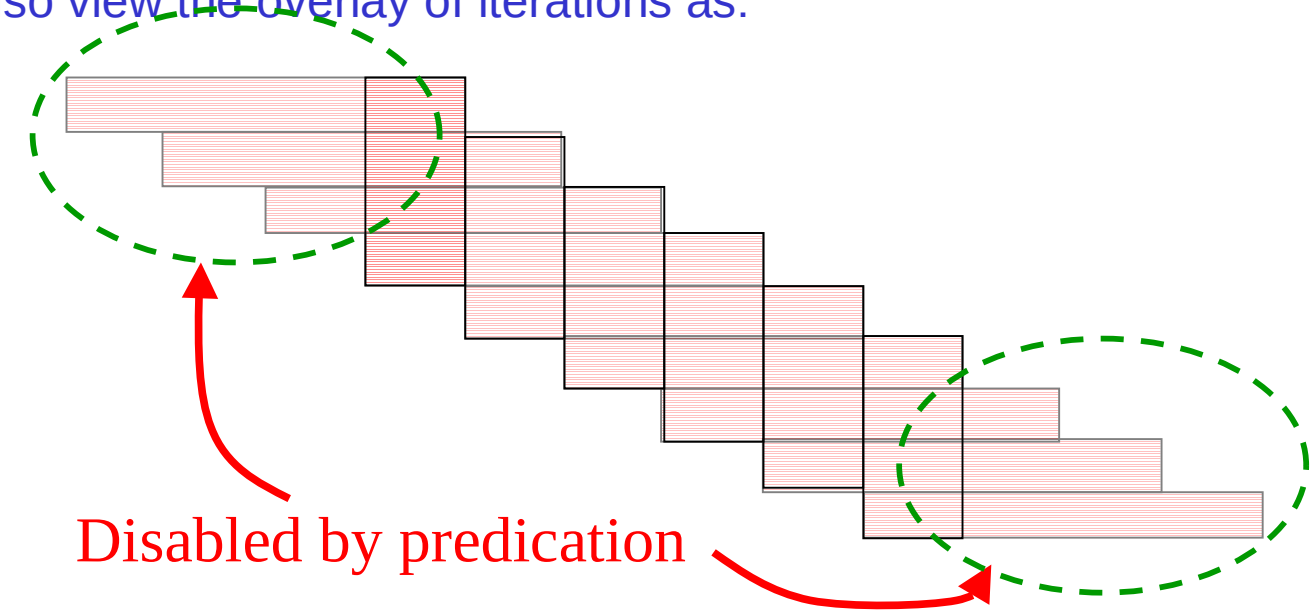
```
        b1  =  PBRR Loop, 1
        s[4] = Mov a
        . . .
        s[1] = Mov a+12
        r[3] = L s[4]
        r[2] = L s[3]
        r[3] = Add r[3],M
        r[1] = L s[2]

Loop:   s[0] = Add s[1],4     ; increment i
               S s[4],r[3]    ; store a[i-3]
        r[2] = Add r[2],M     ; a[i-2]= a[i-2]+M
        r[0] = L s[1]         ; load a[i]
               BRF.B.F.F b1
```

The epilog is handled similarly.

Modulo Scheduling w/ Predication

You can also view the overlay of iterations as:



where the loop kernel is executed in every iteration, but with the undesired instructions disabled by predication.

- Supported by rotating predicate registers.

Modulo Scheduling w/ Predication

Notice that you now need N + (s -1) iterations, where s is the length of each original iteration.

- "ramp down" requires those s-1 iterations, with an additional step being disabled each time.
- The register ESC (epilog stage count) is used to hold this extra count.
- BRF.B.B.F behaves as follows:
  - While LC>0, BRF.B.B.F decrements LC and RRB and writes a 1 into P[0] and branches. This for the Prolog and Kernel.
  - If LC = 0, then while ESC>0, BRF.B.B.F decrements LC and RRB and writes a 0 into P[0] and branches. This is for the Epilog.

Modulo Scheduling w/Predication

Here's the full loop using modulo scheduling, predicated operations, and the ESC register.

```
        s[1]  =     MOV a
        LC    =     MOV N-1
        ESC   =     MOV 4
        b1    =     PBRR Loop,1
Loop:
        s[0]  =     ADD s[1],4   if p[0]
                    S s[4],r[3] if p[3]
        r[2]  =     ADD r[2],M   if p[2]
        r[0]  =     L s[1]       if p[0]
                    BRF.B.B.F b1
```

Modulo Scheduling Performance (matmult)

```
void matmult()
{
   int i,j,k;
   double s1;
   for (i=0 ; i < NUM ; i++)
       for (j=0 ; j < NUM ; j++) {
           s1 = 0.0;
           for (k=0; k < NUM ; k++)
               s1 += a[i][k]*b[k][j];
           c[i][j] = s1;
           printf("c[%d][%d] = %f\n", i, j,s1);
       }
}
```

Modulo Scheduling Performance (matmult)

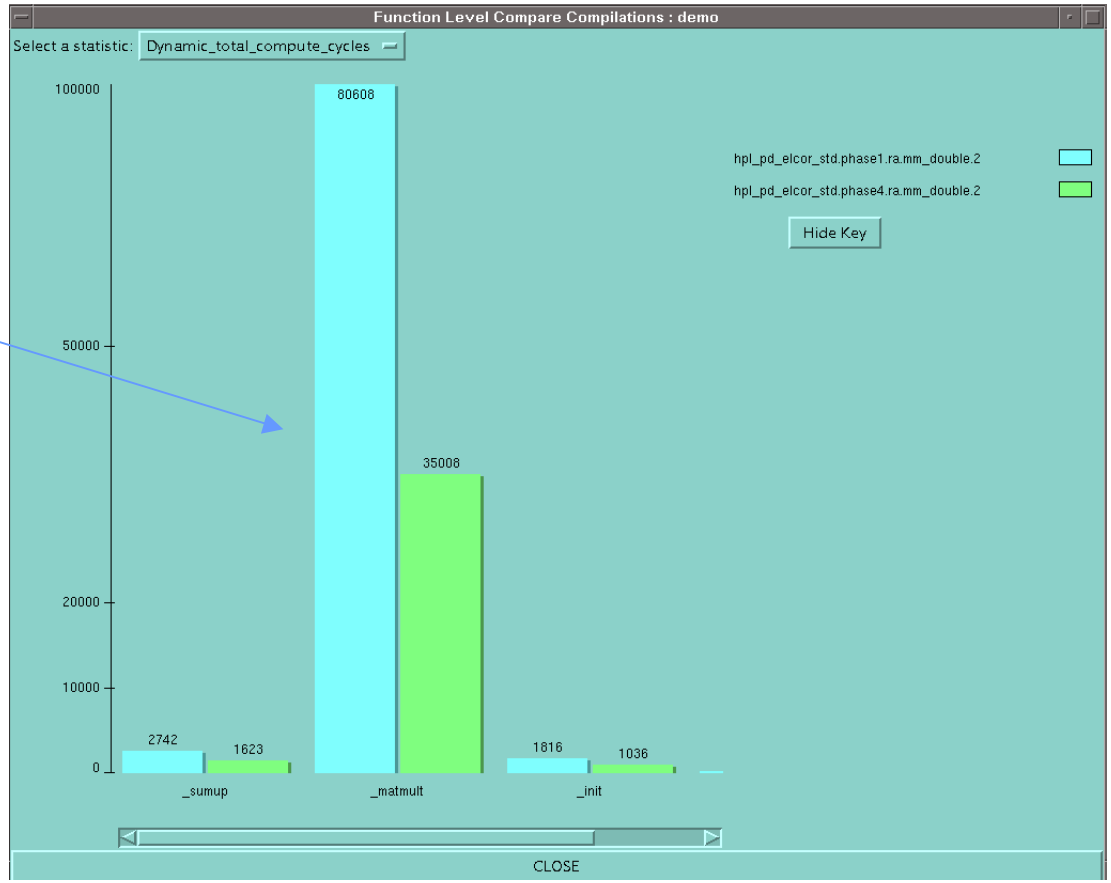
- Speedup of 2.2 due to Modulo scheduling

Total number of cycles

Matmult

Without modulo scheduling
80608 cycles

With modulo scheduling
35008 cycles



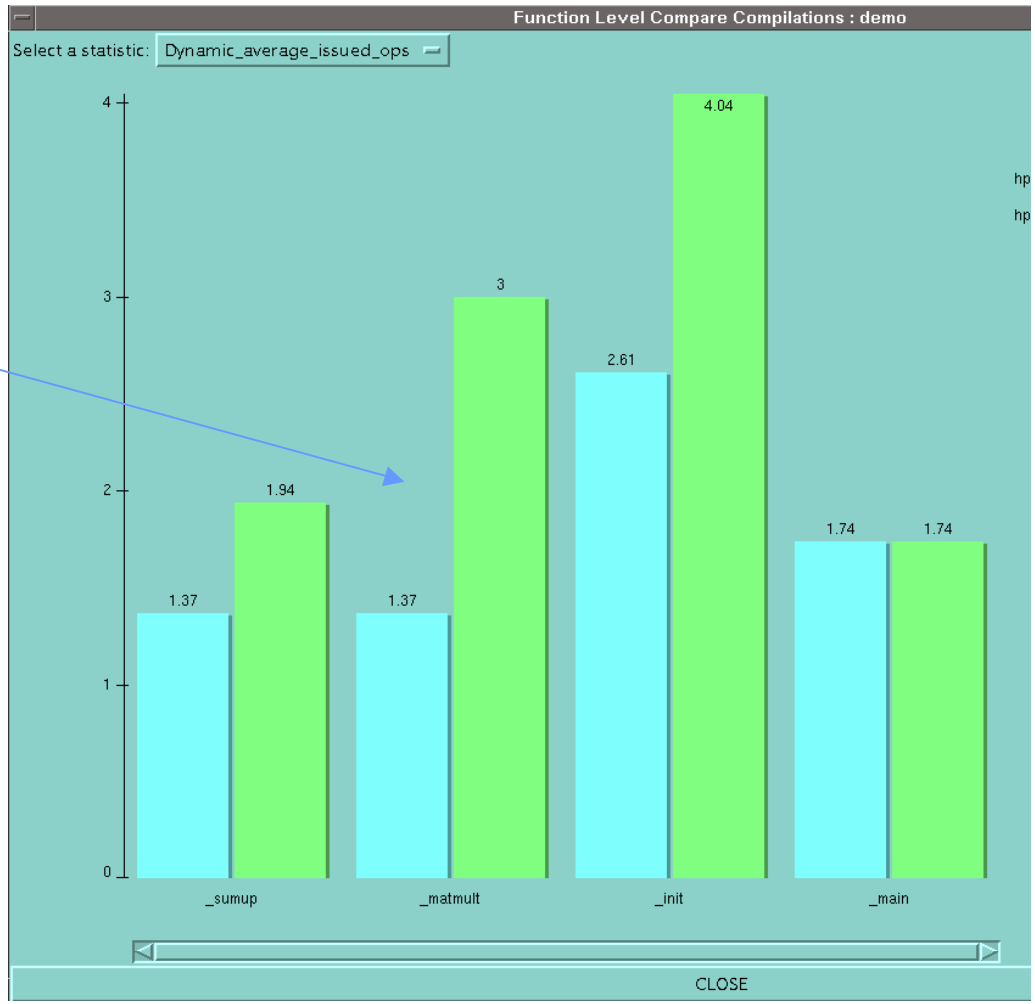
Modulo Scheduling Example (matmult)

ILP Factor:
Average number of ops per cycle

Matmult

Without modulo scheduling
ILP factor = 1.37

With modulo scheduling
ILP factor = 3.0



Summary

- HPL-PD is a flexible ILP architecture
  - encompassing both superscalar and EPIC machine classes
- HPL-PD is a very interesting target for compiler optimizations
  - many useful, novel features
  - increased opportunities for instruction scheduling
    - predication, speculative instructions
  - register allocation
    - EQ model, if desired
  - and other optimizations