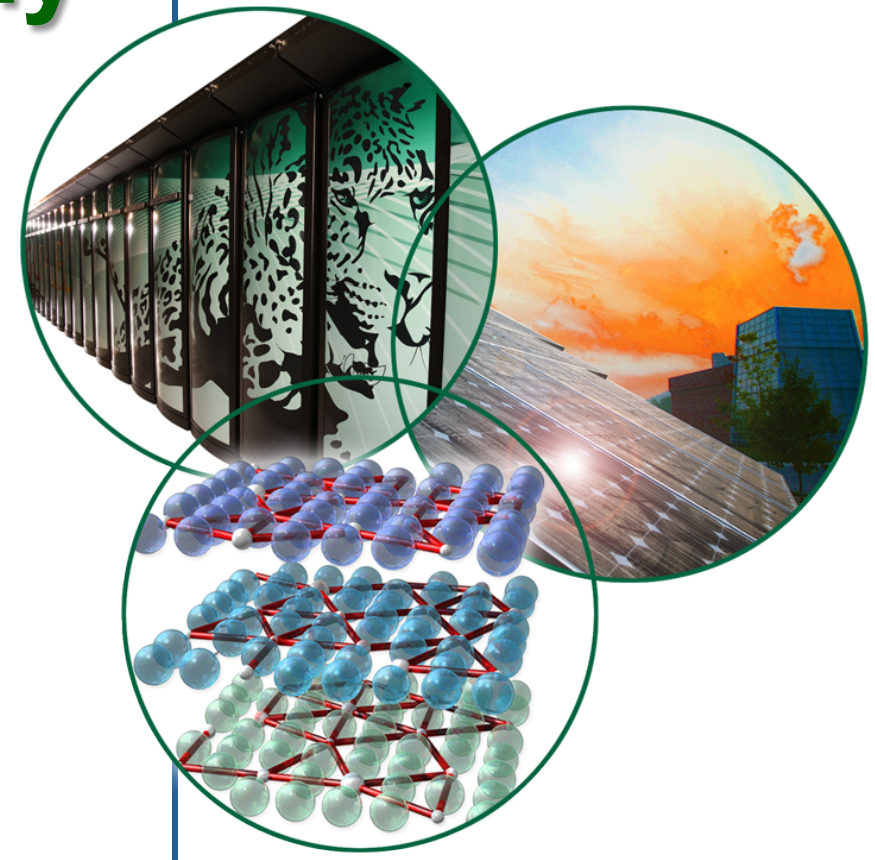
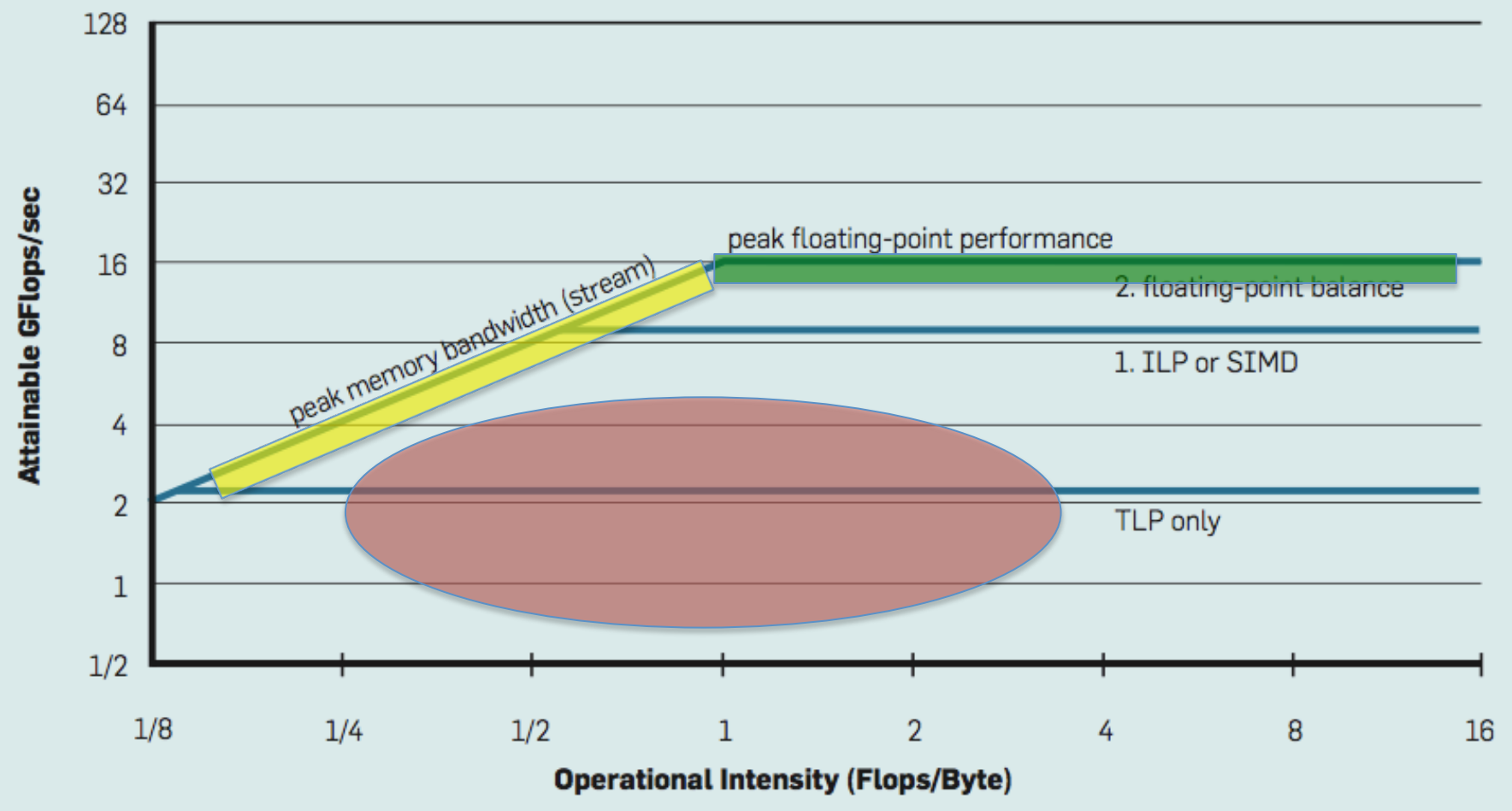
How Fast Should My Application Run?

Understanding performance bottlenecks and what it takes to fix them

Gabriel Marin



The Roofline Model

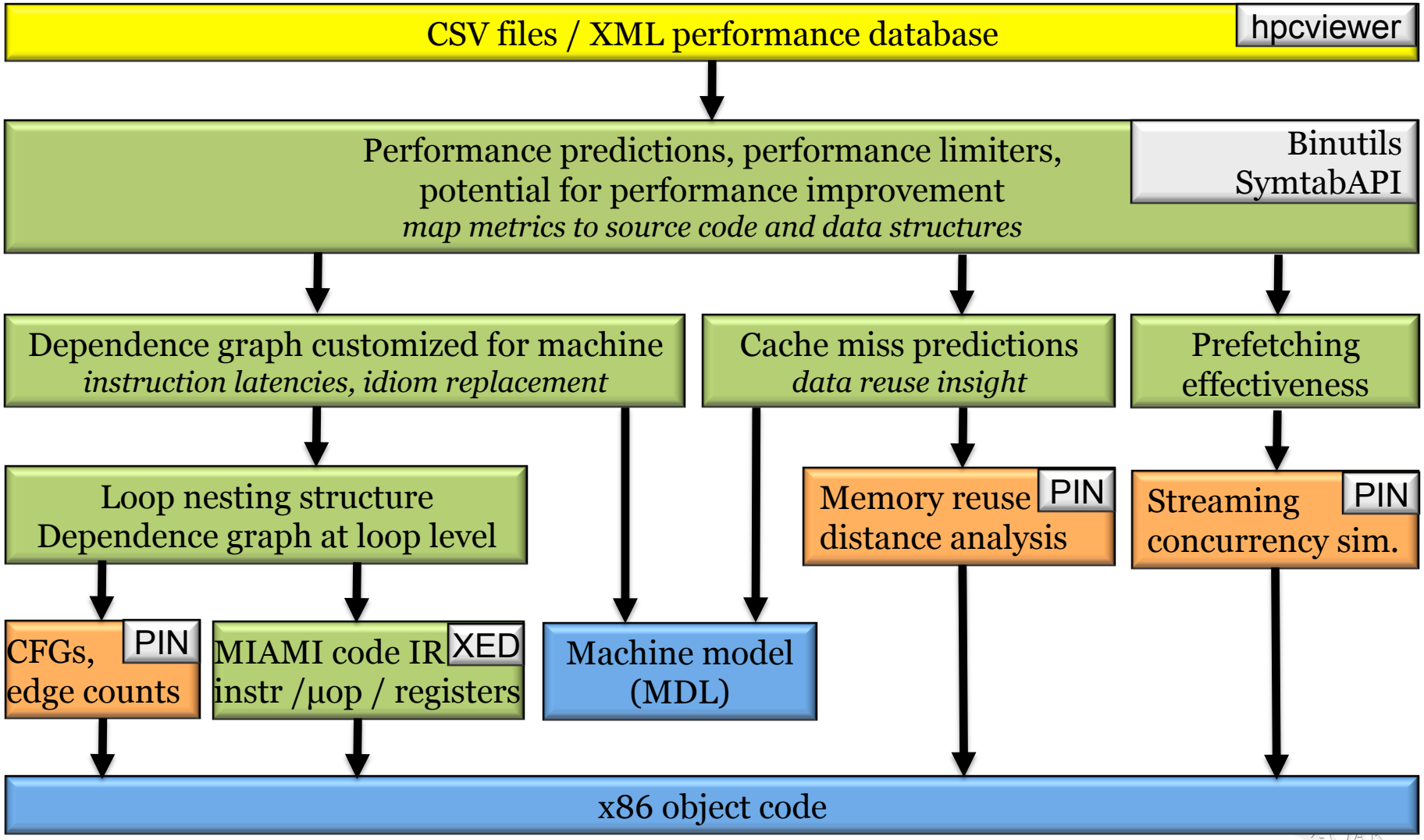


*From: "**Roofline: An Insightful Visual Performance Model for Multicore Architectures**", by S. Williams et al.*

MIAMI

- There is a need for deeper performance analysis
  - Gain insight into performance bottlenecks
  - Root cause analysis
- MIAMI: performance modeling based on static and dynamic analysis of optimized x86-64 binaries
- Application centric, thread level performance models
  - Identify performance limiters at loop level
    - Insufficient ILP, uneven resource utilization, contention on machine resources, memory latency or bandwidth
  - Insight into what code transformations are needed
  - Estimate potential for performance improvement
  - Understand when not to fix an apparent problem

MIAMI Diagram



Binary vs. Source code Analysis

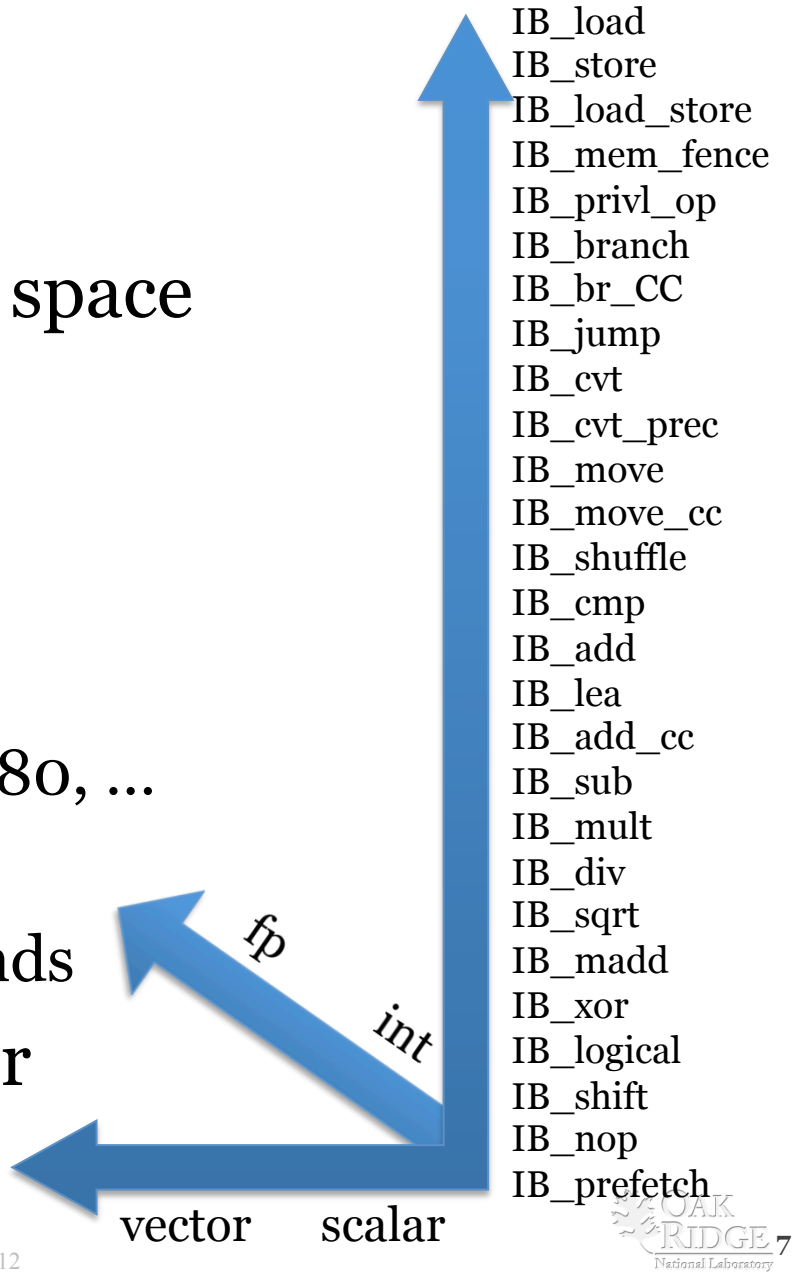
- ✓ Language portability
- ✓ Analysis of optimized code / compiler indep.
- ✓ Full code coverage
- ✗ Loss of high level program information, or more difficult to extract
- ✗ Attribution depends on debug info accuracy
- ✓ Profiling is done post-optimization
- ✓ Machine portability
- ✗ Misses optimization effects, or compiler dependent
- ✗ Blind spots (3rd party libs)
- ✓ High level program information readily available
- ✓ Precise source code attribution
- ✗ Profiling perturbs optimization process

Control Flow Graph (CFG)

- Discovered incrementally at run-time (currently)
  - Light weight tool on top of PIN
  - Selectively insert counters on edges
    - Understand routine entry points, function calls that do not return or return multiple times
  - Save CFGs and selected edge counts
  - 10% – 3x slowdown with PIN
- There are other alternatives
  - Static discovery of CFGs
  - Sampling on the branch target buffer to get edge counts
    - Trade overhead for complexity and some accuracy loss
  - Orthogonal to the rest of the analysis, can be a replacement

Instruction Decoding

- Built on top of XED
- Map instructions onto a 3-D space
  - Instruction type (~ 45 bins)
  - Exec unit style: vector, scalar
  - Operands type: fp, int
- Additional attributes
  - Element bit width: 16, 32, 64, 80, …
  - Vector width: 64, 128, 256, …
  - Source and destination operands
- Represents the MIAMI IR for micro-ops



Instruction Decoding

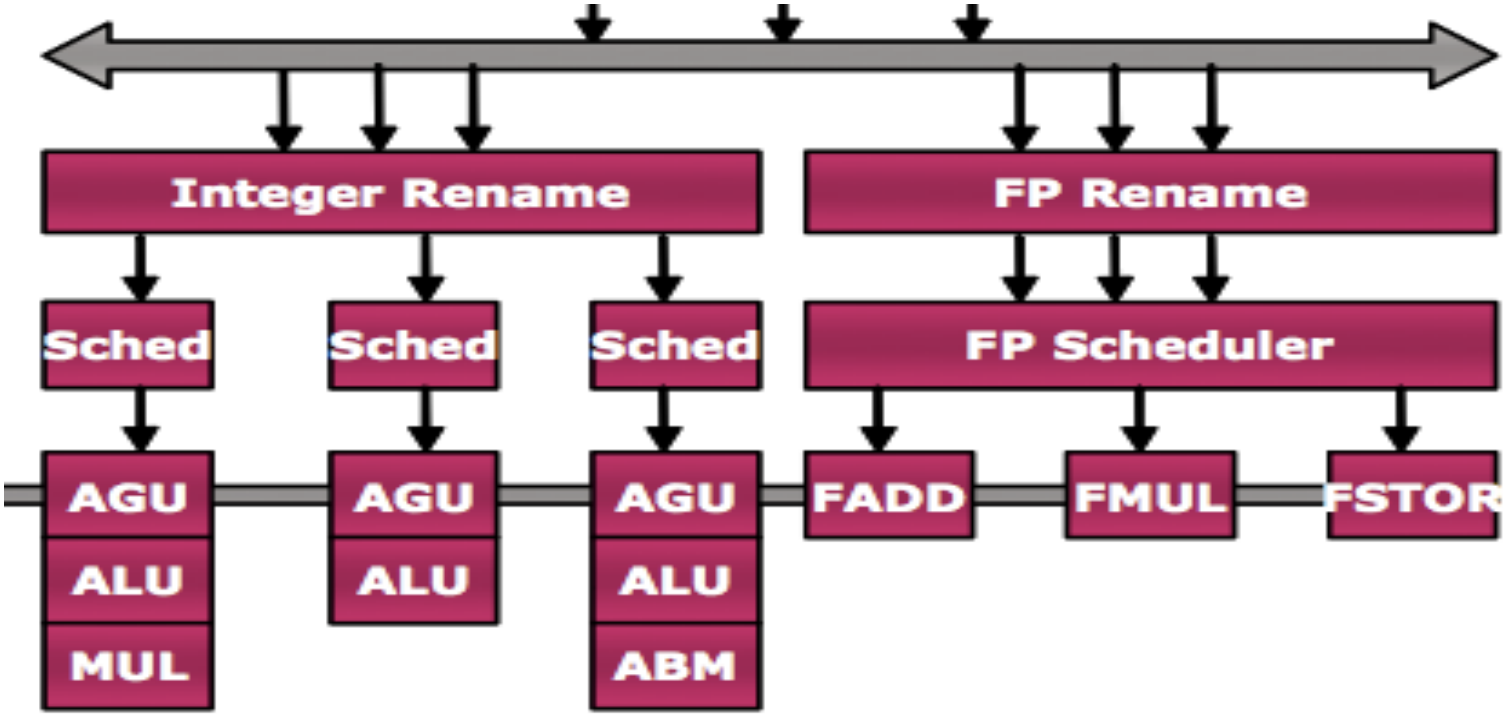
- Only Load, Store and LoadStore micro-ops operate on memory
- For an x86 instruction, each memory operand results into a new Load or Store micro-op, in addition to the micro-op for the main operation
  - Exception: moves that simply copy a value to or from memory
    - they are decoded to a single Store or Load
- Stack push/pop (implicit) operations result in multiple micro-ops (stack pointer increment + mem micro-op)
- REP instructions have a branch micro-op appended
- Care must be taken into assigning original x86 operands to the new micro-ops
  - Instruction dependencies and dataflow analysis are computed on IR

Machine Description Language (MDL)

Construct a model of the target architecture

- Enumerate back-end CPU resources
  - Baseline performance limited by the back-end
- Describe instruction execution templates & resource usage
- Scheduling constraints between resources
- Idiom replacement
  - Account for differences in ISAs, micro-architecture features / optimizations
- Memory hierarchy characteristics
- Other machine features

AMD 10h Architecture




```
CpuUnits = U_ALU * 3, U_AGU * 3, U_Mul, U_ABM,
           U_IDiv, U_LS * 2,
           U_FpAdd, U_FpMul, U_FpStore,
           O_Port * 3;
```

AMD 10h Architecture

```
/* f2iConvert32 */
Instruction Convert{32}:int template = U_FpAdd+U_FpStore+U_ALU, NOTHING*7;
Instruction Convert{32}:int,vec{128} template = U_FpStore, NOTHING*3;

/* f2iConvert64 */
Instruction Convert{64}:int template = U_FpAdd+U_FpStore+U_ALU, NOTHING*7;

/* i2fConvert32 */
Instruction Convert{32}:fp template = U_FpAdd+U_FpStore, NOTHING*8 |
                                      U_FpMul+U_FpStore, NOTHING*8;
Instruction Convert{32}:fp,vec{128} template = U_FpStore, NOTHING*3;

/* i2fConvert64 */
Instruction Convert{64}:fp template = U_FpAdd+U_FpStore, NOTHING*8 |
                                      U_FpMul+U_FpStore, NOTHING*8;
Instruction Convert{64}:fp,vec{128} template = U_FpStore, NOTHING*3;

/* i2fConvert80 - old x87 instruction, only scalar */
Instruction Convert{80}:fp template = U_FpStore, NOTHING*3;

Instruction Div{8}:int template = U_IDiv*17, U_IDiv+O_Port[1,2];
Instruction Div{16}:int template = U_IDiv*26, U_IDiv+O_Port[1,2];
Instruction Div{32}:int template = U_IDiv*43, U_IDiv+O_Port[1,2];
Instruction Div{64}:int template = U_IDiv*74, U_IDiv+O_Port[1,2];
```

AMD 10h Architecture

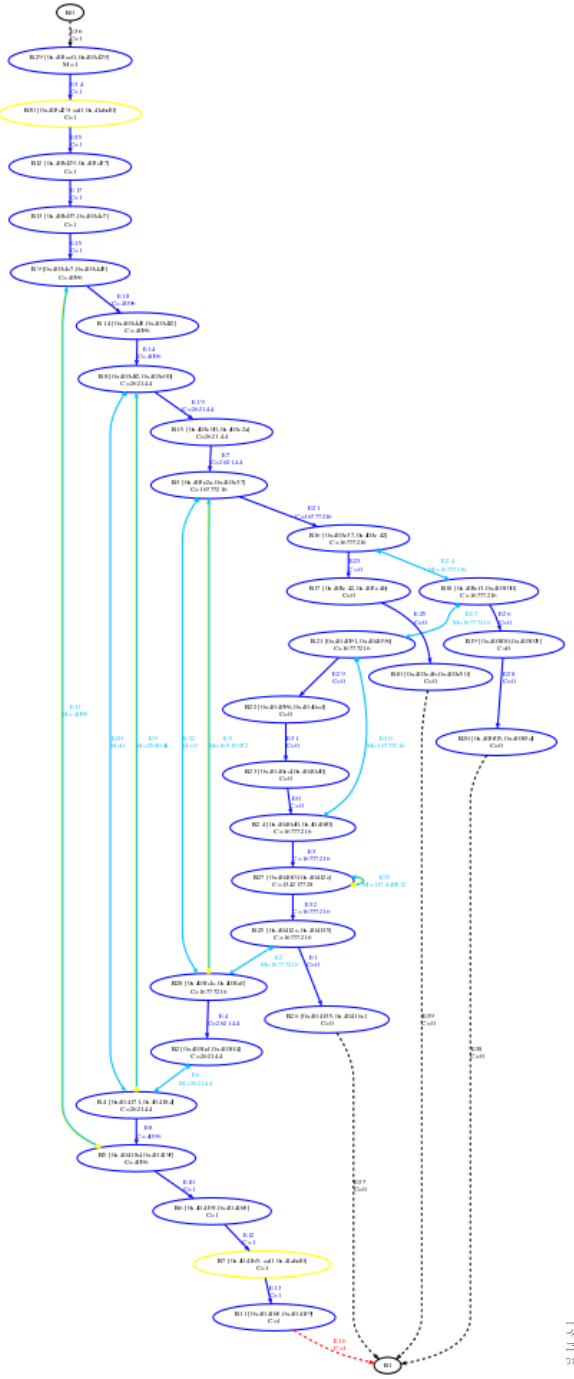
"The L1 data cache can support two 128-bit loads or two 64-bit store writes per cycle or a mix of those. The LSU consists of two queues—LS1 and LS2. LS1 can issue two L1 cache operations (loads or store tag checks) per cycle. It can issue load operations out-of-order, subject to certain dependency restrictions. LS2 effectively holds requests that missed in the L1 cache after they probe out of LS1. Store writes are done exclusively from LS2. 128-bit stores are specially handled in that they take two LS2 entries, and the store writes are performed as two 64-bit writes."

```
/* AMD 10h has only 64 bit stores. 128bit stores are split into
 * two 64bit stores. */
 Replace Store:int,vec{128} $rX -> [$rA] with
       Store:int,vec{64} $rX -> [$rA] +
       Store:int,vec{64} $rX -> [$rA] {"Store 64b int"};

 Replace Store:fp,vec{128} $rX -> [$rA] with
       Store:fp,vec{64} $rX -> [$rA] +
       Store:fp,vec{64} $rX -> [$rA]  {"Store 64b fp"};
```

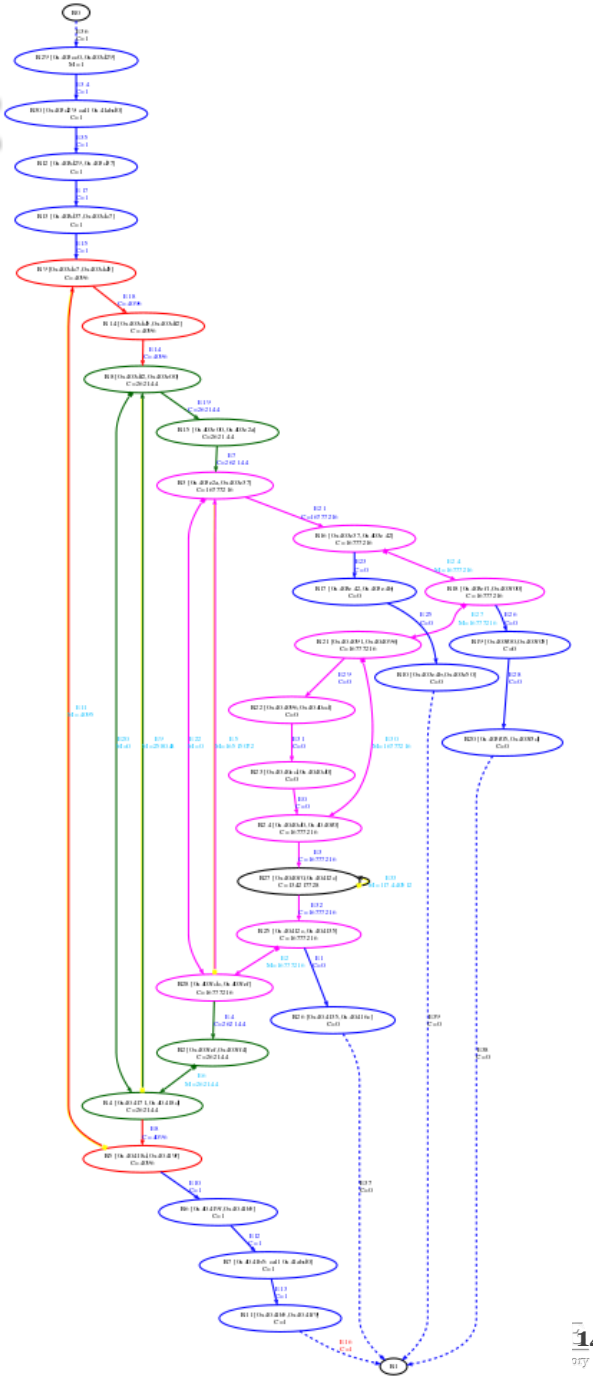
Instruction Schedule Cost

- Read CFG file and recover execution counts for all blocks and edges



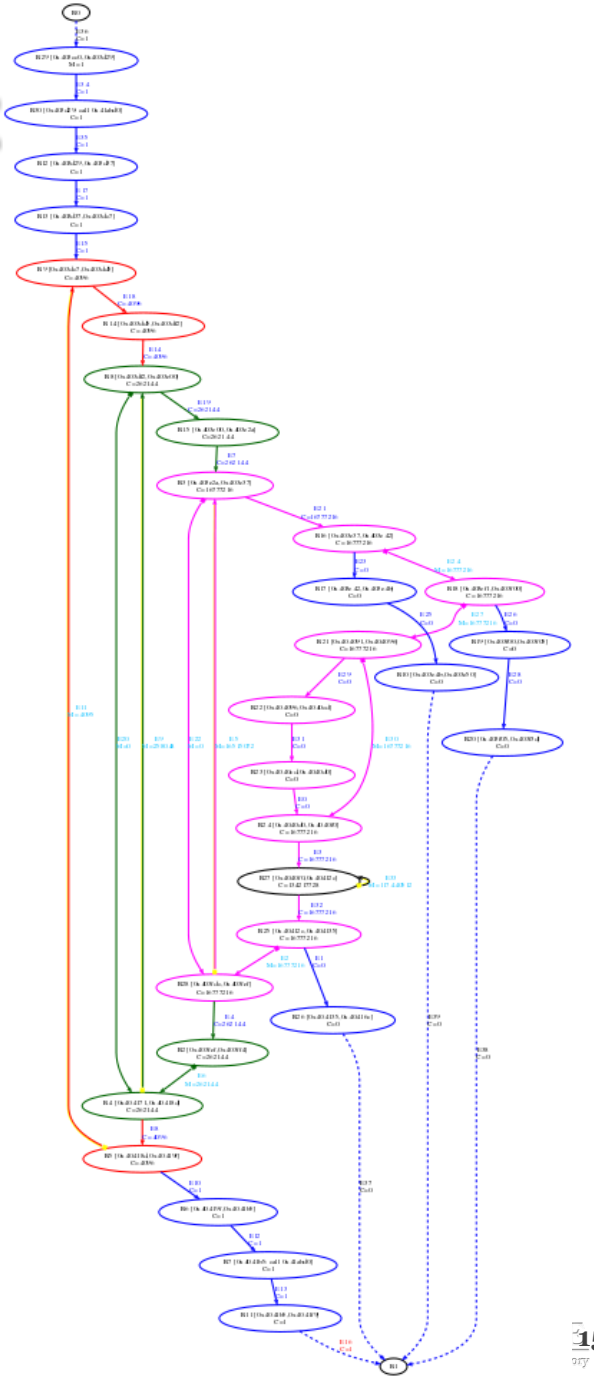
Instruction Schedule Cost

- Read CFG file and recover execution counts for all blocks and edges
- Compute loop nesting structures



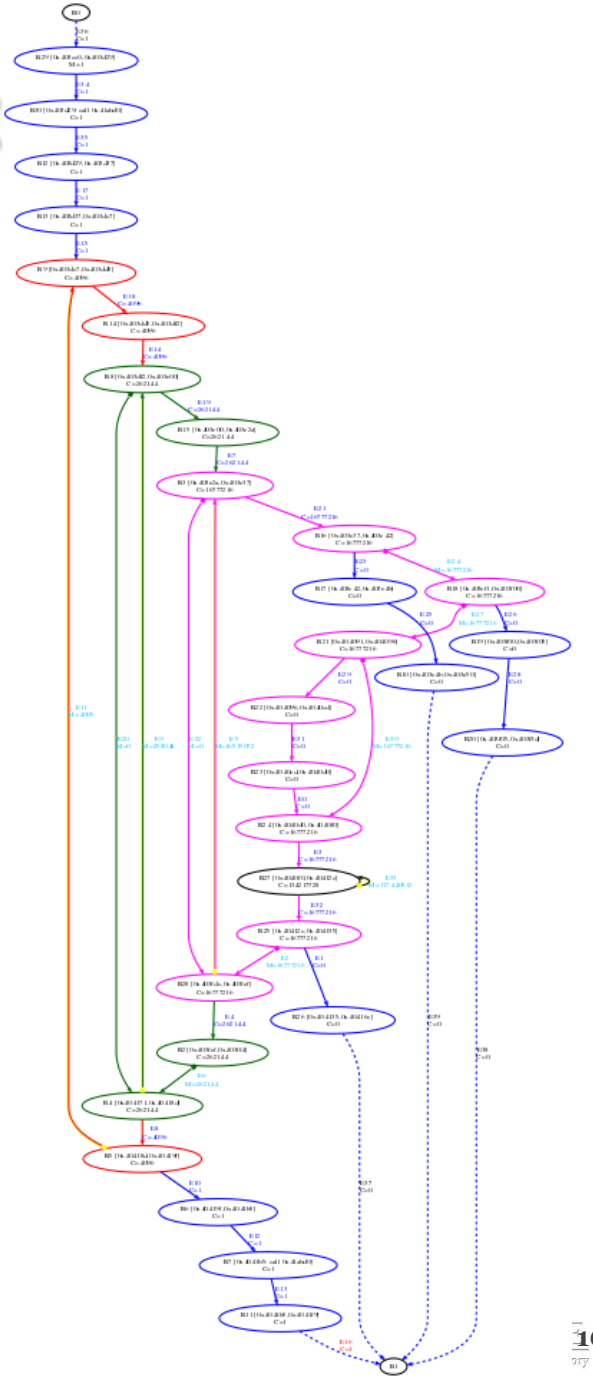
Instruction Schedule Cost

- Read CFG file and recover execution counts for all blocks and edges
- Compute loop nesting structures
- Infer executed paths and their execution frequencies
  - at loop level from the inside out



Instruction Schedule Cost

- Read CFG file and recover execution counts for all blocks and edges
- Compute loop nesting structures
- Infer executed paths and their execution frequencies
  - at loop level from the inside out
- Compute instruction schedule for each path



Instruction Scheduling

- Compute instruction schedule one path at a time
  - Emulates ideal branch predictor
- Build dependence graph for path
- Tailor dependence graph for machine
  - Instantiate scheduler with a machine model
  - Apply replacement rules
  - Instruction latencies determine edges' lengths
- Perform modulo instruction scheduling
  - Provides insight into instruction schedule cost

Matrix Multiply Example

```
register int i, j, k, r;
for (r=0 ; r<reps ; ++r) {
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      for (k = 0; k < n; k++) {
        c[i][j] += a[i][k]*b[k][j];
      }
    }
  }
}
```

Assembly code for inner most loop:

- Intel compiler unrolled the loop 16 times

```
movaps xmm1,XMMWORD PTR [rcx+r9*8+0x609120]
movaps xmm2,XMMWORD PTR [rcx+r9*8+0x609130]
movaps xmm3,XMMWORD PTR [rcx+r9*8+0x609140]
movaps xmm4,XMMWORD PTR [rcx+r9*8+0x609150]
movaps xmm5,XMMWORD PTR [rcx+r9*8+0x609160]
movaps xmm6,XMMWORD PTR [rcx+r9*8+0x609170]
movaps xmm7,XMMWORD PTR [rcx+r9*8+0x609180]
movaps xmm8,XMMWORD PTR [rcx+r9*8+0x609190]
mulpd  xmm1,xmm0
mulpd  xmm2,xmm0
mulpd  xmm3,xmm0
mulpd  xmm4,xmm0
mulpd  xmm5,xmm0
mulpd  xmm6,xmm0
mulpd  xmm7,xmm0
mulpd  xmm8,xmm0
addpd  xmm1,XMMWORD PTR [rsi+r9*8+0x60d920]
addpd  xmm2,XMMWORD PTR [rsi+r9*8+0x60d930]
addpd  xmm3,XMMWORD PTR [rsi+r9*8+0x60d940]
addpd  xmm4,XMMWORD PTR [rsi+r9*8+0x60d950]
addpd  xmm5,XMMWORD PTR [rsi+r9*8+0x60d960]
addpd  xmm6,XMMWORD PTR [rsi+r9*8+0x60d970]
addpd  xmm7,XMMWORD PTR [rsi+r9*8+0x60d980]
addpd  xmm8,XMMWORD PTR [rsi+r9*8+0x60d990]
movaps XMMWORD PTR [rsi+r9*8+0x60d920],xmm1
movaps XMMWORD PTR [rsi+r9*8+0x60d930],xmm2
movaps XMMWORD PTR [rsi+r9*8+0x60d940],xmm3
movaps XMMWORD PTR [rsi+r9*8+0x60d950],xmm4
movaps XMMWORD PTR [rsi+r9*8+0x60d960],xmm5
movaps XMMWORD PTR [rsi+r9*8+0x60d970],xmm6
movaps XMMWORD PTR [rsi+r9*8+0x60d980],xmm7
movaps XMMWORD PTR [rsi+r9*8+0x60d990],xmm8
add    r9,0x10
cmp    r9,0x30
jb     0x400aa0 <main+528>
```

Instruction Scheduler Metrics

Scheduler predictions

Scopes	ExecTime	InfCpuRes	NoDepend	MaxGainExtraRes	MaxGainExtraIP	CPUBottleNeck	BOT_U_LS[0]
Experiment Aggregate Metrics	1.43e09 100.0	3.46e08 100.0	1.15e09 100.0	1.08e09 100.0	2.77e08 100.0	1.11e09 100.0	1.11e09 100.0
main	1.43e09 100.0	3.46e08 100.0	1.15e09 100.0	1.08e09 100.0	2.76e08 100.0	1.11e09 100.0	1.11e09 100.0
loop at source.c: 36	1.43e09 100.0	3.46e08 100.0	1.15e09 100.0	1.08e09 100.0	2.76e08 100.0	1.11e09 100.0	1.11e09 100.0
loop at source.c: 36	1.43e09 100.0	3.46e08 100.0	1.15e09 100.0	1.08e09 100.0	2.76e08 100.0	1.11e09 100.0	1.11e09 100.0
loop at source.c: 36	1.43e09 100.0	3.46e08 100.0	1.15e09 100.0	1.08e09 100.0	2.76e08 100.0	1.11e09 100.0	1.11e09 100.0
loop at source.c: 36	1.34e09 93.5%	2.53e08 73.3%	1.11e09 96.0%	1.08e09 100.0	2.30e08 83.3%	1.11e09 100.0	1.11e09 100.0
Path 1 (x): 2.304E7	2.60e01 0.0%	1.10e01 0.0%	1.60e01 0.0%	1.50e01 0.0%	1.00e01 0.0%	1.60e01 0.0%	1.60e01 0.0%
Path 2: 4.608E7	1.60e01 0.0%	0.00e00 0.0%	1.60e01 0.0%	1.60e01 0.0%	0.00e00 0.0%	1.60e01 0.0%	1.60e01 0.0%
Path 1: 2.3037696E7	4.00e00 0.0%	4.00e00 0.0%	2.00e00 0.0%	0.00e00 0.0%	2.00e00 0.0%		
Path 2 (x): 2304.0	4.00e00 0.0%	4.00e00 0.0%	2.00e00 0.0%	0.00e00 0.0%	2.00e00 0.0%		
Path 1 (x): 48.0	8.00e00 0.0%	8.00e00 0.0%	2.00e00 0.0%	0.00e00 0.0%	6.00e00 0.0%		
Path 2: 2256.0	6.00e00 0.0%	6.00e00 0.0%	2.00e00 0.0%	0.00e00 0.0%	4.00e00 0.0%		
Path 2 (x): 1.0	7.00e00 0.0%	7.00e00 0.0%	3.00e00 0.0%	0.00e00 0.0%	4.00e00 0.0%		
Path 1: 47.0	5.00e00 0.0%	5.00e00 0.0%	3.00e00 0.0%	0.00e00 0.0%			

Main performance limiting factor is the issue bandwidth on the Load/Store units

Scope	CPU_CLK_UNHALTED.[0,0] (I)	DATA_CACHE_MISSES.[0,0] (I)	RETIRED_INSTRUCTIONS.[0,0] (I)	RETIRED_UOPS.[0,0] (I)
Experiment Aggregate Metrics	1.51e+09 100 %	2.30e+04 100 %	2.50e+09 100 %	3.06e+09 100 %
main	1.51e+09 100 %	2.30e+04 100 %	2.50e+09 100 %	3.06e+09 100 %
loop at source.c: 36	1.51e+09 100 %	2.30e+04 100 %	2.50e+09 100 %	3.06e+09 100 %
loop at source.c: 36	1.51e+09 100 %	2.30e+04 100 %	2.50e+09 100 %	3.06e+09 100 %
loop at source.c: 36	1.51e+09 100 %	2.30e+04 100 %	2.50e+09 100 %	3.06e+09 100 %
loop at source.c: 36	1.46e+09 96.7%	2.30e+04 100 %	2.43e+09 97.2%	2.97e+09 97.0%
source.c: 36	1.46e+09 96.7%	2.30e+04 100 %	2.43e+09 97.2%	2.97e+09 97.0%
source.c: 36	4.97e+07 3.3%		7.08e+07 2.8%	9.31e+07 3.0%

HPCToolkit measurements

48*48*8*3 = 55KB

Matrix Multiply Example

- 8 x 128-bit Mul, 8 x 128-bit Add
  - 16 cycles => 50% efficiency with no memory delays
- 16 x 128-bit Load, 8 x 128-bit Store
  - 128-bit Store = 2 x 64-bit Stores
  - Issue bandwidth limited on U_LS
    - Needs blocking for register reuse

```
movaps  xmm1,XMMWORD PTR [rcx+r9*8+0x609120]
movaps  xmm2,XMMWORD PTR [rcx+r9*8+0x609130]
movaps  xmm3,XMMWORD PTR [rcx+r9*8+0x609140]
movaps  xmm4,XMMWORD PTR [rcx+r9*8+0x609150]
movaps  xmm5,XMMWORD PTR [rcx+r9*8+0x609160]
movaps  xmm6,XMMWORD PTR [rcx+r9*8+0x609170]
movaps  xmm7,XMMWORD PTR [rcx+r9*8+0x609180]
movaps  xmm8,XMMWORD PTR [rcx+r9*8+0x609190]
mulpd   xmm1,xmm0
mulpd   xmm2,xmm0
mulpd   xmm3,xmm0
mulpd   xmm4,xmm0
mulpd   xmm5,xmm0
mulpd   xmm6,xmm0
mulpd   xmm7,xmm0
mulpd   xmm8,xmm0
addpd   xmm1,XMMWORD PTR [rsi+r9*8+0x60d920]
addpd   xmm2,XMMWORD PTR [rsi+r9*8+0x60d930]
addpd   xmm3,XMMWORD PTR [rsi+r9*8+0x60d940]
addpd   xmm4,XMMWORD PTR [rsi+r9*8+0x60d950]
addpd   xmm5,XMMWORD PTR [rsi+r9*8+0x60d960]
addpd   xmm6,XMMWORD PTR [rsi+r9*8+0x60d970]
addpd   xmm7,XMMWORD PTR [rsi+r9*8+0x60d980]
addpd   xmm8,XMMWORD PTR [rsi+r9*8+0x60d990]
movaps  XMMWORD PTR [rsi+r9*8+0x60d920],xmm1
movaps  XMMWORD PTR [rsi+r9*8+0x60d930],xmm2
movaps  XMMWORD PTR [rsi+r9*8+0x60d940],xmm3
movaps  XMMWORD PTR [rsi+r9*8+0x60d950],xmm4
movaps  XMMWORD PTR [rsi+r9*8+0x60d960],xmm5
movaps  XMMWORD PTR [rsi+r9*8+0x60d970],xmm6
movaps  XMMWORD PTR [rsi+r9*8+0x60d980],xmm7
movaps  XMMWORD PTR [rsi+r9*8+0x60d990],xmm8
add     r9,0x10
cmp     r9,0x30
jb      0x400aa0 <main+528>
```

5-point Stencil2D Example

```
for (j = 1; j < dy-1; j++)
   for (i = 1; i < dx-1; i++)
      V[j*dx+i] = c0*V[j*dx+i] + c1*(V[j*dx+(i-1)] + V[(j-1)*dx+i] +
                                     V[j*dx+(i+1)] + V[(j+1)*dx+i]);
```

icc -fast -g -inline-level=0

Scopes

Experiment Aggregate Metrics
- stencil2d
  - loop at stencil2d_double.c: 12-17
    - loop at stencil2d_double.c: 14-17
      - Path 2 (x): 2381820.0
      - Path 1: 9.765462E7
    - Path 1 (x): 28355.0
    - Path 2: 2353465.0
  - Path 1 (x): 28355.0

ExecTime	InfCpuRes	NoDepend	MaxGainExtraRes	MaxGainExtraIP	AppDepTime
2.45e09 100.0	2.45e09 100.0	8.17e08 100.0	2.38e06 100.0	1.64e09 100.0	2.44e09 100.0
2.45e09 100.0	2.45e09 100.0	8.17e08 100.0	2.38e06 99.9%	1.64e09 100.0	2.44e09 100.0
2.45e09 100.0	2.45e09 100.0	8.17e08 100.0	2.35e06 98.7%	1.64e09 100.0	2.44e09 100.0
2.43e09 99.2%	2.43e09 99.3%	8.00e08 97.9%	0.00e00 0.0%	1.63e09 99.8%	2.43e09 99.9%
3.80e01 0.0%	3.80e01 0.0%	8.00e00 0.0%	0.00e00 0.0%	3.00e01 0.0%	3.80e01 0.0%
2.40e01 0.0%	2.40e01 0.0%	8.00e00 0.0%	0.00e00 0.0%	1.60e01 0.0%	2.40e01 0.0%
1.00e01 0.0%	1.00e01 0.0%	5.00e00 0.0%	0.00e00 0.0%	5.00e00 0.0%	1.00e01 0.0%
8.00e00 0.0%	7.00e00 0.0%	7.00e00 0.0%	1.00e00 0.0%	1.00e00 0.0%	1.00e00 0.0%
1.20e01 0.0%	1.10e01 0.0%				1.10e01 0.0%

Data dependencies are the main performance limiting factor.

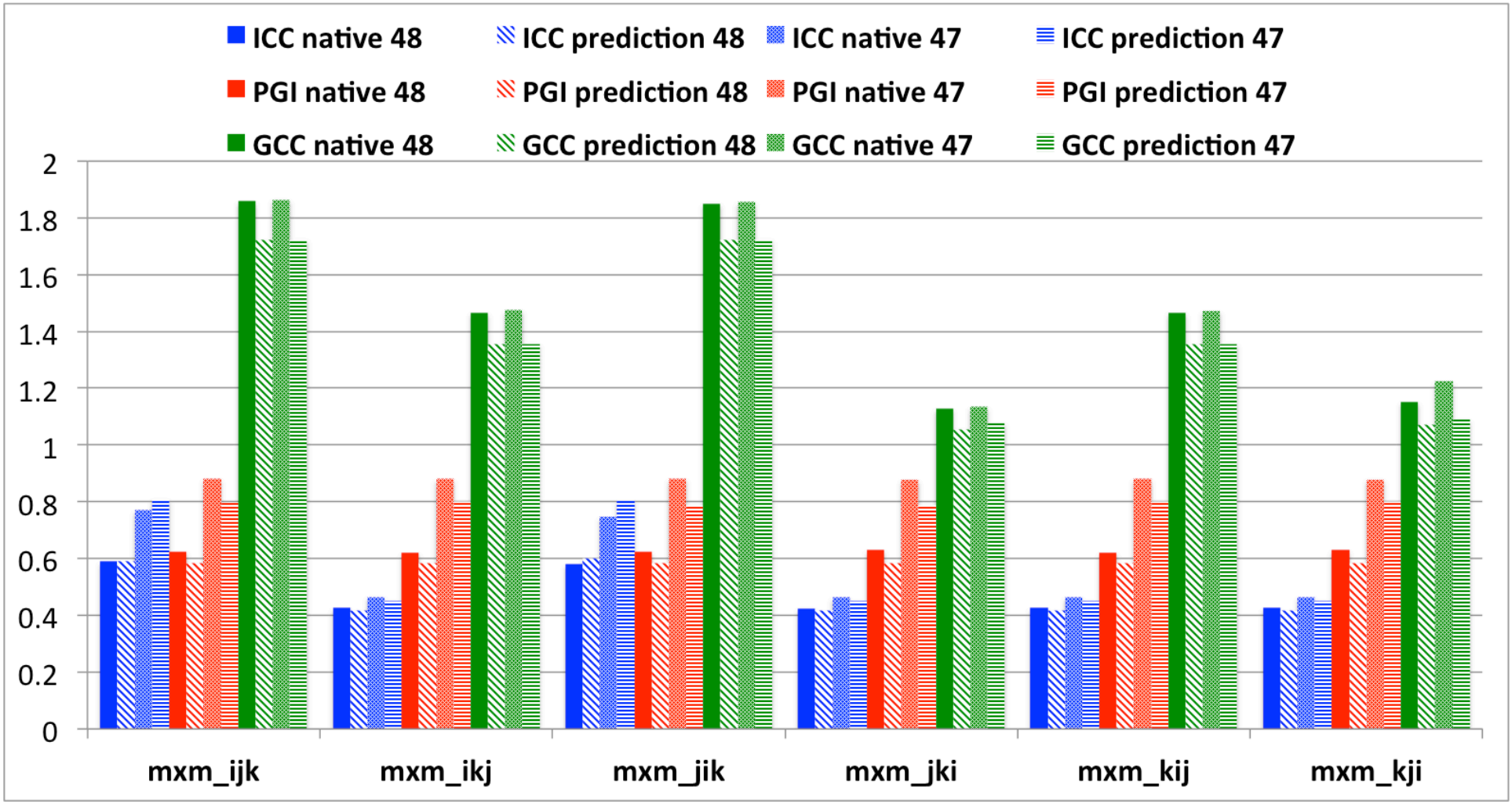
Scopes

Experiment Aggregate Metrics
- stencil2d
  - loop at stencil2d_double.c: 12-14
    - loop at stencil2d_double.c: 14
      - Path 1: 9.765462E7
      - Path 2 (x): 2381820.0
    - Path 1 (x): 28355.0
    - Path 2: 2353465.0
  - Path 1 (x): 28355.0

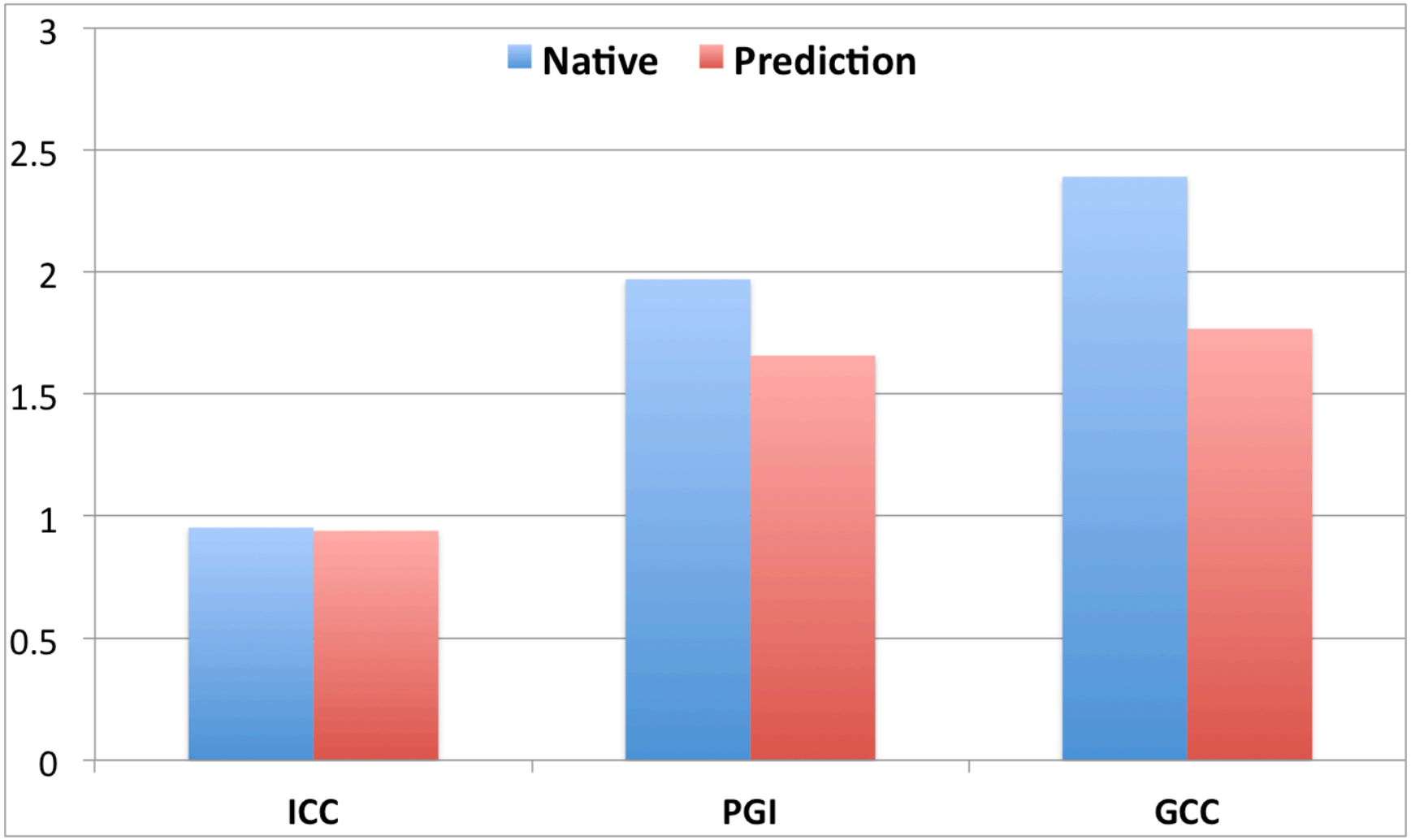
ExecTime	InfCpuRes				AppDepTime
4.33e09 100.0	4.32e09 100.0	8.25e08 100.0	9.53e06 100.0	3.50e09 100.0	4.31e09 100.0
4.33e09 100.0	4.32e09 100.0	8.24e08 100.0	9.50e06 99.7%	3.50e09 100.0	4.30e09 100.0
4.33e09 100.0	4.32e09 100.0	8.24e08 99.9%	9.41e06 98.8%	3.50e09 100.0	4.30e09 100.0
4.30e09 99.4%	4.30e09 99.6%	8.00e08 97.0%	0.00e00 0.0%	3.50e09 99.9%	4.30e09 99.9%
4.30e01 0.0%	4.30e01 0.0%	8.00e00 0.0%	0.00e00 0.0%	3.50e01 0.0%	4.30e01 0.0%
4.30e01 0.0%	4.30e01 0.0%	8.00e00 0.0%	0.00e00 0.0%	3.50e01 0.0%	4.30e01 0.0%
1.30e01 0.0%	1.30e01 0.0%	8.00e00 0.0%	0.00e00 0.0%	5.00e00 0.0%	1.30e01 0.0%
1.10e01 0.0%	7.00e00 0.0%	1.00e01 0.0%	4.00e00 0.0%	1.00e00 0.0%	1.00e00 0.0%
1.30e01 0.0%	1.00e01 0.0%	9.00e00 0.0%	3.00e00 0.0%	4.00e00 0.0%	1.00e01 0.0%

pgcc -fastsse -gopt

Matrix Multiply Prediction Accuracy



Stencil2D Prediction Accuracy



Insight from the Scheduler

- Understand losses due to insufficient ILP
- Utilization of various machine resources
  - If vector units are available and not used
    - Can we understand the reason for failed vectorization?
    - Lack of ILP or another machine specific reason
- Contention on machine resources
  - Few options from an application perspective, must change instruction mix
  - Contention on load/store unit -> improve register reuse
- Challenge
  - How to present relevant insight to the user

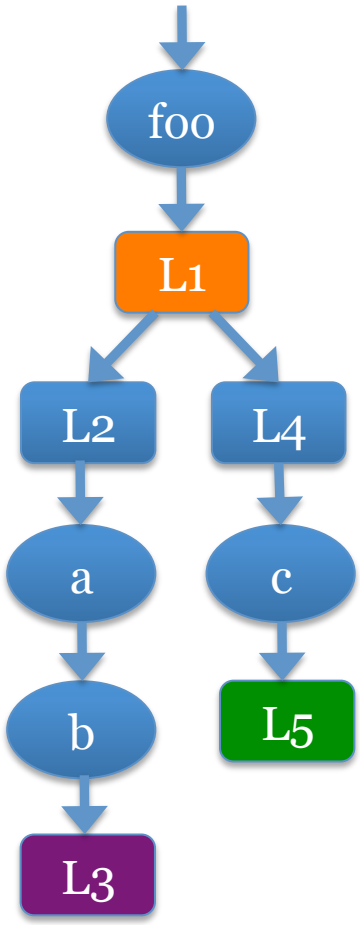
Understanding Memory Behavior

- Instruction schedule cost
  - Depends on mostly deterministic factors
    - Enables analytical time predictions
  - Depends only on local information
    - CPU back-ends do not have long term state
    - Enables computing lower bounds on instruction schedule cost
- Memory behavior
  - We can measure or predict number of cache misses
  - Latency hidden by overlap with code or with other memory accesses
  - Time penalty predictions require understanding memory parallelism
    - Really hard to predict actual memory parallelism analytically

Understanding Memory Behavior

- Lower bound on the number of cache misses?
  - Data reuse is not a local phenomenon
    - Cache state depends on program history
    - How much other data did we touch since previously accessing the same datum?
    - Where did we access the data previously?
    - Computing a lower bound is prohibitively expensive
  - "Compulsory misses" represent a loose lower bound
- Provide insight to the user on how data is reused
  - Inform what is required to shorten the reuse
  - Let the user decide if it is worth the effort

Understanding Data Reuse Patterns

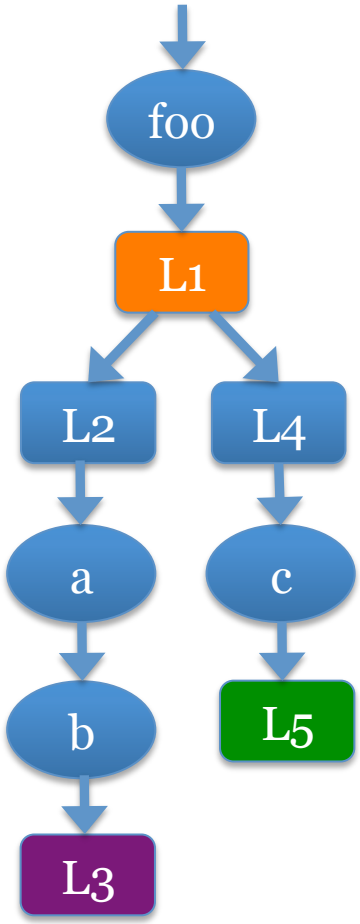


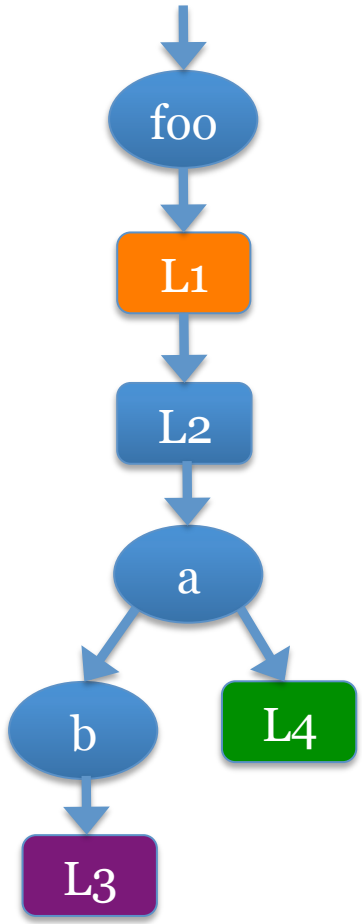
L3 – the source of the reuse; previous access to the datum

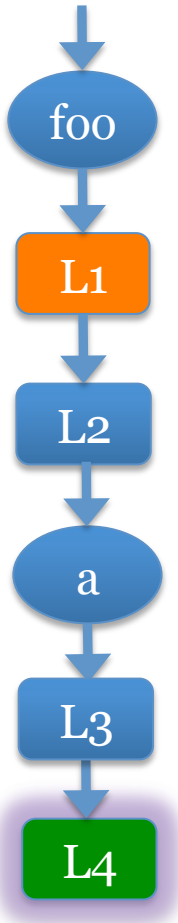
L5 – the destination of the reuse; current access to the datum

L1 – the carrier of the reuse; the least common ancestor (LCA) of the source and the destination scopes in the dynamic program context tree

Understanding Data Reuse Patterns





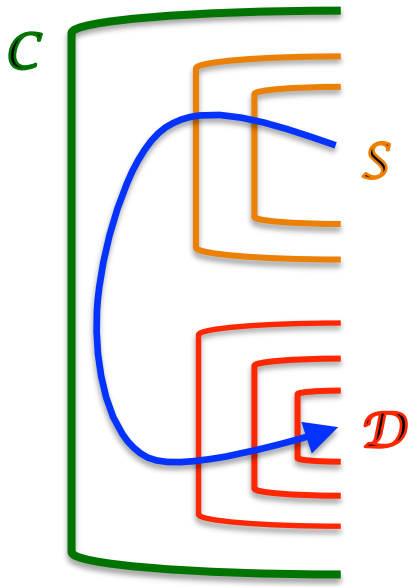


Carrier scope – the most recently entered program scope, which has stayed active during both the source and the destination of the reuse

Interpreting Data Reuse Information

S: source scope, $\mathcal{D}$: destination scope, C: carrying scope of a reuse pattern

- Reuse carried within the same iteration of the carrier scope (also same invocation of a routine body)
  - S and $\mathcal{D}$ must be the same scope as C (reuse between different statements), or in disjoint loop nests or routines

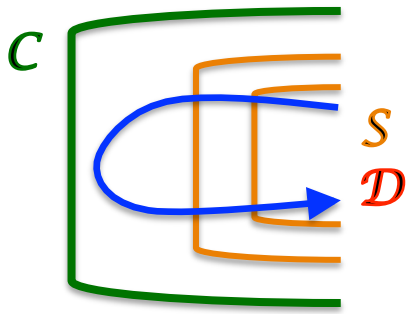


- If S, $\mathcal{D}$ and C are in the same routine
  - fuse S and $\mathcal{D}$
- S and/or $\mathcal{D}$ are in routines called from C, e.g. reuse between different sub-steps of a computation
  - strip-mine S and $\mathcal{D}$ with the same stripe; promote the loops over stripes outside of C and fuse them
  - the further removed the carrying scope from S and $\mathcal{D}$, the harder it is to shorten the reuse

Interpreting Data Reuse Information

S: source scope, $\mathcal{D}$: destination scope, *C*: carrying scope of a reuse pattern

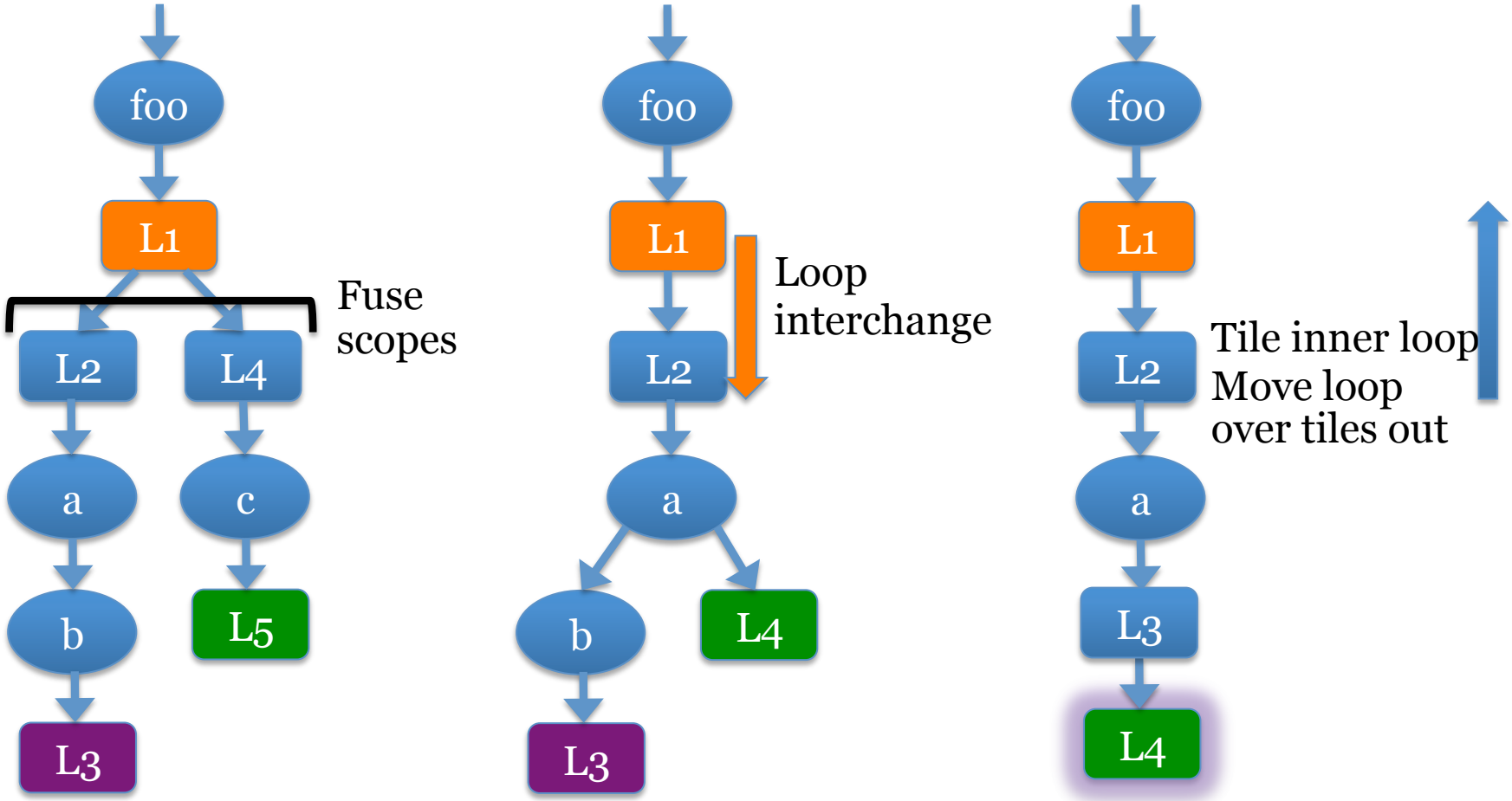
- Reuse carried across iterations of *C*
  - *S* = $\mathcal{D}$, or in the same loop nest



•*C* iterates over the array's inner dimension or array indexing independent of *C*
  - •apply loop interchange, or
  - •apply dimension interchange on the array(s), or
  - •apply blocking on a loop inside of *C* and move the loop over blocks outside of *C*

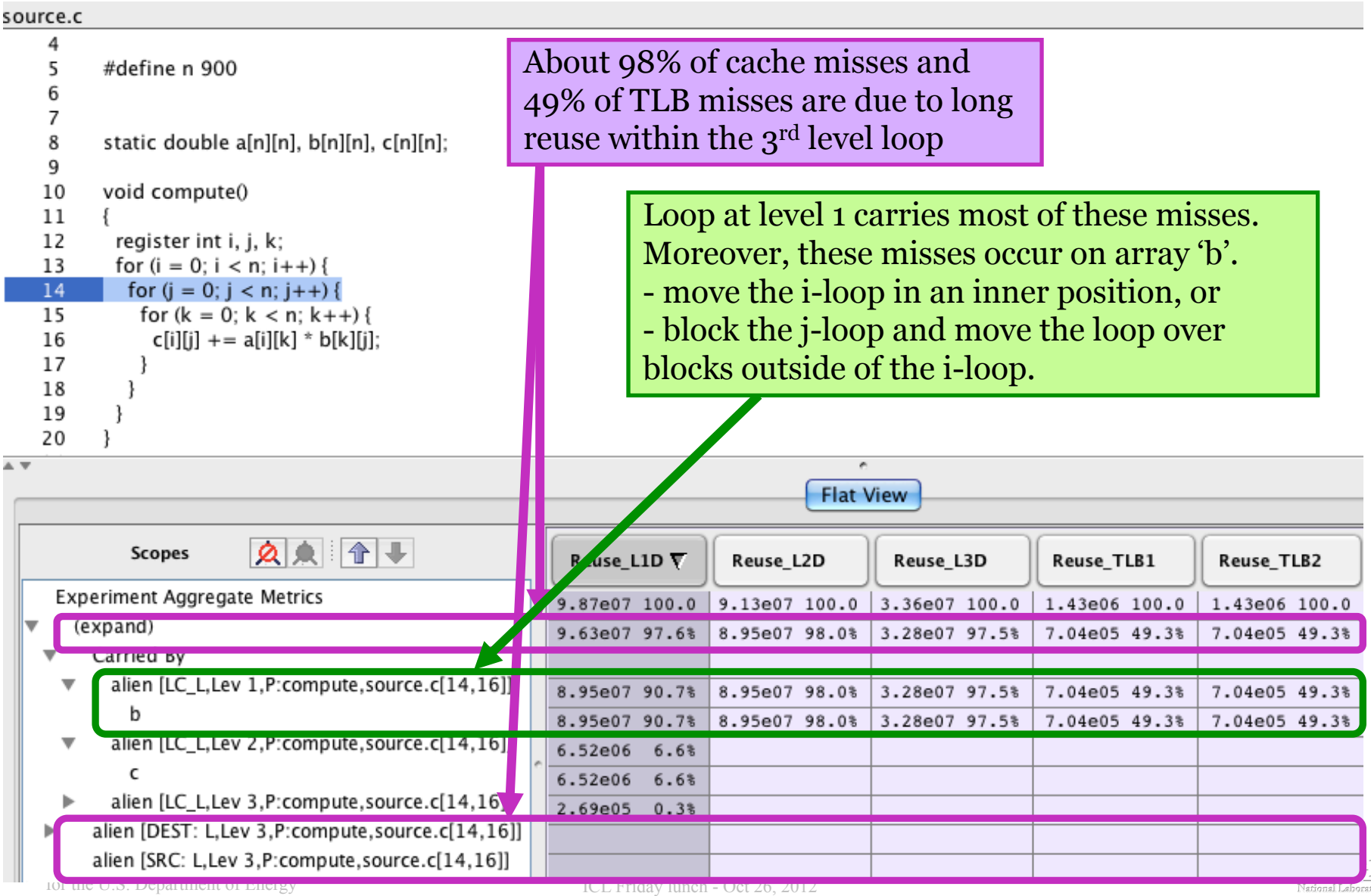
  - *S* and $\mathcal{D}$ in disjoint loop nests or routines
    - combination of the previous two cases; apply loop fusion + blocking/loop interchange
    - usually, it is harder to optimize
  - Large number of irregular misses and *S* = $\mathcal{D}$
    - apply data or computation reordering

Understanding Data Reuse Patterns



The farther removed the carrier scope, the more difficult to shorten the reuse

Data Reuse Example

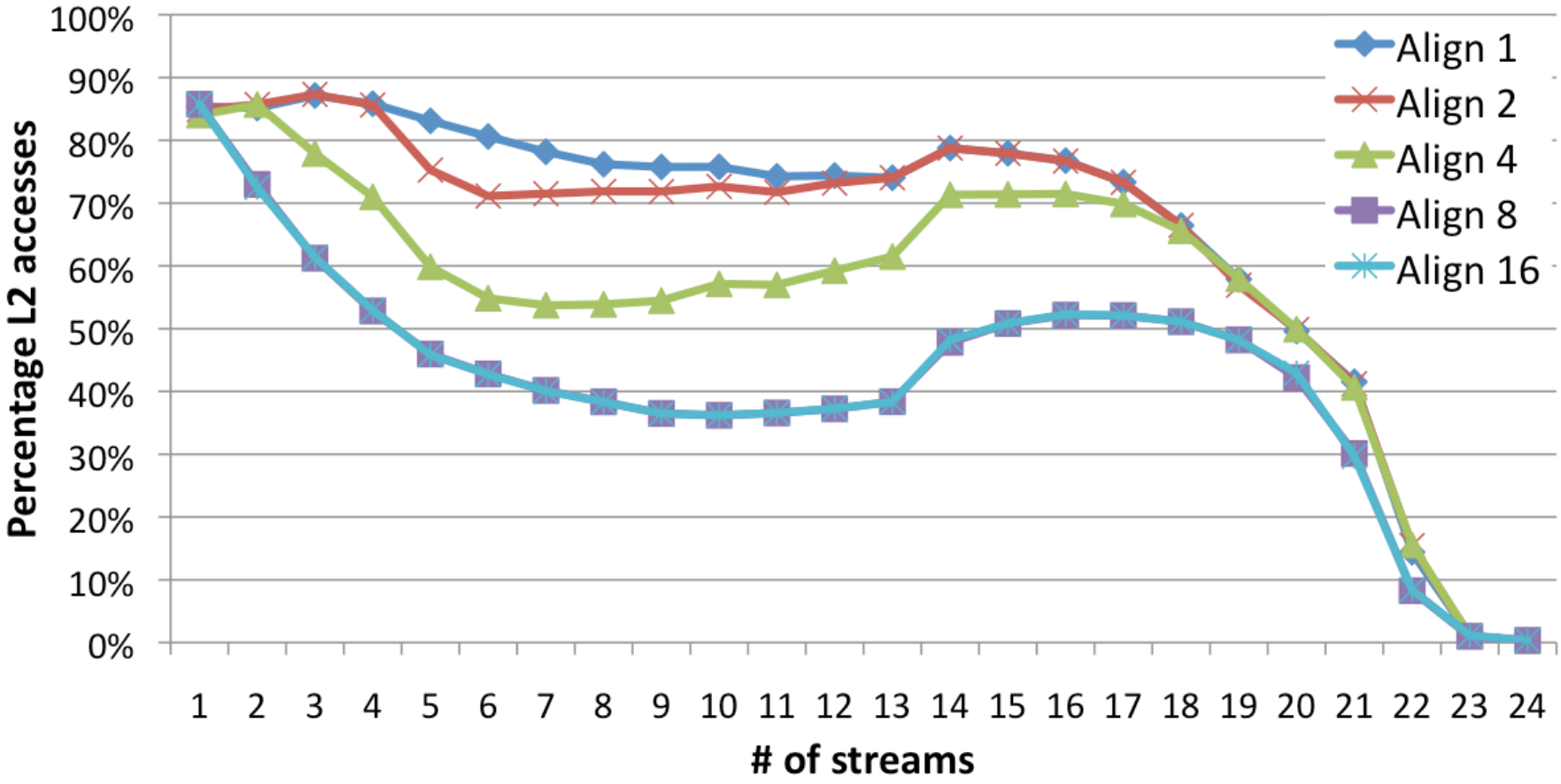


Understanding Prefetching Performance

- The application experiences cache misses that cannot be removed
  - Memory parallelism helps to hide latency
  - Prefetching increases both memory parallelism and overlapping with computation
- Hardware prefetchers
  - Effective at detecting and fetching streaming accesses
  - Can track only a limited number of concurrent streams
  - Maximum streaming stride is limited
- AMD 10H architecture has two prefetchers
  - DC and MC prefetchers

AMD 10H DC Prefetcher

- DC prefetcher effectiveness vs. # of streams

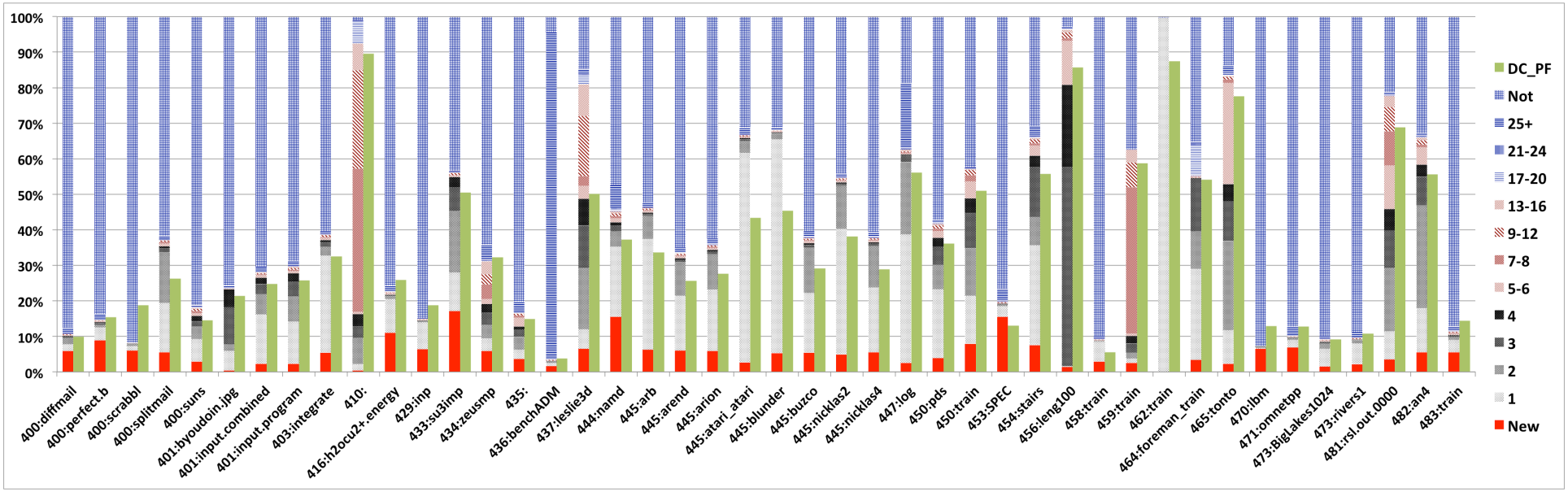


Application Streaming Concurrency

- PIN based tool to understand logical data streams in applications
- Classifies memory accesses as
  - Non-streaming
  - Starting a new stream
  - Part of an existing stream
- For streaming accesses record number of concurrent live streams

SPEC CPU2006 Streaming Results

- Application streaming concurrency vs. AMD 10H DC hardware prefetching effectiveness

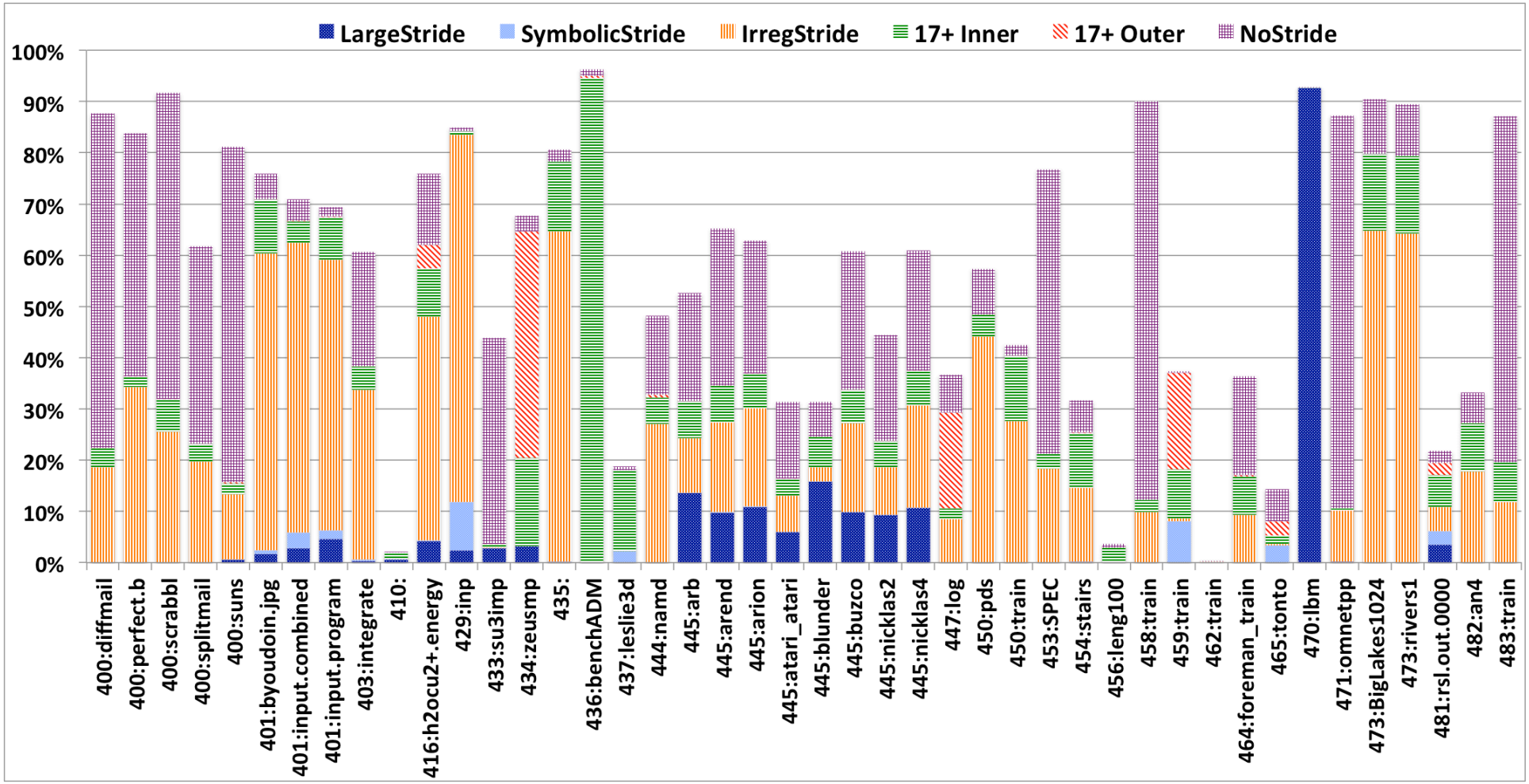


$$DC\_PF = \frac{L2\_REQ:HW\_PREF}{L2\_REQ:DATA}$$

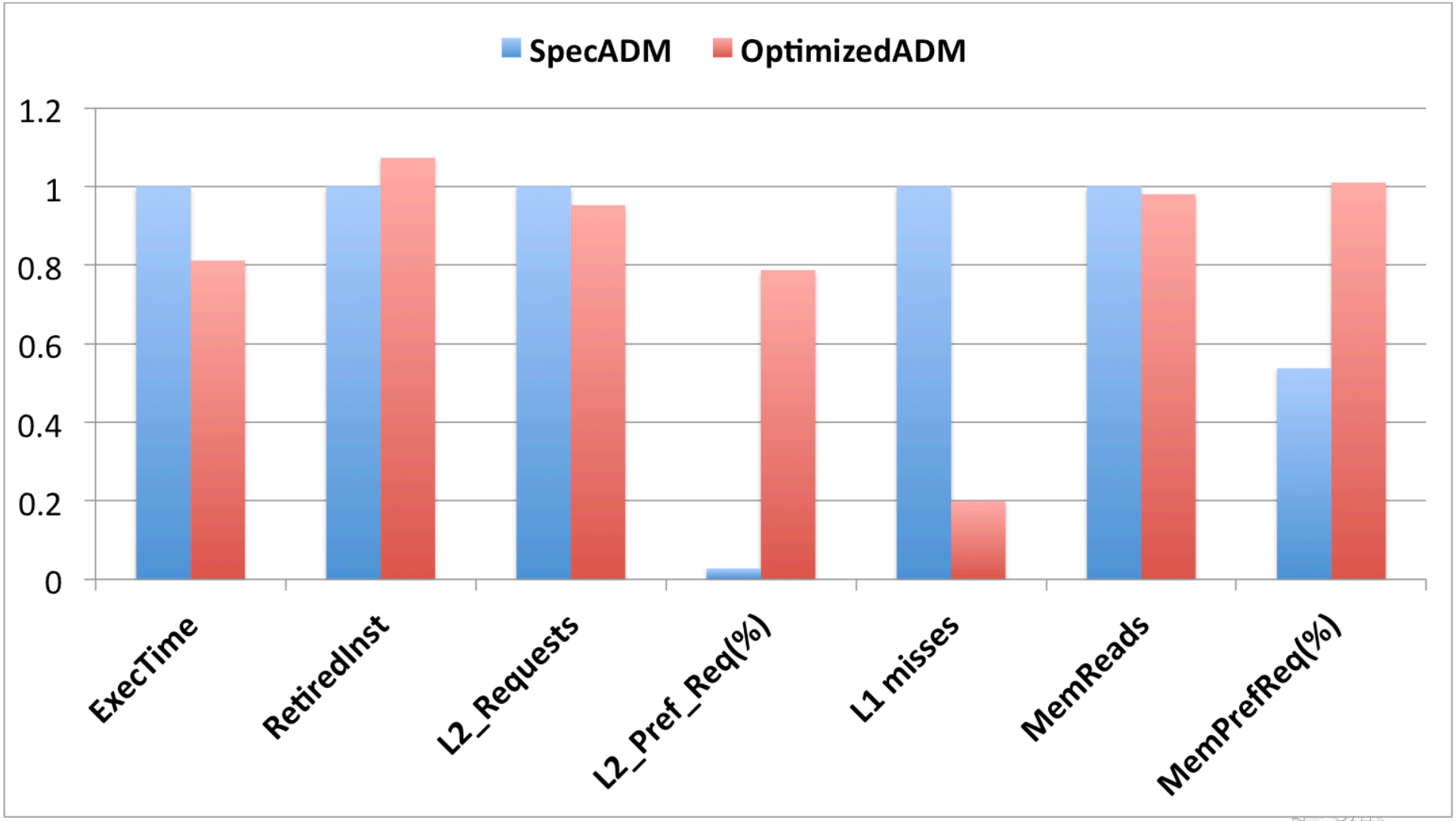
Identify Prefetch Preventing Conditions

- Non-streaming accesses
  - Irregular accesses
  - Strided accesses with a large stride
- Streaming accesses with large concurrency
  - Streams created by innermost loop
  - Streams created by an outer loop
- Identify memory references with the above four characteristics
  - Uses MIAMI static analysis

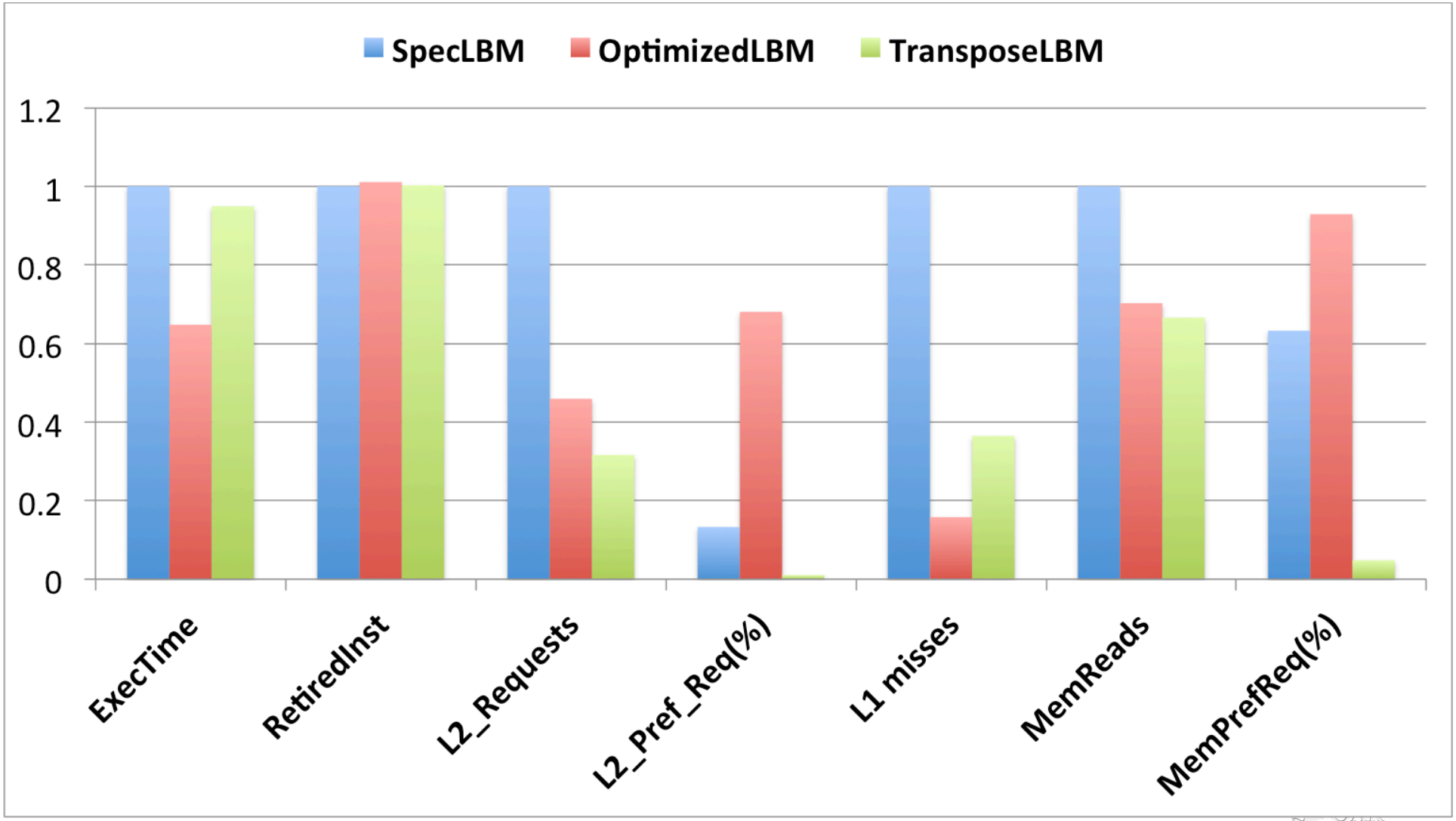
SPEC CPU2006 Results



436.CactusADM Prefetch Optimizations



470.LBM Prefetch Optimizations



Summary

- Most performance analysis tools pinpoint places where time is spent, or where cache misses occur
  - Explaining the "why" and determining if code can be improved are left to the user
    - Not an easy task most of the time
- MIAMI builds application-centric models
  - Uses machine model to understand code-architecture interactions
  - Outputs metrics that expose various performance bottlenecks, with bounds on performance improvement
  - Users can understand the root cause and what code changes are typically needed in each situation
  - Legality of code transformations and effort worthiness are left to the user to determine

Summary

- Challenges
  - How to most effectively expose performance details to the user without going into data overload
  - How to prioritize between the different types of performance bottlenecks
    - Automatic ranking of performance improvement opportunities