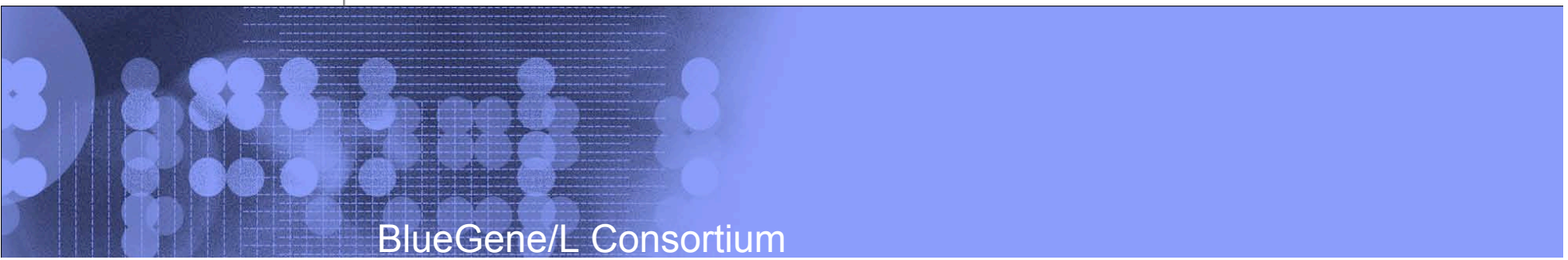
A Tutorial on BG/L Dual FPU Simdization

Alexandre Eichenberger, Rohini Nair, and Peng Wu / TPO

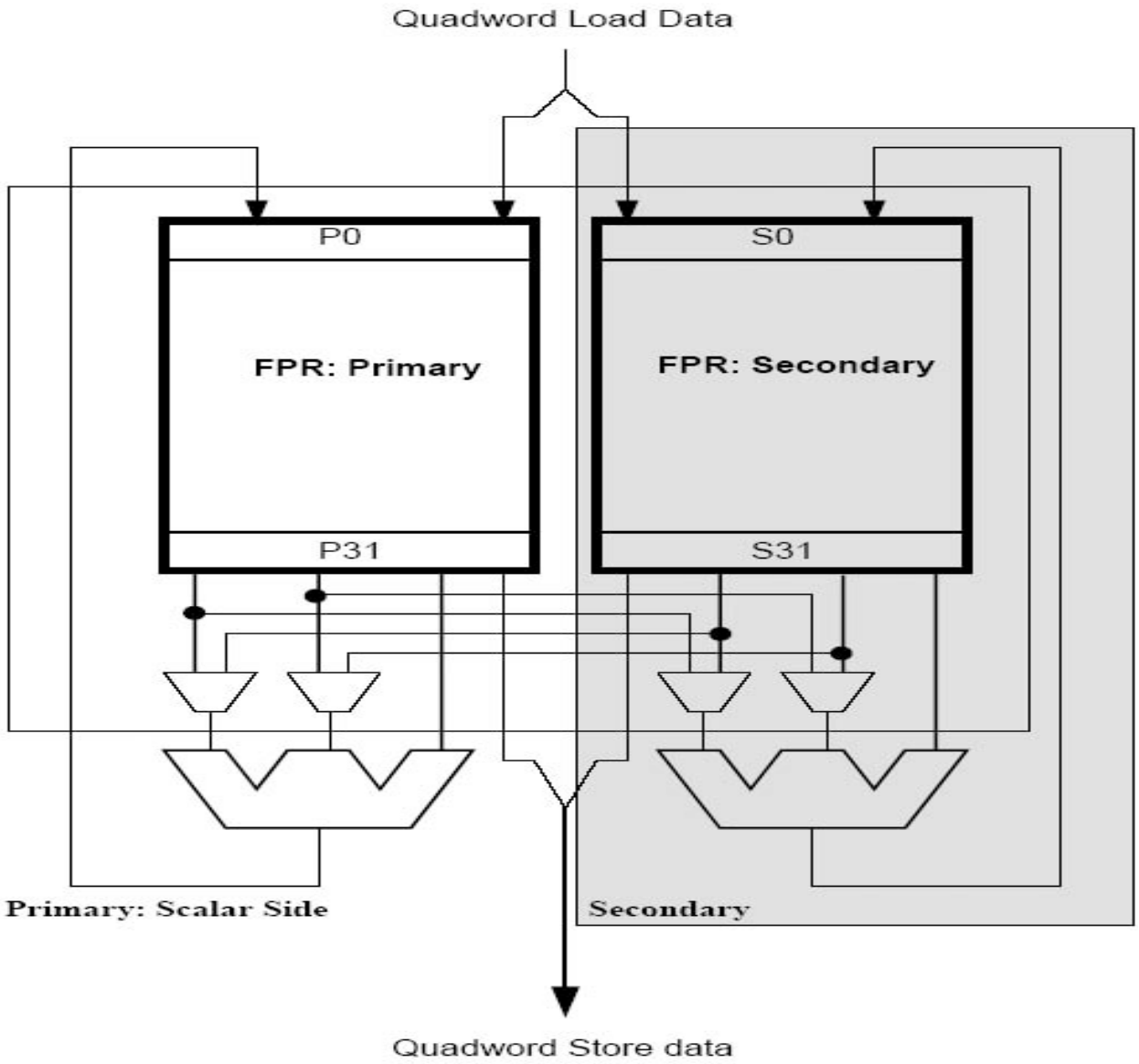
Mark Mendell / Tobey



Outline

- Background
- How to use the compiler
- Diagnostic info and tuning
- Alignment handling
- Experimental results

BlueGene/L Dual Floating Point Unit



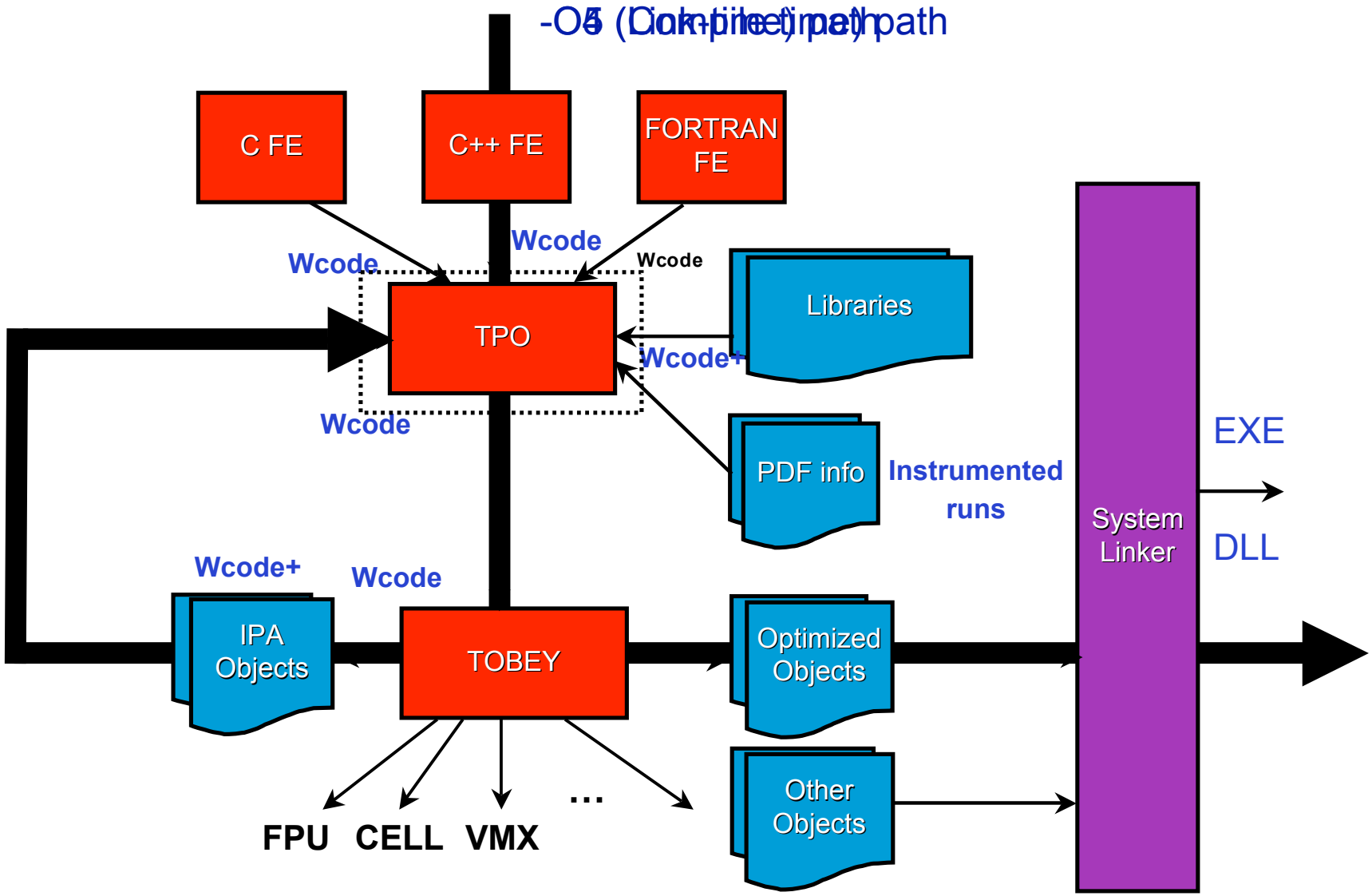
Architecture Constraints of Dual FPU Unit

- Only stride-one memory accesses use full bandwidth
  - "stride-one" means "stored consecutively in memory"
  - lower bandwidth for non-stride-one accesses (non major, a[2i+1], indirect access
- Access efficiently only 16-byte aligned data
  - a[i] = b[i] +c [i] vs. a[i] = b[i+1] + c[i]
- Misaligned data can be loaded using cross-instructions
  - data realignment pattern is encoded in the instructions,
  - makes handling of runtime alignment difficult
- Non-uniform instruction set for dual unit
  - double precision floating point only
- Simdization → SIMD vectorization

Outline

- Background
- How to use the compiler
- Diagnostic info and tuning
- Alignment handling
- Experimental results

The XL Compiler Architecture



Where does Simdization Occur?

- Some occurs in TPO (high-level inter-procedural optimizer)
  - computations that stream over double floats
  - TPO does most loop level/inlining/cloning optimizations
- Some occurs in Tobey (low-level backend optimizer)
  - complex arithmetic on double floats is an ideal target
  - other non-regular double floats are also packed
  - Tobey does most code motion/scheduling/machine specific optimizations

This talk focus mainly on TPO level simdizatic

3-Step Program to Enable Simdization

- Compile for the right machine
  - -qarch=440d –qtune=440 (in this order)
- Turn on the right optimizations
  - -O5 (link-time, whole-program analysis & simdization)
  - -O4 (compile time, limited scope analysis & simdization)
  - -O3 –qhot=simd (compile time, less optimization & simdization)
- Tune your programs
  - use TPO compiler feedback (-qxflag=diagnostic) to guide you
  - help the compiler with extra info (directive/pragmas)
  - modify algorithms (hint: more stride-one memory accesses)

2-Step Program to Disable Simdization

- Compile for the wrong machine
  - to completely disable simdization: -qarch=440 –qtune=440
- Turn off the right optimizations
  - compile for –qarch=440d –qtune=440
  - disable TPO simdization (keep Tobey simdization)
    - for a loop: #pragma nosimd | !IBM* NOSIMD
    - completely: -qhot=nosimd
  - disable Tobey simdization (keep TPO simdization)
    - not supported, may not work, try at your own risks
    - completely: -qxflag=nhummer:ncmplx

green is for C | red is for for

5 to 7 Steps to Help Us

- Found a correctness bug?
  - play with options to see at which level it fails
  - isolate the error (code as small as possible)
  - simdize only the loop that fails
  - give us all the info (all sources, header, make files, compiler options)
  - report the problem

- Found a performance bug?
  - test the correctness of your code (verify results if possible)
  - try to estimate a good lower bound (number of mem/fma/…)
  - apply above 5 steps

Outline

- Background
- How to use the compiler
- Diagnostic info and tuning
- Alignment handling
- Experimental results

Examples of TPO Simdization Success Diagnostic

Examine loop <1> on line 12
(simdizable) []

Examine loop <2> on line 20
(simdizable) [misalign(compile time) shift(3 compile-time)]

Examine loop <3> on line 26
(simdizable) [misalign(runtime)][versioned(relative-align)]

TPO Diagnostic Information on Success

- Simdizable loops
  - diagnostic reports "`(simdizable)[features][version]`"

- [feature] further characterizes simdizable loops
  - "`misalign(compile time store)`": simdizable loop with misaligned access
  - "`shift(4 compile time)`": simdizable loop with 4 stream shift inserted
  - "`priv`": simdizable loop has private variable
  - "`reduct`": simdizable loop has a reduction construct

- [version] further characterizes if/why versioned loops where created
  - "`relative align`": versioned for relative alignment
  - "`trip count`": versioned for short runtime trip count

-qxflag=diagnostic report on TPO Simdization o

Examples of TPO Simdization Failure Diagnostic

```
Examine loop <id=1> on line 1647
    not single block loop
(non_simdizable)

Examine loop <id=1> on line 2373
    dependence at level 0 from  ( 0 73 100 )
(non_simdizable)

Examine loop <id=2> on line 2356
    dependence due to aliasing
(non_simdizable)

Examine loop <1> on line 4
    no intrinsic mapping for <ADD int>: a[]0[$.CIV0] + b[]0[$.CIV0]
(non_simdizable)
```

TPO Diagnostic Information on Failure

- Alignment:
  - "`misalign(….)`": simdizable loop with misaligned accesses
    - "`non-natural`": non naturally aligned accesses
    - "`runtime`": runtime alignment

⇒Action:

- align data for the compiler: double a[256] **__attribute__((aligned(16));**
  - all dynamically allocated memory (malloc,alloca) are 16-byte aligned
  - all global objects are 16-byte aligned
  - inside struct / common block, you are on your own
- tell the compiler it's aligned: __alignx(16, p); | call alignx(16,a[5]);
  - like a function call, no code is issued
  - can be placed anywhere in the code, preferably close to the loop
- tell compiler that all references are naturally aligned
  - -qxflag=simd_nonnat_aligned
- use array references instead of pointers when possible

green is for C | red is for for

TPO Diagnostic Information on Failure (Cont')

- Structure of the loop
  - "`irregular loop structure (while-loop)`" (handle only for/do loops)
  - "`contains control flow`": (no if/then/else allowed)
  - "`contains function call`": (no function calls)
  - "`trip count too small`": (short loops not profitable)

⇒Action:

- convert while loops into do loops when possible
- limited if conversion support
  - handle best if-then-else with same array defined on both sides
  - can try data select
- inline function calls
  - automatically (-O5 more aggressive, use inline pragma/directives)
  - manually

TPO Diagnostic Information on Failure (Cont')

- Dependence
  - "`dependence due to aliasing`"

⇒Action:

  - help the compiler with aliasing info
    - use -O5 (does interprocedural analysis)
    - tell the compiler when its disjoint: #pragma disjoint (*a, *b)
    - use fewer pointers when possible

- Scalar references
  - "`non-simdizable reductions`"
  - "`non-simdizable scalar var`"

⇒Action:

  - reductions that are used in the loops can not be simdized

TPO Diagnostic Information on Failure (Cont')

- Array references
  - "`access not stride one`":
  - "`mem accesses with unsupported alignment`"
  - "`contains runtime shift`"

⇒Action:

- interchange the loops to enhance stride-one, when possible
- sometime TPO may interchange loops for you, in a way that you don't want
  - disable unimodular transformation: -qxflag=nunimod
- runtime alignment not feasible on BG/L
  - compiler version the loop
  - one of the two version may report "`(non-simdizable)`"

TPO Diagnostic Information on Failure (Cont')

- Pointer references
  - "non normalized pointer accesses"

⇒Action:

  - simple pointer arithmetic should be well tolerated
  - otherwise, try using arrays

- Native Mapping and native data types
  - "non supported vector element types"
  - "no intrinsic mapping for <op type>:"

⇒Action:

  - none: BG/L supports only double precision floating point SIMD

Other Tuning

- Loop unrolling can interact with simdization
  - there is some support for simdizing unrolled loop, but its harder
  - try to not manually unroll the loop for better TPO simdization
  - unroll directive: #pragma nounroll | #pragma unroll(2)
- Math libraries:
  - currently, we don't simdize sqrt,…
    - we split the loop, simdize the one without sqrt
    - you can do the same, short loop that compute all the sqrt, store in a temp ar
    - use optimized libraries to compute vectors of sqrt
    - then use it in the old loop, that one will simdize
- Use literal constant loop bounds
  - e.g. #define when possible
- Tell compiler not to simdize a loop if not profitable (e.g., trip count too low)
  - #pragma nosimd (right before the innermost loop)

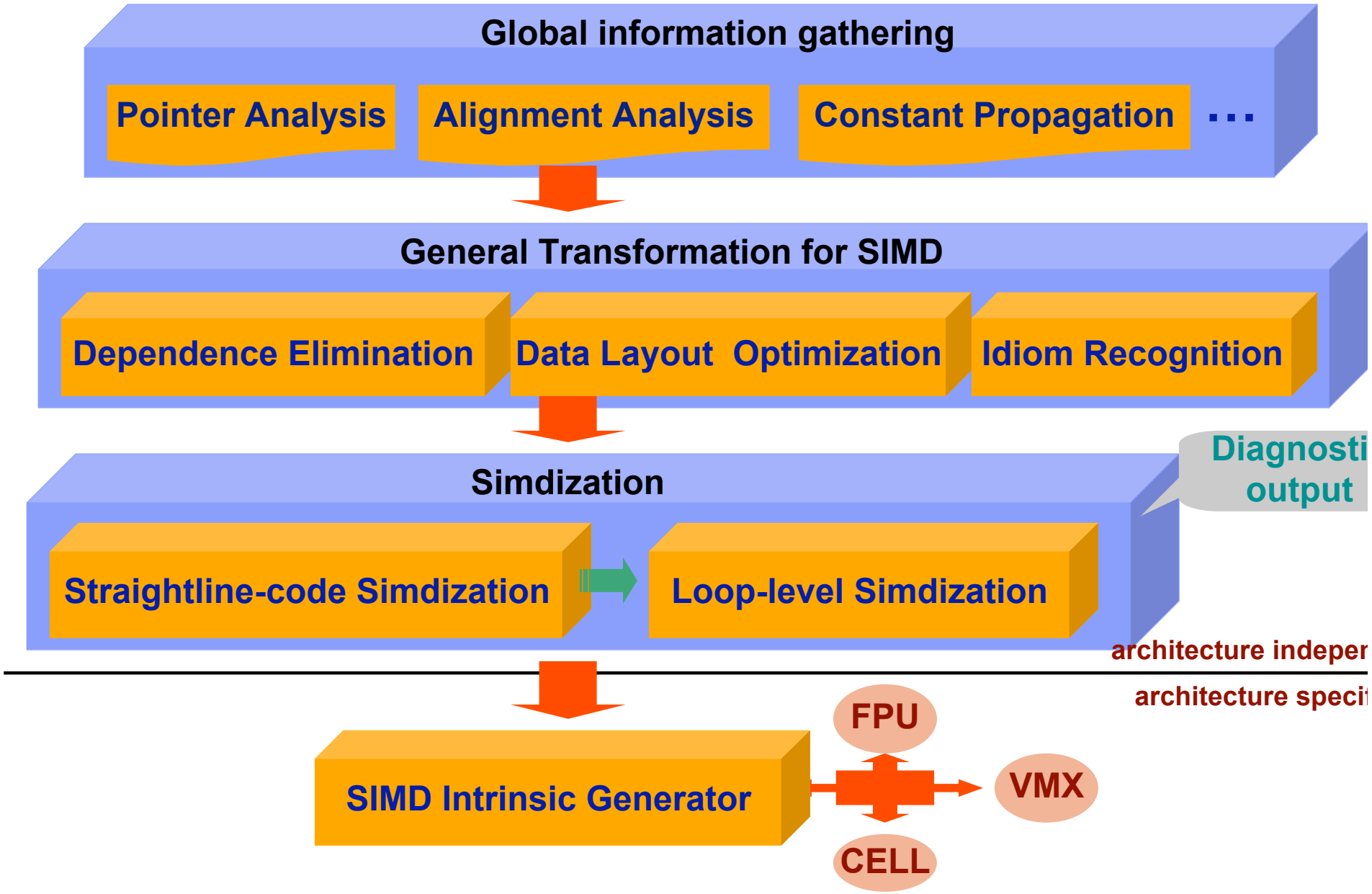
More pragma/directive info

- Some generally available info is here
  - http://publib.boulder.ibm.com/infocenter/comphelp/index.jsp
  - some useful links on this site:
    - Fortran/Language references/Directives
    - Fortran/Language references/Intrinsic procedures/Hardware specific
    - C/Language references/Preprocessor directives/Pragma directives

Outline

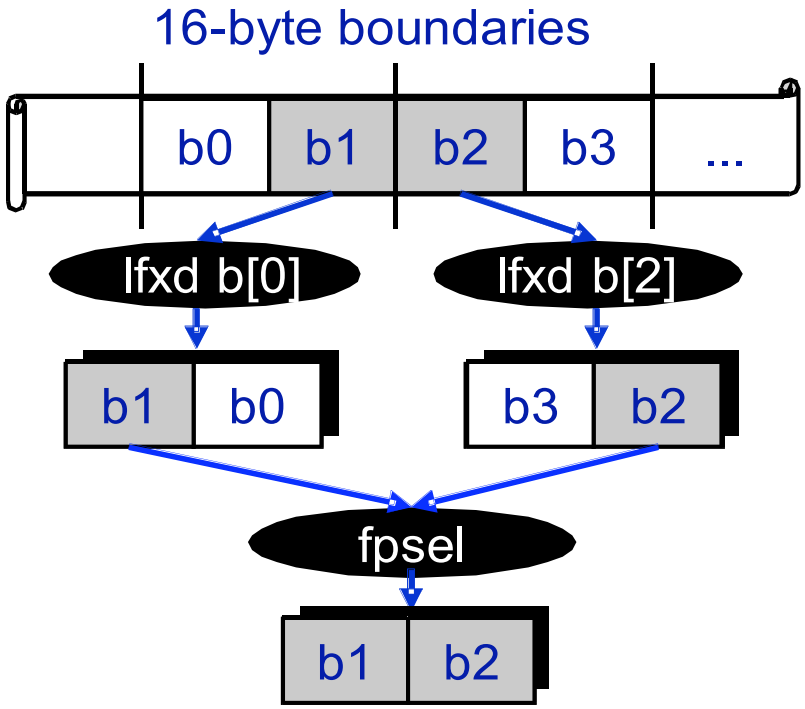
- Background
- How to use the compiler
- Diagnostic info and tuning
- Alignment handling
- Experimental results

A Unified Simdization Framework



How to load from misaligned memory?

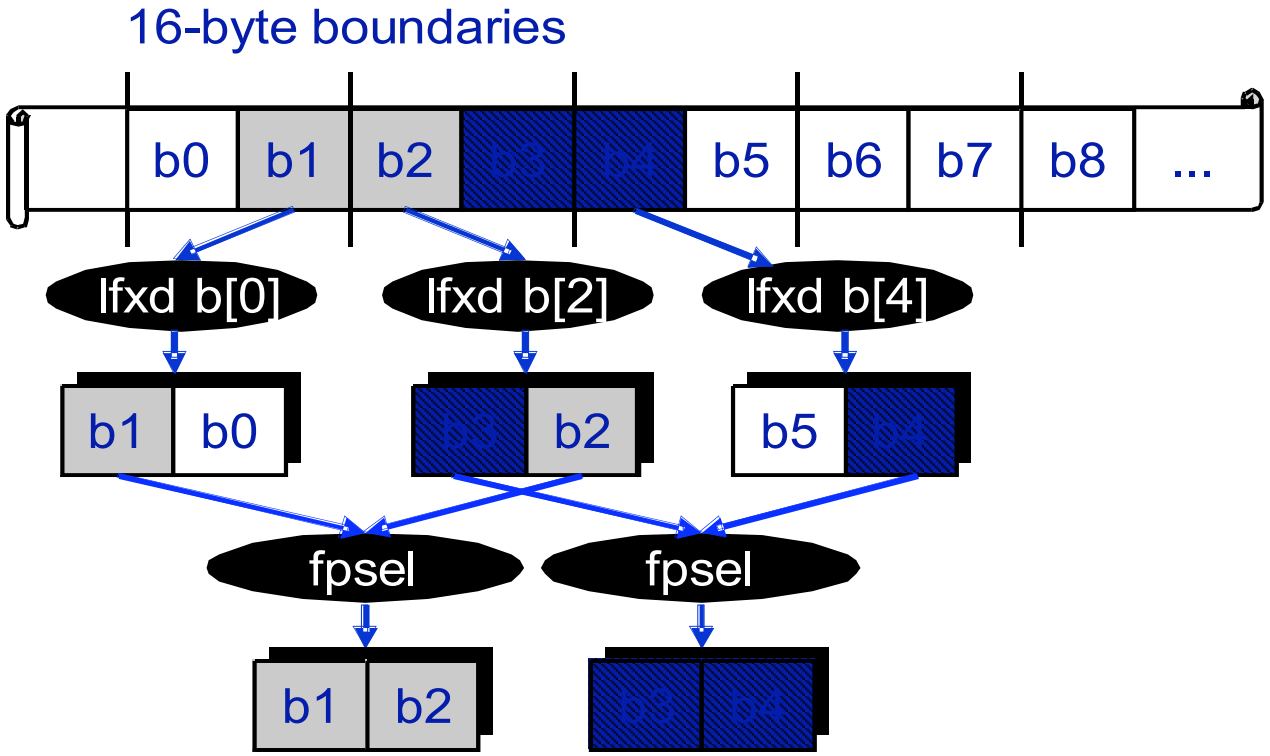
- Load one misaligned quad:



1 misaligned-quad load costs 2 aligned-quad cross-loads + 1 select

How to access misaligned memory (cont')?

- Load multiple consecutive misaligned quad data:
  - reuse quad load-across



1 misaligned-quad load costs on avg. 1 aligned-quad cross-loads + 1 sele

When misalignment handling is needed?

- for (i=0; i<100; i++) a[i] = b[i] + c[i+1] ;
  - aligned: a[i], b[i]
  - misaligned : c[i+1]
  - action: realign c[i+1]

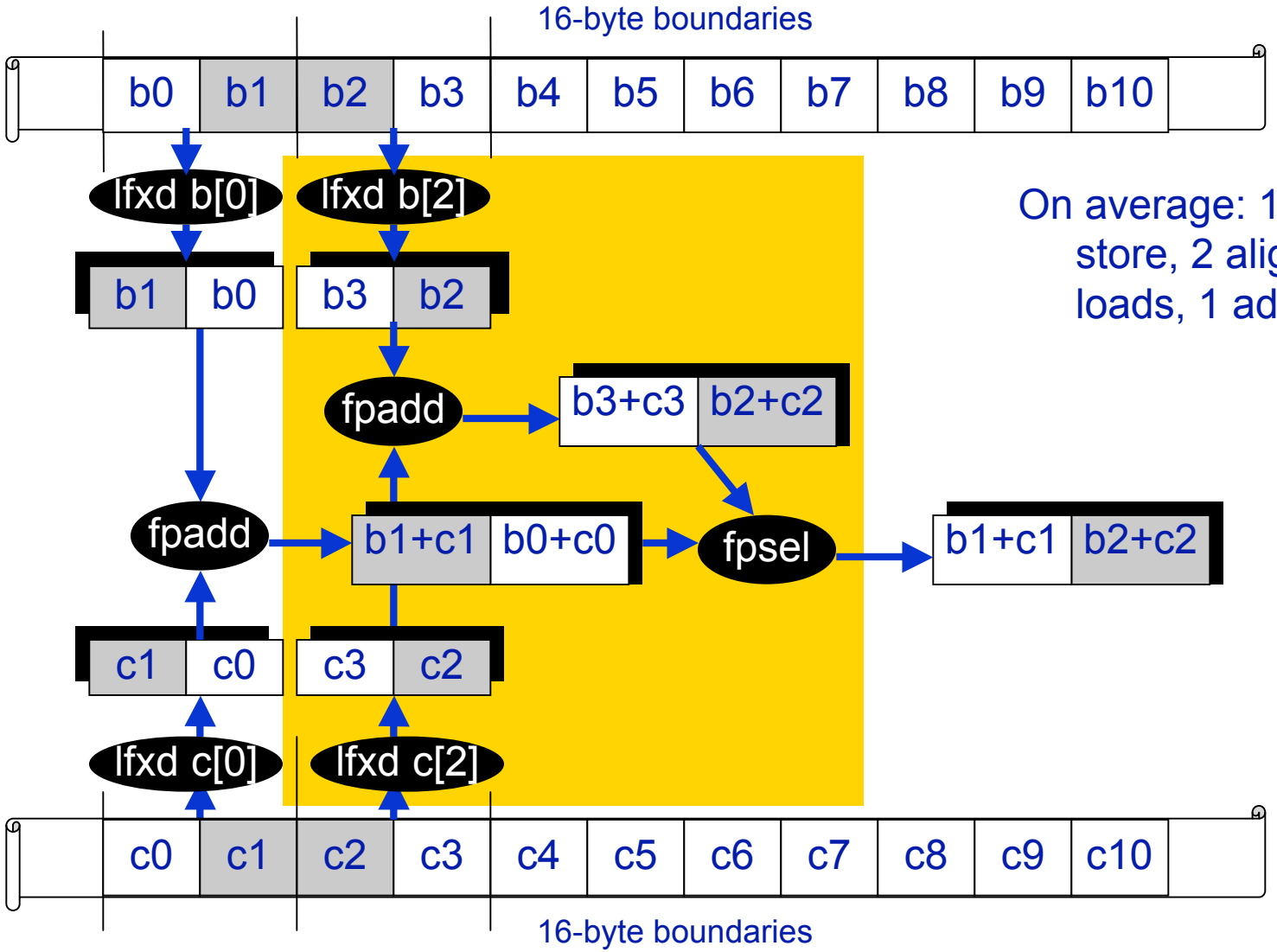
- for (i=0; i<100; i++) a[i+1] = b[i+1] + c[i+1];
  - misaligned, but relatively aligned: a[i+1], b[i+1], c[i+1]
  - action: peel first iteration

- for (i=0; i<100; i++) a[i+1] = b[i+1] + c[i];
  - misaligned, but relatively aligned: a[i+1], b[i+1]
  - aligned: c[i] is aligned
  - action: peel first iteration, realign c[i]

a[0], b[0], c[0] assumed alig

Minimizing data reorganization overhead

- for (i=0; i<100; i++) a[i] = b[i+1] + c[i+1] ;

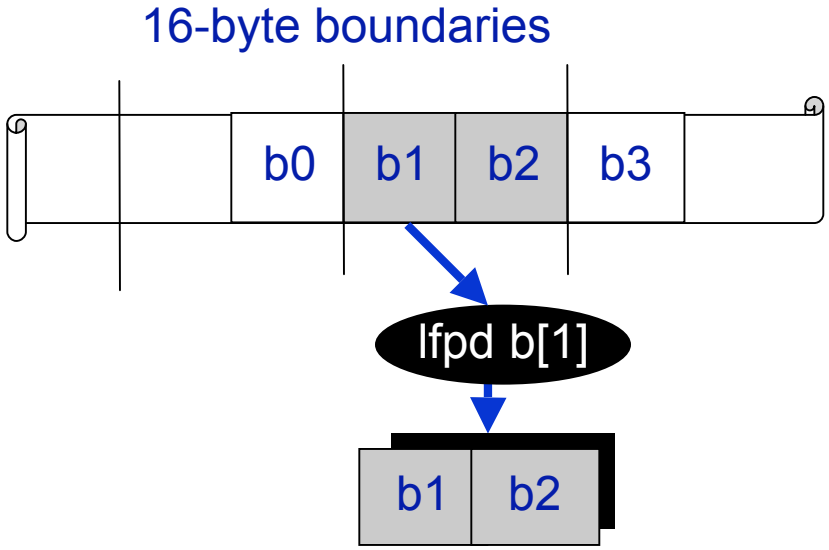


On average: 1 aligned store, 2 aligned cross-loads, 1 add, 1 select

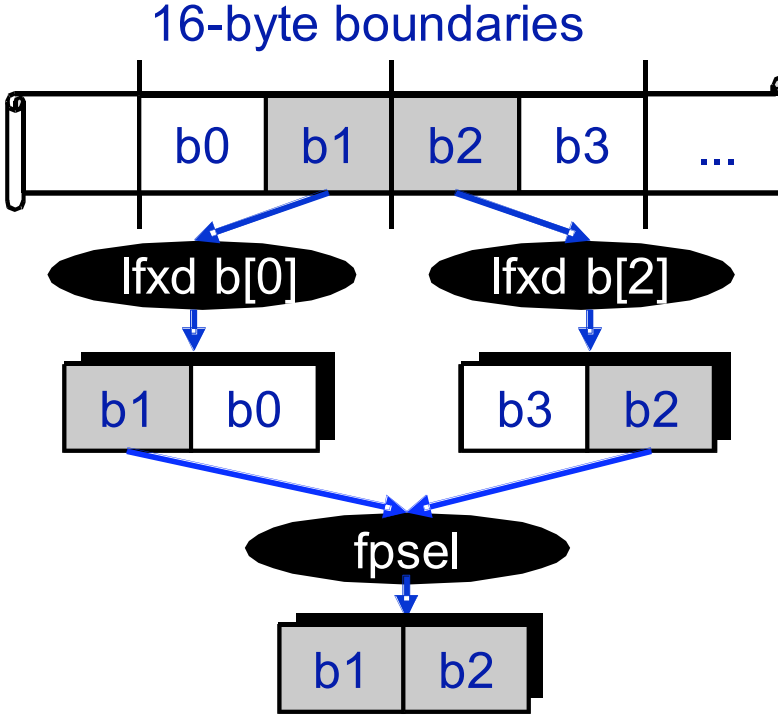
Issues with Runtime Alignment

- Depending on the alignment, different code sequences may be generated
  - When alignment is runtime, the compiler does not know which code sequence to generate

1. when b[1] is aligned



2. when b[1] is misaligned



Versioning for relative alignment

- Solution to loops with runtime alignment
  - versioning for relative alignment

- When versioning is needed?
  - for (i=0; i<100; i++) a[i+n] = b[i+1+n] + c[i+1+n];
    - n is runtime loop invariant
    - a[i+n], b[i+1+n], c[i+1+n]: runtime alignments, but relatively aligned
    - no versioning is necessary

  - for (i=0; i<100; i++) p[i] = q[i] + r[i];
    - p, q, and r are pointers, alignment & relative alignment unknown
    - versioning is necessary
    - bet on them being relatively aligned

      if ((p-q) mod 16 == 0 && (p-r) mod 16 == 0) $\Rightarrow$ SIMD version

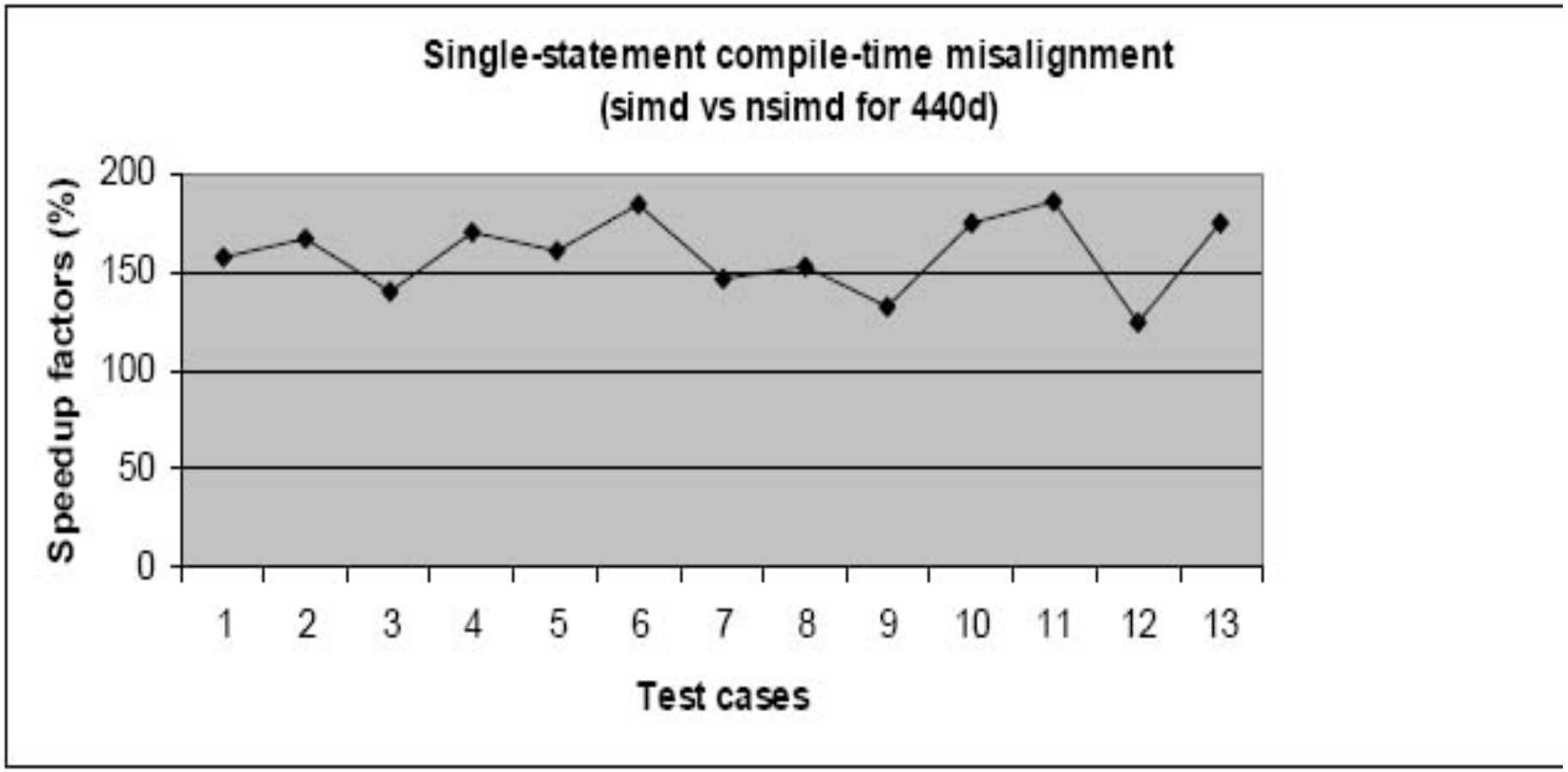
Outline

- Background
- How does the compiler simdize
- Diagnostic info and tuning
- Alignment handling
- Experimental results

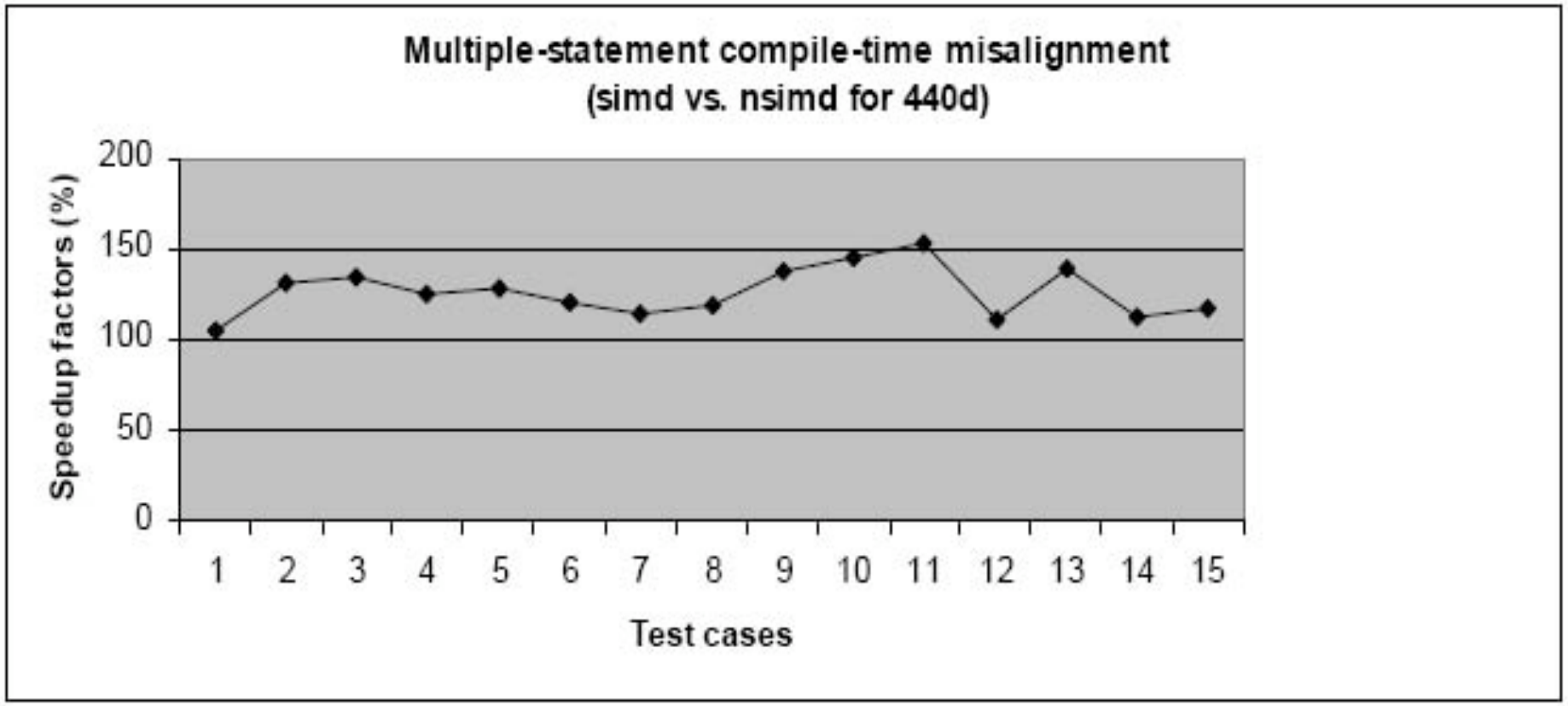
Evaluation of Alignment Handling

- Measurements on a set of kernel loops
  - Harmonic means of a set of 50 loops with identical characteristics
    - 3 loads, 2 adds, 1 store per statement
    - 3 statements per loop for multiple statement loops
    - 500 iterations per loop
    - Randomly generated memory alignments

Single-statement loop with compile-time misalignment



Multiple-statement loops with compile-time misalignment



HPCC/StreamC Simdization performance

- Compiler simdizes all 4 stream tests, speedup factor 1.39 ~ 1.97.