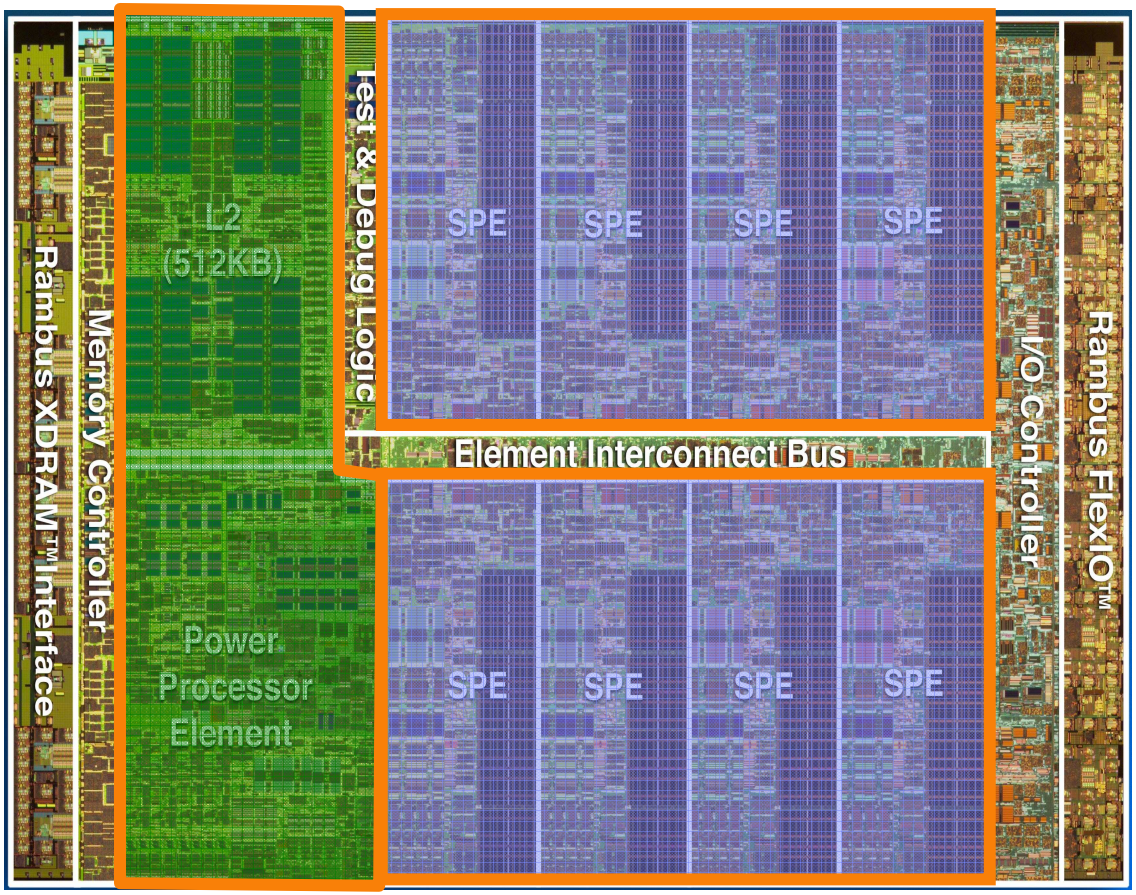
Optimizing Compiler for the Cell Processor

Alexandre Eichenberger, Kathryn O'Brien, Kevin O'Brien,
Peng Wu, Tong Chen, Peter Oden, Daniel Prener,
Janice Shepherd, Byoungro So, Zehra Sura, Amy Wang,
Tao Zhang, Peng Zhao, and Michael Gschwind

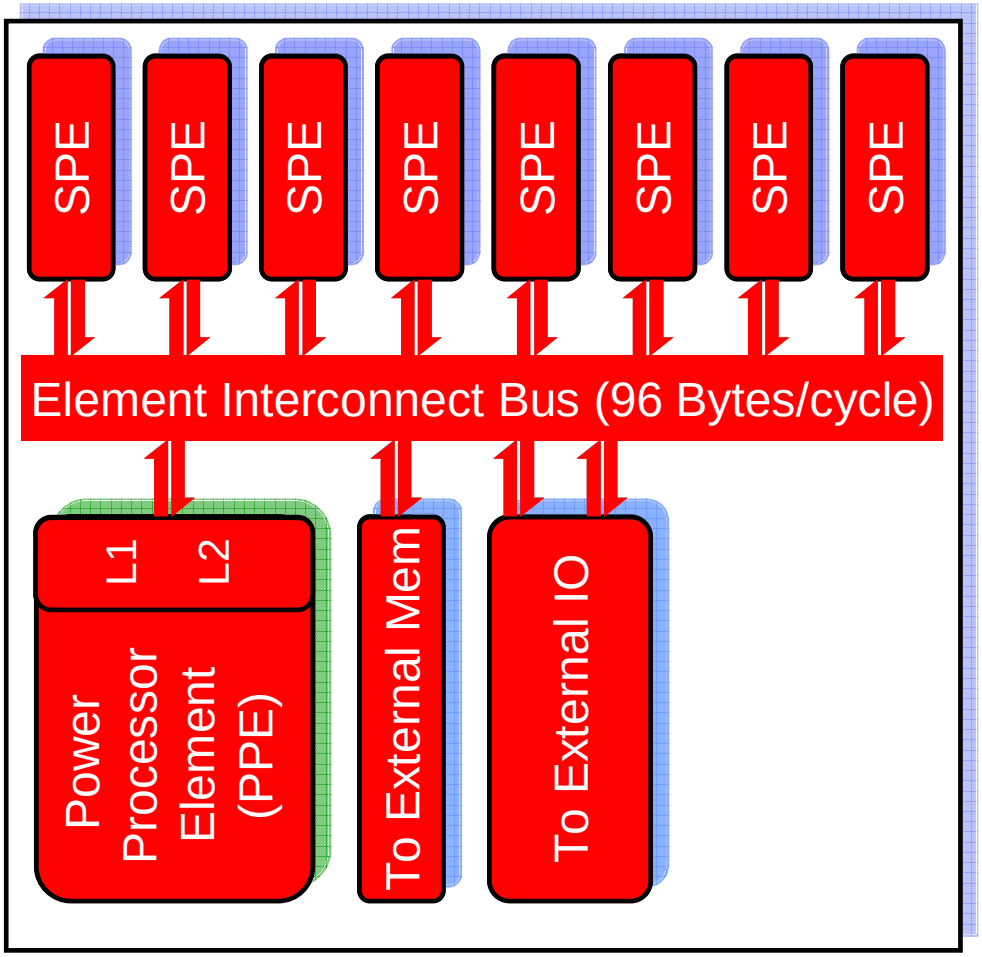
www.research.ibm.com/cellcompiler/compiler.htm

Cell Broadband Engine



- **Multiprocessor on a chip**
  - 241M transistors, 235mm^2
  - 200 GFlops (SP) @3.2GHz
  - 200 GB/s bus (internal) @ 3.2GHz
- **Power Proc. Element (PPE)**
  - general purpose
  - running full-fledged OSs
- **Synergistic Proc. Element (SPE)**
  - optimized for compute density

Cell Broadband Engine Overview



- **Heterogeneous, multi-core engine**
  - 1 multi-threaded power processor
  - up to 8 compute-intensive-ISA engines
- **Local Memories**
  - fast access to 256KB local memories
  - globally coherent DMA to transfer data
- **Pervasive SIMD**
  - PPE has VMX
  - SPEs are SIMD-only engines
- **High bandwidth**
  - fast internal bus (200GB/s)
  - dual XDR™ controller (25.6GB/s)
  - two configurable interfaces (76.8GB/s)
  - numbers based on 3.2GHz clock rate

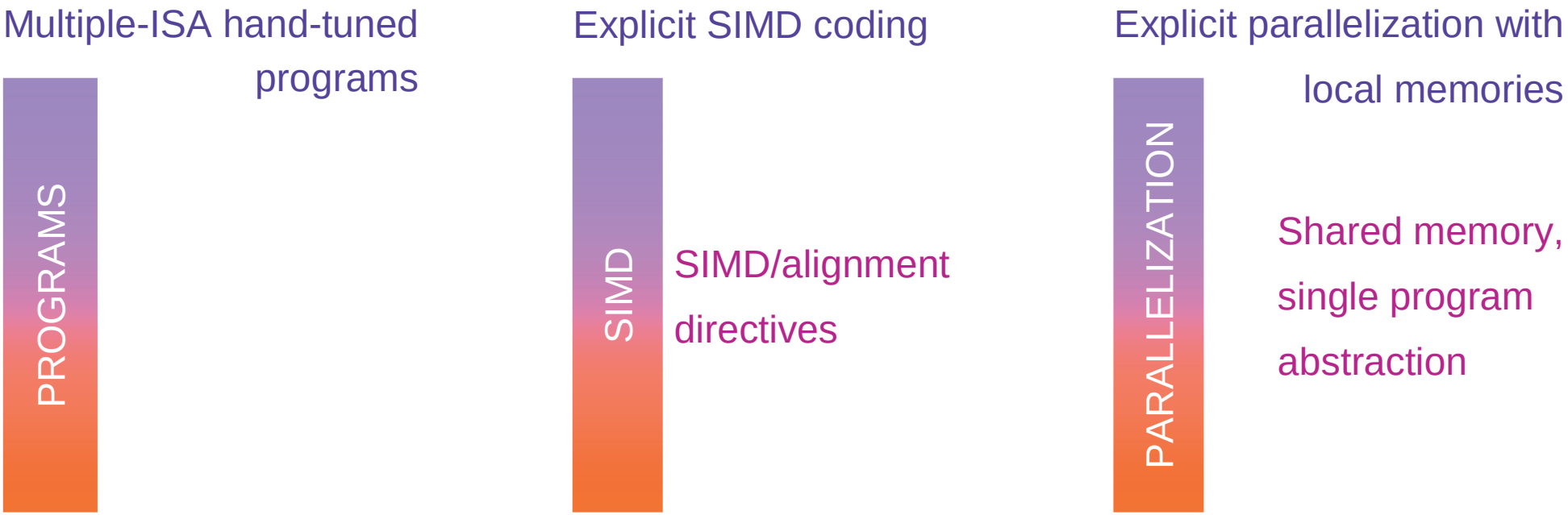
8 Bytes (per dir)

16Bytes (one dir)

128Bytes (one dir)

Supporting a Broad Range of Expertise to Program Cell

Highest performance with help from programmers



Highest Productivity with fully automatic compiler technology

Outline

Part 1:
Automatic SPE tuning

Multiple-ISA hand-tuned programs

Automatic tuning for each ISA

Part 2:
Automatic simdization

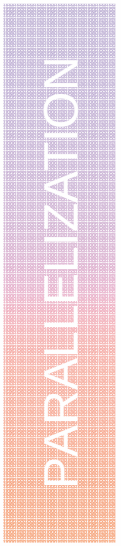
Explicit SIMD coding

SIMD/alignment directives

Automatic simdization

Part 3:
Shared memory & single program abstr.

Explicit parallelization with local memories

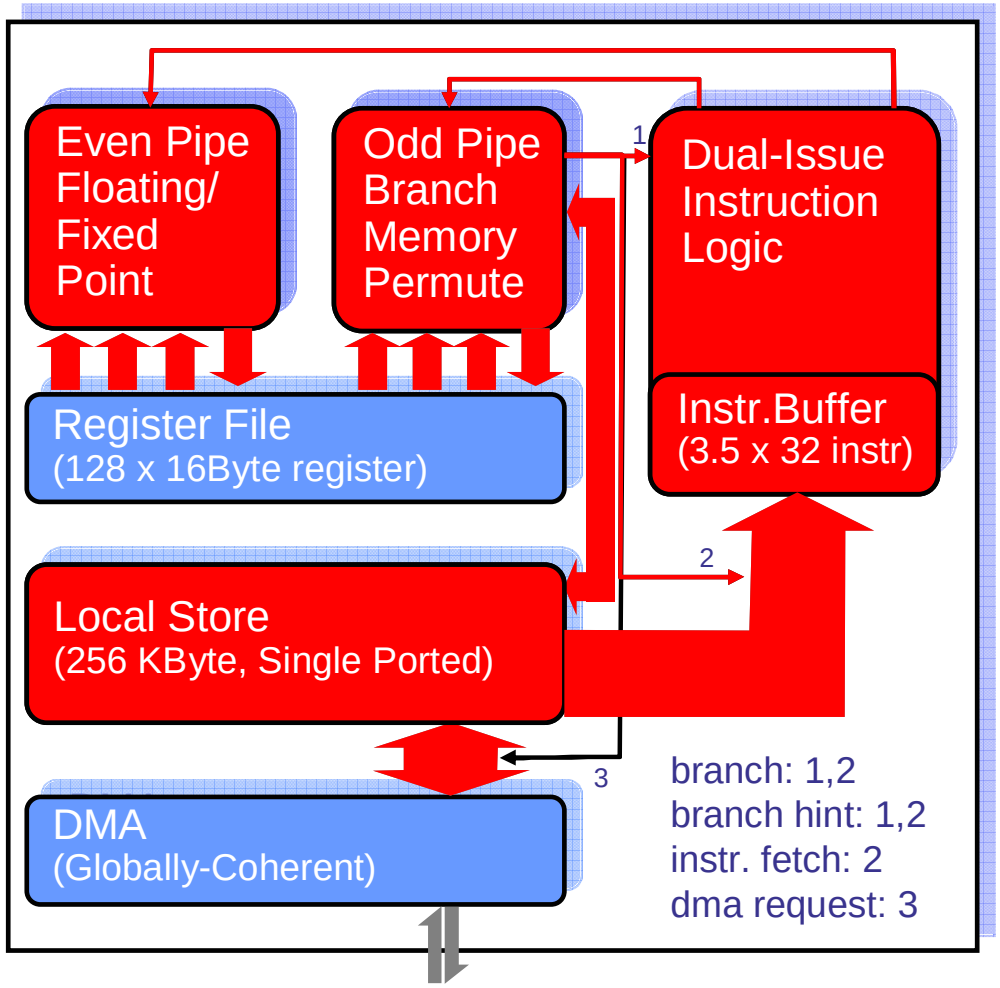


Shared memory, single program abstraction

Automatic parallelization

Architecture: Relevant SPE Features

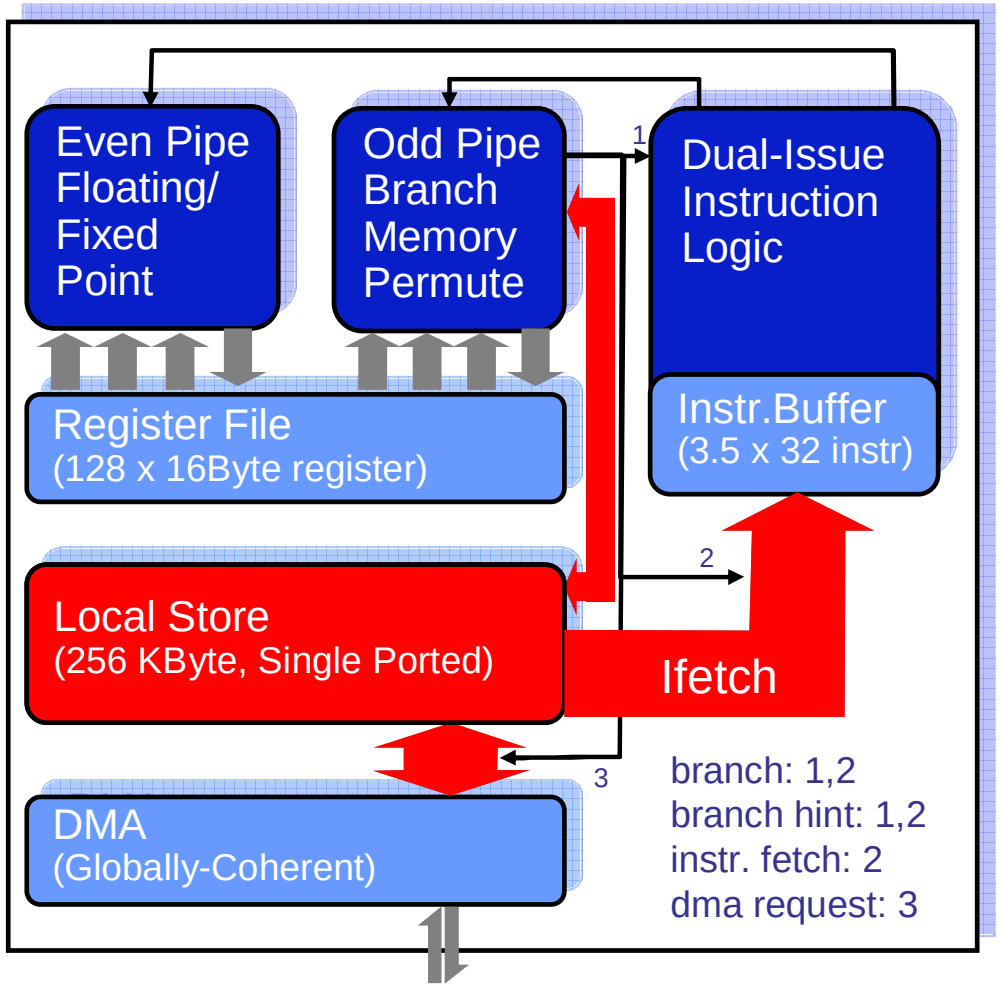
Synergistic Processing Element (SPE)



- **SIMD-only functional units**
  - 16-bytes register/memory accesses
- **Simplified branch architecture**
  - no hardware branch predictor
  - compiler managed hint/predication
- **Dual-issue for instructions**
  - full dependence check in hardware
  - must be parallel & properly aligned
- **Single-ported local memory**
  - aligned accesses only
  - contentions alleviated by compiler

Feature 1: Single-Ported Local Memory

SPE



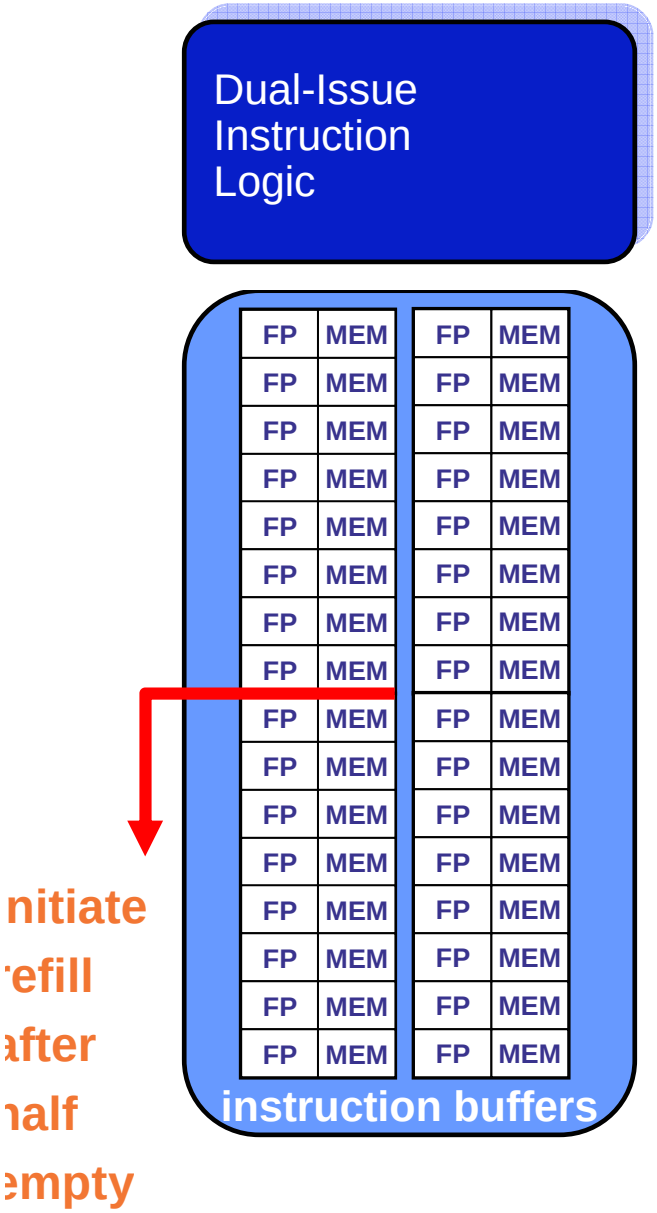
8 Bytes (per dir)

16Bytes (one dir)

128Bytes (one dir)

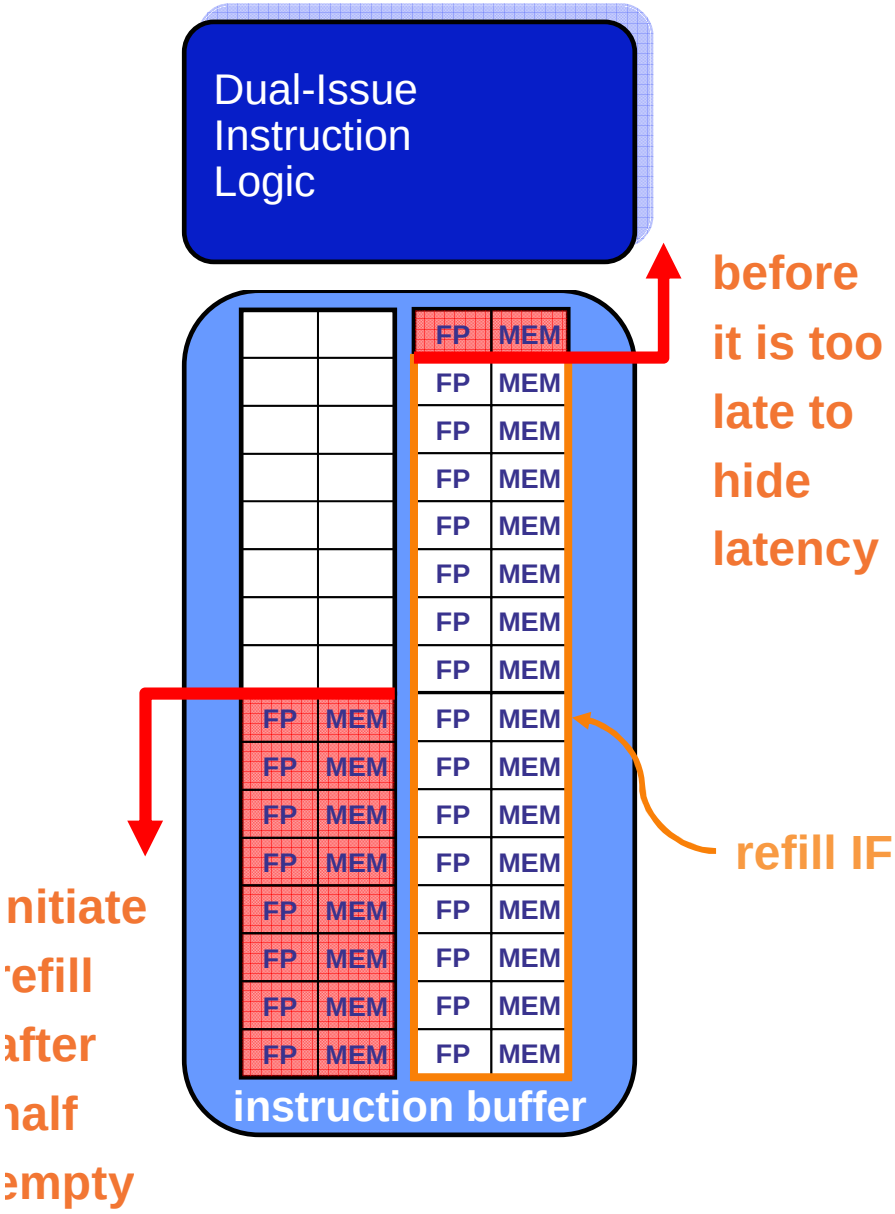
- **Local store is single ported**
  - less expensive hardware
  - asymmetric port
    - 16 bytes for load/store ops
    - 128 bytes for IFETCH/DMA
  - static priority
    - DMA > MEM > IFETCH
- **If we are not careful, we may starve for instructions**

Instruction Starvation Situation



- **There are 2 instruction buffers**
  - up to 64 ops along the fall-through path
- **First buffer is half-empty**
  - can initiate refill
- **When MEM port is continuously used**
  - starvation occurs (no ops left in buffers)

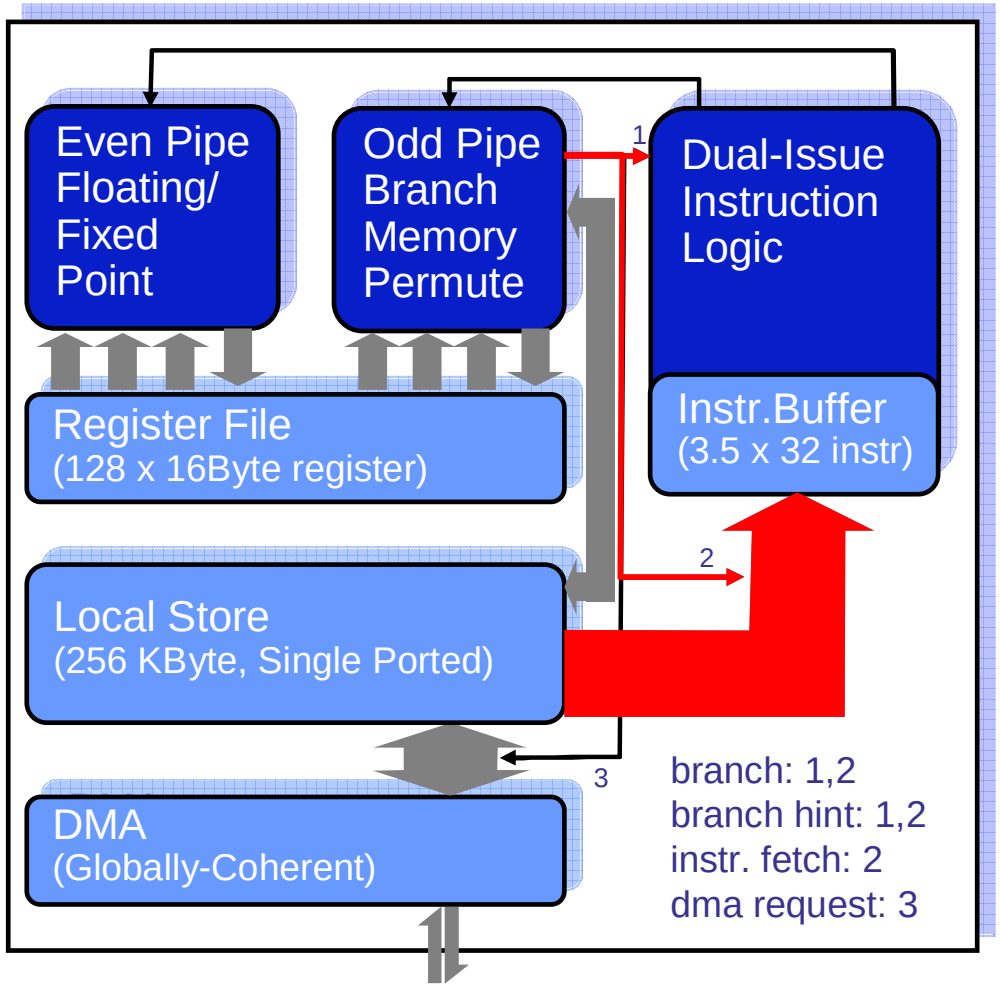
Instruction Starvation Prevention



- **SPE has an explicit IFETCH op**
  - which initiates a instruction fetch
- **Scheduler monitors starvation situation**
  - when MEM port is continuously used
  - insert IFETCH op within the (red) window
- **Compiler design**
  - scheduler must keep track of code layout

Feature #2: Software-Assisted Branch Architecture

SPE



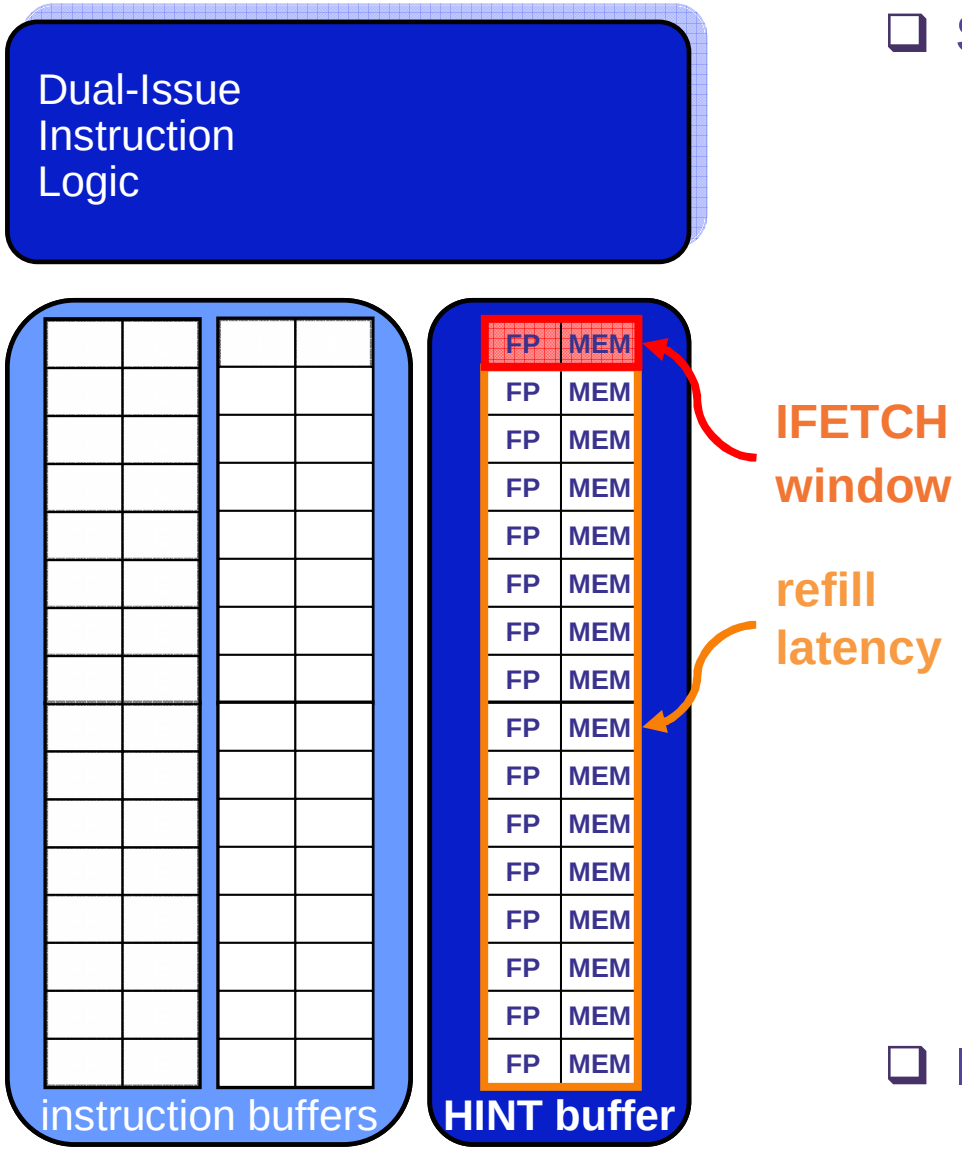
8 bytes (per dir)

16 bytes (one dir)

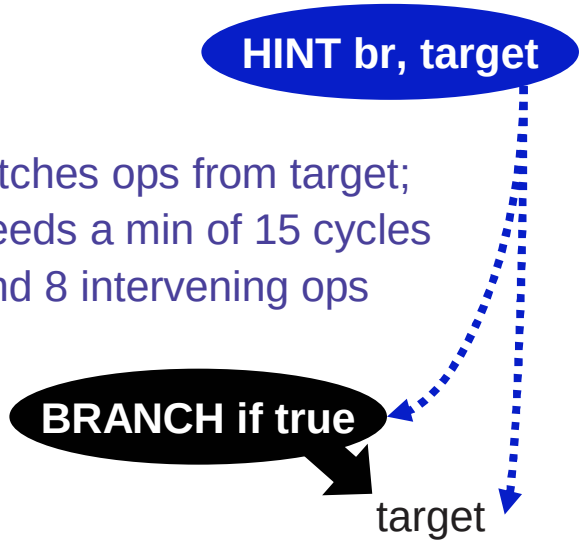
128 bytes (one dir)

- **Branch architecture**
  - no hardware branch-predictor, but
  - compare/select ops for predication
  - software-managed branch-hint
  - one hint active at a time
- **Lowering overhead by**
  - predicating small if-then-else
  - hinting predictably taken branches

Hinting Branches & Instruction Starvation Prevention



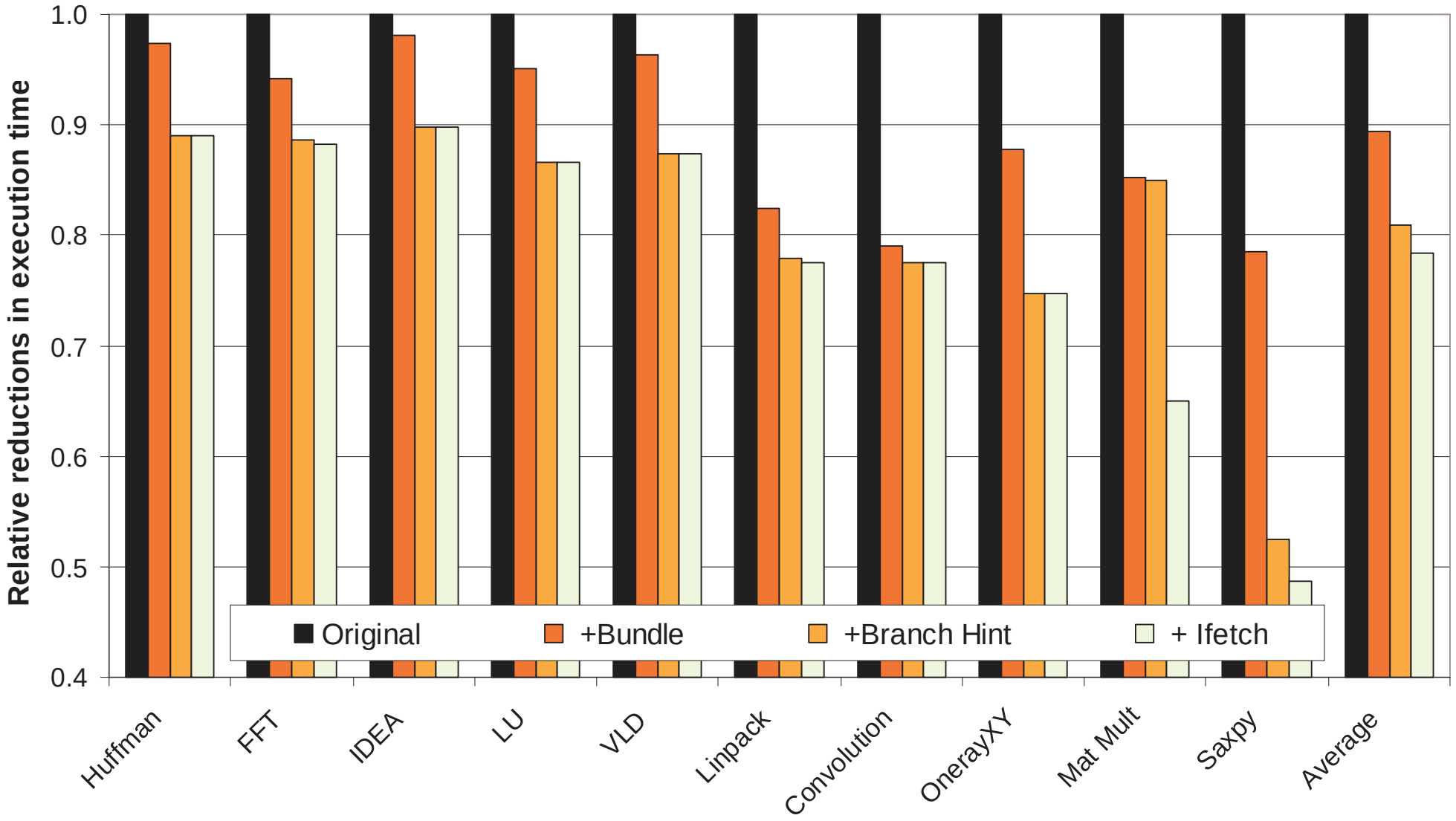
- **SPE provides a HINT operation**
  - fetches the branch target into HINT buffer
  - no penalty for correctly predicted branches



  - compiler inserts hints when beneficial

- **Impact on instruction starvation**
  - after a correctly hinted branch, IFETCH window is smaller

SPE Optimization Results



single SPE performance, optimized, simdized code (avg 1.00 → 0.78)

Outline

Part 1:
Automatic SPE tuning

Part 2:
Automatic simdization

Part 3:
Shared memory &
single program abstr.

Multiple-ISA hand-tuned programs

Explicit SIMD coding

Explicit parallelization with local memories

SIMD/alignment directives

Shared memory, single program abstraction

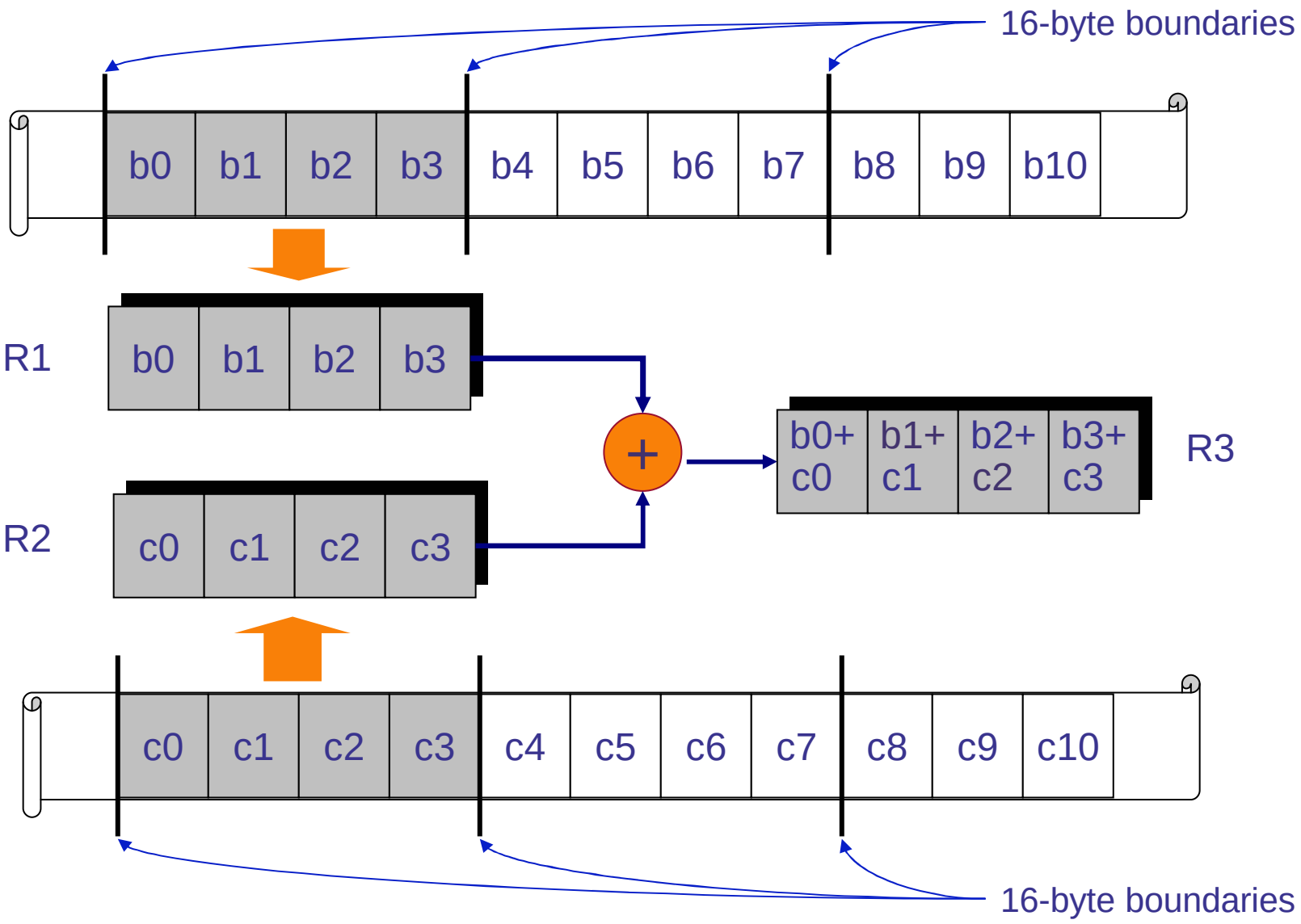
Automatic tuning for each ISA

Automatic simdization

Automatic parallelization

Single Instruction Multiple Data (SIMD) Computation

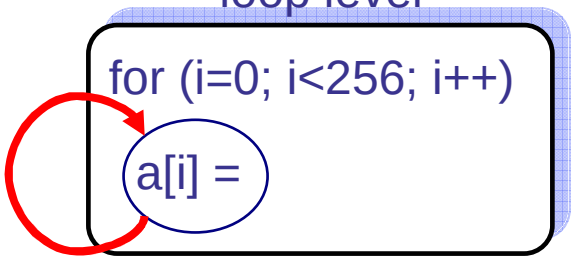
Process multiple “b[i]+c[i]” data per operations



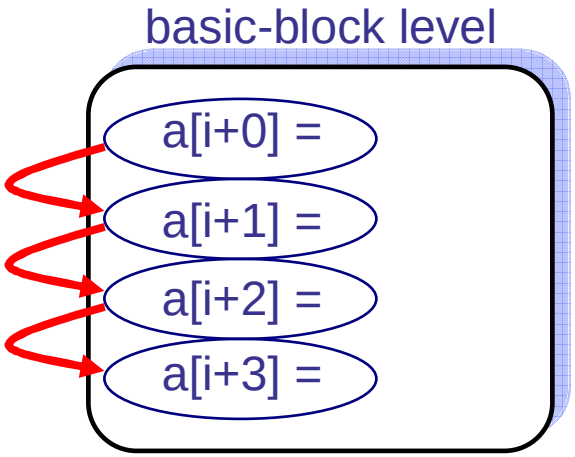
Successful Simdization

Extract Parallelism

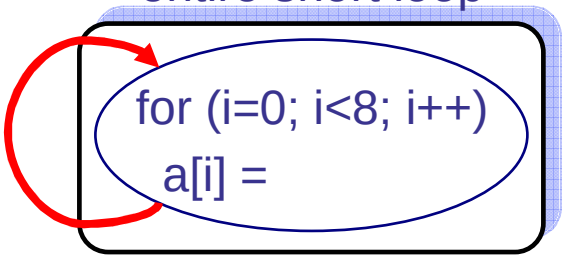
loop level

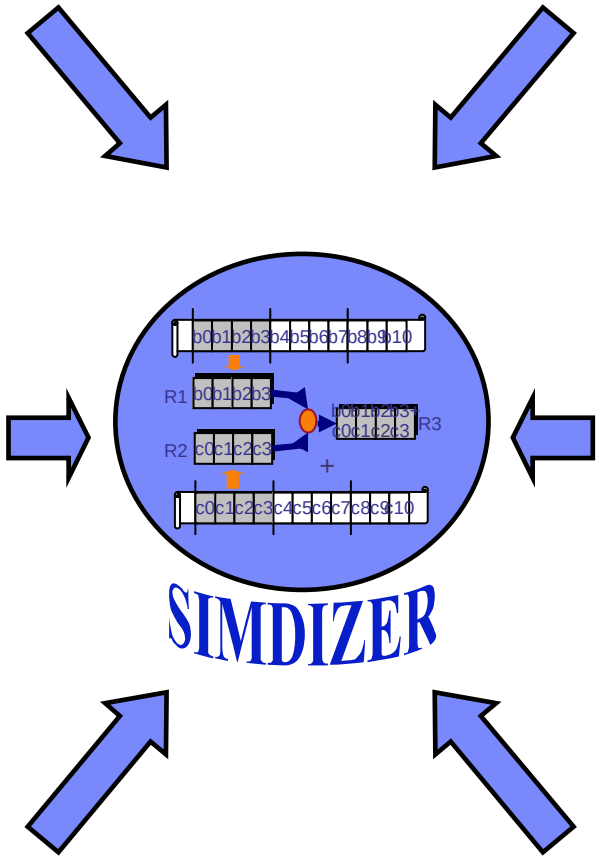


basic-block level



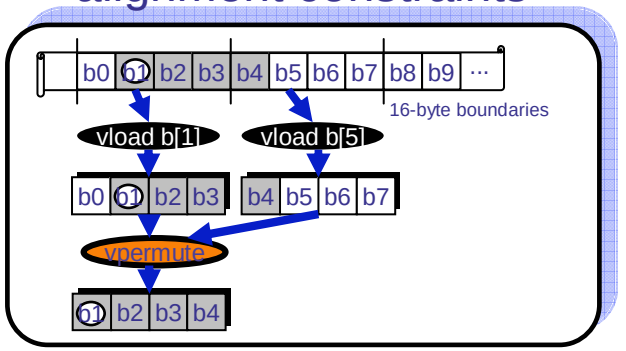
entire short loop



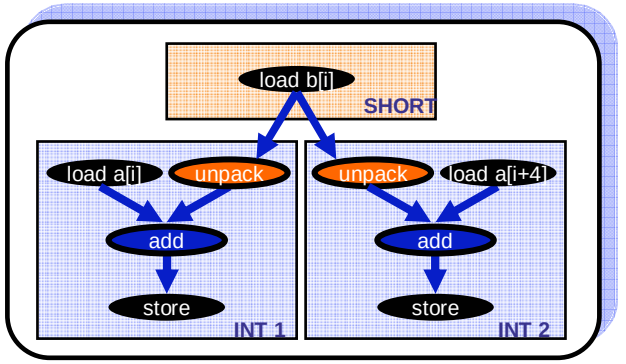


Satisfy Constraints

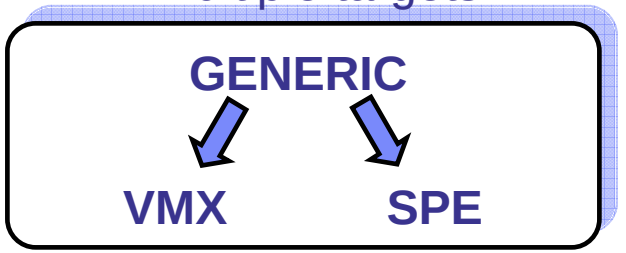
alignment constraints



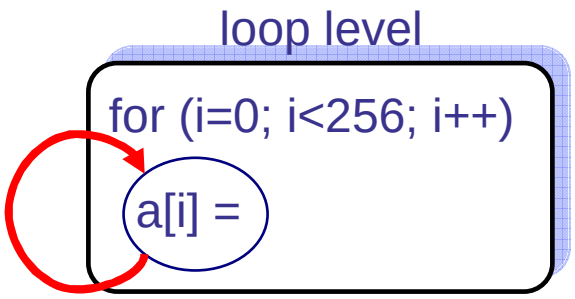
data size conversion

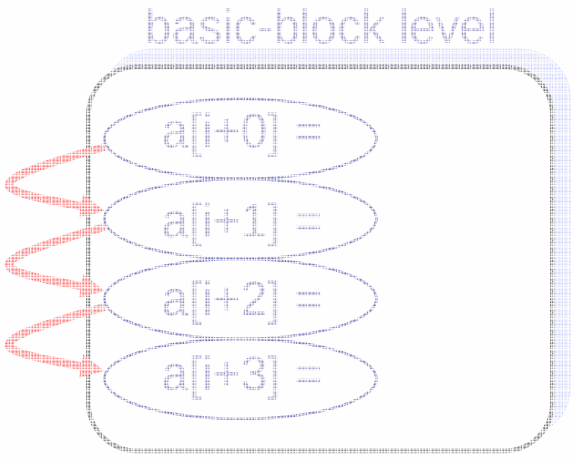


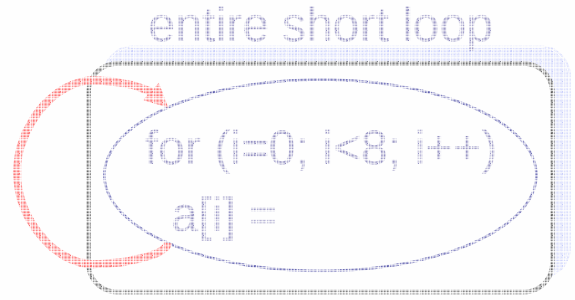
multiple targets



Example of SIMD-Parallelism Extraction







- **Loop level**
  - SIMD for a single statement across consecutive iterations
  - successful at:
    - efficiently handling misaligned data
    - pattern recognition (reduction, linear recursion)
    - leverage loop transformations in most compilers

[Bik *et al*, IJPP 2002]

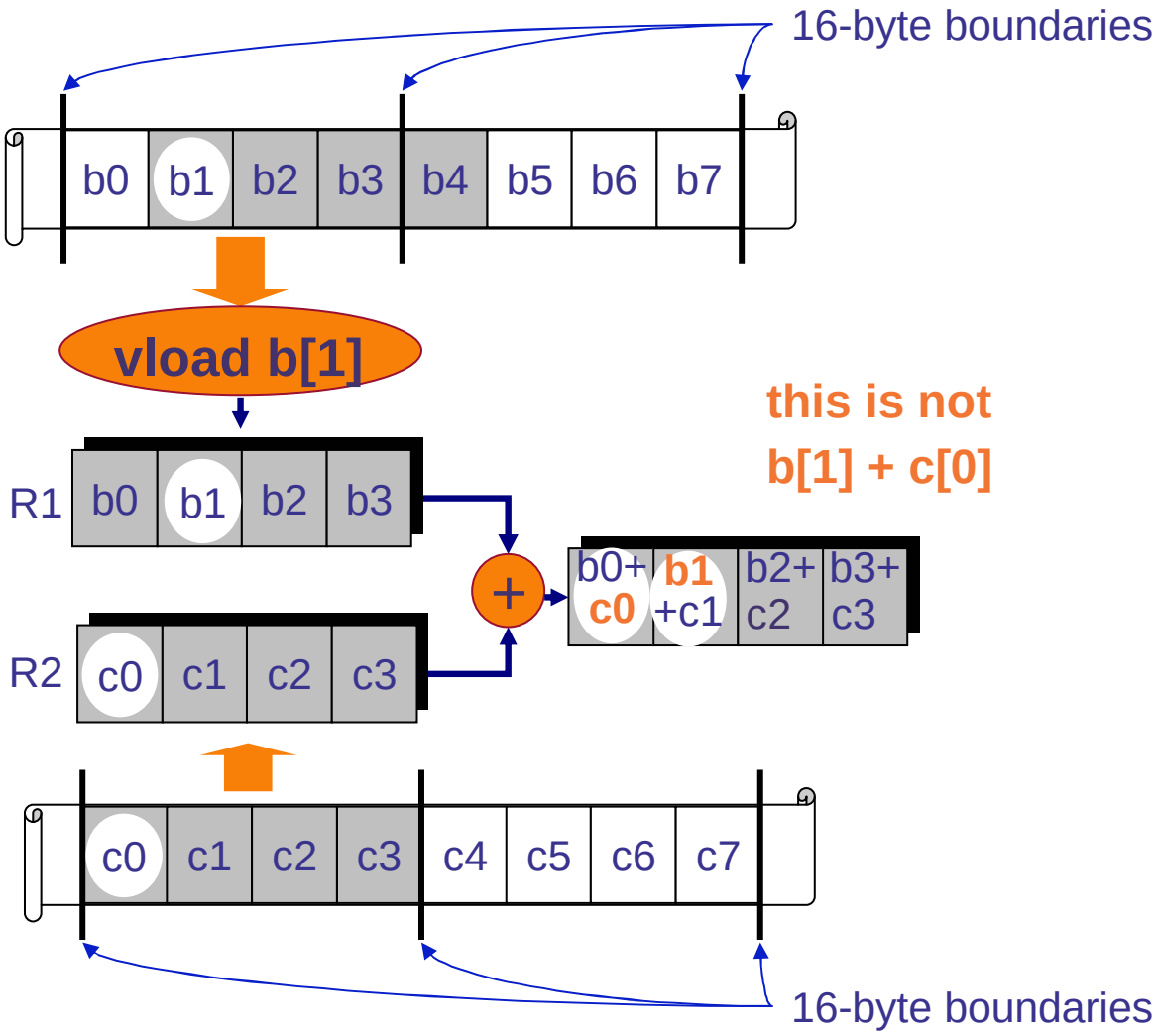
[VAST compiler, 2004]

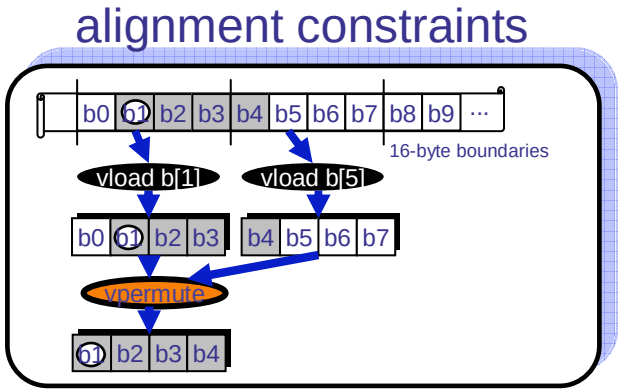
[Eichenberger *et al*, PLDI 2004] [Wu *et al*, CGO 2005]

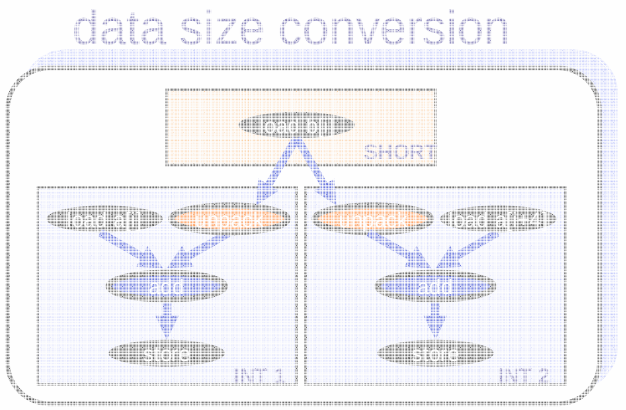
[Naishlos, GCC Developer's Summit 2004]

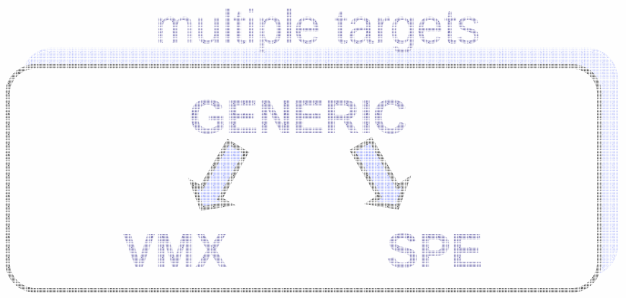
Example of SIMD Constraints

- **Alignment in SIMD units matters:**
  - consider “b[i+1] + c[i+0]”



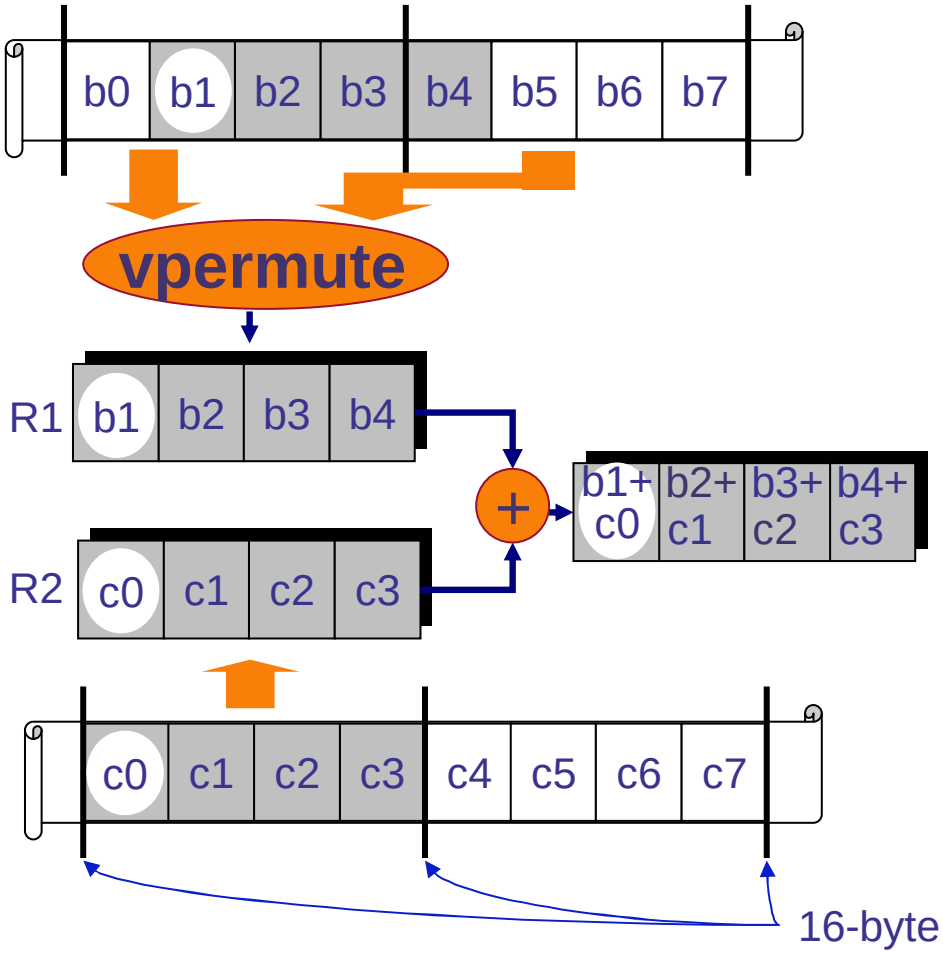


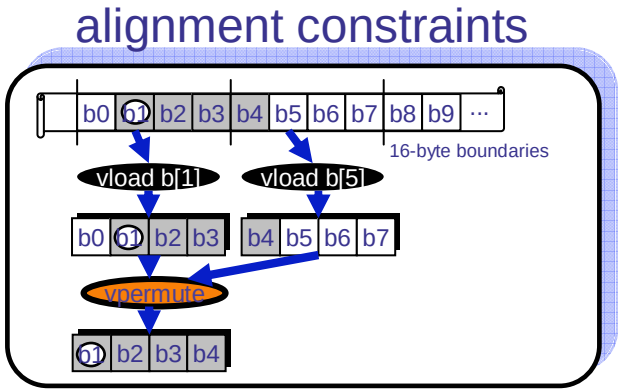


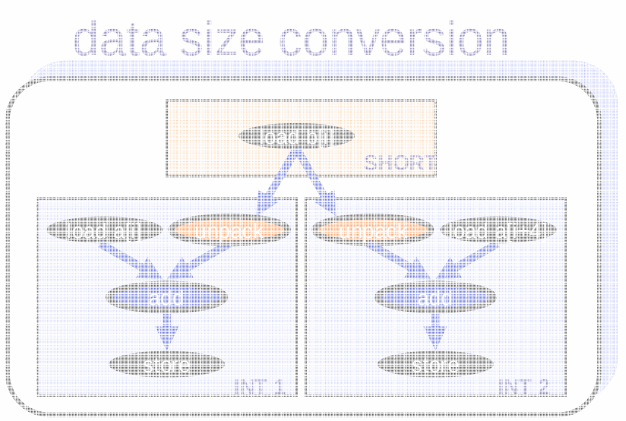


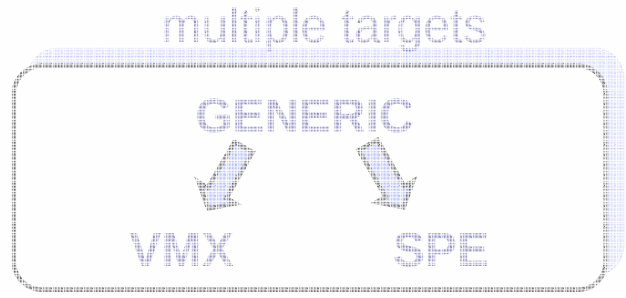
Example of SIMD Constraints (cont.)

- **Alignment in SIMD units matters**
  - when alignments within inputs do not match
  - must realign the data



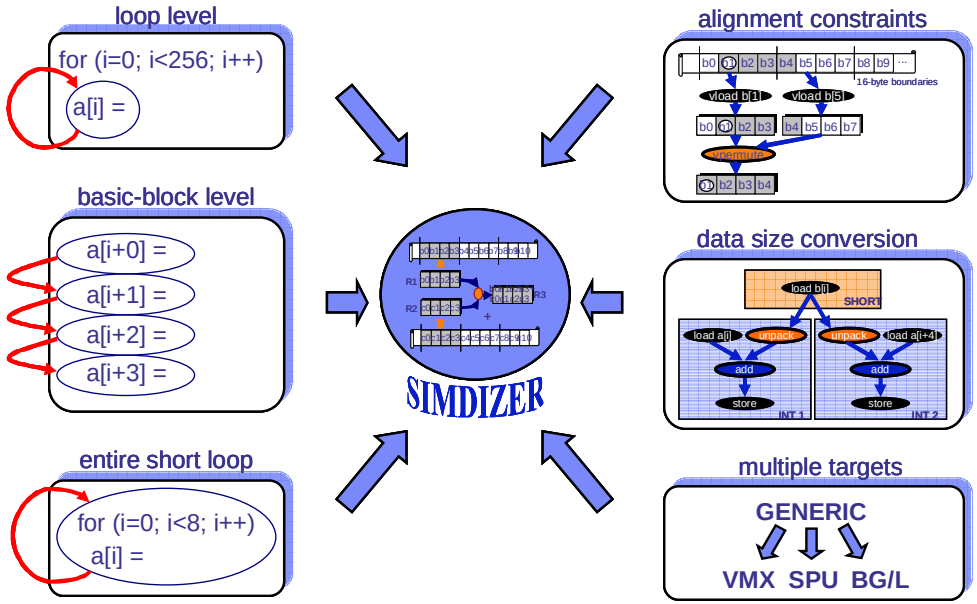






Automatic Simdization for Cell

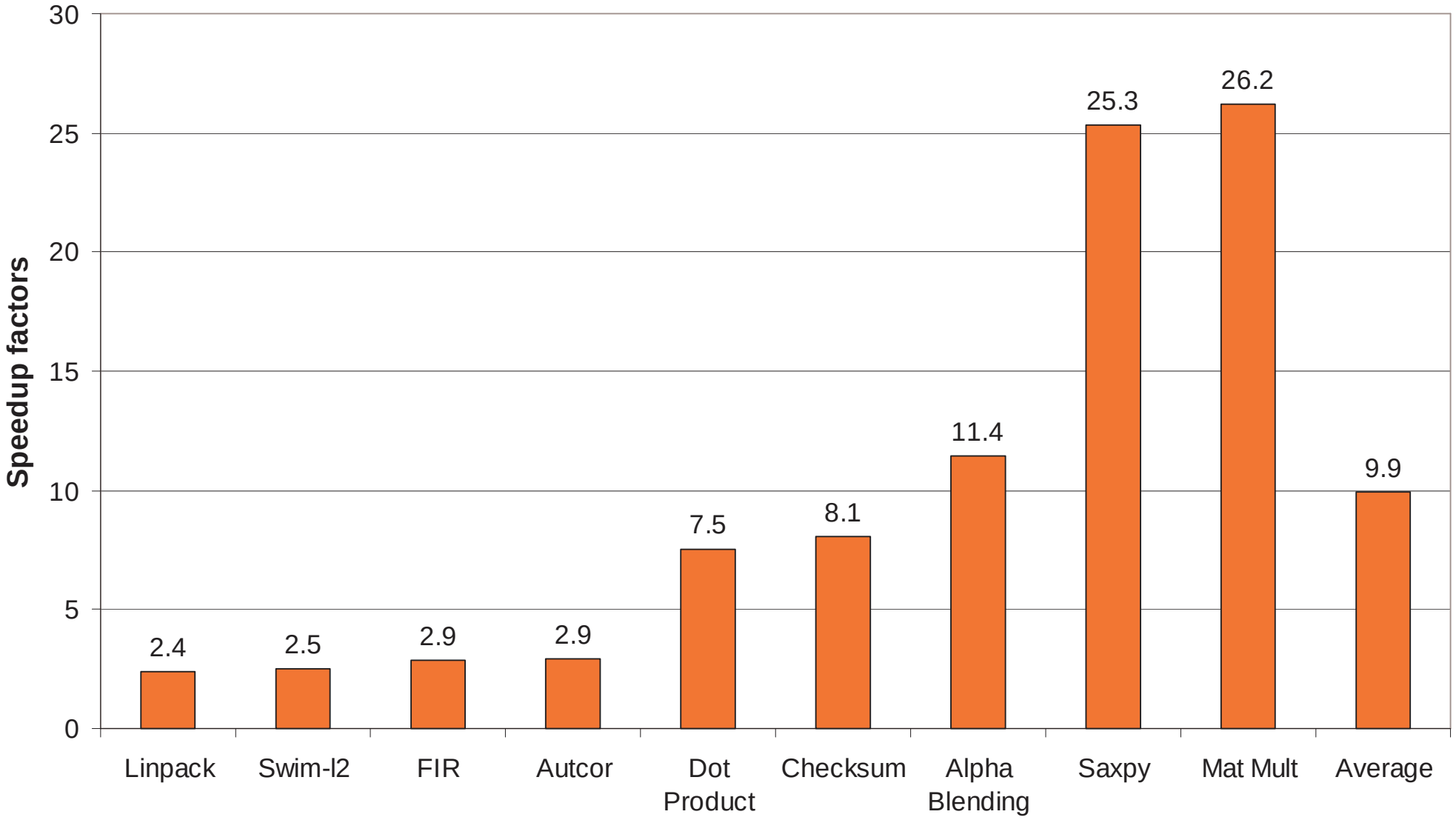
- **Integrated Approach**
  - extract at multiple levels
  - satisfy all SIMD constraints
  - use “virtual SIMD vector” as glue



- **Minimize alignment overhead**
  - lazily insert data reorganization
  - handle compile time & runtime alignment
  - simdize prologue/epilogue for SPEs
    - memory accesses are always safe on SPE

- **Full throughput computations**
  - even in presence of data conversions
  - manually unrolled loops...

SPE Simdization Results



single SPE, optimized, automatic simdization vs. scalar code

Outline

Part 1:
Automatic SPE tuning

Part 2:
Automatic simdization

Part 3:
Shared memory & single program abstr.

Multiple-ISA hand-tuned programs

Explicit SIMD coding

Explicit parallelization with local memories

SIMD/alignment directives

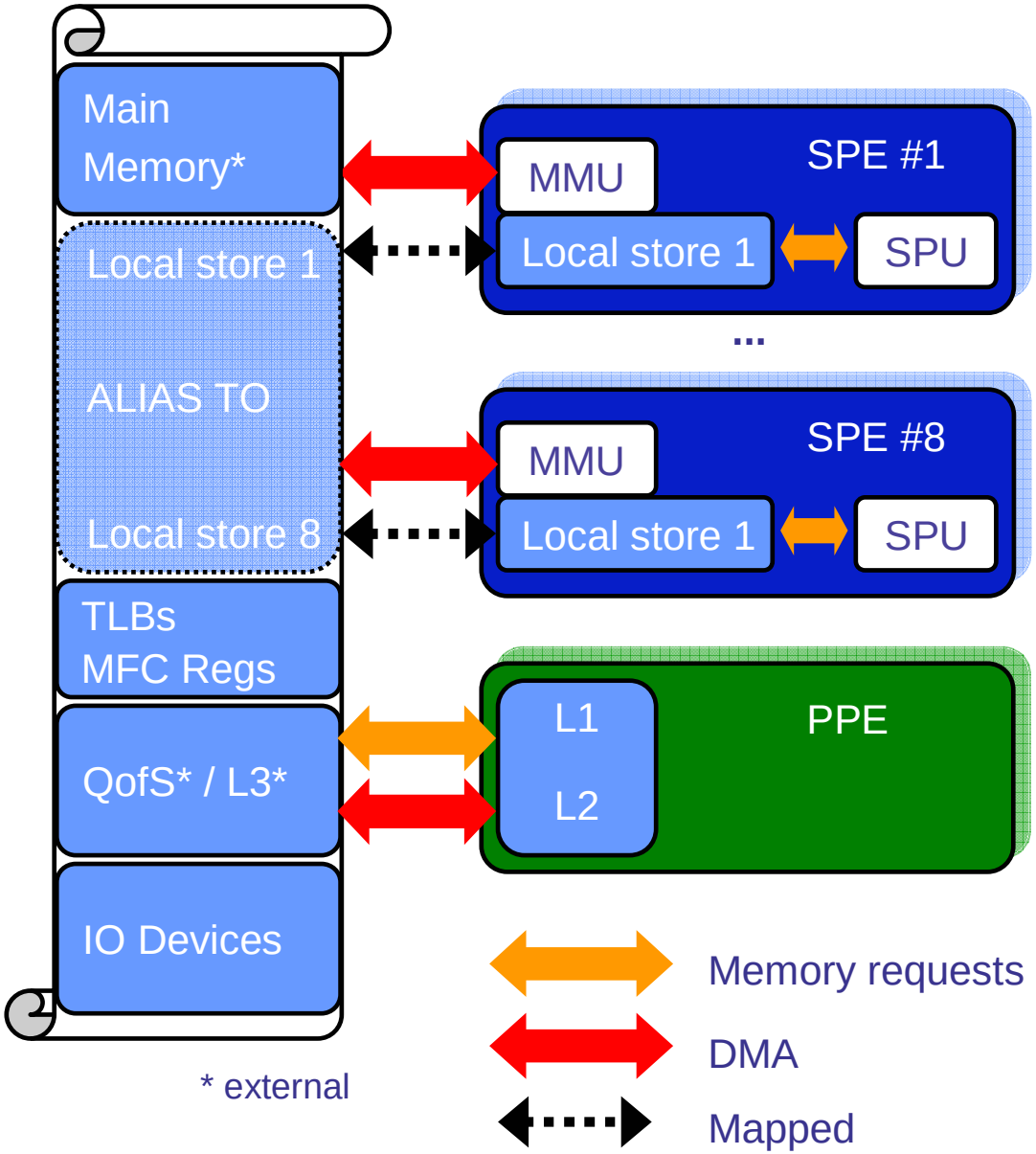
Shared memory, single program abstraction

Automatic tuning for each ISA

Automatic simdization

Automatic parallelization

Cell Memory & DMA Architecture

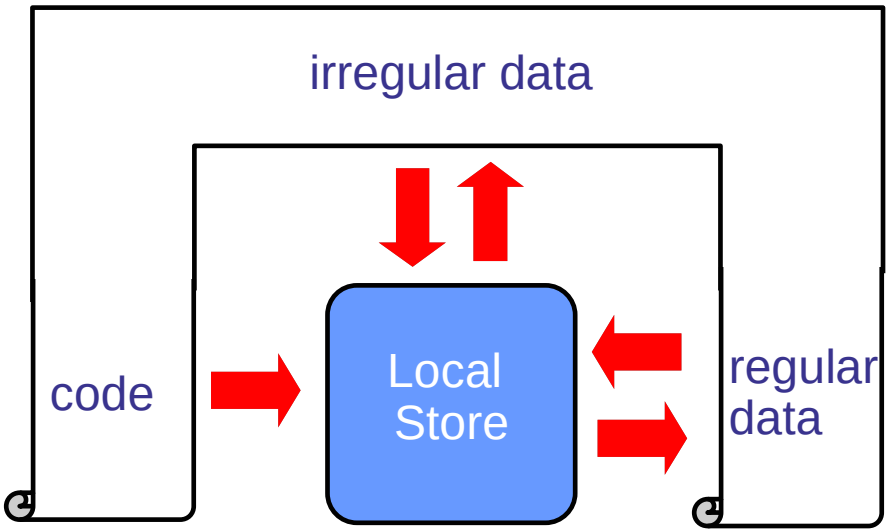


- **Local stores are mapped in global address space**
- **PPE**
  - can access/DMA memory
  - set access rights
- **SPE can initiate DMAs**
  - to any global addresses,
  - including local stores of others.
  - translation done by MMU
- **Note**
  - all elements may be masters, there are no designated slaves

Competing for the SPE Local Store

Local store is fast, need support when full.

Provided compiler support:



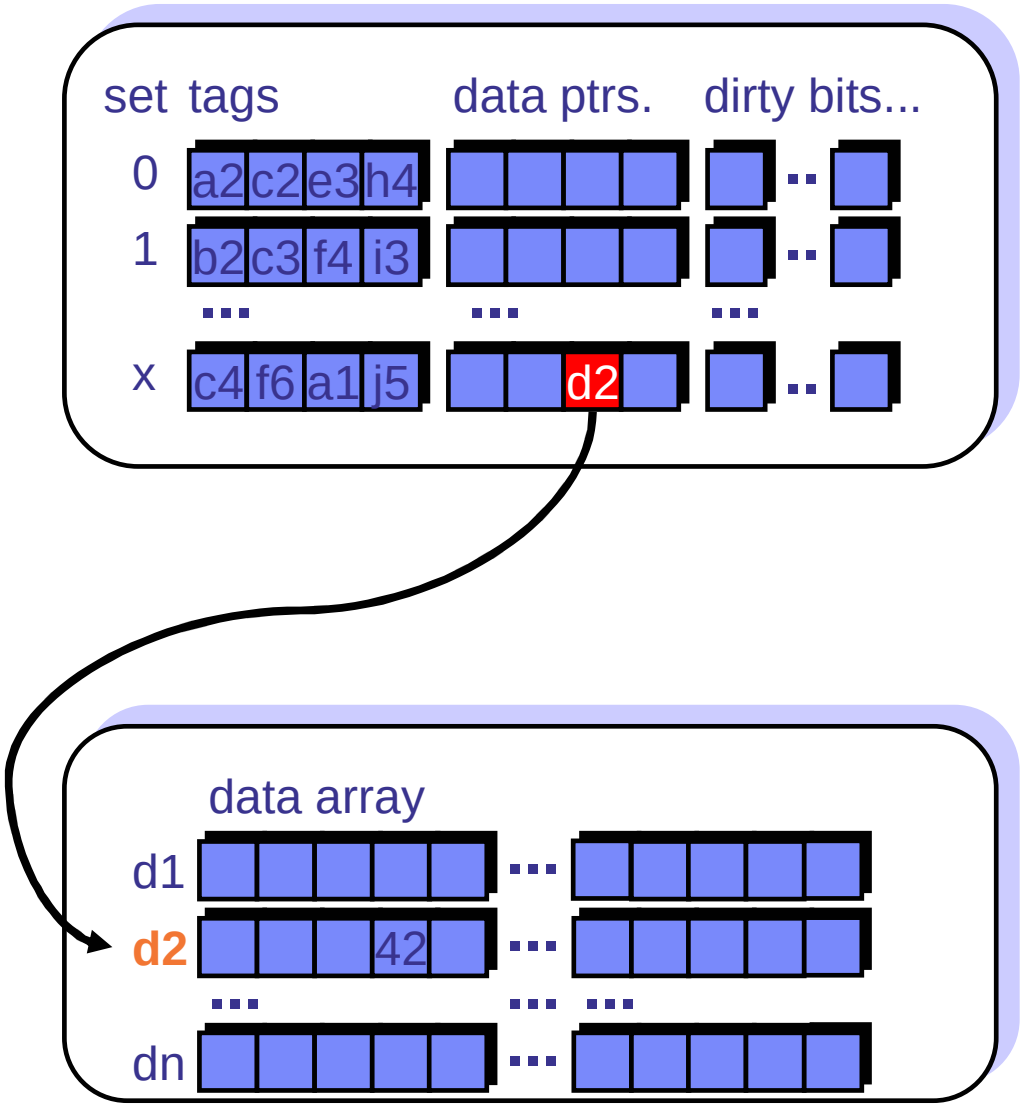
- **SPE code too large**
  - compiler partitions code
  - partition manager pulls in code as needed
- **Data with regular accesses is too large**
  - compiler stages data in & out
  - using static buffering
  - can hide latencies by using double buffering
- **Data with irregular accesses is present**
  - e.g. indirection, runtime pointers...
  - use a software cache approach to pull the data in & out (last resort solution)

Software Cache for Irregular Accesses

- **Data with irregular accesses**
  - cannot reside permanently in the SPE's local memory (typically)
  - thus reside in global memory
  - when accessed,
    - must translate the global address into a local store address
    - must pull the data in/out when its not already present

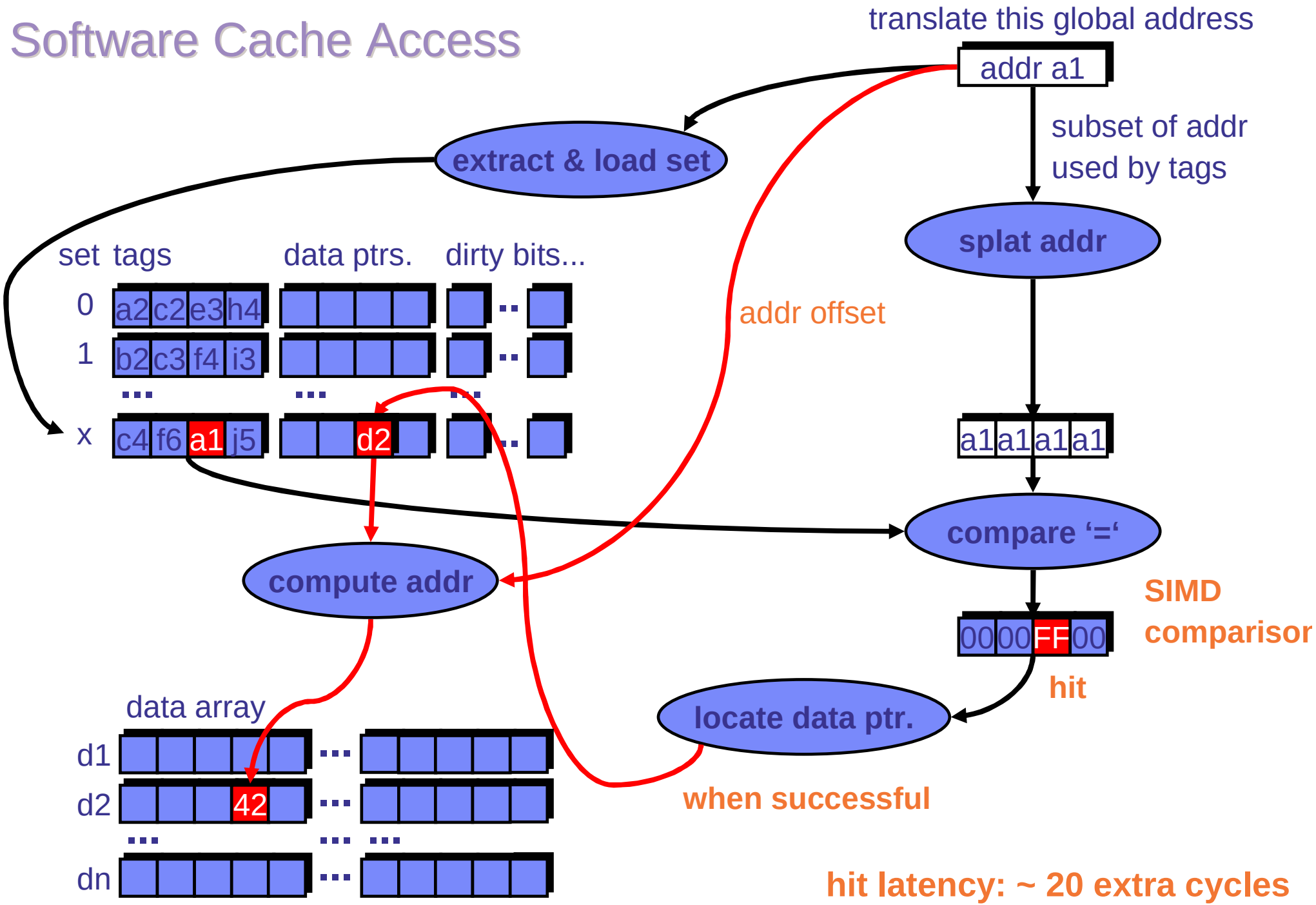
- **Use a software cache**
  - managed by the SPEs in the local store
  - generate DMA requests to transfer data to/from global memory
  - use 4-way set associative cache to naturally use the SIMD units of the SPE

Software Cache Achitecture



- **Cache directory**
  - 128-set, 4-way set associative
  - pointers to data lines
  - use 16KByte of data
- **Data in a separate structure**
  - 512 x 128B lines
  - use 64KByte of data

Software Cache Access



“Single Source” Compiler, using OpenMP

```
#pragma Omp parallel for
for( i =0; i<10000; i++)
   A[i] = x* B[i];
```

code with OpenMP directives

outline omp region

```
void foo()
   #pragma Omp parallel for
      for( i =0; i<10000; i++)
         A[i] = x* B[i];
```

clone for SPE

clone for PPE

```
void foo_PPE()
   init_omp_rte();
   for( i=LB;  i<UB;  i++) ´
      A[i] = x * B[i];
```

```
void foo_SPE()
   for( k=LB;  k<UB;  k++)
      DMA 100 B elements into B´
      for (j=0;   j<100;   j++)
         A´[j] = cache_lookup(x) * B´[j];
      DMA 100 A elements out of A´
```

Runtime:

initialize OpenMP runtime

compute its own work

Runtime:

DMA in/out array, lookup software cache

compute its own work

Single Source Compiler Results

- **Results for Swim, Mgrid, & some of their kernels**

baseline: execution on one single PPE

Conclusions

- **Cell Broadband Engine architecture**
  - heterogeneous parallelism
  - dense compute architecture
- **Present the application writer with a wide range of tool**
  - from support to extract maximum performance
  - to support to achieve maximum productivity with automatic tools
- **Shown respectable speedups**
  - using automatic tuning, simdization, and support for shared-memory abstraction

Questions

For additional info:

www.research.ibm.com/cellcompiler/compiler.htm

Extra

Compiler Support For Single Source Program