Compiler Mediated Performance of a Cell Processor

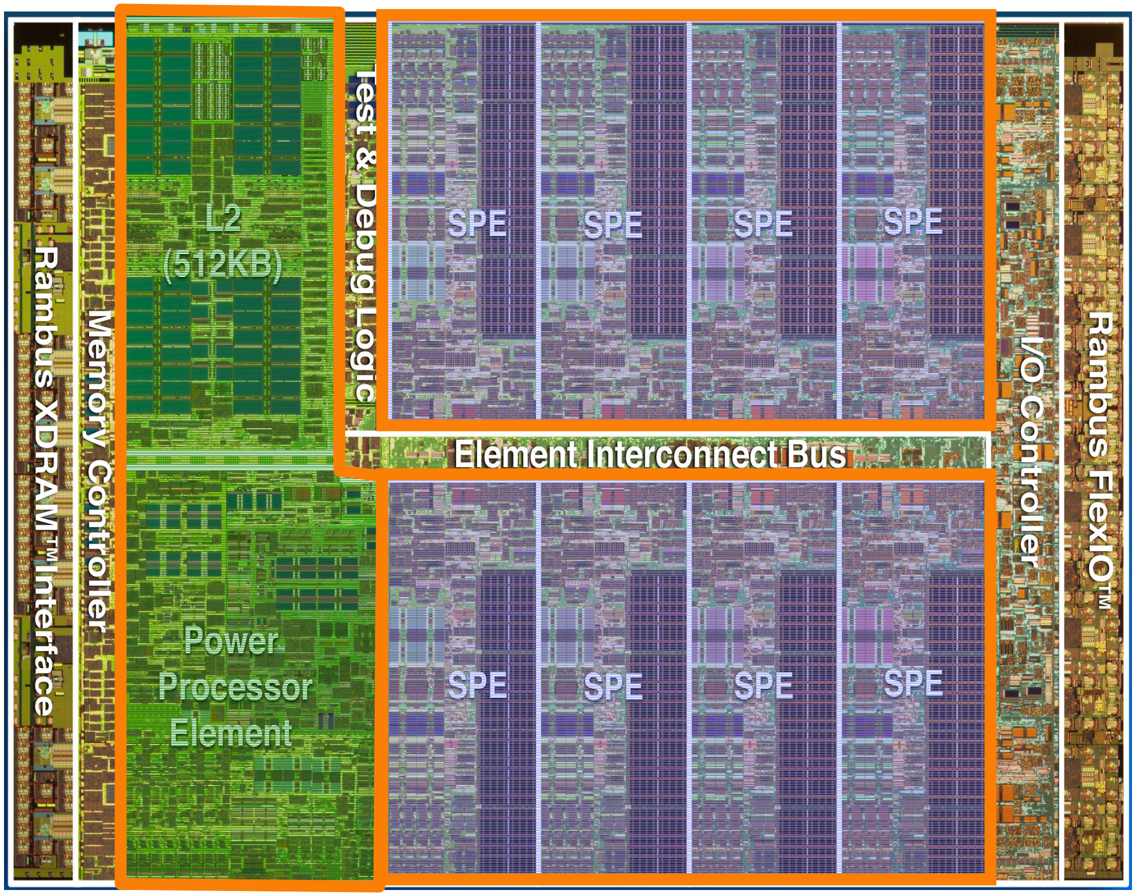
Alexandre Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter Oden, Daniel Prener, Janice Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao,Yaoqing Gao

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

Background

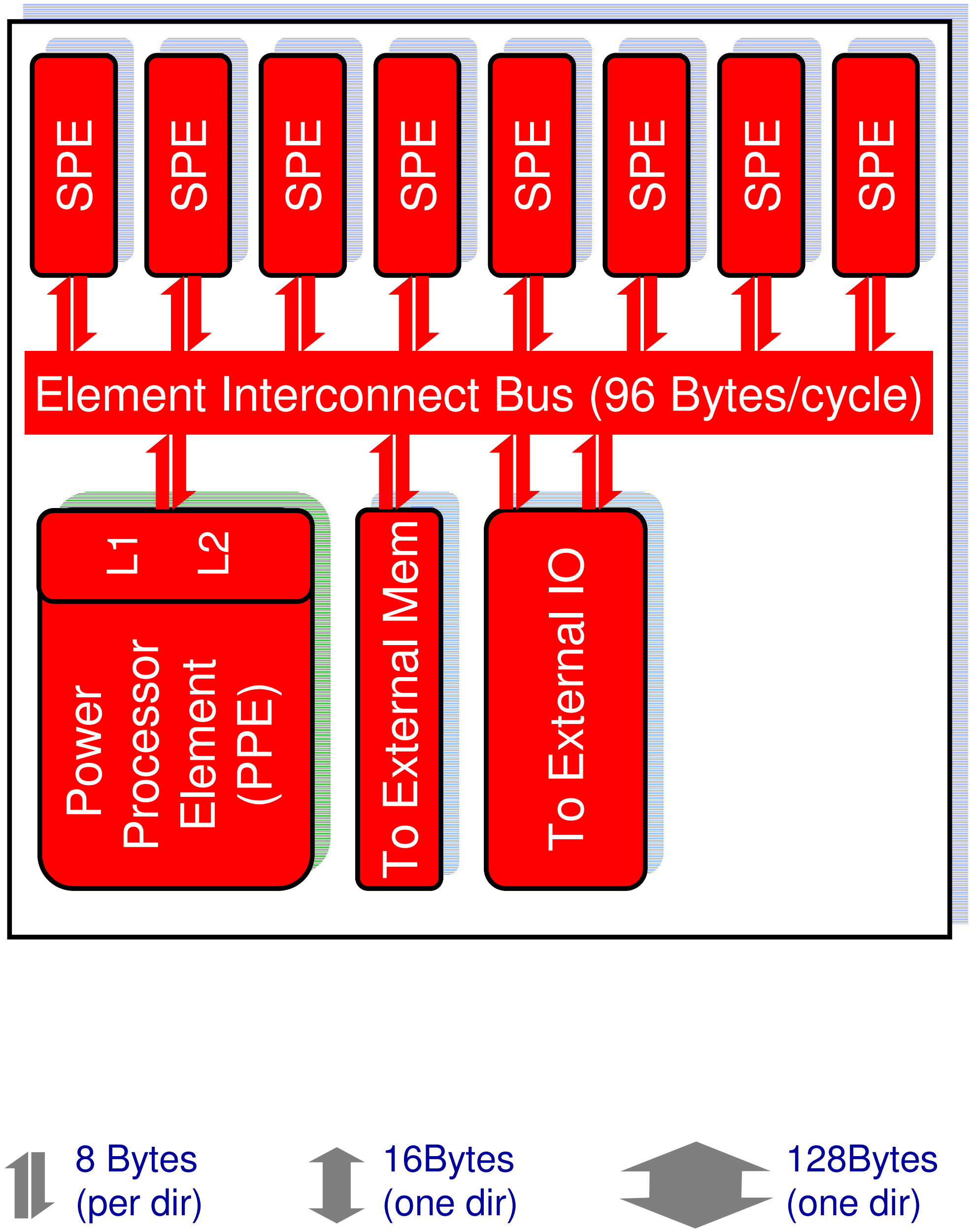
- **Prototype compiler**
  - Some functionality shipped in Alphaworks Cell xlc compiler
- **Based on and integrated with the product code base**
- **Collaboration with compiler development group**
- **Many research issues still in progress**

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

Compiler support
for the full spectrum of Cell programming expertise

Highest performance with programmer control

Hand tuned for multipe ISAs

PROGRAMS

Automatic tuning for each ISA

Explicit SIMD coding

SIMD

SIMD/alignment directives

Automatic simdization

Explicit parallelization with local memories

PARALLELIZATION

Shared memory abstraction through single source compilation

Automatic parallelization

Highest Productivity with fully automatic compiler technology

Outline

Part 1:
Automatic SPE tuning

Multiple-ISA hand-tuned programs

PROGRAMS

Automatic tuning for each ISA

Part 3:
Automatic simdization

Explicit SIMD coding

SIMD

SIMD/alignment directives

Automatic simdization

Part 2:
Parallelization and memory abstraction

Explicit parallelization with local memories

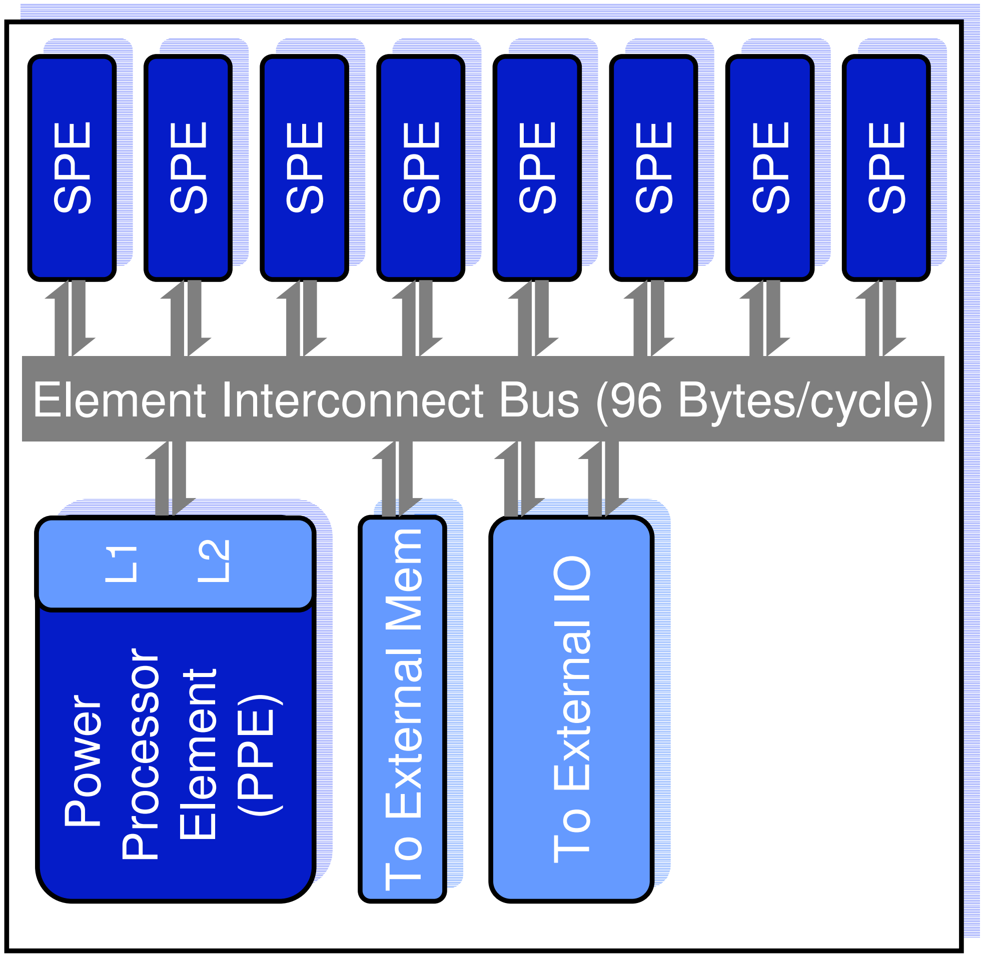
PARALLELIZATION

Shared memory, single program abstraction

Automatic parallelization

Cell Architecture Overview

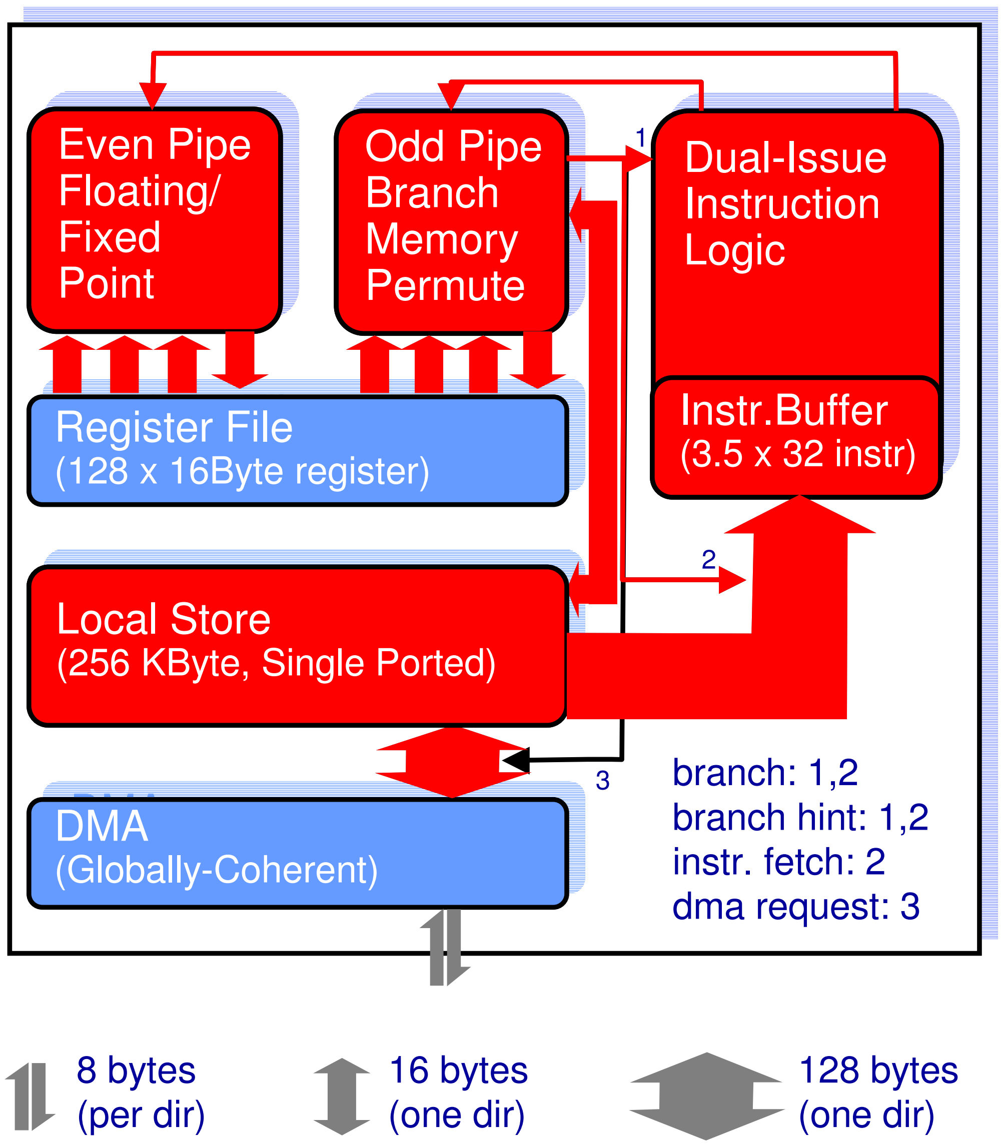
Cell Processor



- **PPE executes traditional code**
- **SPE optimized for**
  - high compute throughput
  - small area footprint
- **SPE tradeoff**
  - lower hardware complexity
  - peak performance requires compiler assistance

Architecture: Relevant SPE Features

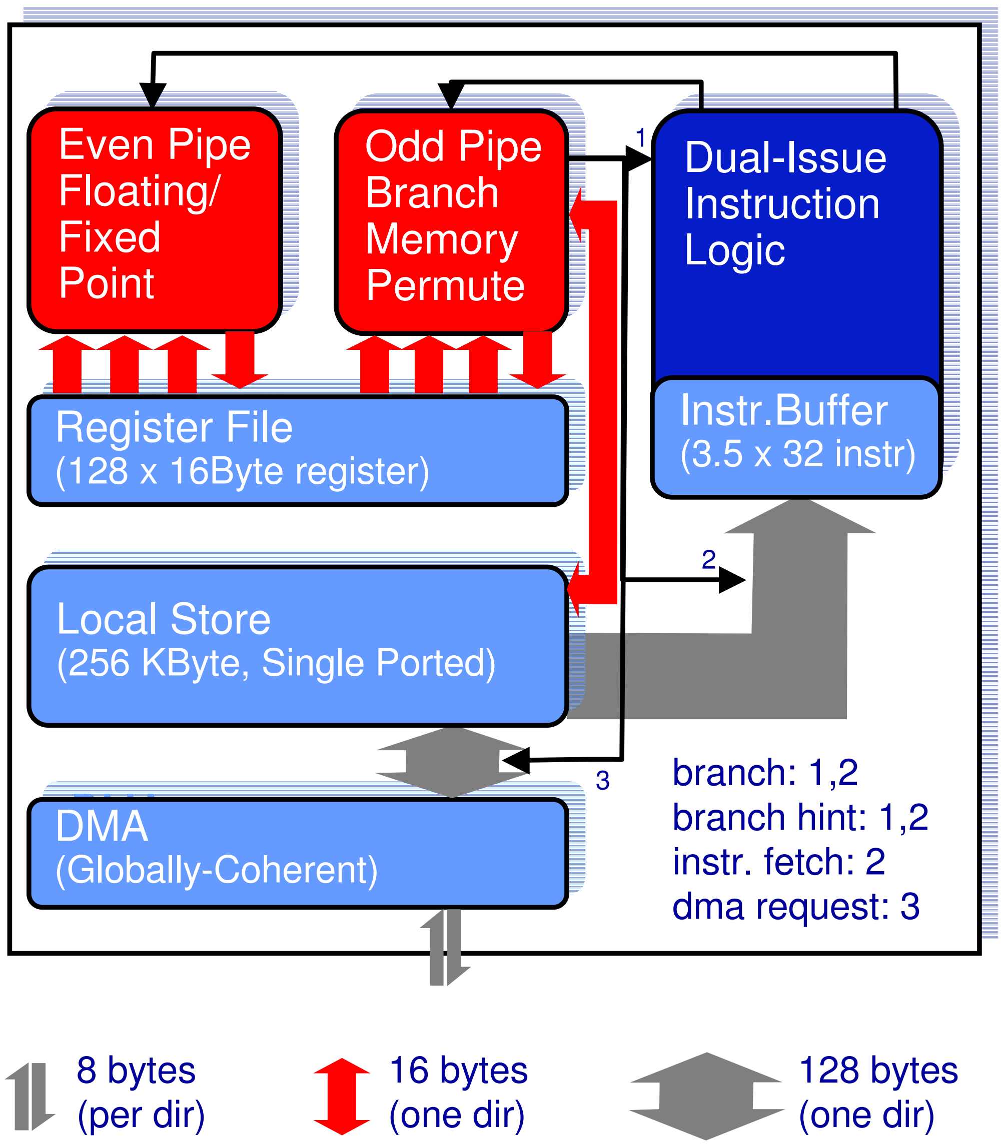
Synergistic Processing Element (SPE)



- **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: Functional Units are SIMD Only

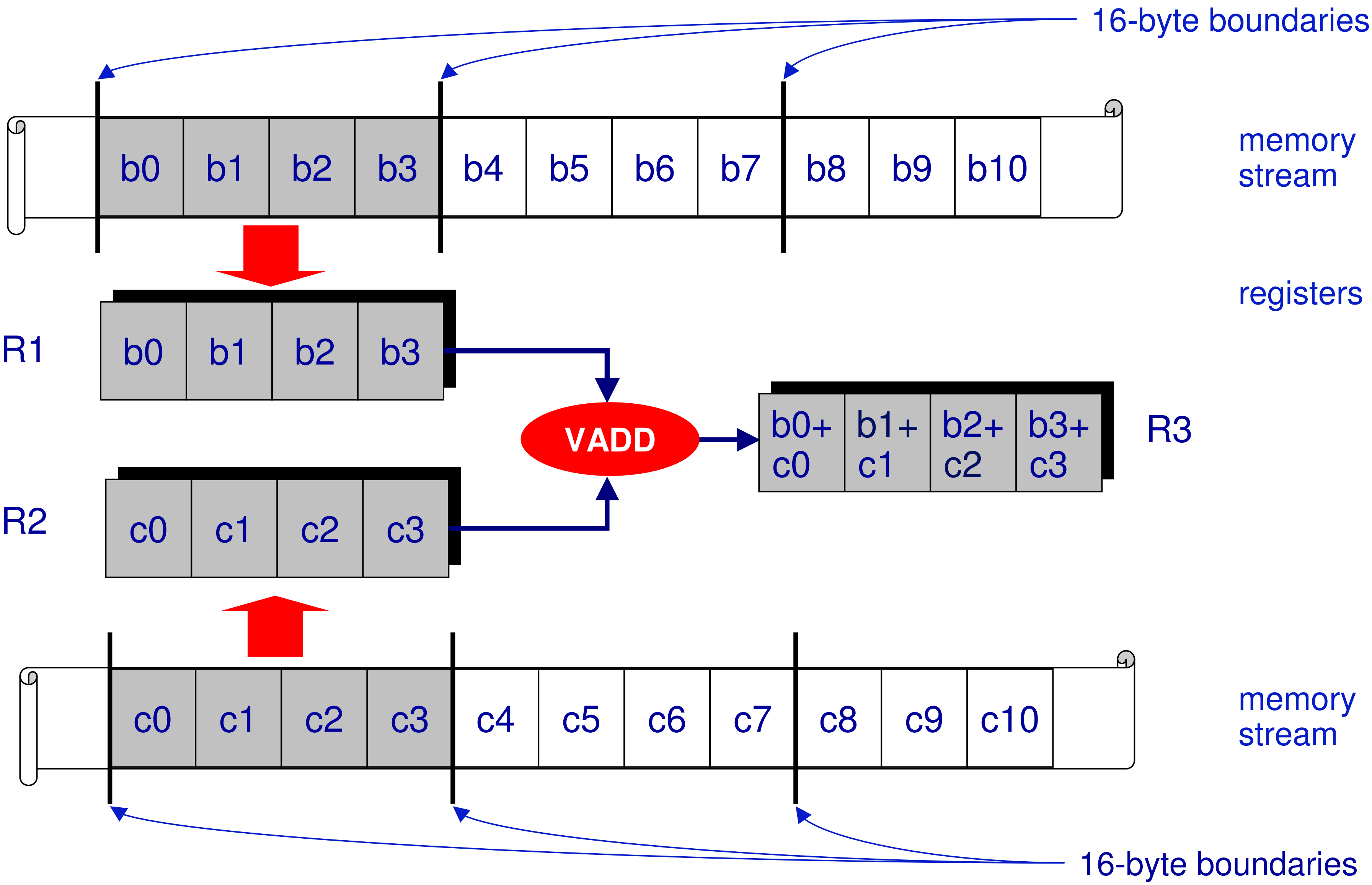
SPE



- **Functional units are SIMD only**
  - all transfers are 16 Bytes wide,
  - including register file and memory
- **How to handle scalar code?**

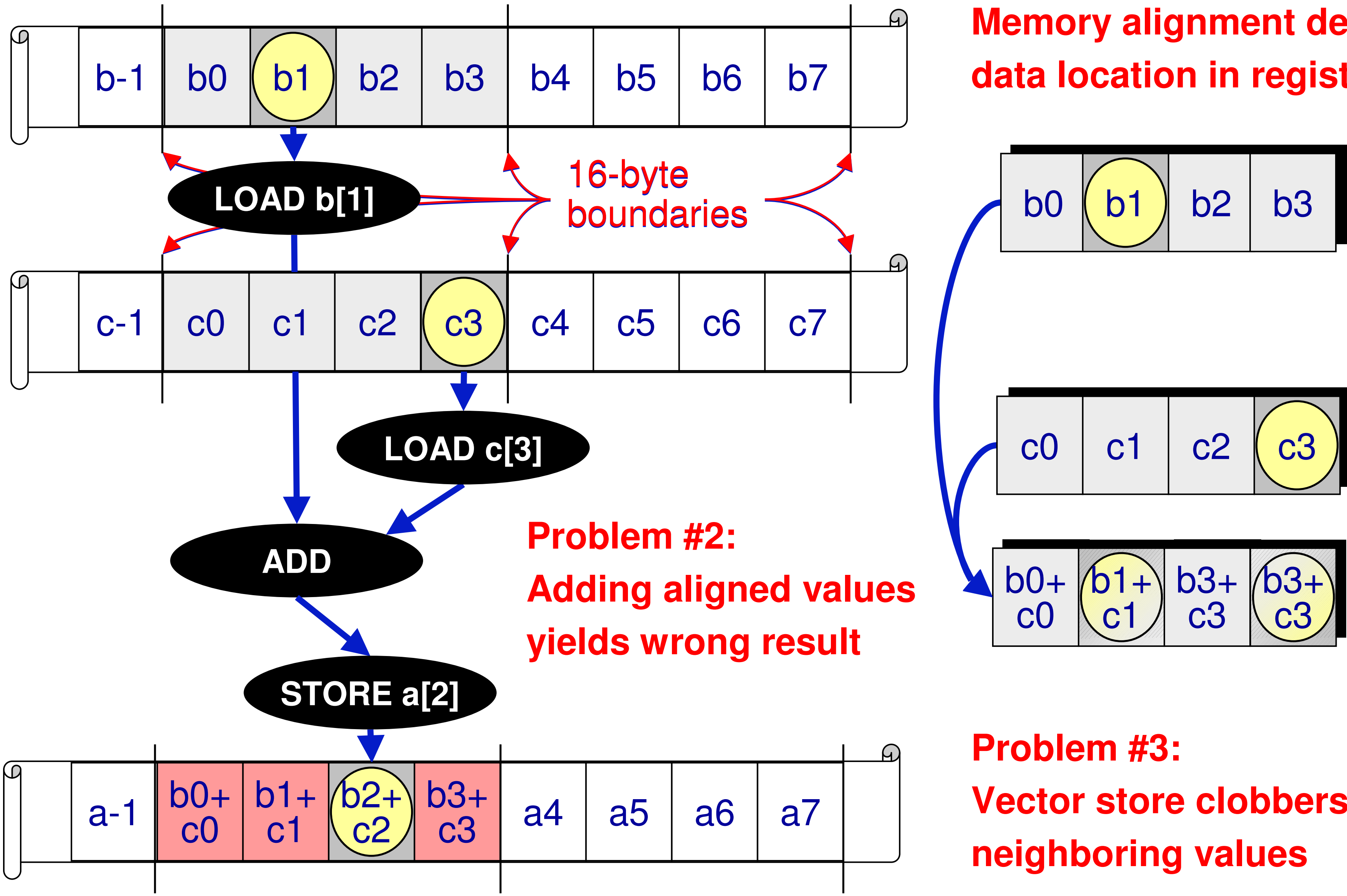
Single Instruction Multiple Data (SIMD)

Meant to process multiple "b[i]+c[i]" data per operation



Scalar Code on SIMD Functional Units

- **Example: a[2] = b[1] + c[3]**



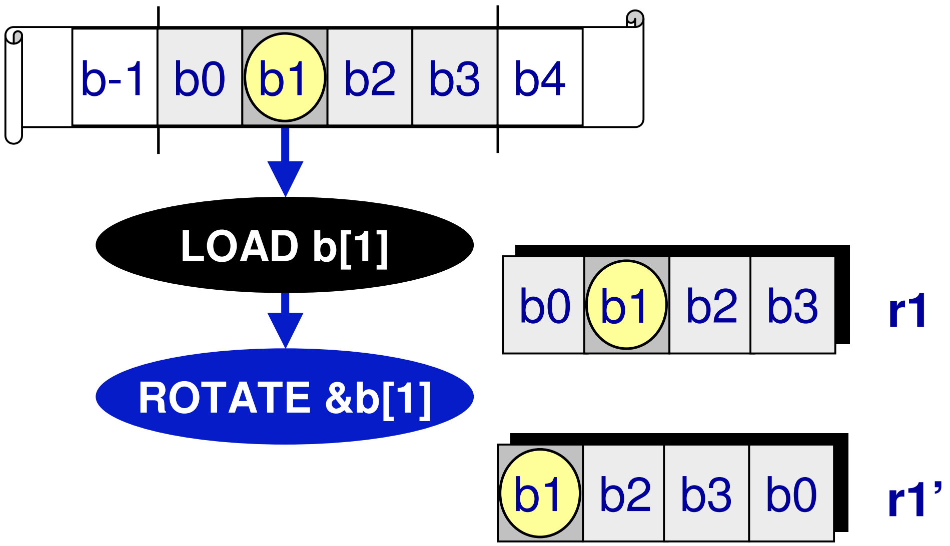
Problem #1: Memory alignment defines data location in register

Problem #2: Adding aligned values yields wrong result

Problem #3: Vector store clobbers neighboring values

Scalar Load Handling

❑ Use read-rotate sequence



❑ Overhead (1 op, in blue)

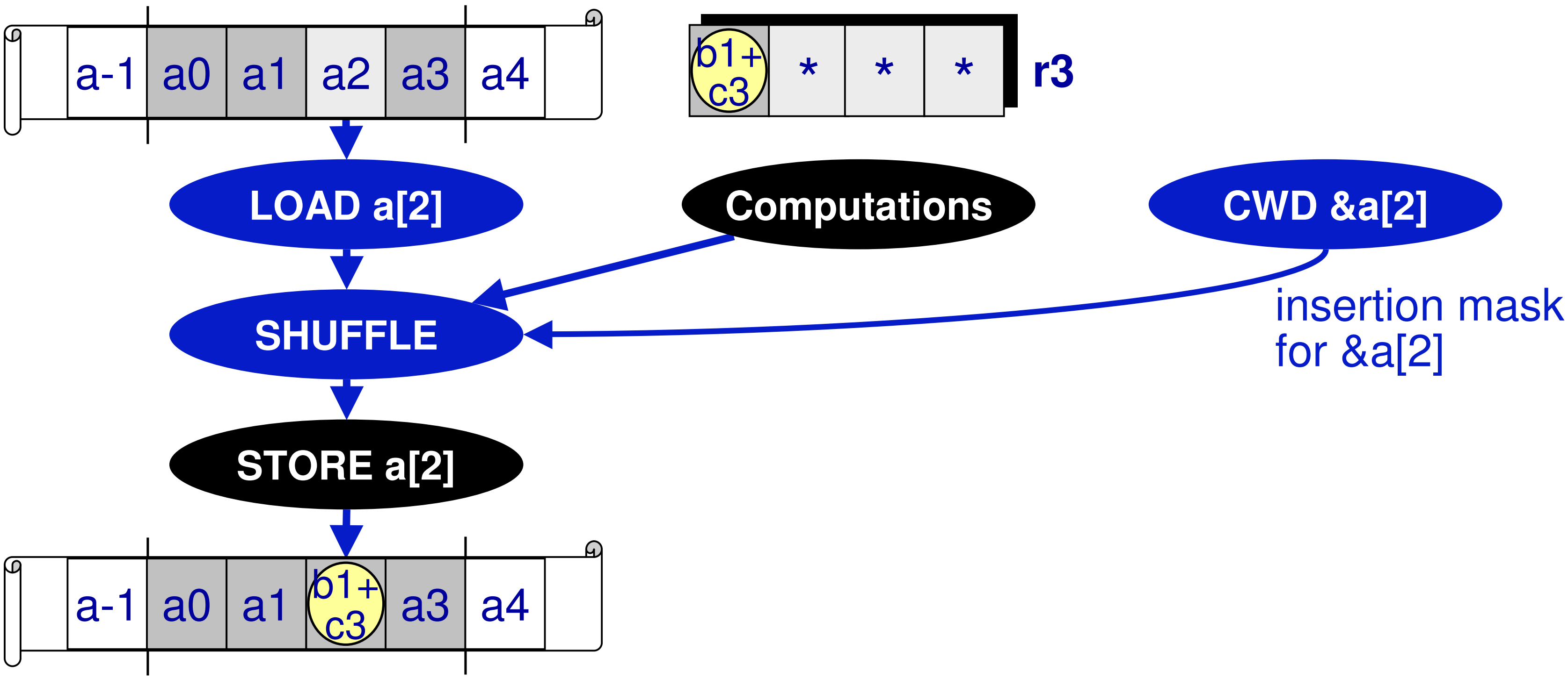
- one quad-word byte rotate

❑ Outcome

- desired scalar value always in the first slot of the register
- this addresses Problems 1 & 2

Scalar Store Handling

- **Use read-modify-write sequence**



- **Overhead (1 to 3 ops, in blue)**
  - one shuffle byte, one mask formation (may reuse), one load (may reuse)

- **Outcome**
  - SIMD store does not clobber memory (this addresses Problem 3)

Optimizations for Scalar on SIMD

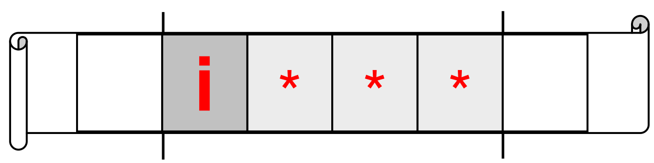
❑ Significant overhead for scalar load/store can be lowered

❑ For vectorizable code

- generate SIMD code directly to fully utilize SIMD units
- done by expert programmers or compilers (see SIMD presentation)

❑ For scalar variable

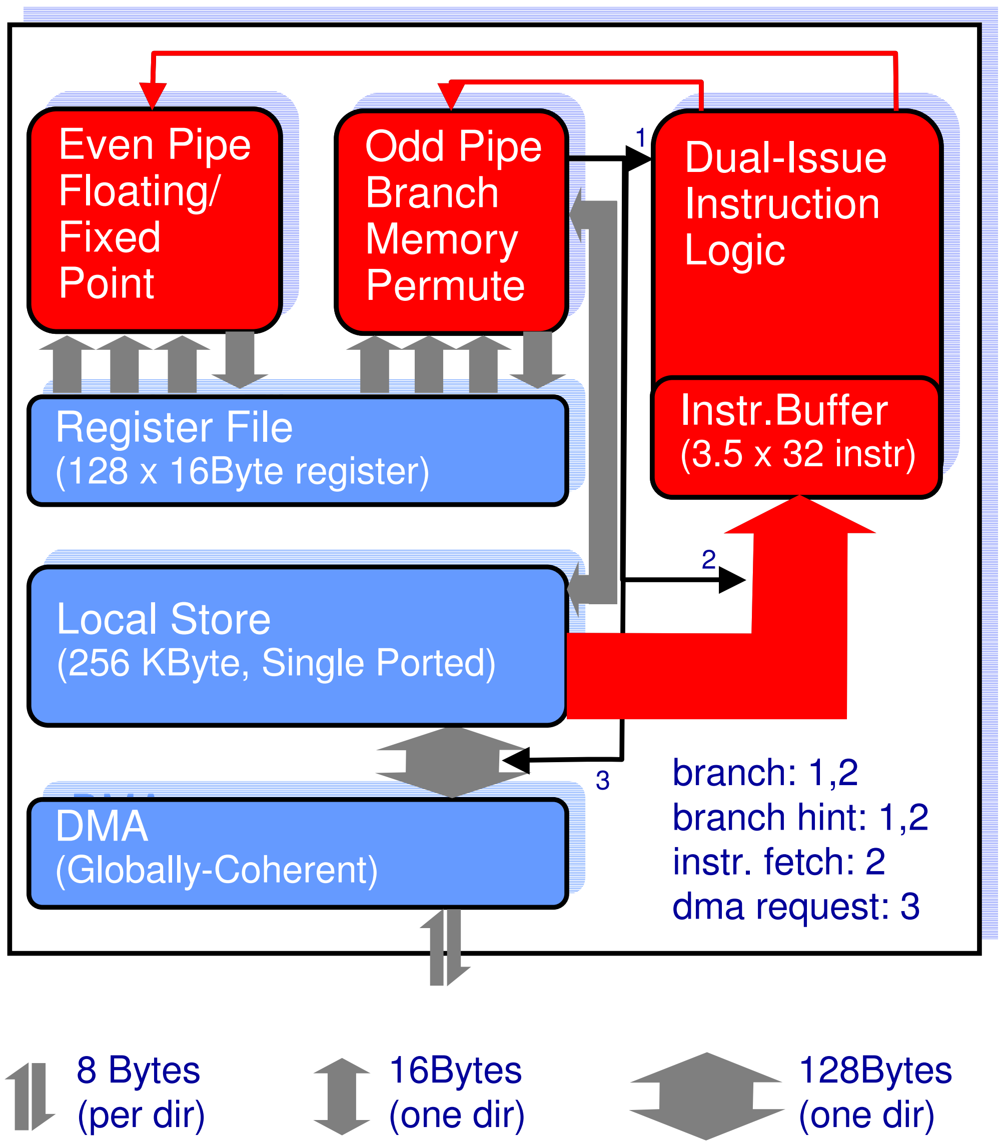
- allocate scalar variables in first slot, by themselves



- eliminate need for rotate when loading
  - data is guaranteed to be in first slot (Problems 1 & 2)
- eliminate need for read-modify-write when storing
  - other data in 16-byte line is garbage (Problem 3)
- can also deal with the alignment of scalar data

Feature #2: Software-Assisted Instruction Issue

SPE



- **Dual-issue for Instructions**
  - can dual-issue parallel instructions
  - code layout constrains dual issuing
  - full dependence check in hardware
- **Alleviate constraints by**
  - making the scheduler aware of code layout issue

Alleviating Issue Restriction

Scheduling finds the best possible schedule

- dependence graph modified to account for latency of false dependences

Bundling ensures code layout restrictions

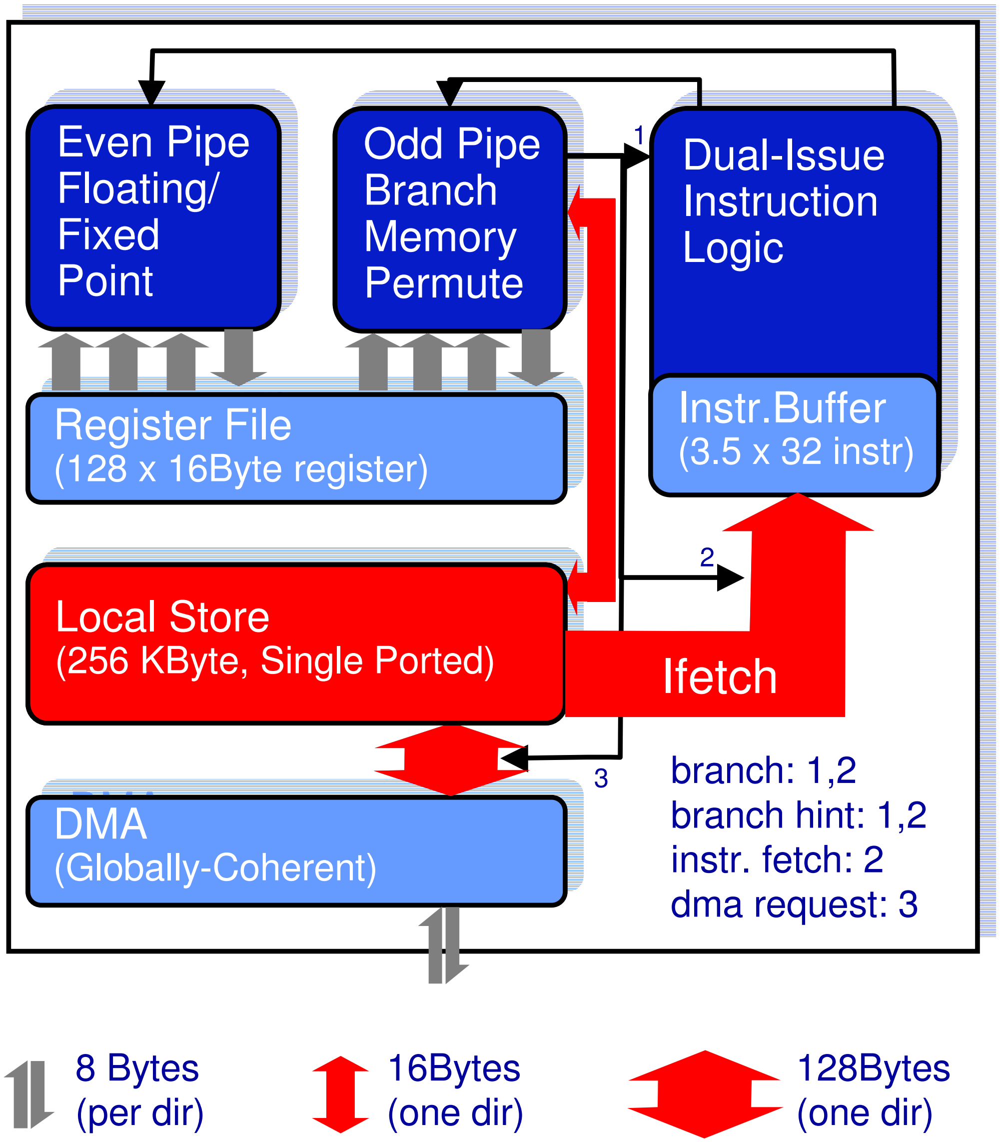
- keep track of even/odd code layout at all times
- swap parallel ops when needed
- insert (even or odd) nops when needed

Engineering issues

- each function must start at known even/odd code layout boundary
- one cannot add any instructions after the last scheduling phase as it would impact the code layout and thus the dual-issuing constraints

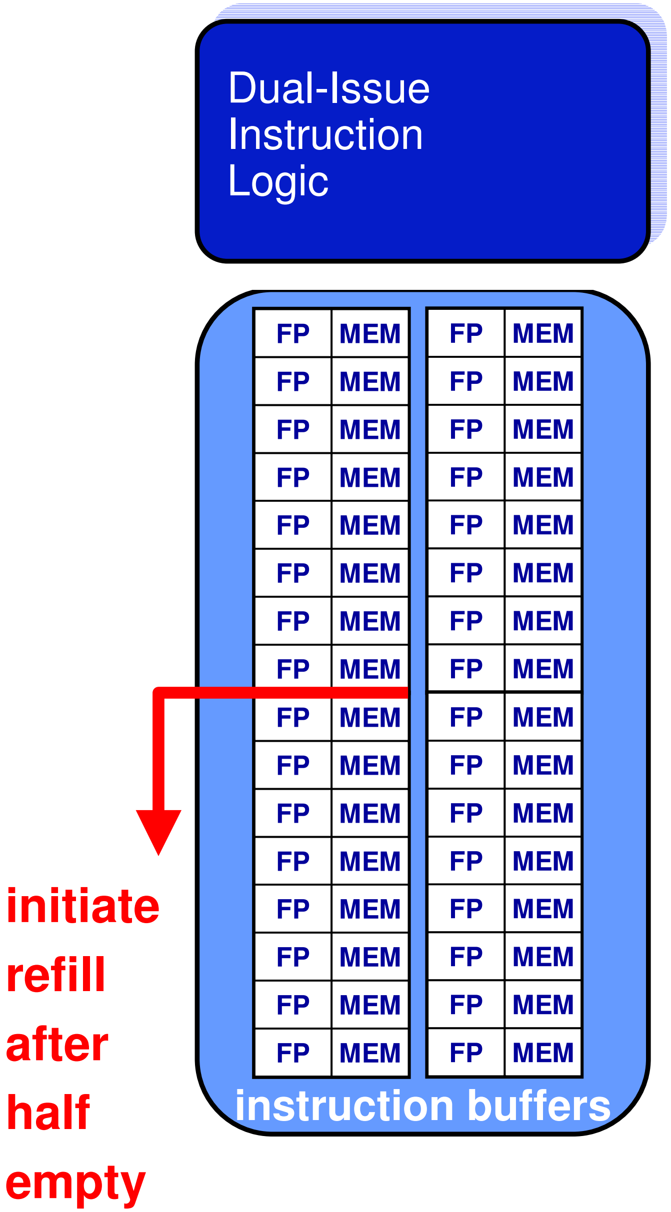
Feature 3: Single-Ported Local Memory

SPE



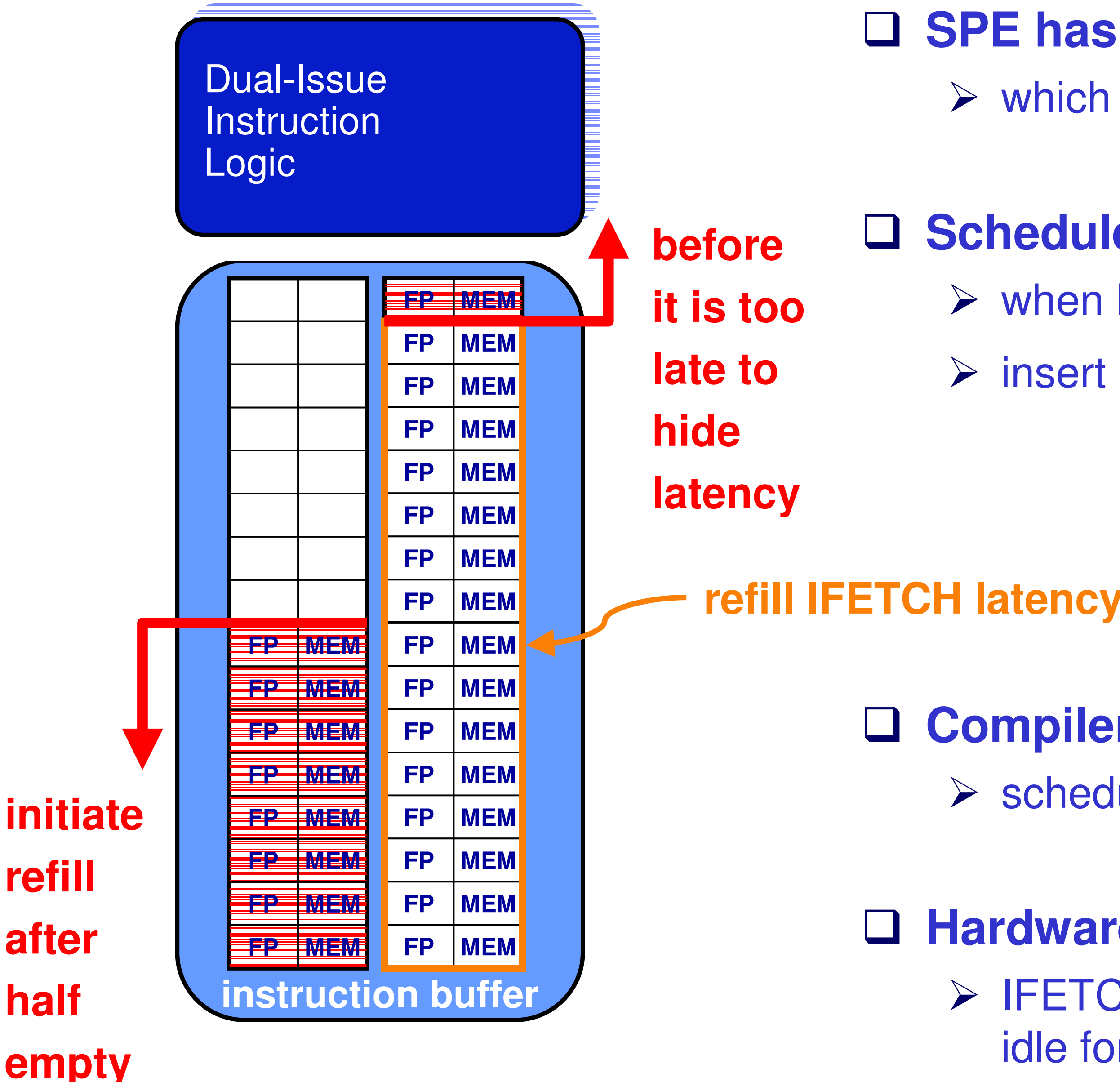
- **Local store is single ported**
  - denser hardware
  - asymmetric port
    - 16 bytes for load/store ops
    - 128 bytes for IFETCH/DMA
  - static priority
    - DMA > MEM > IFETCH
- **If we are not careful, we may starve for instructions**

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)

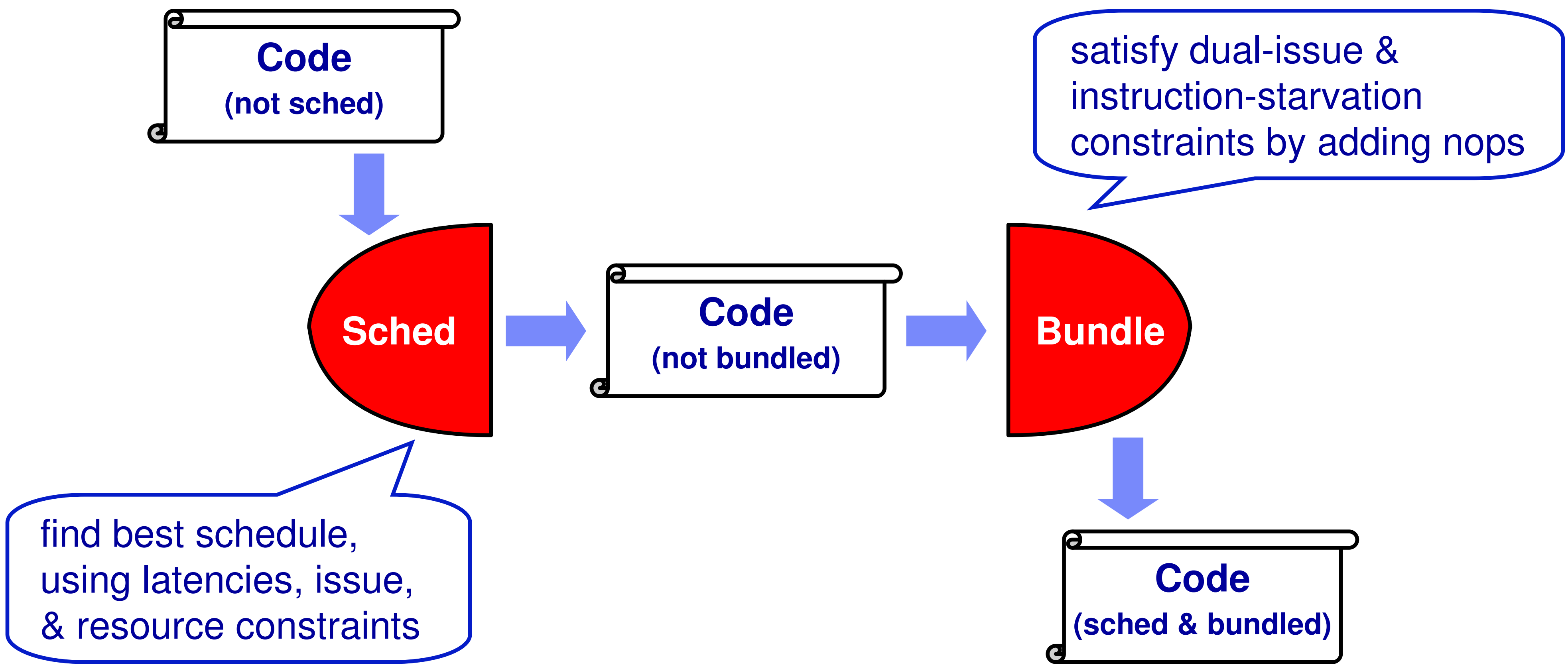
Instruction Starvation Prevention



- **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
- **Hardware design**
  - IFETCH op is not needed if memory port is idle for one or more cycles within the red window

Engineering Issues for Dual-Issue & Starvation Prevention

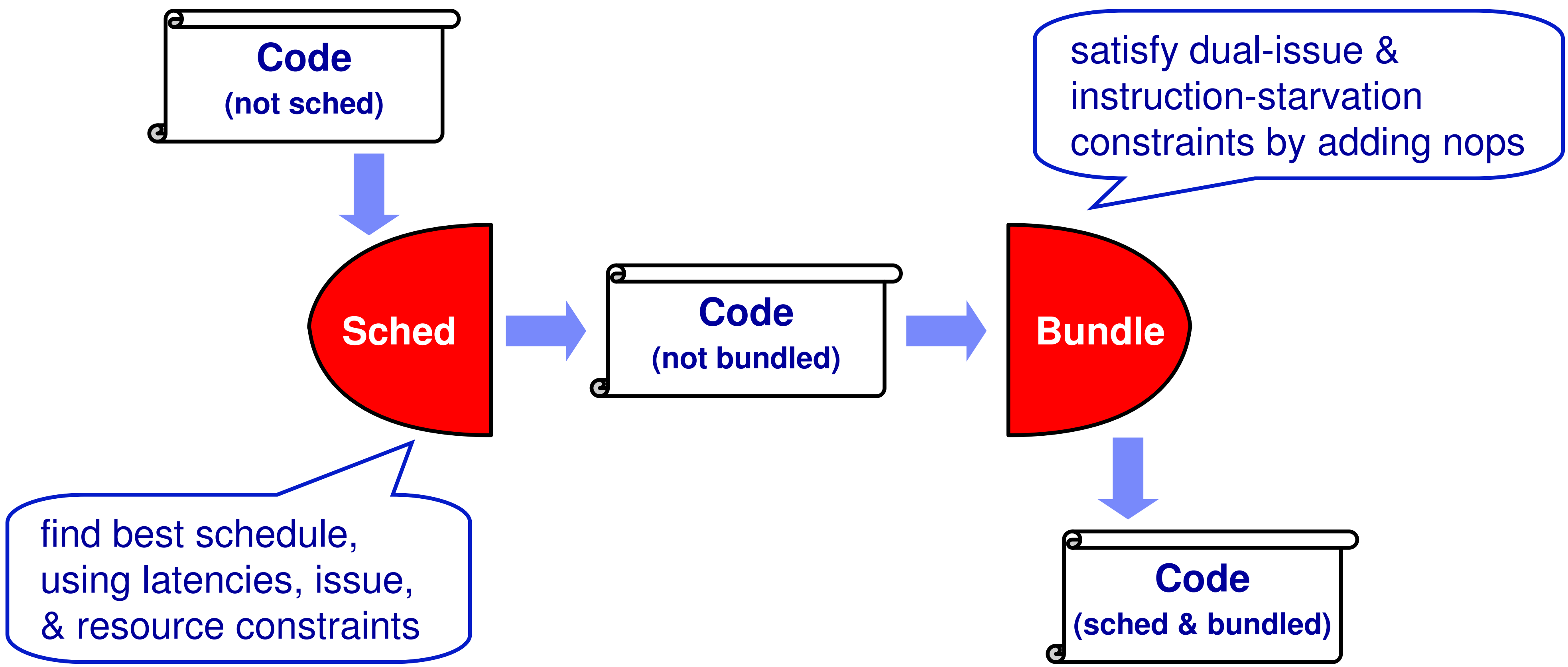
- Initially, the scheduling and bundling phases were separate



Problem: Bundler adds an IFETCH to prevent starvation.
A better schedule could be found if the scheduler had known that.
But the schedule is already "finalized".

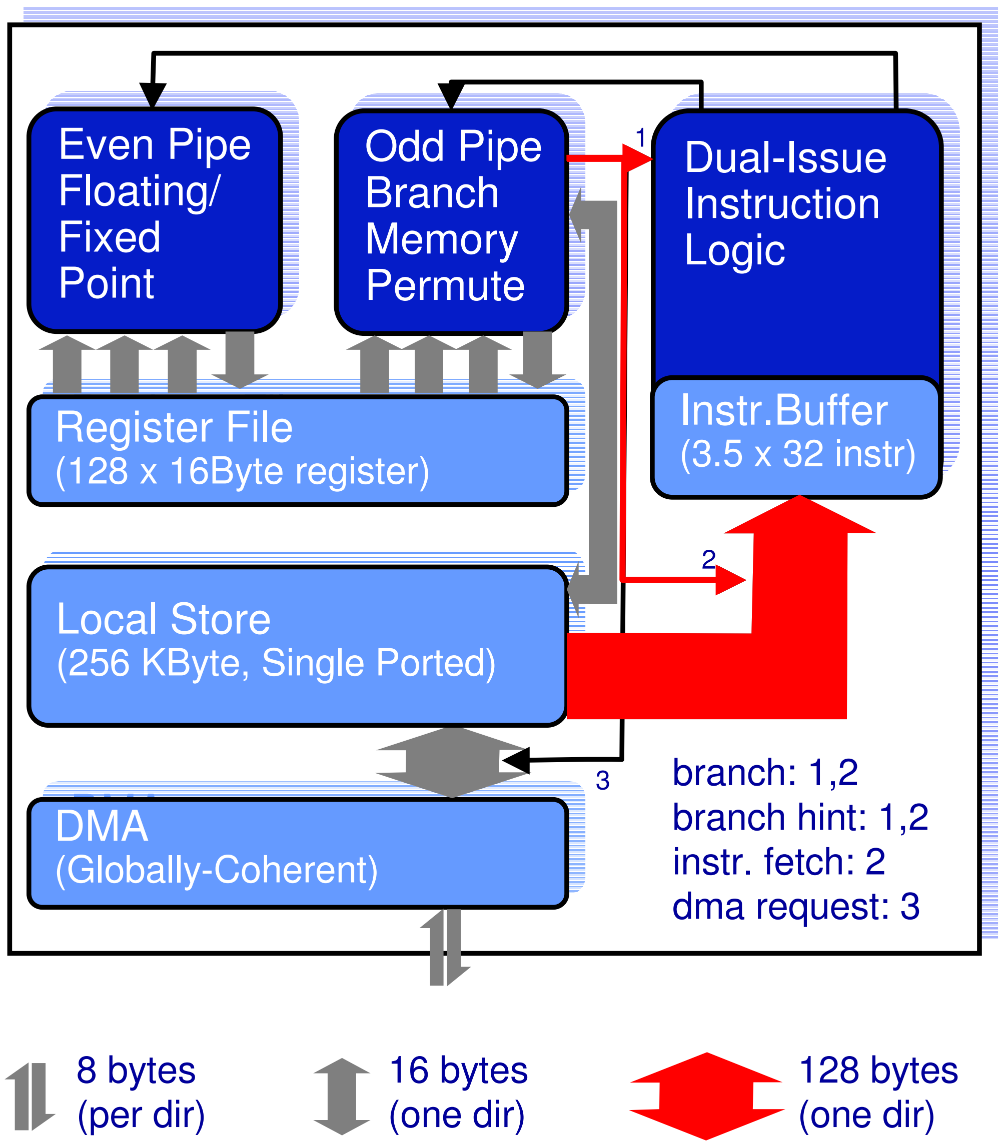
Engineering Issues for Dual-Issue & Starvation Prevention

- We integrate Scheduling and Bundling tightly, on a cycle per cycle basis



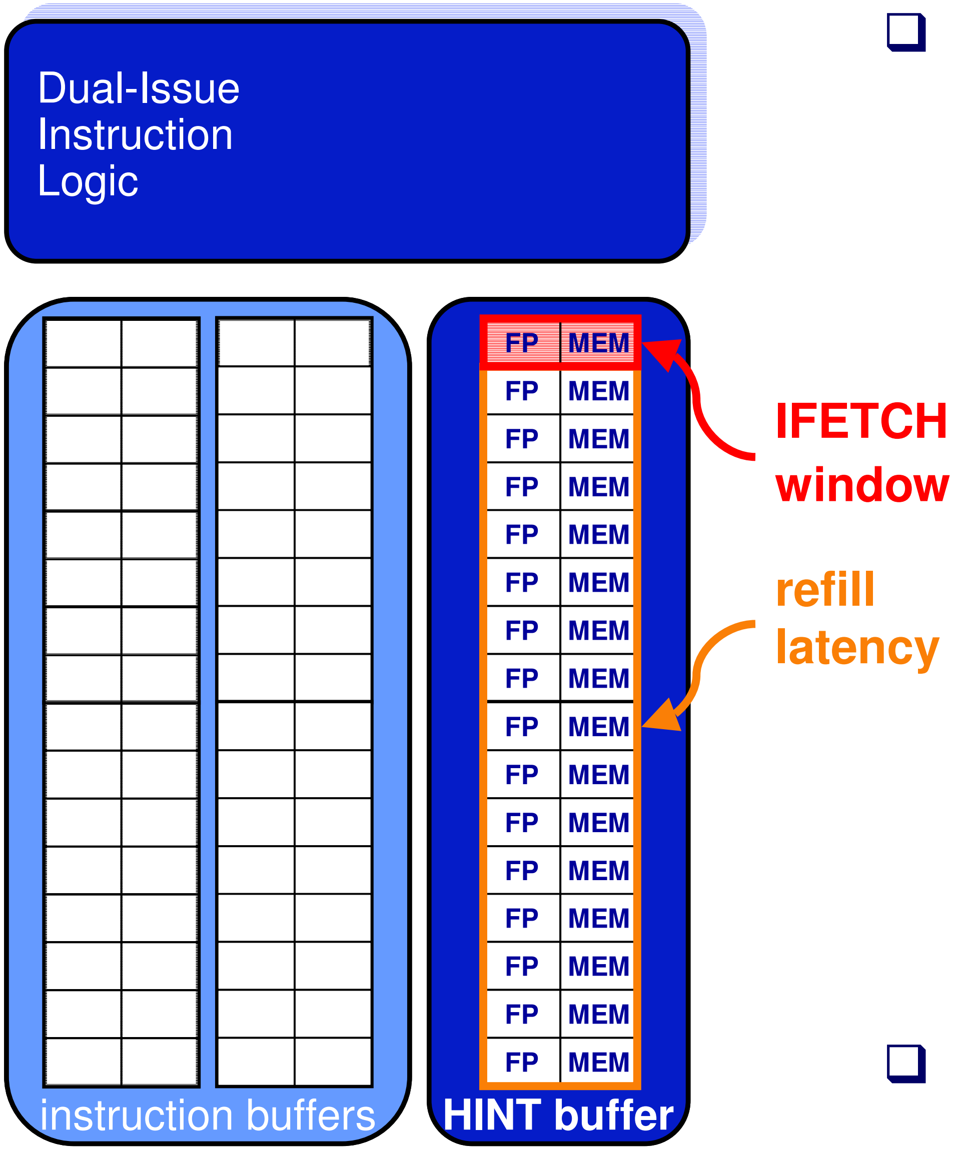
Feature #4: Software-Assisted Branch Architecture

SPE

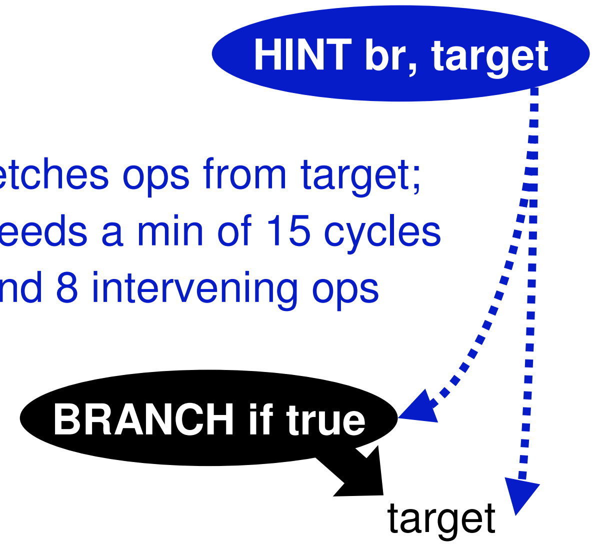


- **Branch architecture**
  - no hardware branch-predictor, but
  - compare/select ops for predication
  - software-managed branch-hint
  - one hint active at a time
- **Lowering overhead by**
  - predicating small if-then-else
  - hinting predictably taken branches

Hinting Branches & Instruction Starvation Prevention



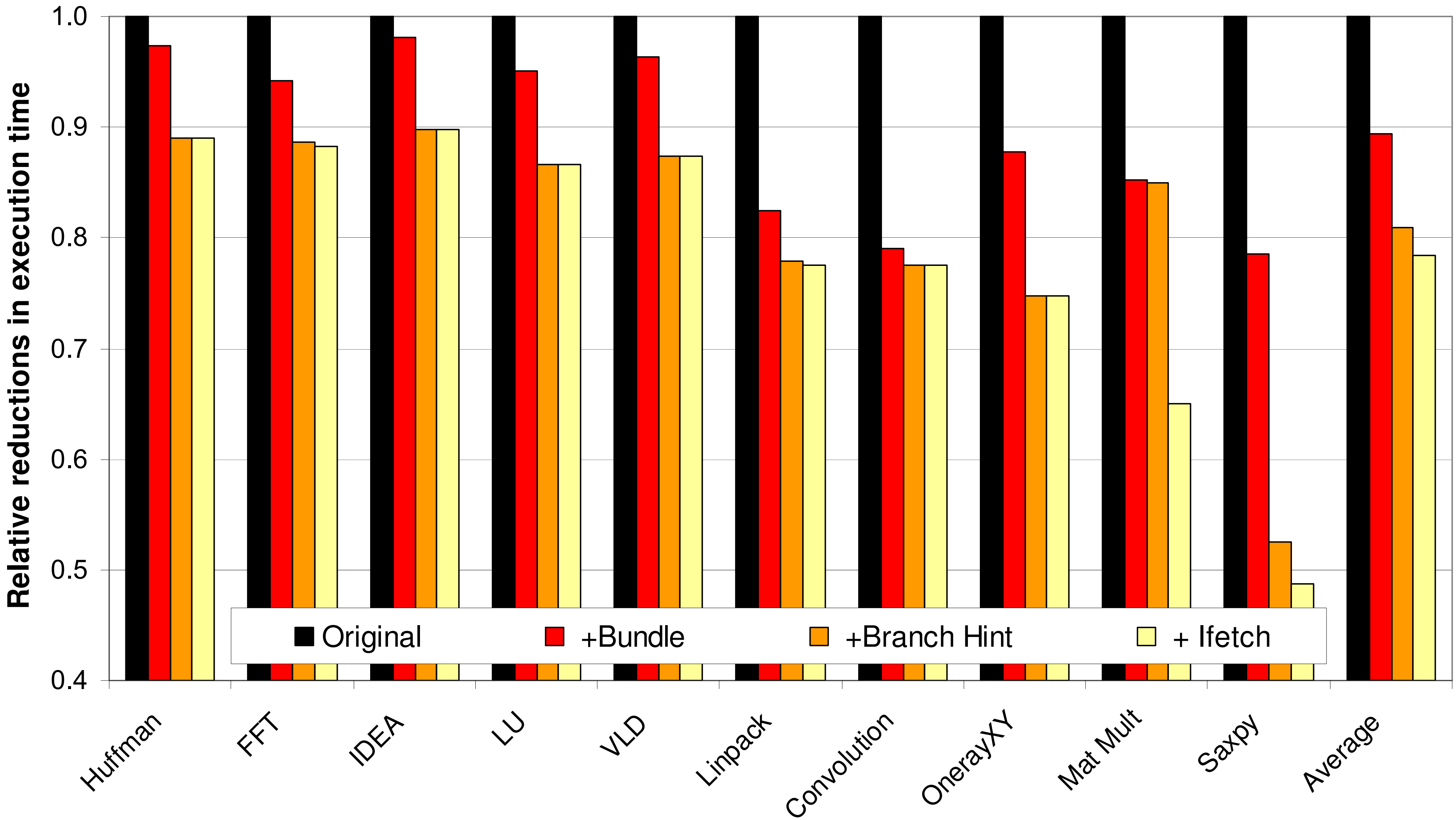
- **SPE provides a HINT operation**
  - fetches the branch target into HINT buffer
  - no penalty for correctly predicted branches



  - 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

Multiple-ISA hand-tuned programs

PROGRAMS

Automatic tuning for each ISA

Part 3:
Automatic simdization

Explicit SIMD coding

SIMD

SIMD/alignment directives

Automatic simdization

Part 2:
Parallelization and memory abstraction

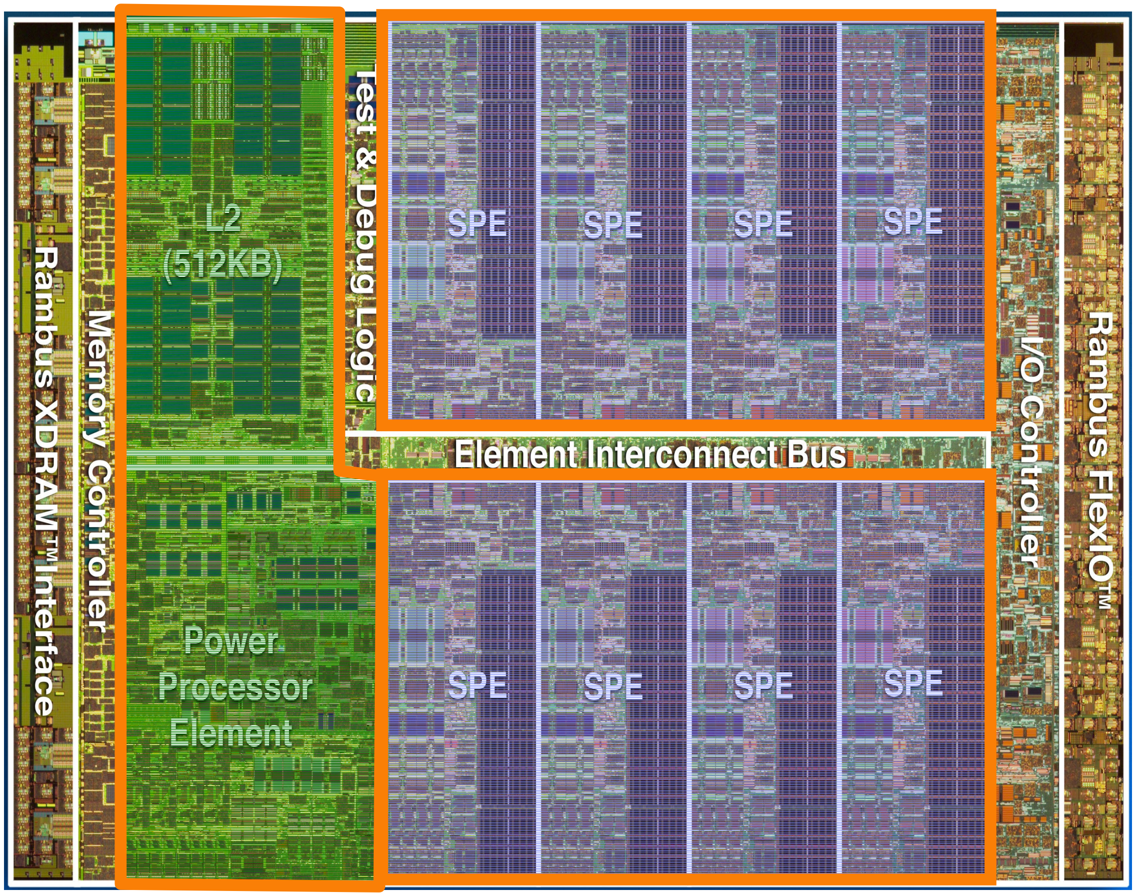
Explicit parallelization with local memories

PARALLELIZATION

Shared memory, single program abstraction

Automatic parallelization

Cell Broadband Engine



- **Multiprocessor on a chip**
  - Power Proc. Element (PPE)
    - general purpose
    - running full-fledged OSs
  - Synergistic Proc. Element (SPE)
    - optimized for compute density
- **Compiler support for parallelizing across the heterogeneous processing elements**

Single Source Compiler

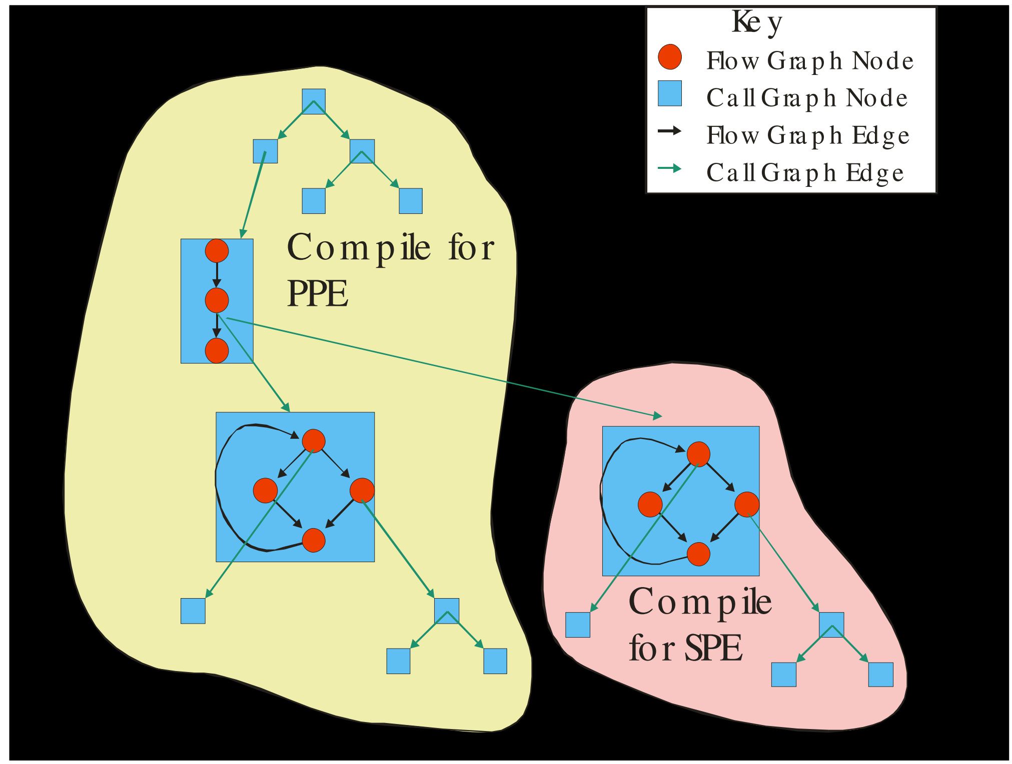
- user prepares an application as a collection of one or more source files containing user directives targetting the Cell architecture
- compiler, guided by these directives, takes care of the code partitioning between PE and SPE
- compiler takes care of data transfers.
  - identifies accesses in SPE functions that refer to data in system memory locations
  - uses software cache or static buffers to manage transfer of this data to/from SPE local stores
- compiler takes care of code size
  - explore extending Code partitioning to Single Source, i.e. automatic partitioning based on functionality rather than size

Using OpenMP to partition/parallelize across Cell

- Single source program contains C or Fortran with user directives or pragmas
- Compiler ‘outlines’ all code within the pragmas into separate functions compiled for the SPE.
- Replaces ‘outlined’ code with call to the parallel runtime and compiles this code for the PPE
- Master thread continues to execute on PPE and participate in workshare constructs
- PPE Runtime
  - places outlined functions on a work queue containing information about number of iterations to execute, or ‘chunk’ size for each SPE
  - Creates up to 8 SPE threads to pull work items (outlined parallel functions) from queue and execute on SPEs
- May wait for SPE completion, or proceed with other PPE statement execution

Cloning for heterogeneous processors

- **Outlined function becomes new node in call graph**
- **In pass 2 of TPO, using whole program call graph, outlined function is cloned, then specialized to create a ppe and an spe version**
- **All called functions must also be cloned**
- **SPE call sites modified to call SPE versions of cloned subroutines**
- **Partitioning pass creates SPE and PPE partitions and invokes lower level optimizer for machine specific optimization**



"Single Source" Compiler, using OpenMP

```
Foo1();
#pragma Omp parallel for
for( i =0; i<10000; i++)
    A[i] = x* B[i];
Foo2();
```

outline omp region

```
void ol$1()
        for( i =0; i<10000; i++)
            A[i] = x* B[i];
```

Compile for PPE

```
Foo1();
Call omp-rte-init;
Call omp_rte_do_par;
Foo2();
```

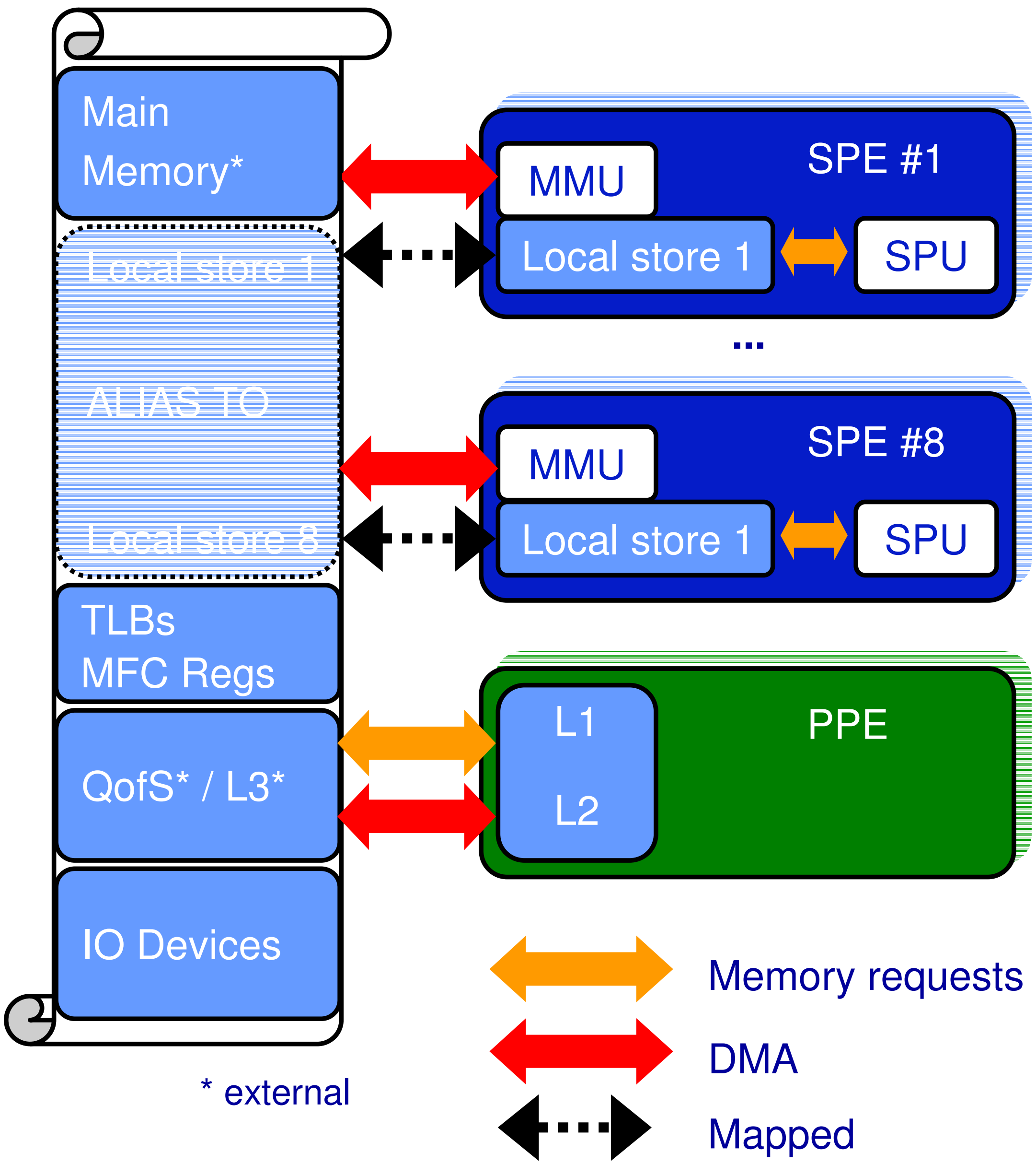
clone for SPE

```
void OL$1_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´
```

clone for PPE

```
void OL$1_PPE()
    for( i=LB;  i<UB;  i++) ´
        A[i] = x * B[i];
```

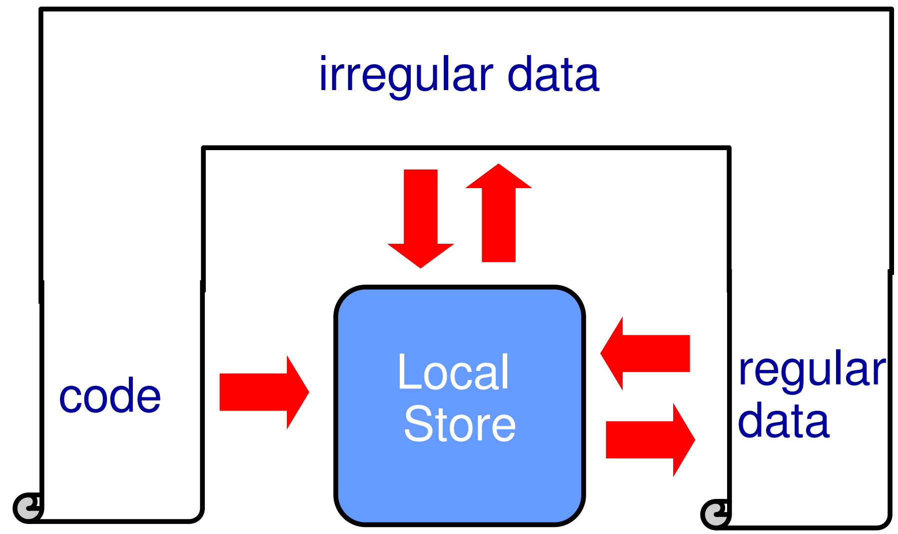
Cell Memory & DMA Architecture



- **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)

How can we run programs that have lots of data?

- **Data won't all fit in Local Store**

  - Solution: Use System Memory, its large and virtual
    - But the SPU doesn't have Load/Store access to it
  - The compiler can automatically manage the transferring of Data between System Memory, and Local Store
  - We call this Data Partitioning

Data Partitioning

- **Single Source assumption: all data lives in System Memory**
- **Naïve implementation, every load and store requires a dma operation**
  - Too costly (~300 cycles per load or store)
  - MP will require locking on every reference
- **What can be done to make this acceptable?**

Reducing the Overhead of Accesses to System Memory

- **Rather than executing a DMA transaction for every variable access, we would like to bring data over in larger pieces, and keep it in Local Store for some portion of its lifetime**

- **There are several techniques that can accomplish this**
  - Prefetching predictable references, and accumulating writes
  - Software Cache
    - on demand, but leverage spatial and temporal locality
    - requires additional instructions inline, so it is essentially a fallback strategy
  - We can apply Tiling to increase reuse for both of these

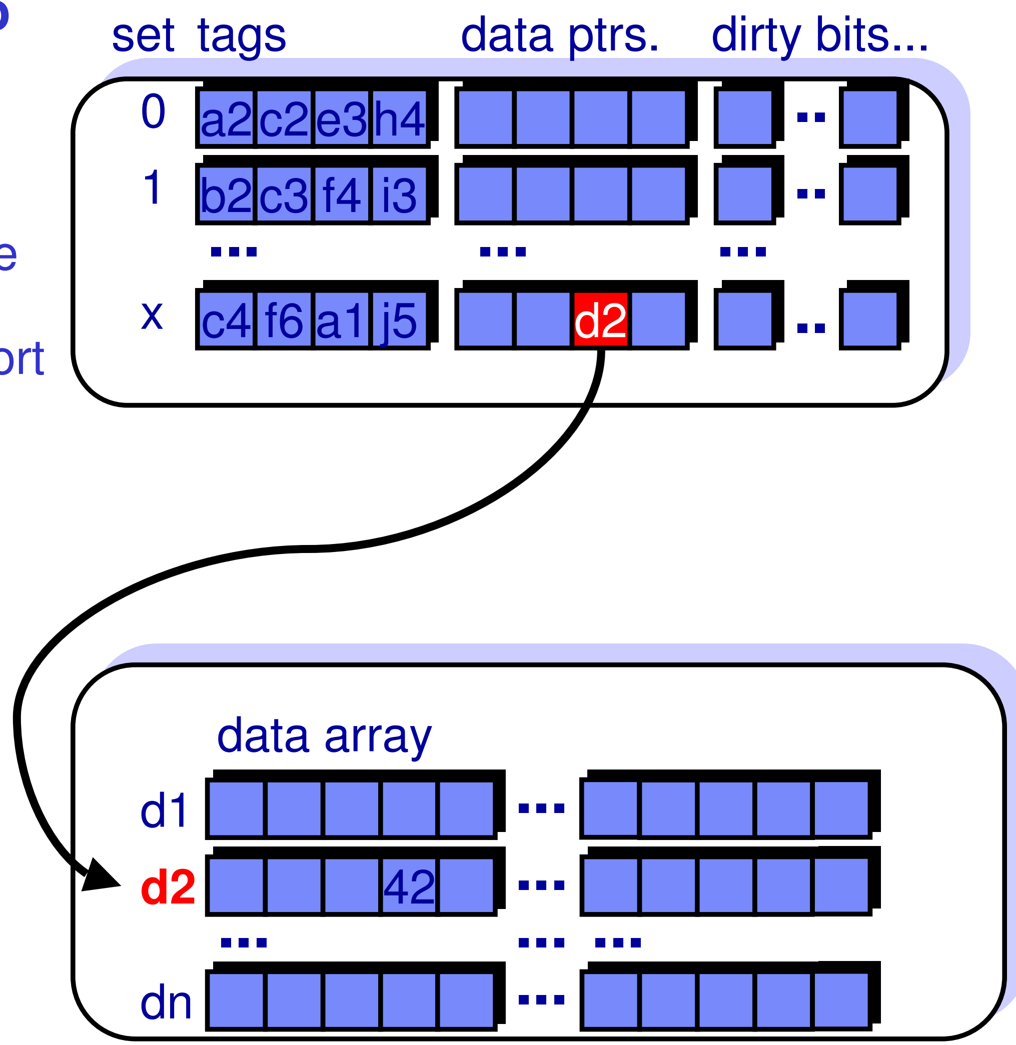
Software Cache for Irregular Accesses

- **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 Overview

- **A portion of Local Store is set aside to hold the cache**
- **Cache is made up of two arrays**
  - Tag Array: Contains the comparands for the address lookups and pointers to the data lines, also contains “dirty-bits” for MP support
  - Data Array: Contains the data lines
- **Geometry**
  - Currently, 4-way Set Associative
  - line size is 128 bytes, and there are 128 of them in each set
  - these parameters can be changed
- **Total size, Tags (16K) + Data(64K) is 80K**

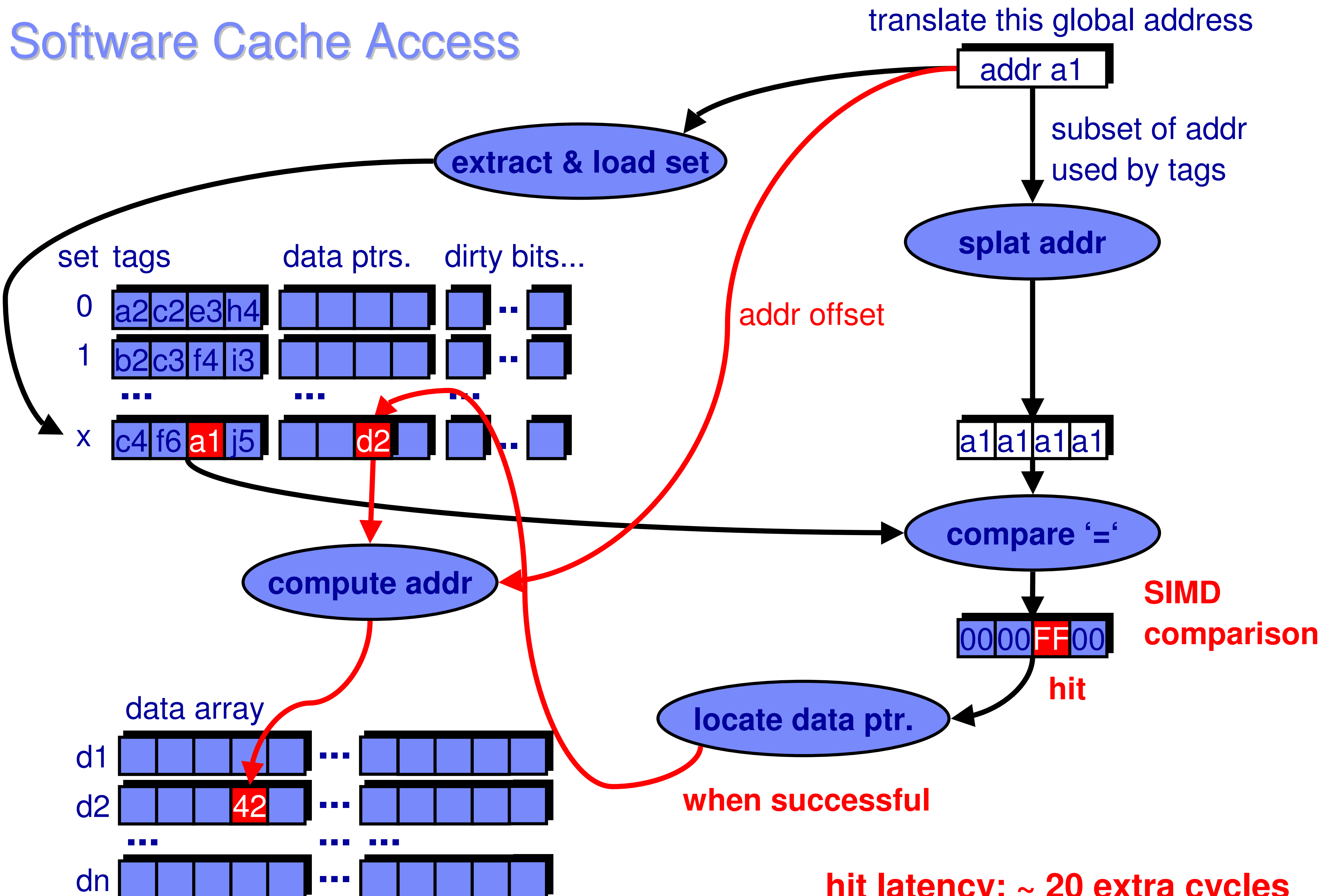


Software Cache Lookup

- The cache lookup code is inserted inline at each Read or Write reference to a Cacheable Variable
- This code is optimized along with all the other instructions
- The branch to the misshandler is treated as a regular instruction throughout optimization, and expanded only at the very end of compilation
- The misshandler uses a tailored register convention, so that it has no impact on the hit path

```
20:    VAND     vr844=gr664,vr842
20:    LI       vr847=0x10203
20:    VLR     *vr846=vr847
20:    VSHUFB   vr848=gr664,gr664,vr846
20:    VLR     *vr845=vr848
20:    VAND     vr849=vr843,vr845
20:    VLQ      vr850=.L_tagaddr(relative,0)
20:    A        gr851=vr844,vr850
20:    VLQ      vr852=.L_tagarray(gr851,0)
20:    VLQ      vr853=.L_tagarray(gr851,16)
20:    VANDC    vr854=gr664,vr843
20:    VCEQW    vr855=vr849,vr852
20:    VGBB     vr856=vr855
20:    VCNTLZ   vr857=vr856
20:    VQBR     vr858=vr853,vr857,gr1
20:    MISS    *vr858,.L_tagarray=gr664,vr856,vr858
20:    A        gr859=vr854,vr858
20:    VLR     *gr799=gr859
20:    LI       vr847=0x10203
20:    VLR     *vr846=vr847
20:    VSHUFB   vr848=gr664,gr664,vr846
20:    VLR     *vr845=vr848
20:    VLQ      vr800=c[]0(gr799,0)
20:    LR      *vr663=vr800
```

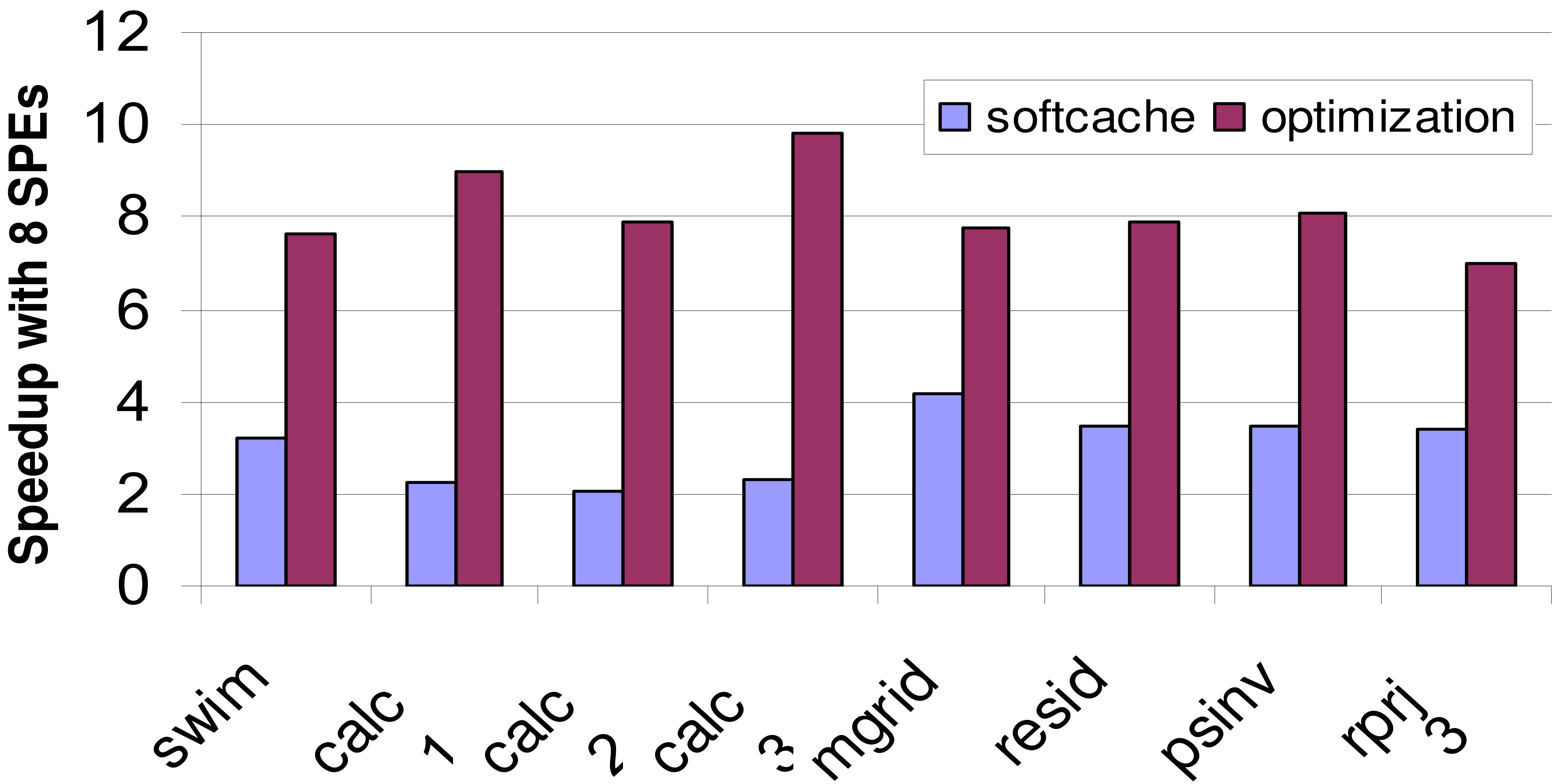
Software Cache Access



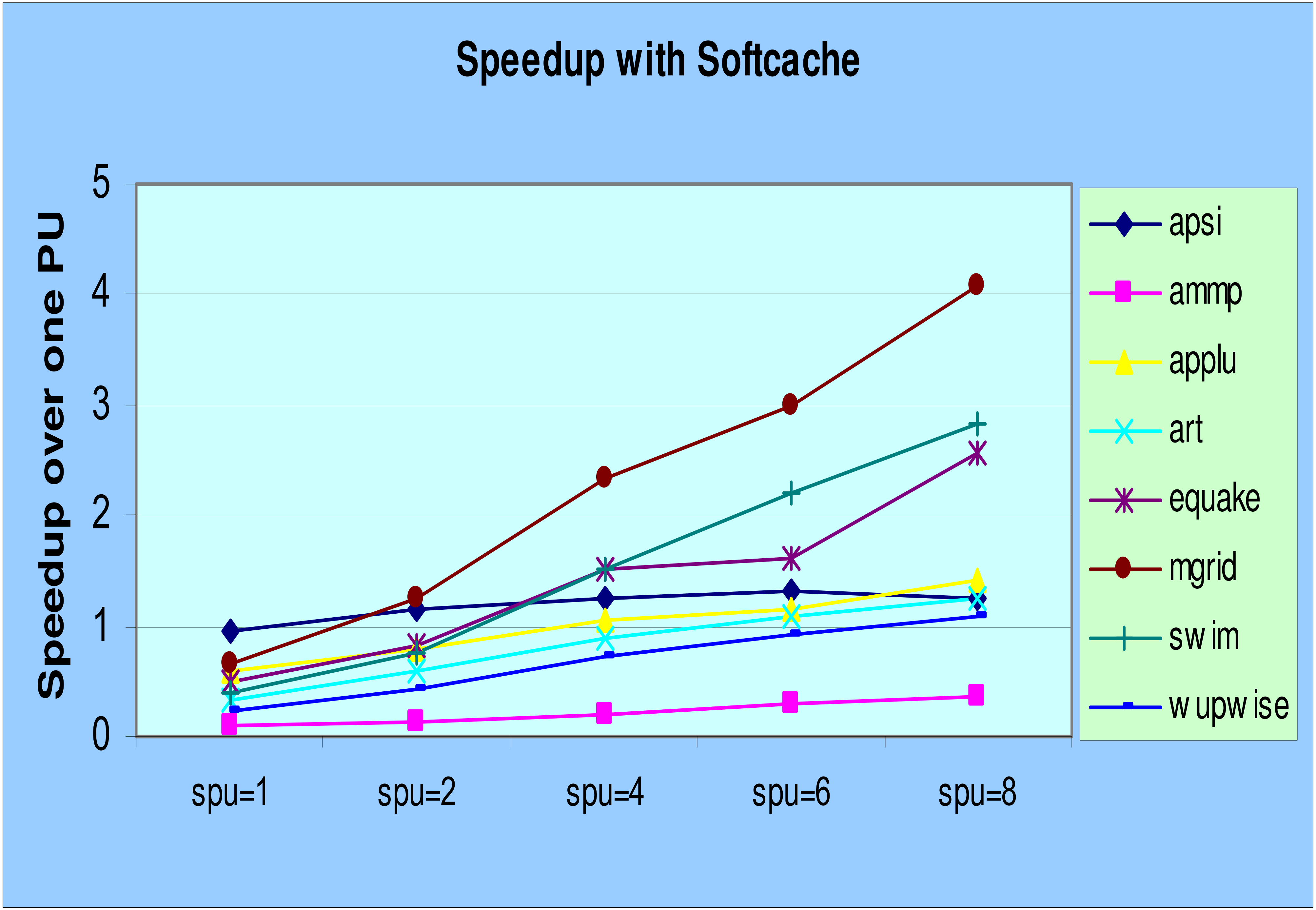
hit latency: ~ 20 extra cycles

Single Source Compiler Results

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



baseline: execution on one single PPE


Speedup with Softcache
Speedup over one PU
0
1
2
3
4
5
spu=1
spu=2
spu=4
spu=6
spu=8
apsi
ammp
applu
art
equake
mgrid
sw im
w upw ise

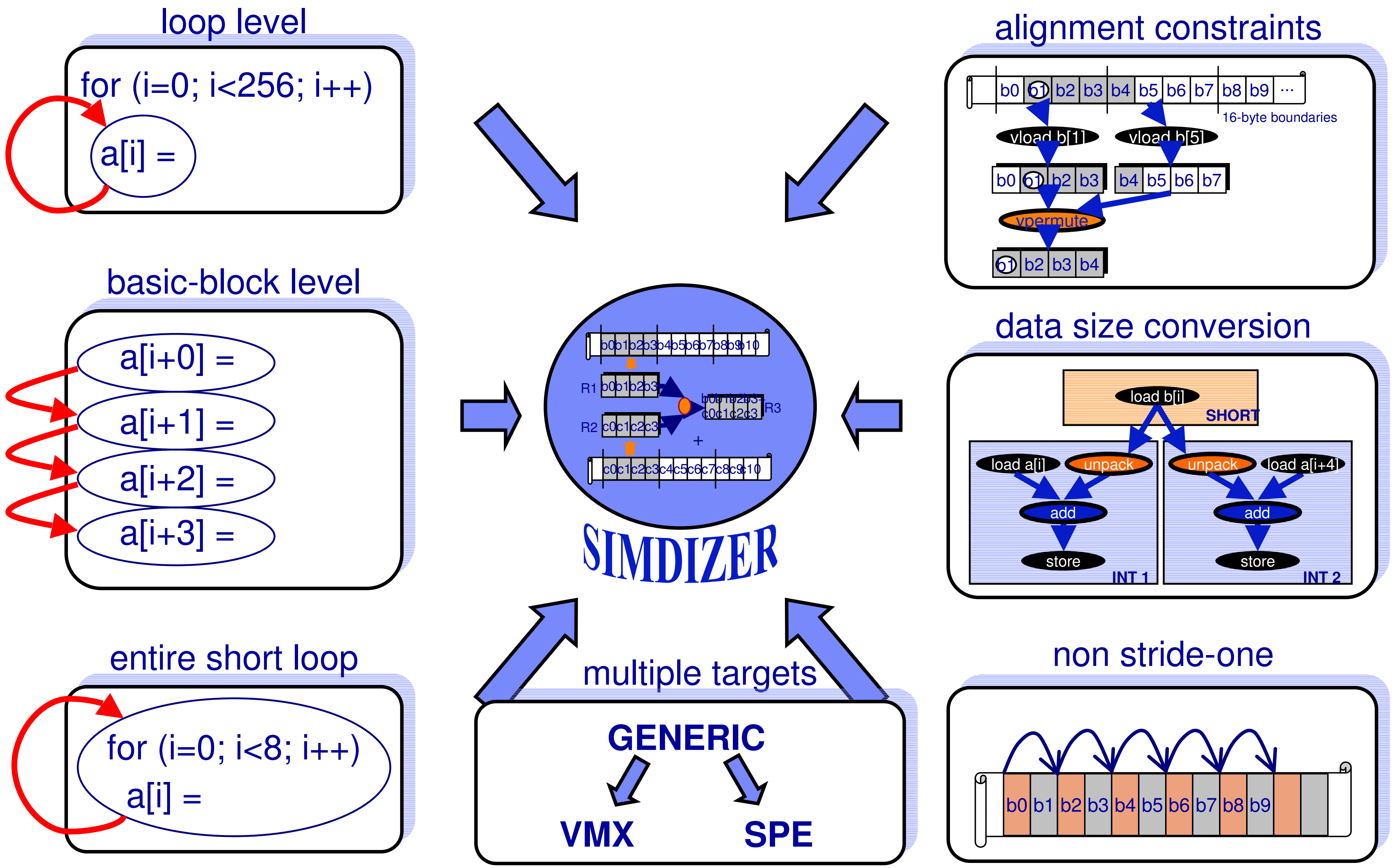
Outline

Part 1: Automatic SPE tuning	Part 3: Automatic simdization	Part 2: Parallelization and memory abstraction
Multiple-ISA hand-tuned programs	Explicit SIMD coding	Explicit parallelization with local memories
PROGRAMS	SIMD	PARALLELIZATION
	SIMD/alignment directives	Shared memory, single program abstraction
Automatic tuning for each ISA	Automatic simdization	Automatic parallelization

Successful Simdizer

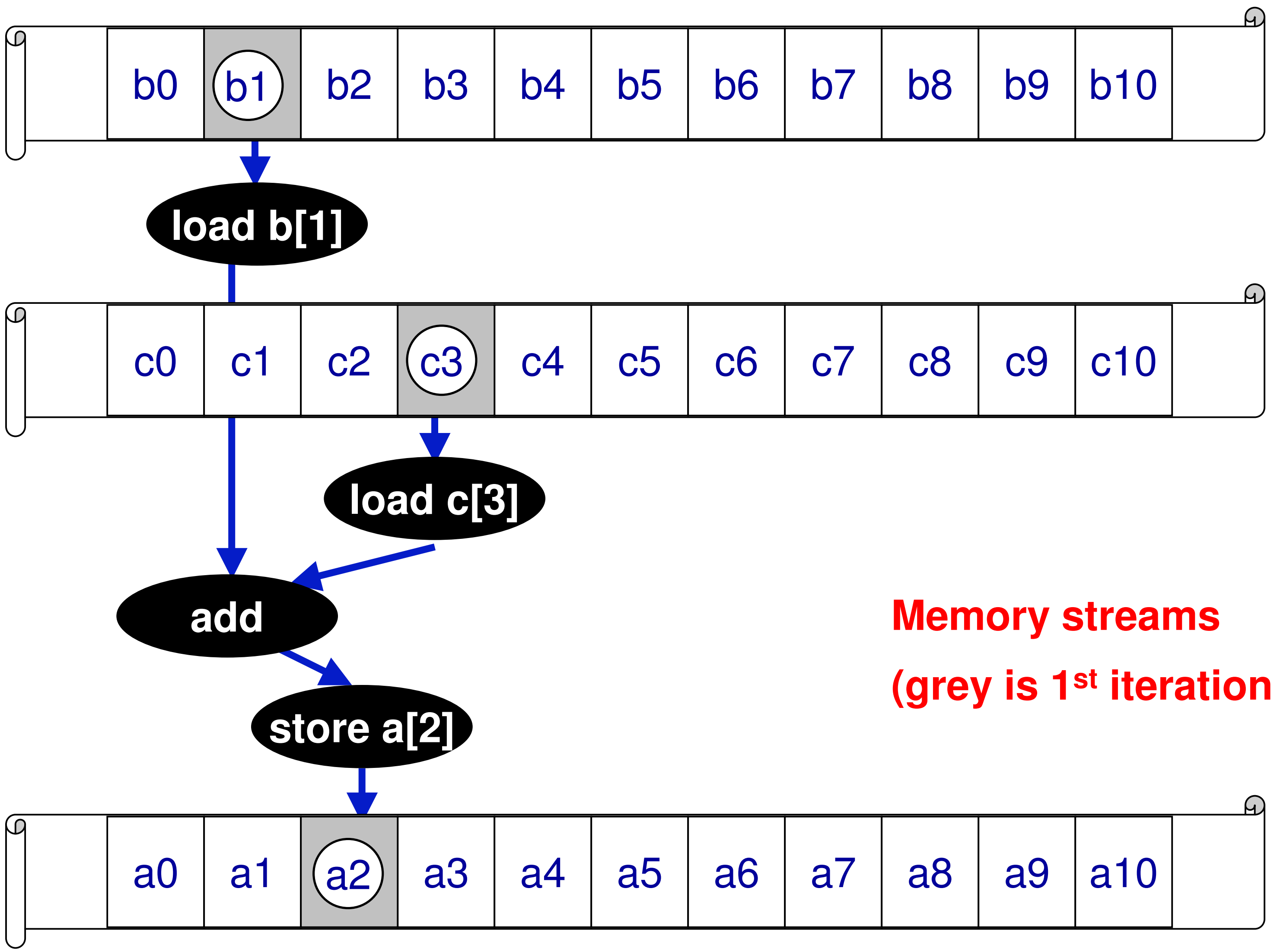
Extract Parallelism

Satisfy Constraints



Traditional Execution of a Loop

- **Sequential execution of** "for (i=0;i<64;i++) **a[i+2] = b[i+1] + c[i+3]** "

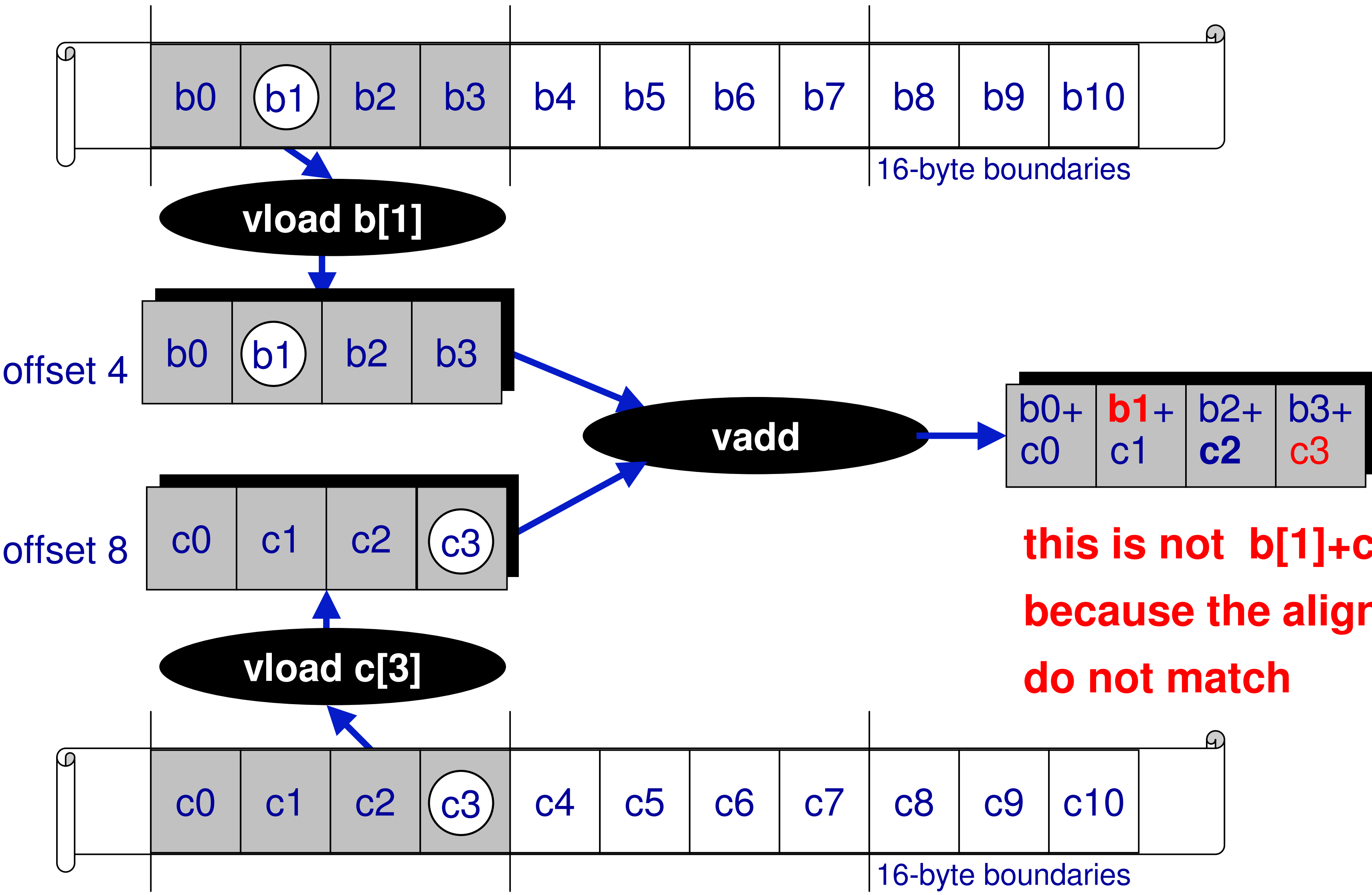


Memory streams

(grey is 1st iteration)

SIMD Alignment Problem

- **SIMD execution of** “for(i=0;i<64;i+) **a[i+2] = b[i+1] + c[i+3]**”



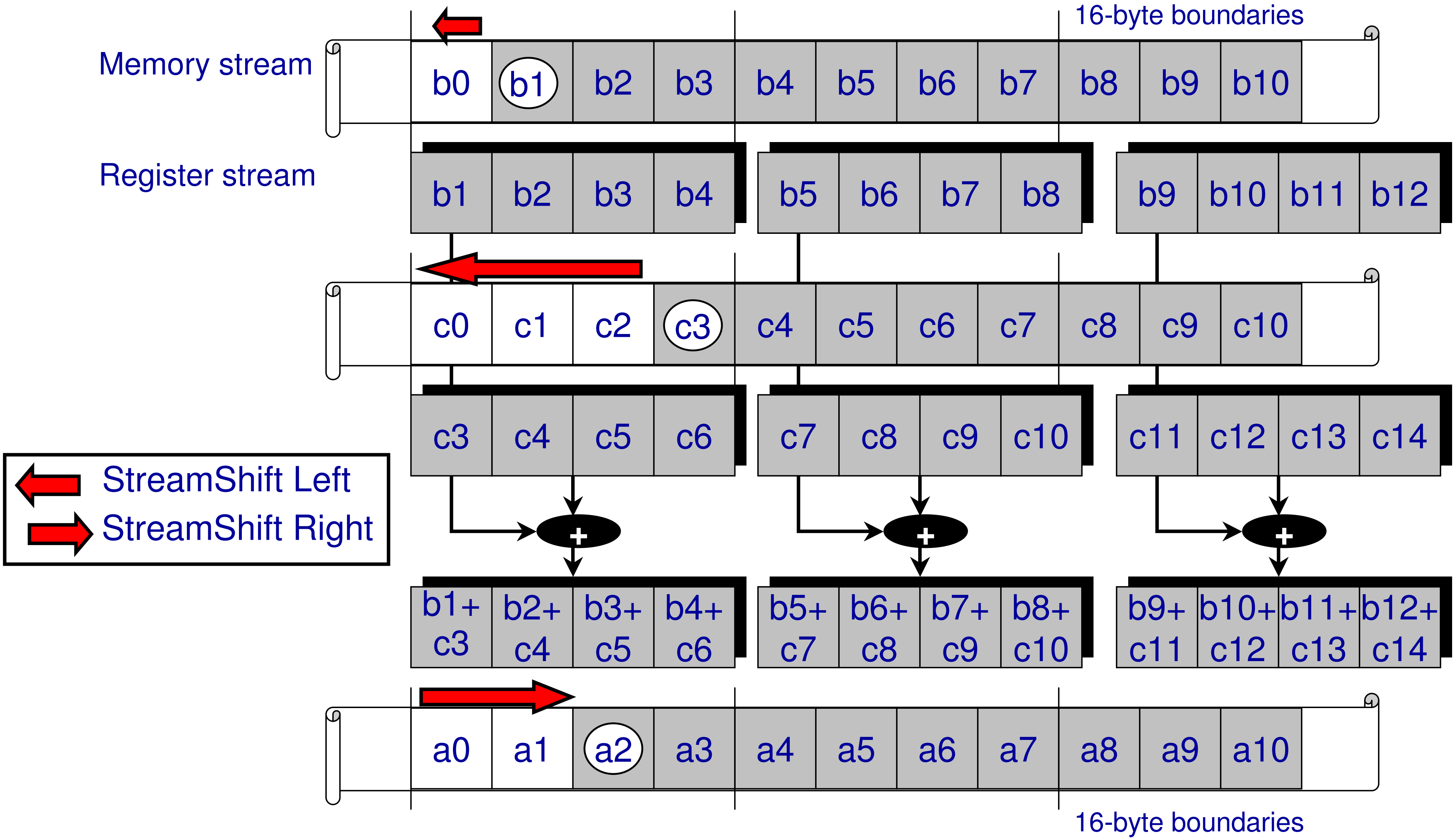
this is not b[1]+c[3], ...

because the alignments

do not match

Loop-Level Simdization, Naïve Way

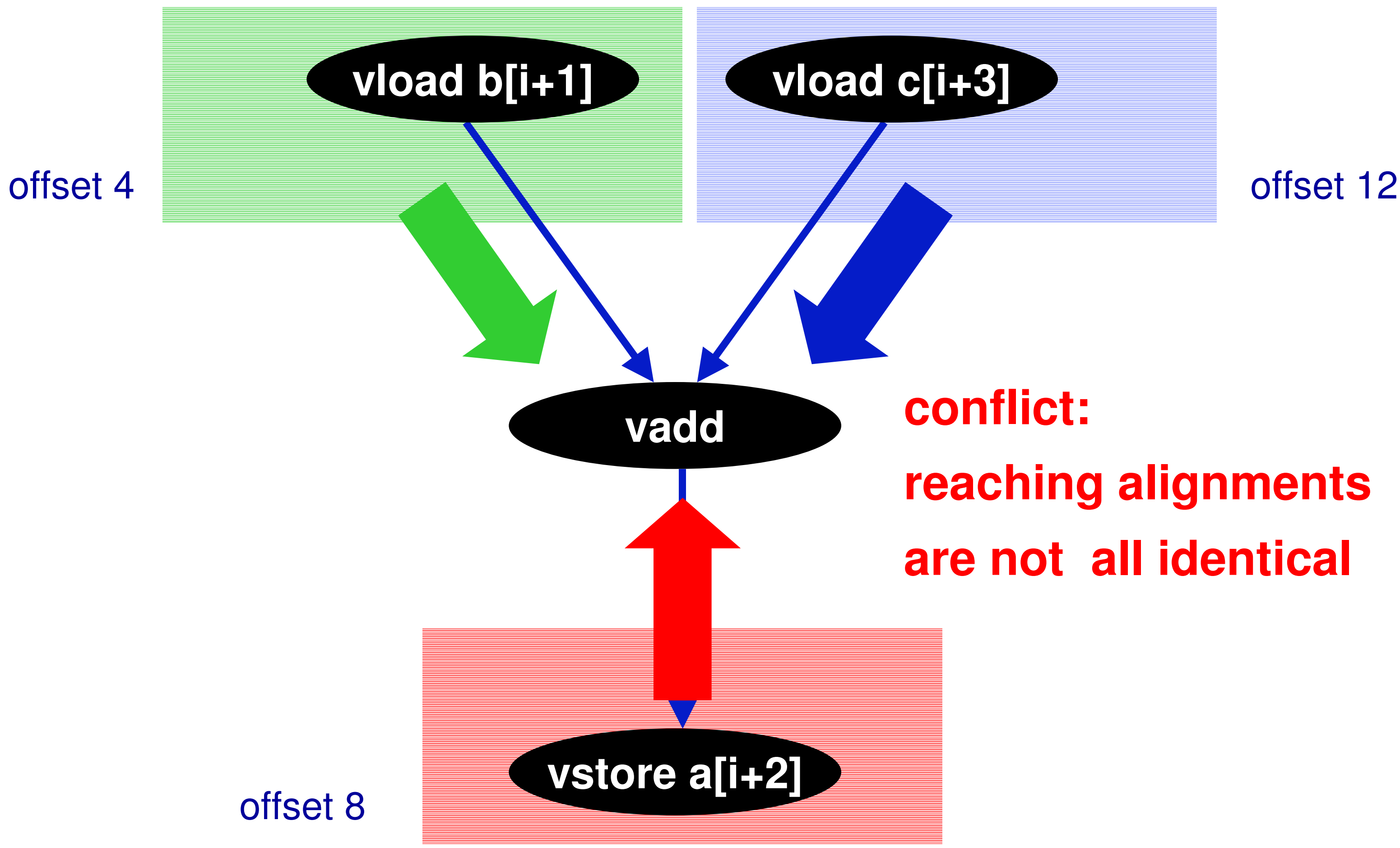
- **SIMD execution of** "for(i=0;i<64;i+) **a[i+2] = b[i+1] + c[i+3]**"



Solving the Alignment Problem

❑ Data Reorganization Graph

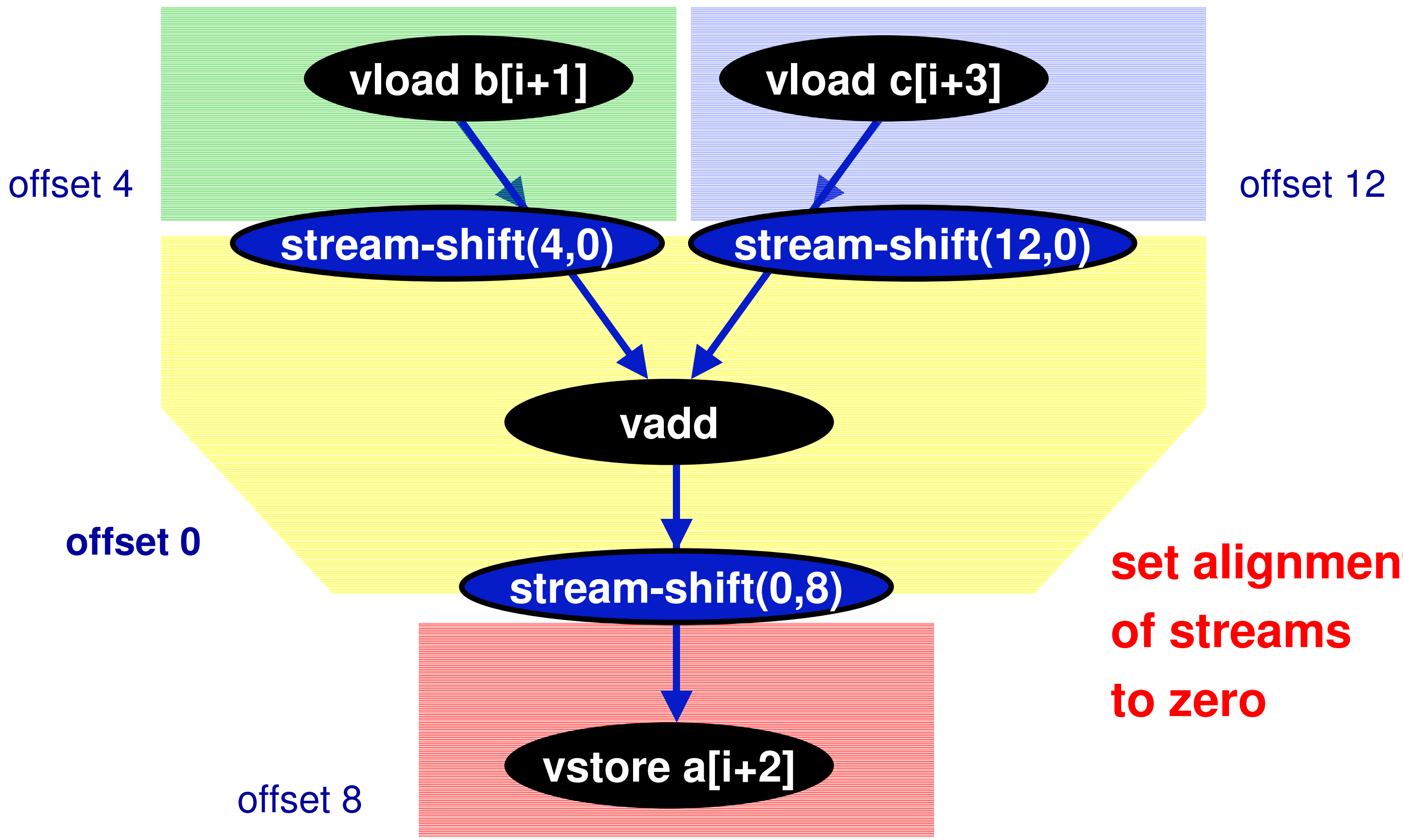
- original graph with alignment label each load/store
- resolve alignment conflicts



Solving the Alignment Problem

❑ Data Reorganization Graph

- original graph with alignment label each load/store
- resolve alignment conflicts by adding "stream-shift" aligning operations

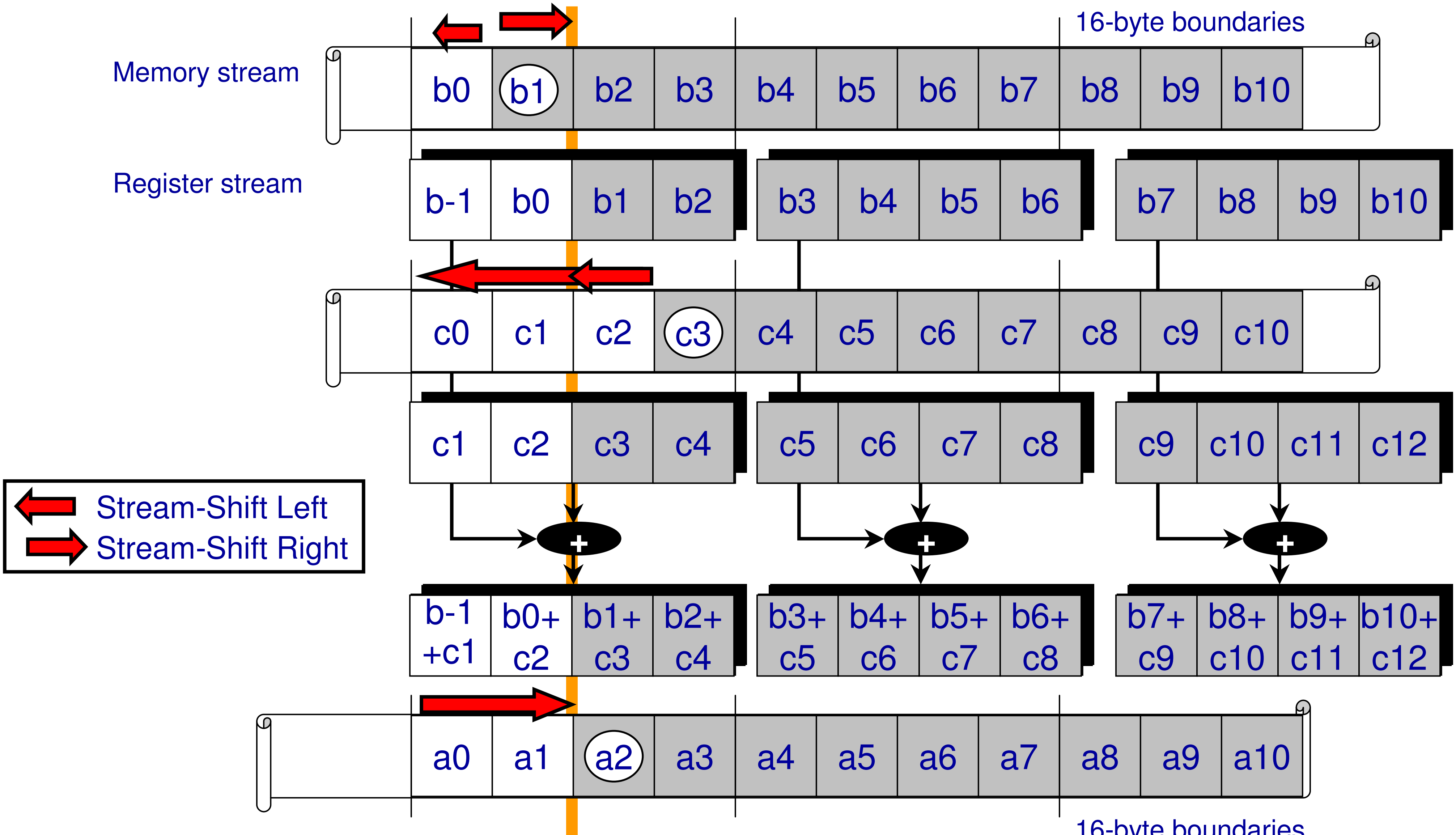


set alignment of streams to zero

[VAST]

Loop-Level Simdization, Optimized Way

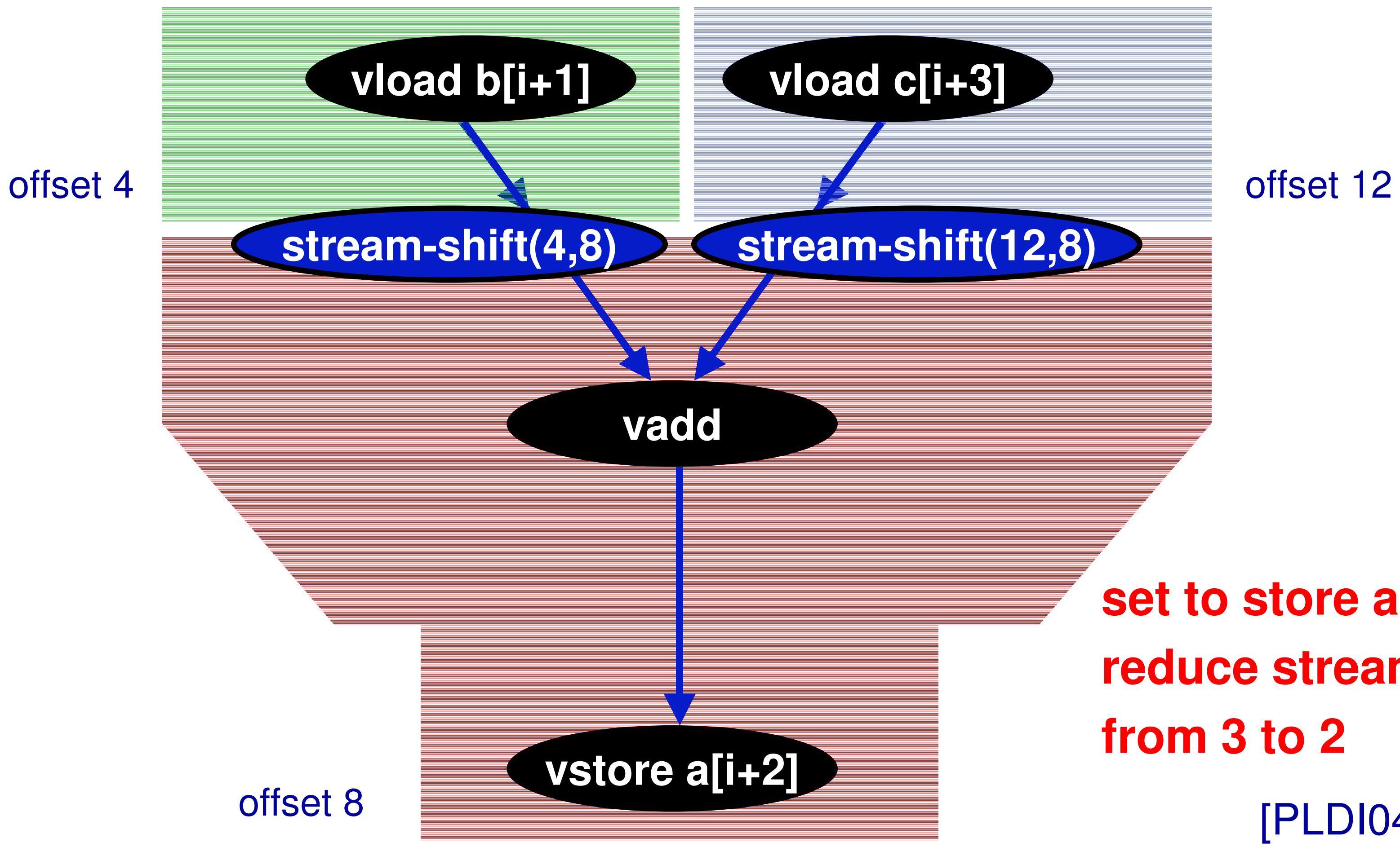
- SIMD execution of "for(i=0;i<64;i+) **a[i+2] = b[i+1] + c[i+3]**"



Optimized Solving of the Alignment Problem, Eager Policy

❑ Data Reorganization Graph

- eagerly align to store alignment

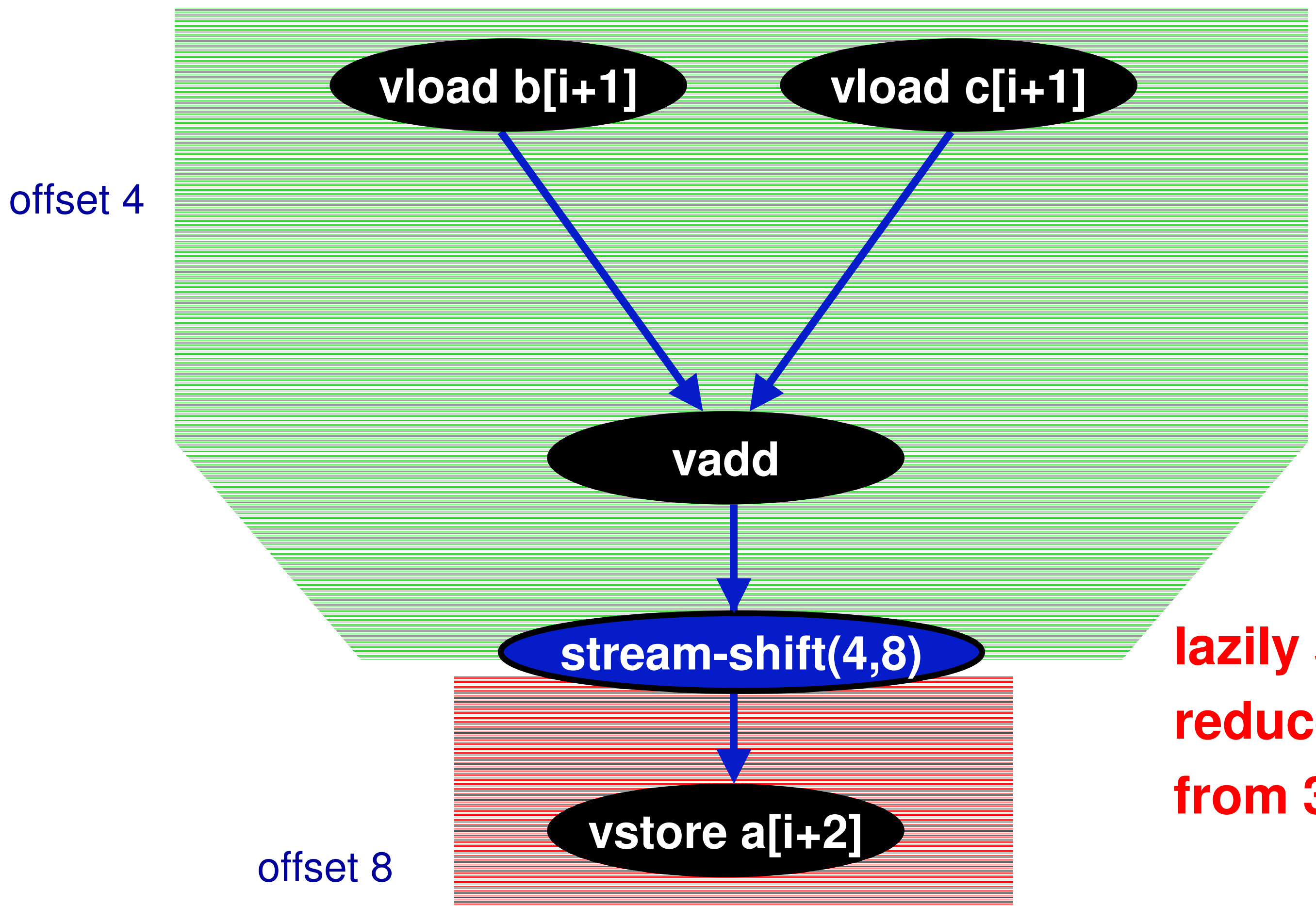


set to store alignment; reduce stream-shift # from 3 to 2

[PLDI04, CGO05]

Optimized Solving of the Alignment Problem, Lazy Policy

- **Data Reorganization Graph (slightly different “b[i+1] + c[i+1]” example)**
  - lazily align to store alignment



lazily set to store align; reduce stream-shift # from 3 to 1

[PLDI04, CGO05]

SIMD Code Generation

- **SIMD codes are generated from a valid data reorganization graph**

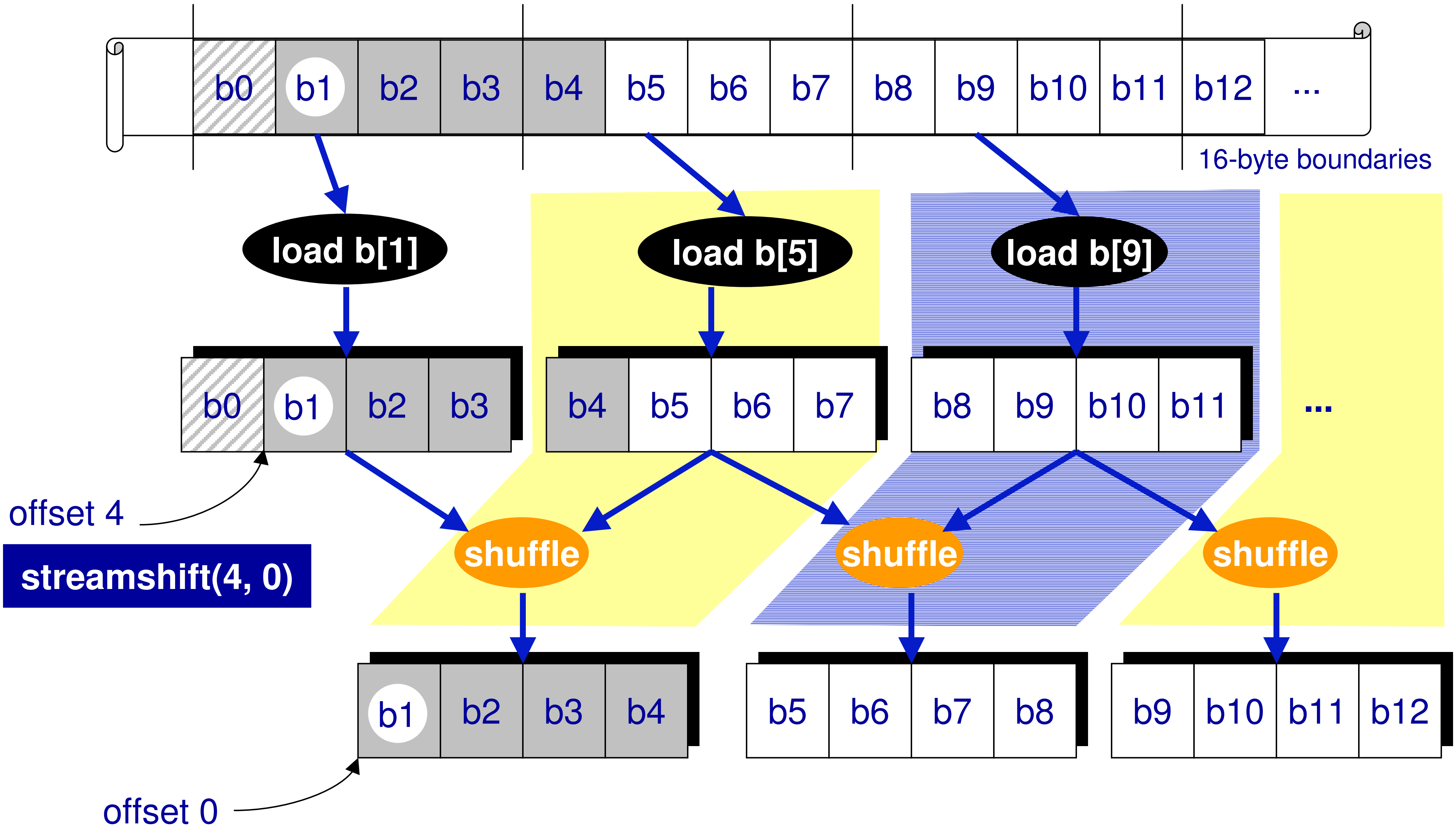
- **Code generation for simdized loop**

```
<simdized prologue>
for(i=0; i<100; i+=4)
   <simdized loop body>
<simdized epilogue>
```

  - simdized loop (steady state)
  - simdized prologue if store is misaligned
  - simdized epilogue depending on store alignment and tripcount

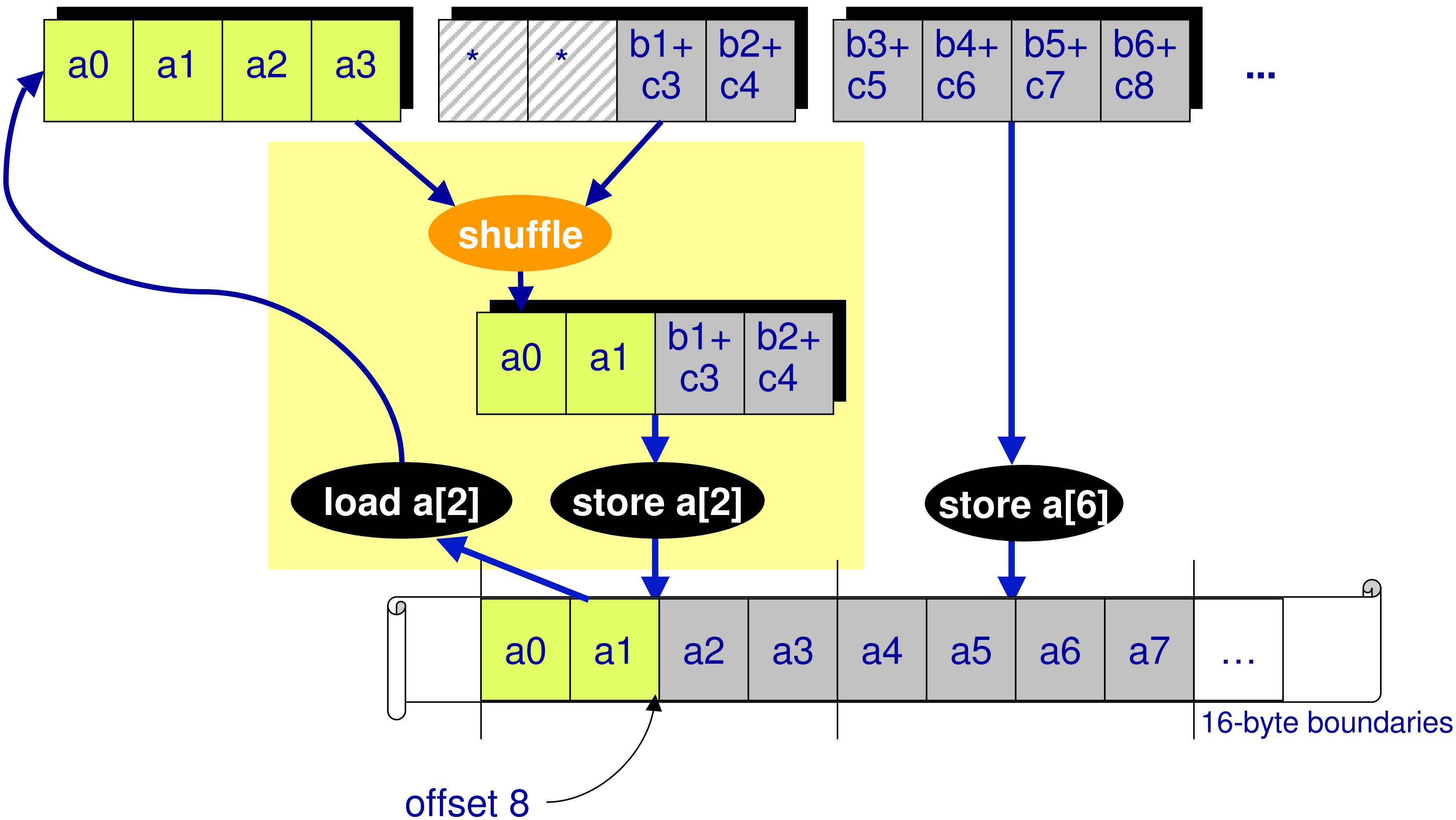
Code Generation for Stream-Shift

- **When you need a <u>stream</u> of vectors starting at address b[1]**



Code Generation for Partial Store

for (i=0; i<100; i++) a[i+2] = b[i+1] + c[i+3];

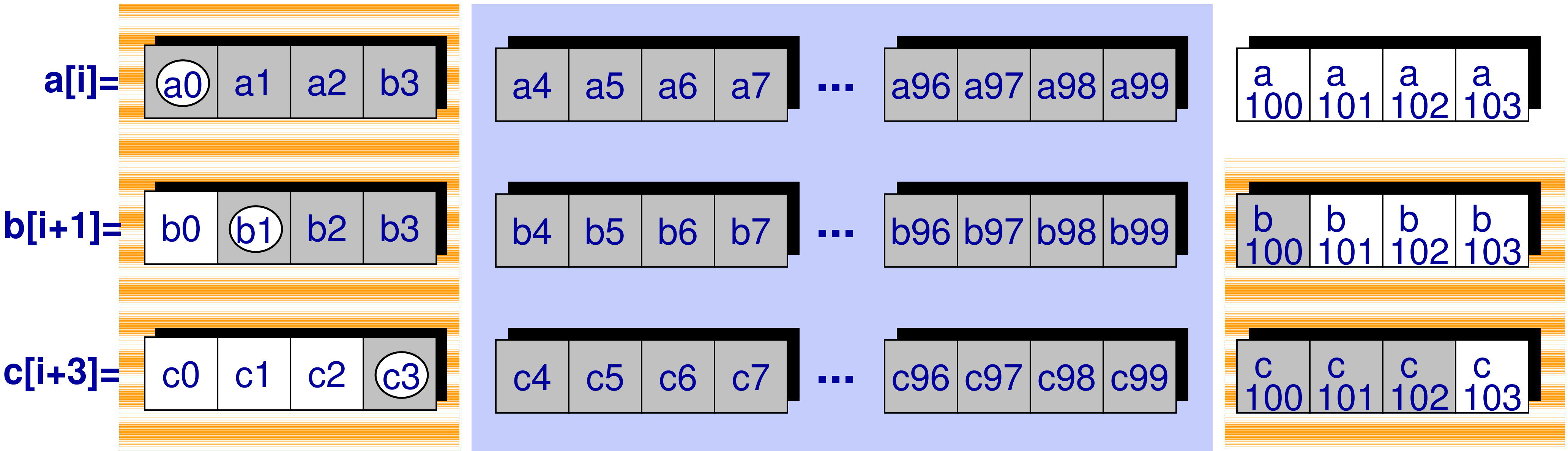


Code Generation for Loops (Multiple Statements)

```
for (i=0; i<n; i++) {
  a[i] = …;
  b[i+1] = …;
  c[i+3] = …;
}
```

Implicit loop skewing (steady-state)

```
a[i+4] = …
b[i+4] = …
c[i+4] = …
```



loop prologue (simdized)

loop steady state (simdized)

loop epilogue (simdized)

Machine-Independent Code After Simdization

Pseudo codes after simdization (no software pipelining on adjacent loads)

- all operations are vectors
- loads/stores normalized to truncated address
- splice, shiftpairl, and shiftpairr are pseudo data reorganization operations to be mapped to spu_shuffle or spu_sel

```
i = 0;
a[i] = splice(a[i], shiftpairr(b[i - 4], b[i], 4) + shiftpairl(c[i - 4], c[i],4), 8);
do {
    a[i + 4] = shiftpairr(b[i], b[i + 4], 4) + shiftpairl(c[i], c[i + 4],4);
    i = i + 4;
} while (i < 95);
a[i + 4] = splice(shiftpairr(b[i], b[i + 4],4) + shiftpairl(c[i], c[i + 4], 4), a[i + 4],8));
```

Machine-Dependent Intrinsic Codes after Simdization

- after software pipelining, loop normalization, and address truncation
- spu_shuffle, spu_sel, spu_add, spu_mask are SPU intrinsics
- <0x08090a0b, …> is a vector literal

```
a[0] = spu_sel(a[0],spu_add(spu_shuffle(b[-4],b[0], <0x08090a0b,0x0c0d0e0f,0x10111213,0x14151617>),
spu_shuffle(c[-4],c[0], <0x0c0d0e0f,0x10111213,0x14151617,0x18191a1b>),spu_maskb(15));
    oldSPCopy0 = c[0];
    oldSPCopy1 = b[0];
    i = 0;
    do {
      tc = c[i*4+4];
      tb = b[i*4+4];
      a[i*4+4] = spu_add(spu_shuffle(oldSPCopy1,tb,<0x08090a0b,0x0c0d0e0f,0x10111213,0x14151617>),
spu_shuffle(oldSPCopy0, tc,<0x0c0d0e0f,0x10111213,0x14151617,0x18191a1b>);
      oldSPCopy0 = tc;
      oldSPCopy1 = tb;
      i = i + 1;
    } while (i < 24);
a[100] = spu_sel(spu_add(spu_shuffle(b[96],b[100],<0x08090a0b,0x0c0d0e0f,0x10111213, 0x14151617>),
spu_shuffle(c[96], c[100], <0x0c0d0e0f,0x10111213,0x14151617,0x18191a1b>), a[100], spu_maskb(15));
```

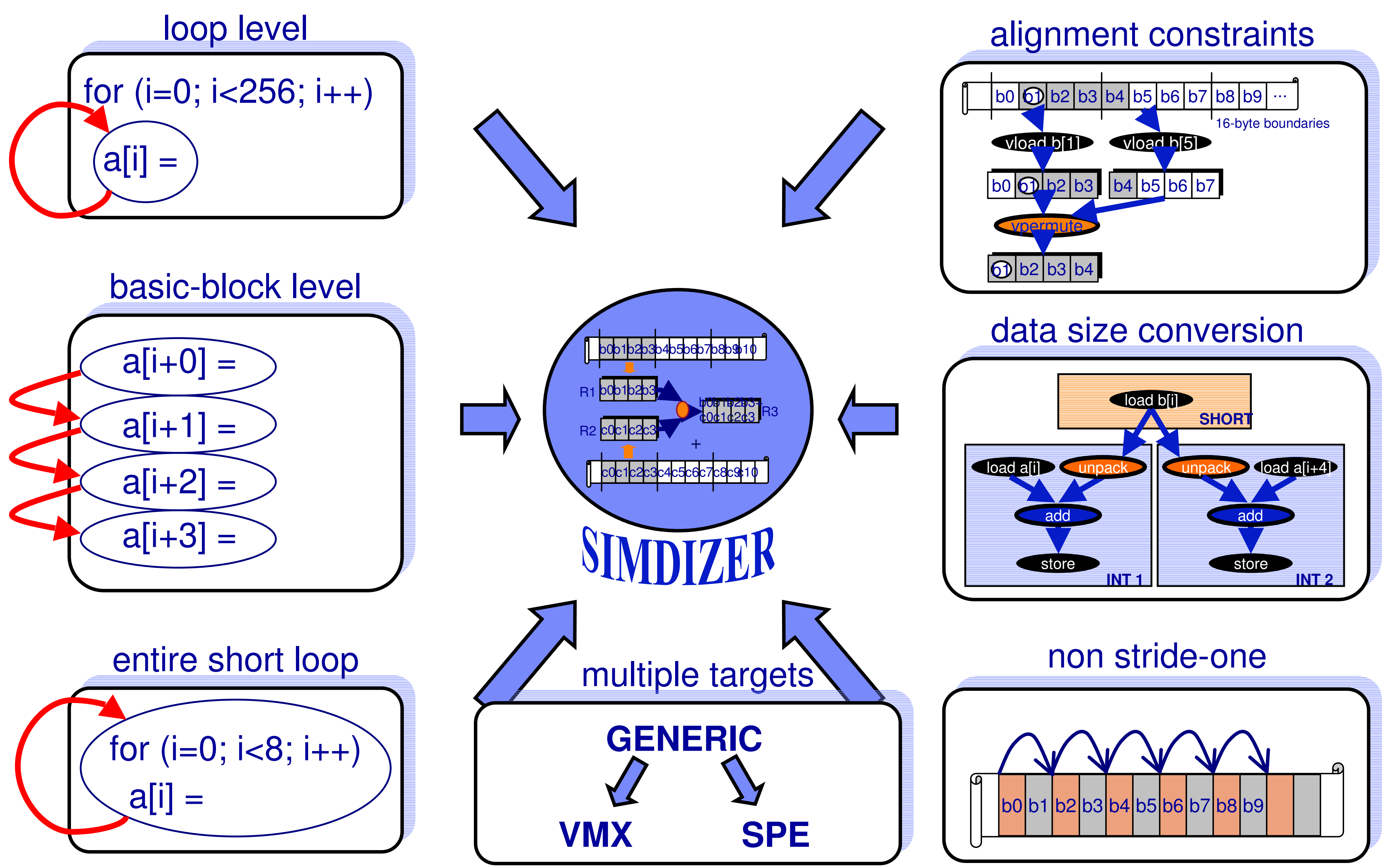
Outline

- Simdization example
- **Integrated simdization framework**

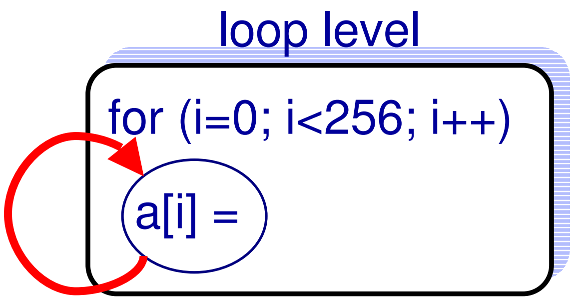
Successful Simdizer

Extract Parallelism

Satisfy Constraints



Multiple Sources of SIMD Parallelism



- **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
    - amortize overhead (versioning, alignment handling) and employ cost models

[Bik *et al*, IJPP 2002]

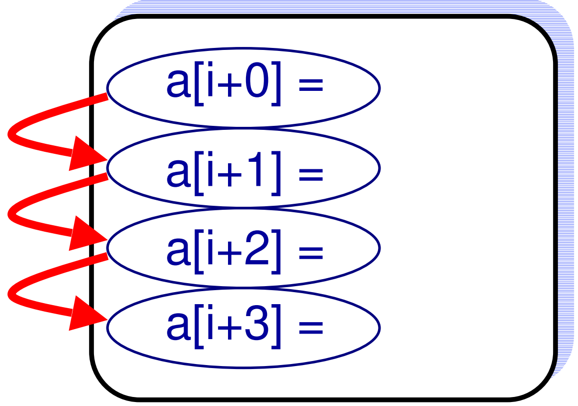
[VAST compiler, 2004]

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

[Naishlos, GCC Developer's Summit 2004]

Multiple Sources of SIMD Parallelism (cont.)

basic-block level



- **Basic-block level**
  - SIMD across multiple isomorphic operations
  - successful at
    - handling unrolled loops (manually or by compiler)
    - extracting SIMD parallelism within structs, e.g.

```
a[i].x =
a[i].y =
a[i].z =
```

    - extracting SIMD parallelism within a statement

```
s +=  a(i)*b(i) + a(i+1)*b(i+1) + a(i+2)*b(i+2) +
      a(i+3)*b(i+3) + a(i+4)*b(i+4)
```

[Larsen *et al*, PLDI 2000]

[Shin *et al*, PACT 2002]

Multiple Sources of SIMD Parallelism (cont.)

- **Short-loop level**
  - SIMD across entire loop iterations
  - effectively collapse innermost loop
  - we can now extract SIMD at the next loop level
  - e.g. FIR

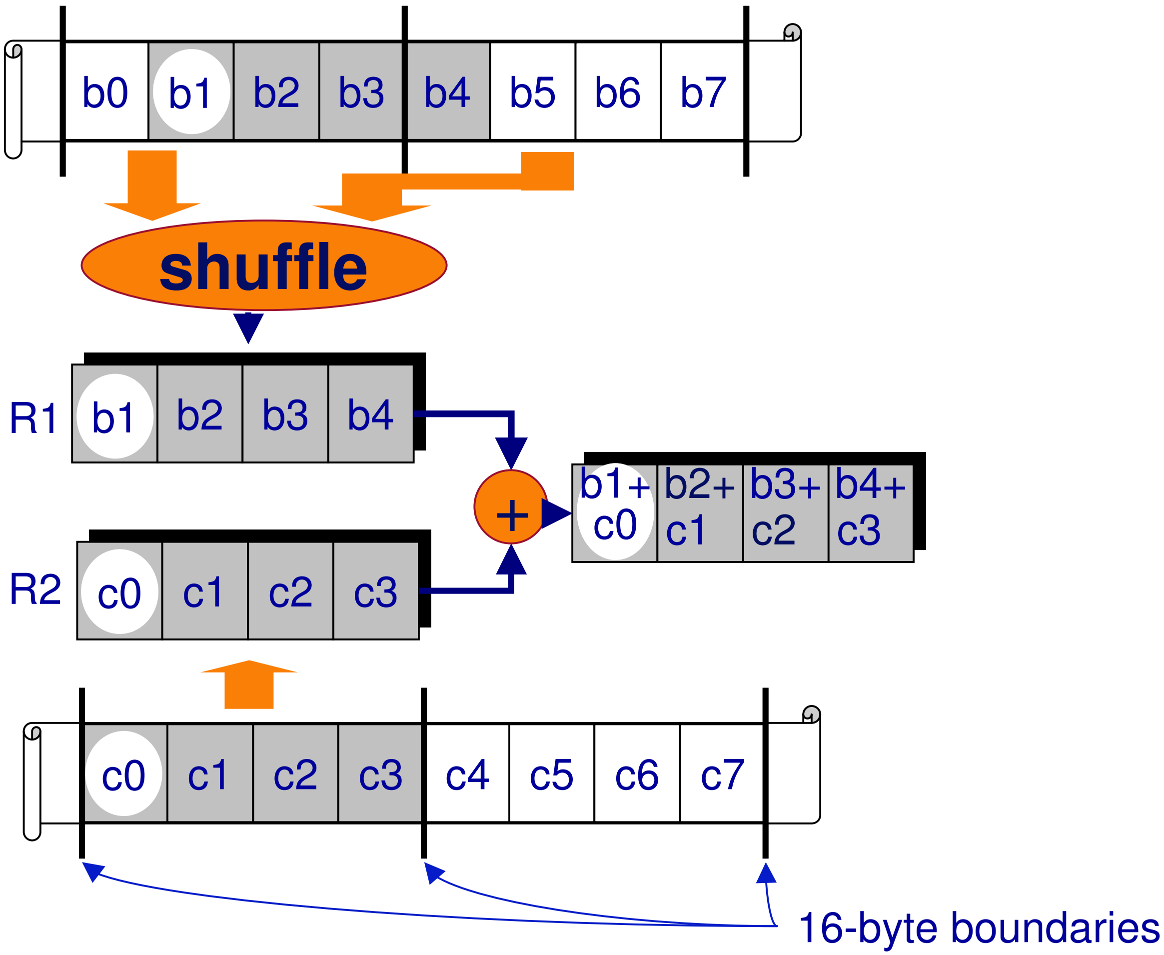
```
for (k=0; k<248; k++)
    for (i=0; i<8; i++)
        res[k] += in[k+i] * coef[k+i];
```

entire short loop

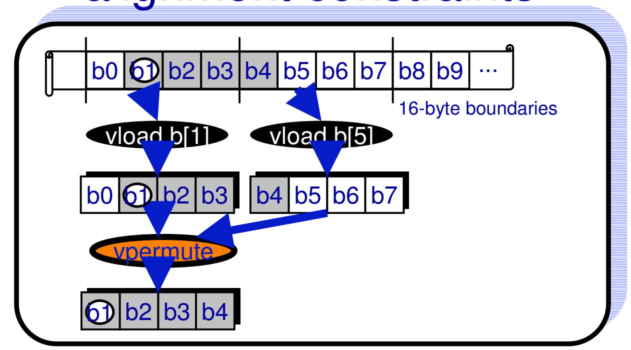
```
for (i=0; i<8; i++)
  a[i] =
```

Multiple SIMD Hardware Constraints

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

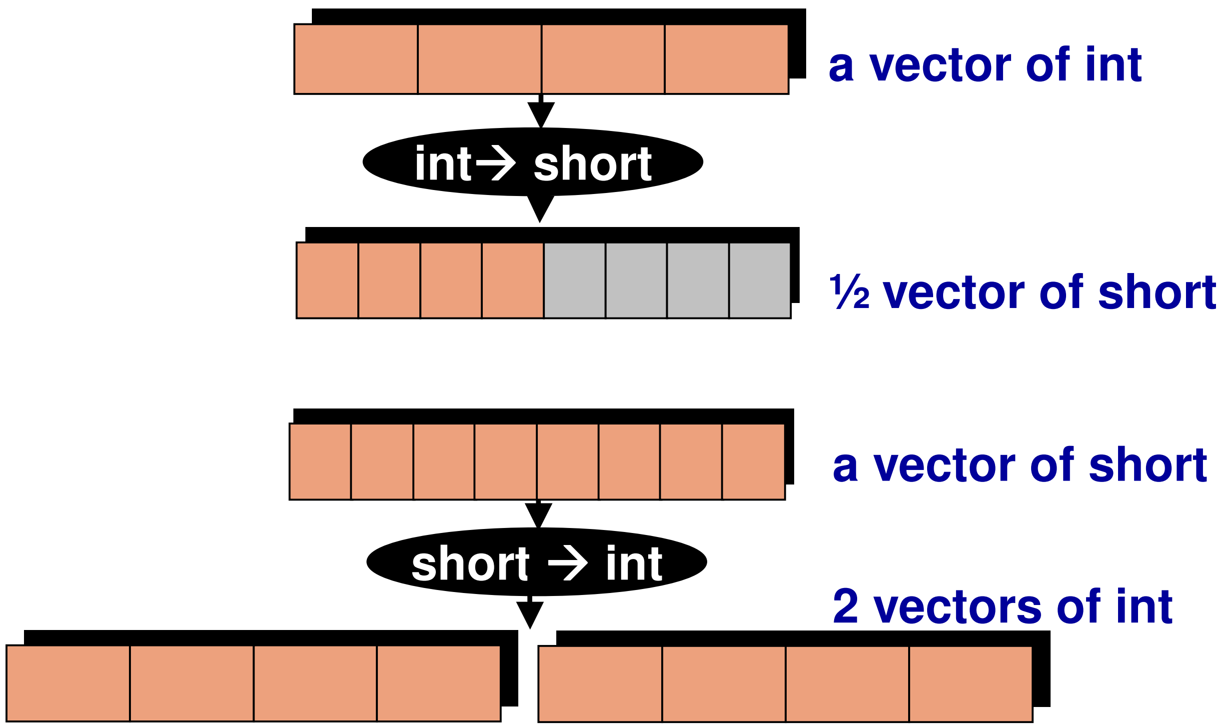


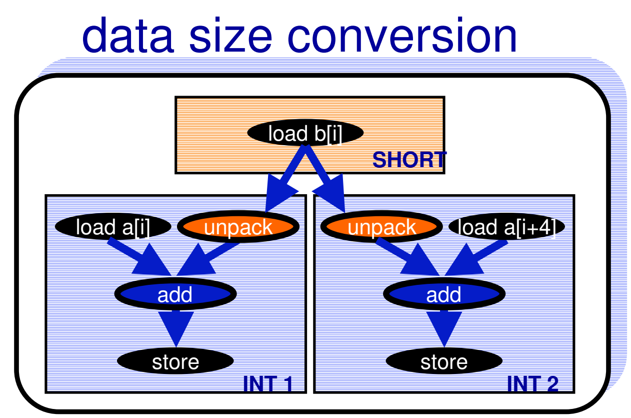
alignment constraints



Multiple SIMD Hardware Constraints (cont.)

- ❑ **Size of data in SIMD registers matters**



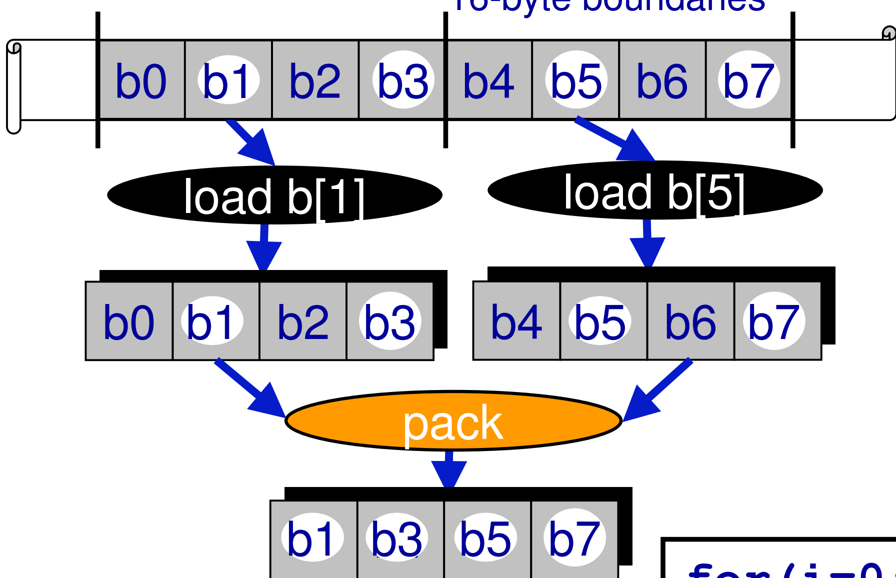


- ❑ **E.g. when converting from short to integer**
  - ➢ we must issue 2x integer SIMD operations

Multiple SIMD Hardware Constraints (cont.)

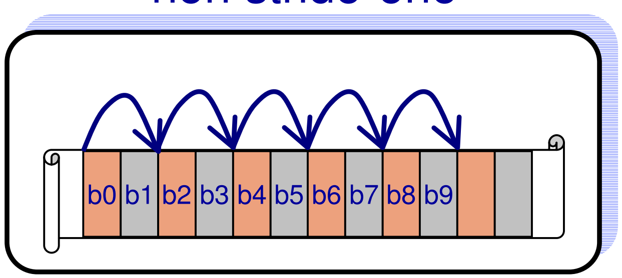
- **Hardware supports 16-byte continuous access only**

- **Non stride-one load requires packing**




```
for(i=0;i<n;i++)
 …= a[i] + b[2i+1]
```

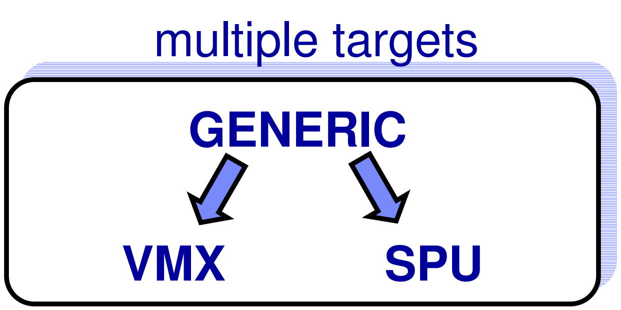
non stride-one

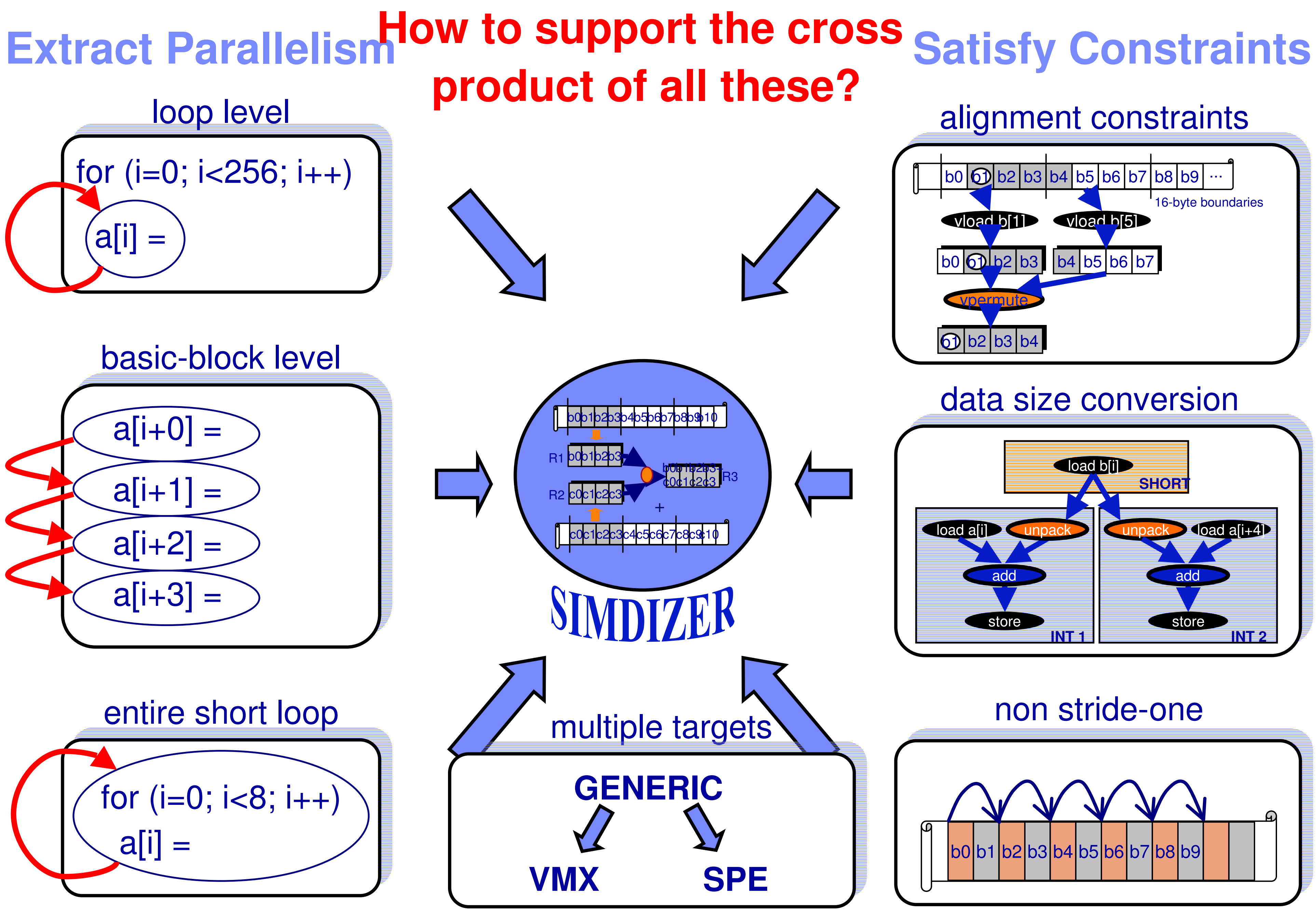


- **Non stride-one store requires unpacking**

Multiple SIMD Hardware Constraints (cont.)

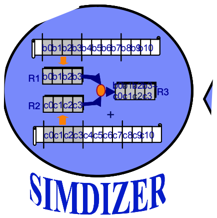
- **Different platforms have varying SIMD support**
  - e.g. VMX / SPE have SIMD permute instructions
  - e.g. SPE has no memory page fault, VMX does




Extract Parallelism
How to support the cross product of all these?
Satisfy Constraints
loop level
for (i=0; i<256; i++)
a[i] =
basic-block level
a[i+0] =
a[i+1] =
a[i+2] =
a[i+3] =
entire short loop
for (i=0; i<8; i++)
a[i] =
alignment constraints
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ...
16-byte boundaries
vload b[1]
vload b[5]
b0 b1 b2 b3
b4 b5 b6 b7
vpermute
b1 b2 b3 b4
R1
R2
R3
SIMDIZER
data size conversion
load b[i]
SHORT
load a[i]
unpack
unpack
load a[i+4]
add
add
store
store
INT 1
INT 2
multiple targets
GENERIC
VMX
SPE
non stride-one
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9

Key Abstraction: Virtual SIMD Vector

- **Virtual SIMD Vector**
  - has arbitrary length
  - has no alignment constraints
- **Extraction of SIMD Parallelism**
  - use virtual vector as representation
  - abstract away all the hardware complexity
- **Progressive "de-virtualization" of the virtual vector**
  - until each vector in the loop satisfies all hardware constraints
  - or revert vectors back to scalars (if too much overhead)

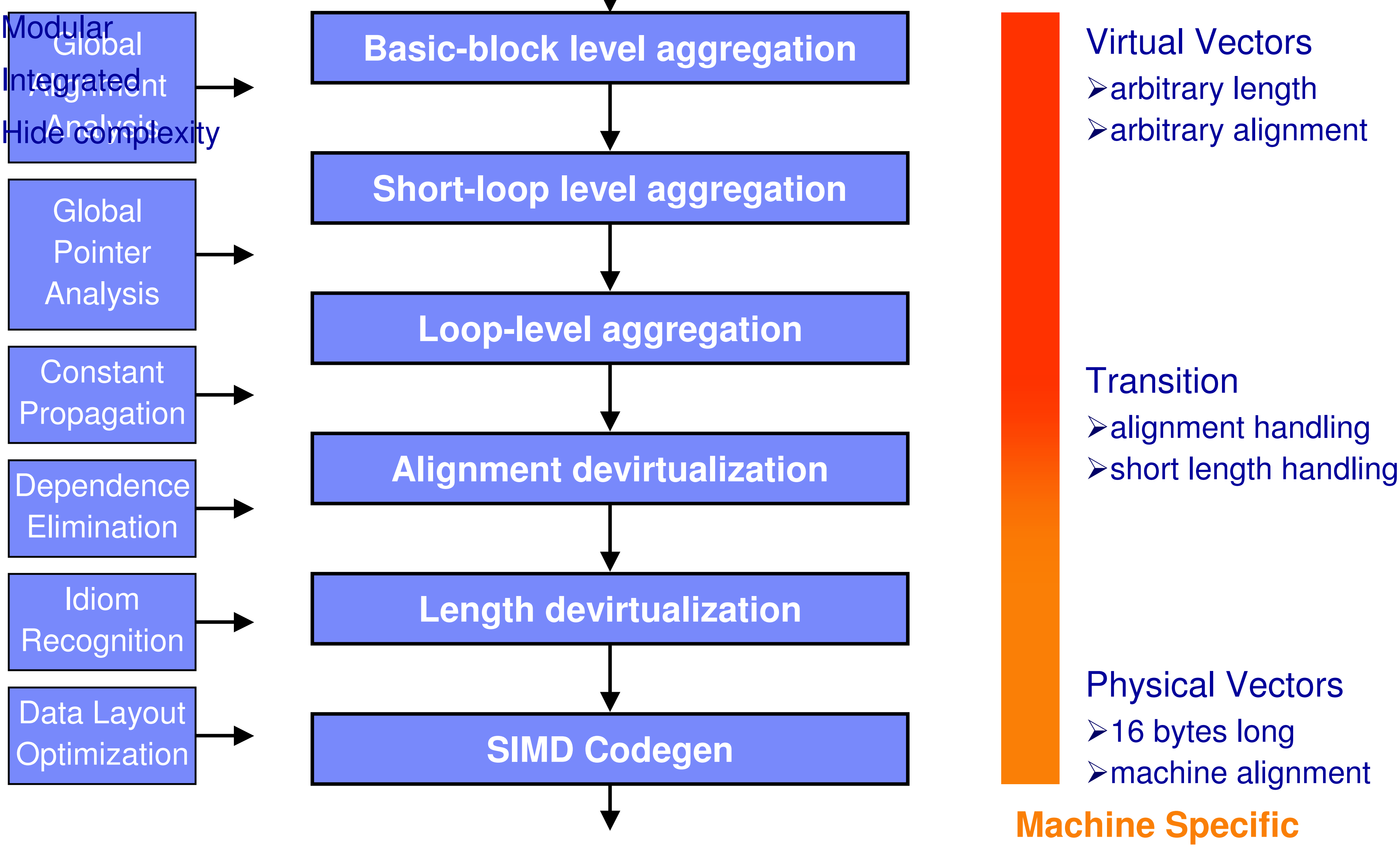


Integrated Simdization Framework

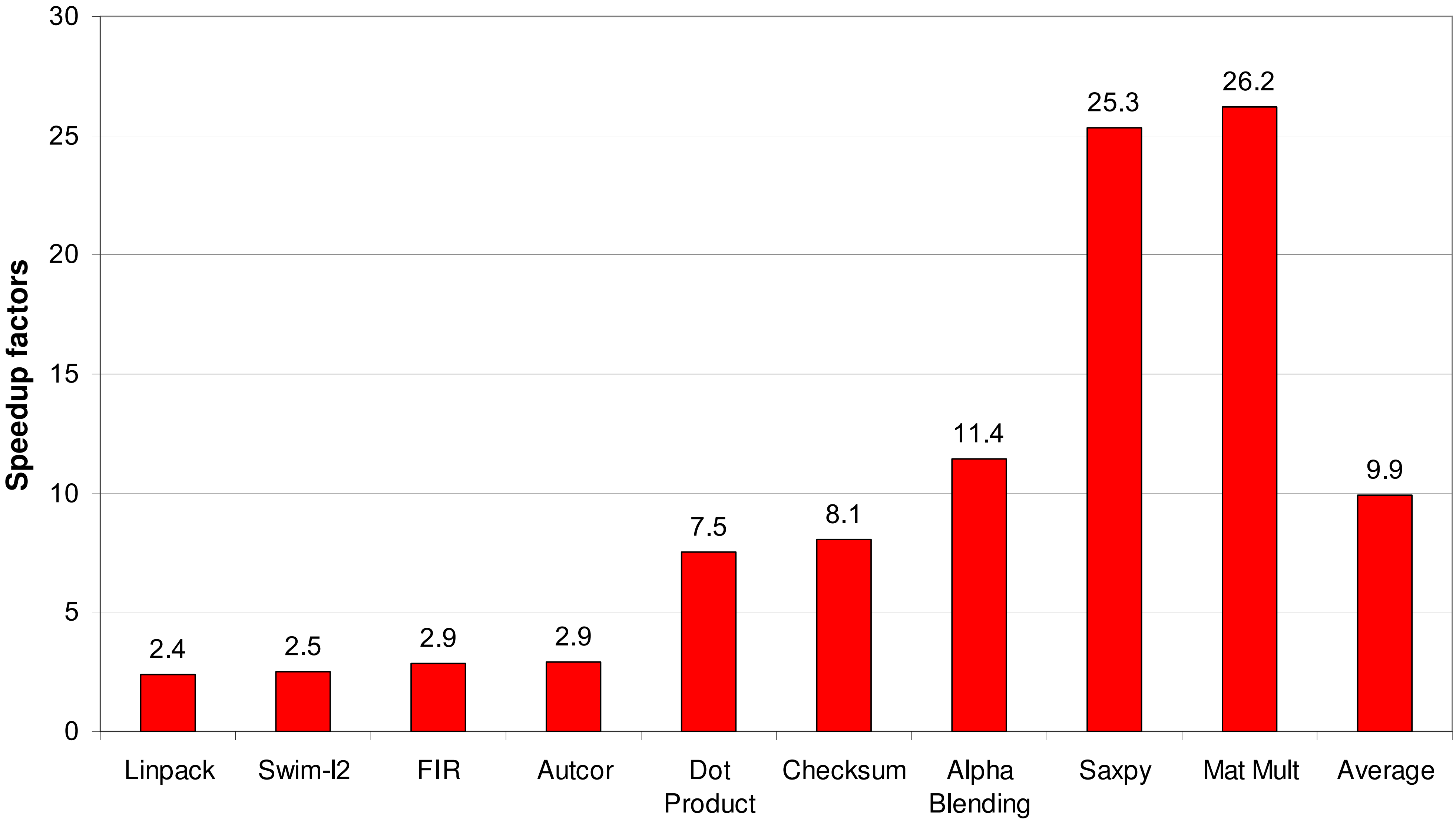
Characteristics:

- Modular
- Integrated
- Hide complexity

Support:



SPE Simdization Results



single SPE, optimized, automatic simdization vs. scalar code

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:

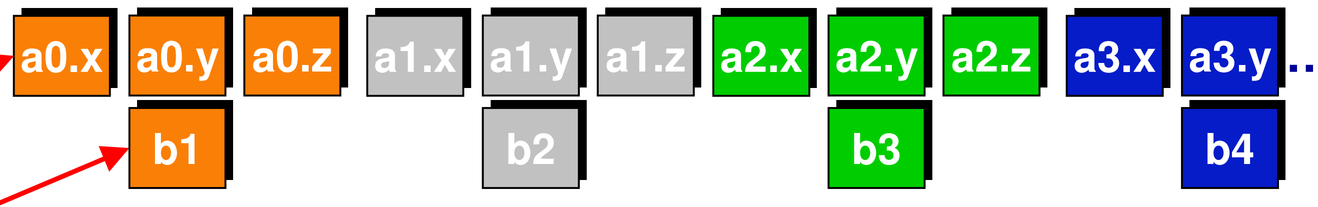
www.research.ibm.com/cellcompiler/compiler.htm

First Example: Basic-Block & Loop Level Aggregation

Original loop

```
for (i=0; i<256; i++) {
   a[i].x =
   a[i].y =
   a[i].z =
   b[i+1] =
}
```

Value streams



4 iterations shown here for the purpose of illustration

1st iter   2nd iter   3rd iter   4th iter

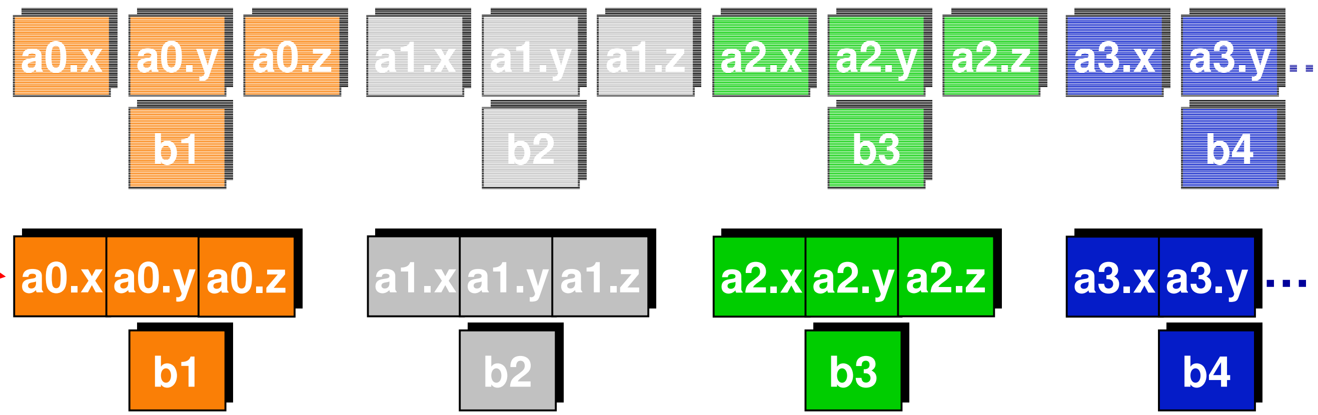
Phase 1: Basic-Block Level Aggregation

Original loop

```
for (i=0; i<256; i++) {
  a[i].x =
  a[i].y =
  a[i].z =
  b[i+1] =
}
```

pack

Value streams



Basic-Block level aggregation

- pack a[i].x, a[i].y, a[i].z into a vector of 3 elements
- pack regardless of alignment

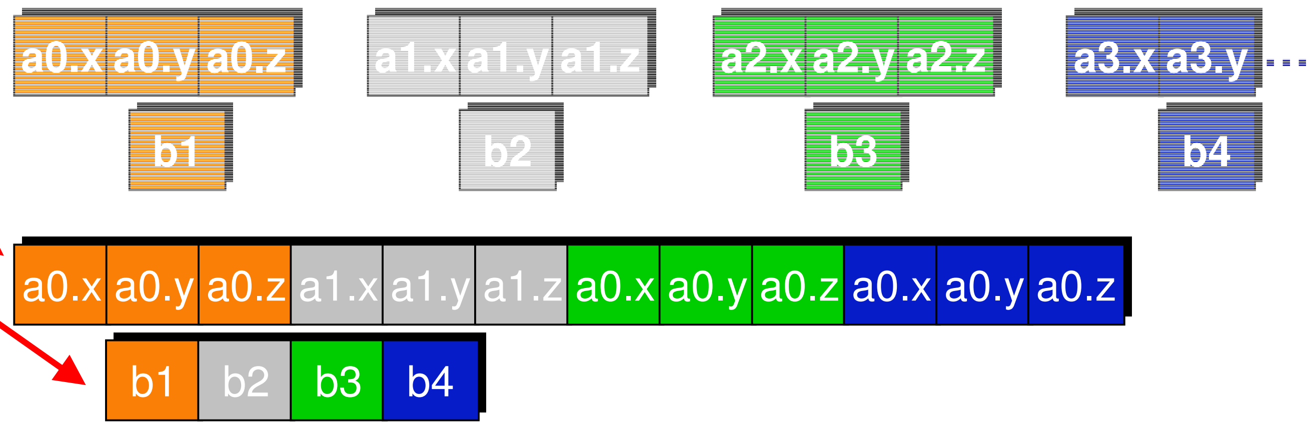
Phase 2: Loop-Level Aggregation

BB-aggregated loop

Value streams

```
for (i=0; i<256; i++) {
 (a[i].x,y,z) =
  b[i+1] =
}
```

pack with self



Loop-level aggregation

- pack each statement with itself across consecutive iterations
- final vector lengths must be multiple of 16 bytes

- scalar “b[i]” or vector “(a[i].x,y,z)” are treated alike
- pack regardless of alignment

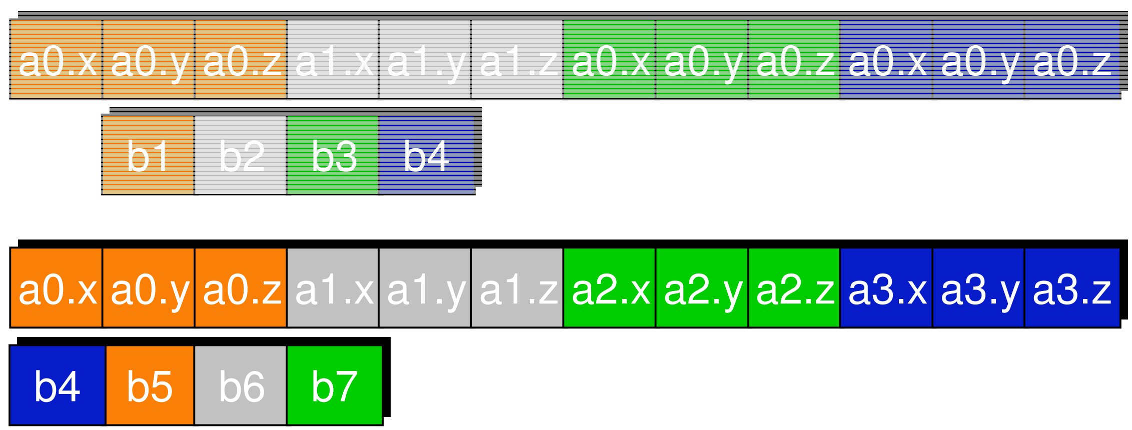
Phase 3: Alignment Devirtualization

Loop-aggregated

```
for (i=0; i<256; i+=4) {
  (a[i].x,…,a[i+3].z) =
  (b[i+1],…,b[i+4]) =
}
```

align access

Value streams



Alignment *

- shift misaligned streams
- skew the computations so that loop computes (b[i+4]…b[i+7])

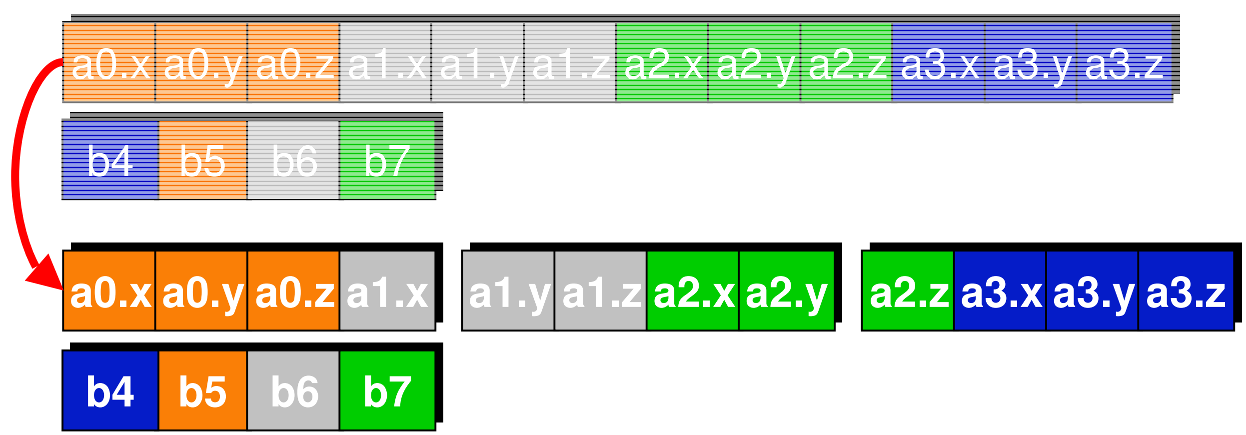
* Arrays (e.g. &a[0], &b[0],…) are assumed here 16-byte aligned.

Phase 4: Length Devirtualization

Aligned loop

```
(b[1]…b[3]) =
for (i=0; i<252; i+=4) {
  (a[i].x,…,a[i+3].z) =
  (b[i+4],…,b[i+7]) =
}
(a[252].x,…,a[255].z) =
b[256] =
```

Value streams



Length

- break into 16-byte chunks

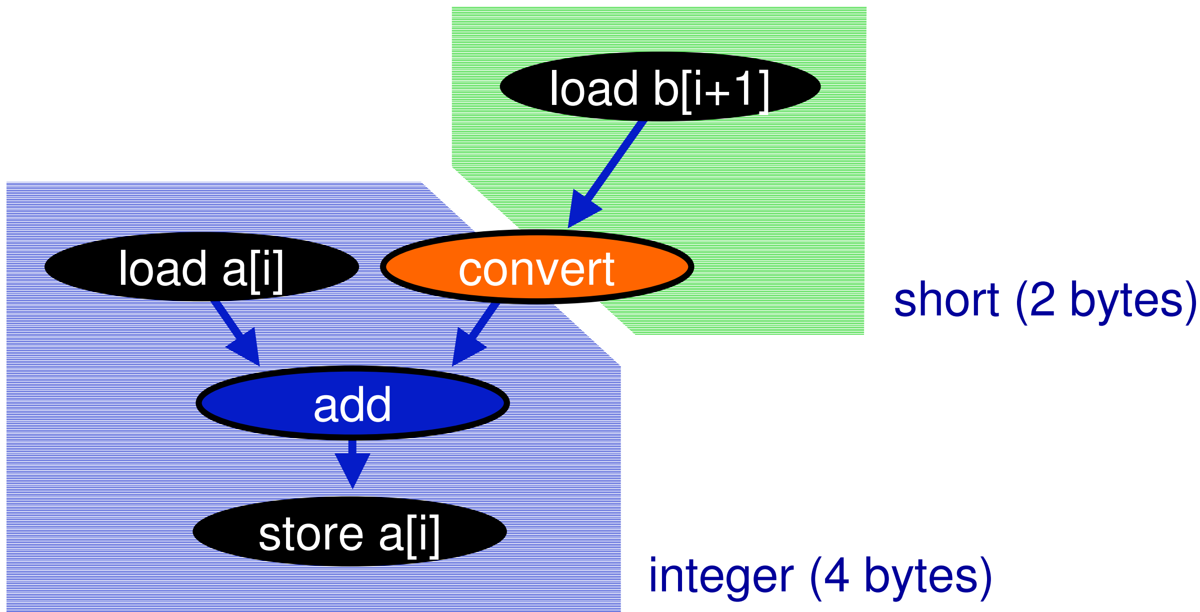
Second Example: Data-Size Conversion and Misalignment

Original loop

```
for (i=0; i<256; i++) {
  a[i]  +=  (int) b[i+1]
}
```

short computations

integer computations

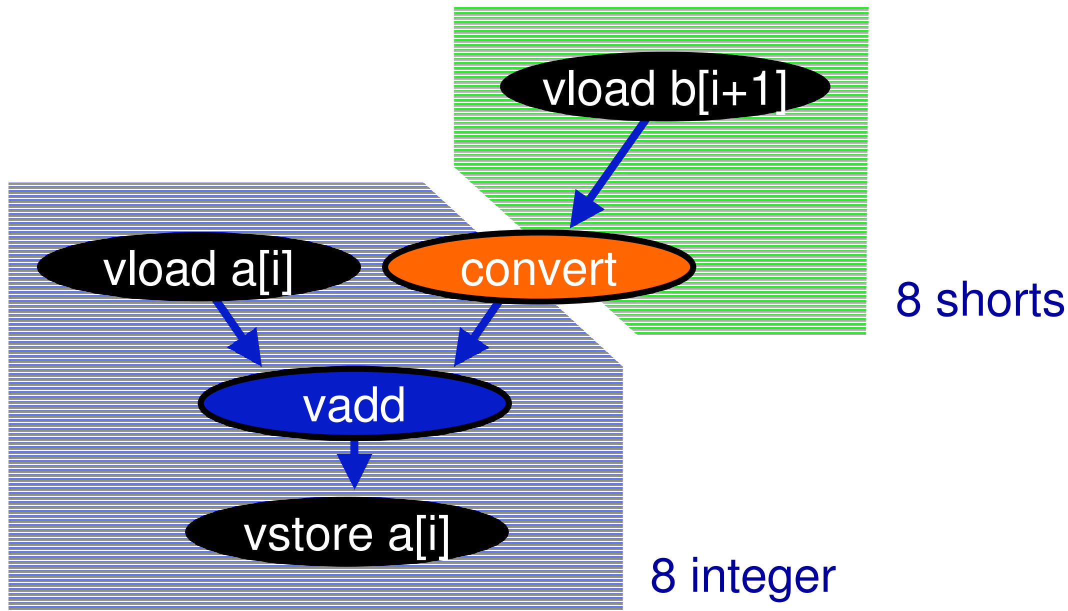


Phase 1: Loop-Level Aggregation

Original loop

```
for (i=0; i<256; i++) {
  a[i]  +=  (int) b[i+1]
}
```

pack with self



Loop-level aggregation

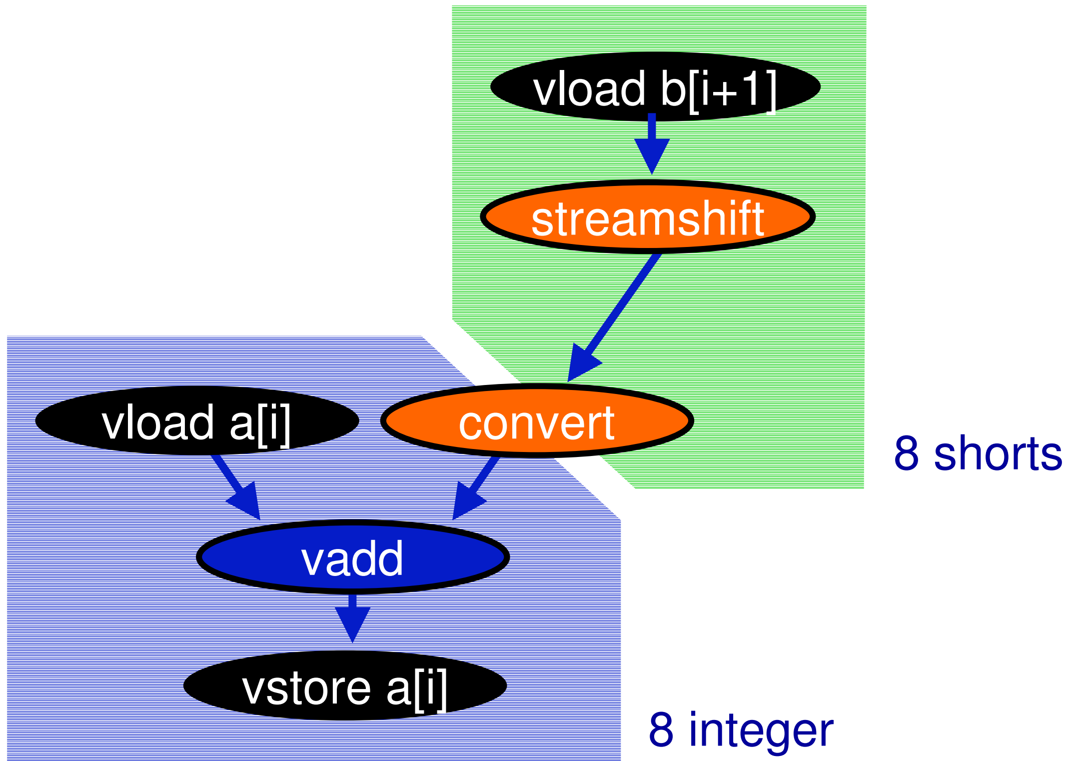
- pack each statement with itself across consecutive iterations
- virtual vectors have uniform number of elements, even when
  - vector of 8 integer = 32 bytes of data
  - vector of 8 short = 16 bytes of data

Phase 2: Alignment Devirtualization

Original loop

```
for (i=0; i<256; i++) {
  a[i]  +=  (int) b[i+1]
}
```

align b[i+1]



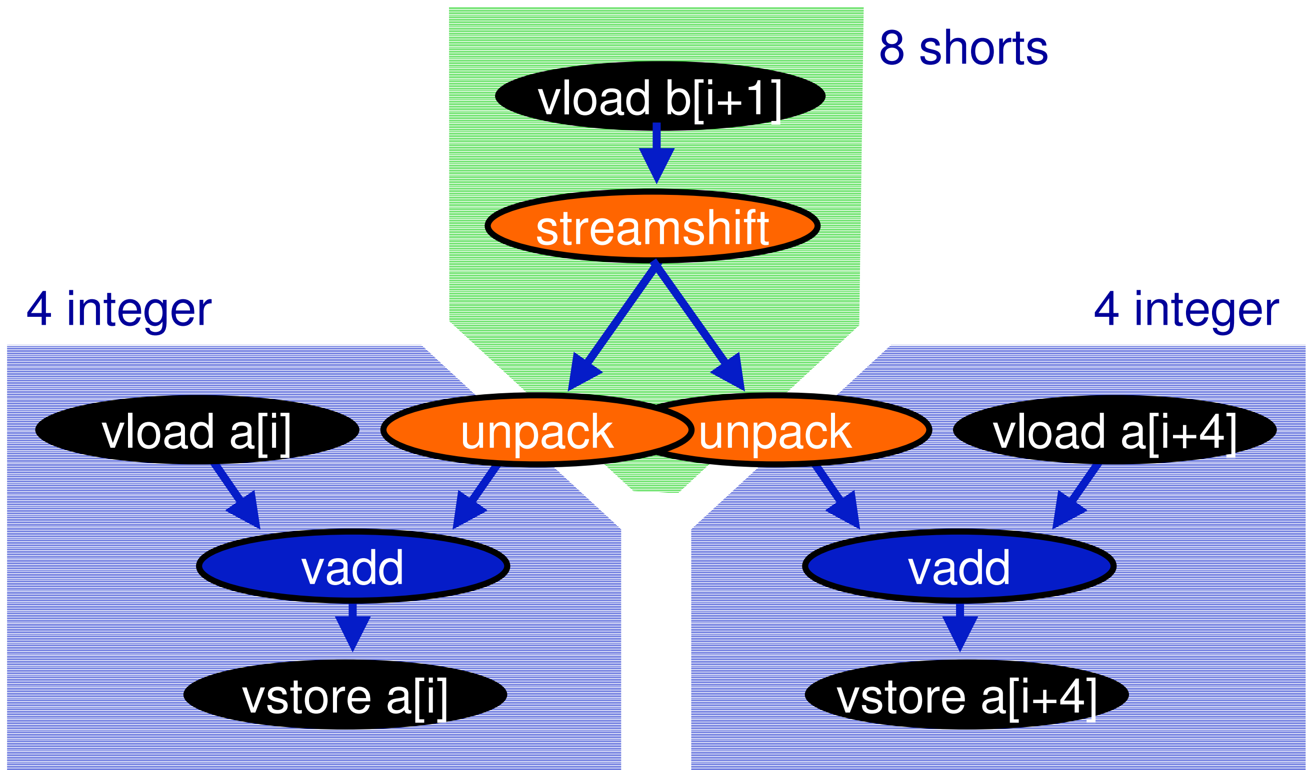
Alignment

- shift misaligned streams
- easy to do as we are still dealing with long vectors

Phase 3: Length Devirtualization

Original loop

```
for (i=0; i<256; i++) {
  a[i]  +=  (int) b[i+1]
}
```



Length

- 8 shorts fit into a 16-byte register
- 8 integers do not fit; must replicate integer registers and associated instructions