Speculative Dynamic Vectorization to Assist Static Vectorization in a HW/SW Co-designed Environment

Rakesh Kumar [Ψ]

Alejandro Martínez [λ]

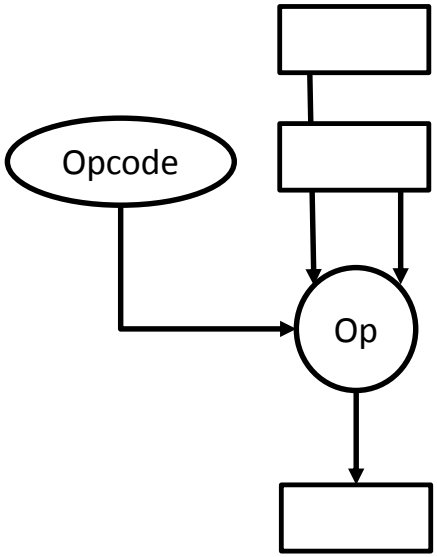
Antonio González [Ψ,λ]

Ψ Dept. Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona, Spain
rkumar@ac.upc.edu

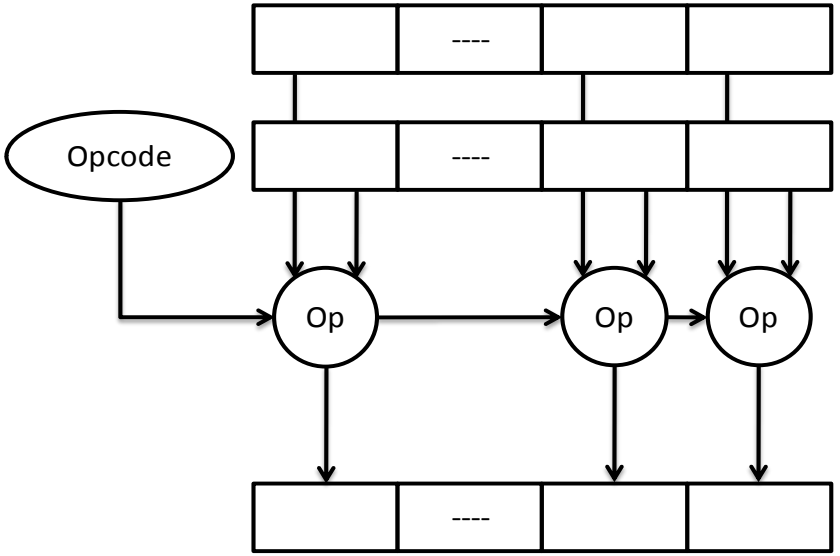
λ Intel Barcelona Research Center
Intel Labs - UPC
Barcelona, Spain
alejandro.martinez@intel.com
antonio.gonzalez@intel.com

SIMD Execution Model

- Compiler
  - Multiple scalar instructions → one vector instruction
- Hardware
  - Performs multiple operations of same kind in parallel



Scalar Execution

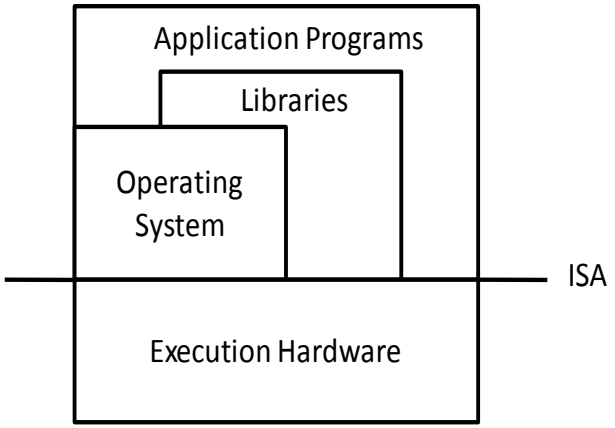


SIMD Execution

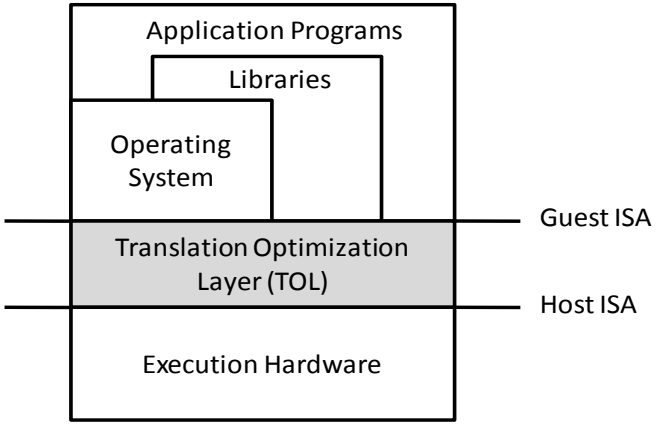
SIMD Execution

- Advantages
  - Performance
    - Parallel execution
    - Instruction cache hit rate
    - More instructions/operations in instruction window
  - Energy Efficiency
    - Front-end has to handle (fetch/decode) less instructions
- Problem
  - SIMD unit usages depends on compilers ability to vectorize
  - Compilers vectorize conservatively (for correctness)
- Solution
  - Speculative Dynamic Vectorization
    - Speculatively assumes two memory reference will not alias
    - Requires hardware support to detect speculation failure and recovery

HW/SW Co-designed Processor



Conventional Processor



HW/SW Co-designed Processor

- Translation and Optimization Layer
  - Resides between software stack and hardware
  - Translates from guest ISA to host ISA
  - Instruments/optimizes the guest binary to find hot code
- Salient Features
  - Power, Complexity, Performance, Multiple ISA support, Binary Compatibility, Security

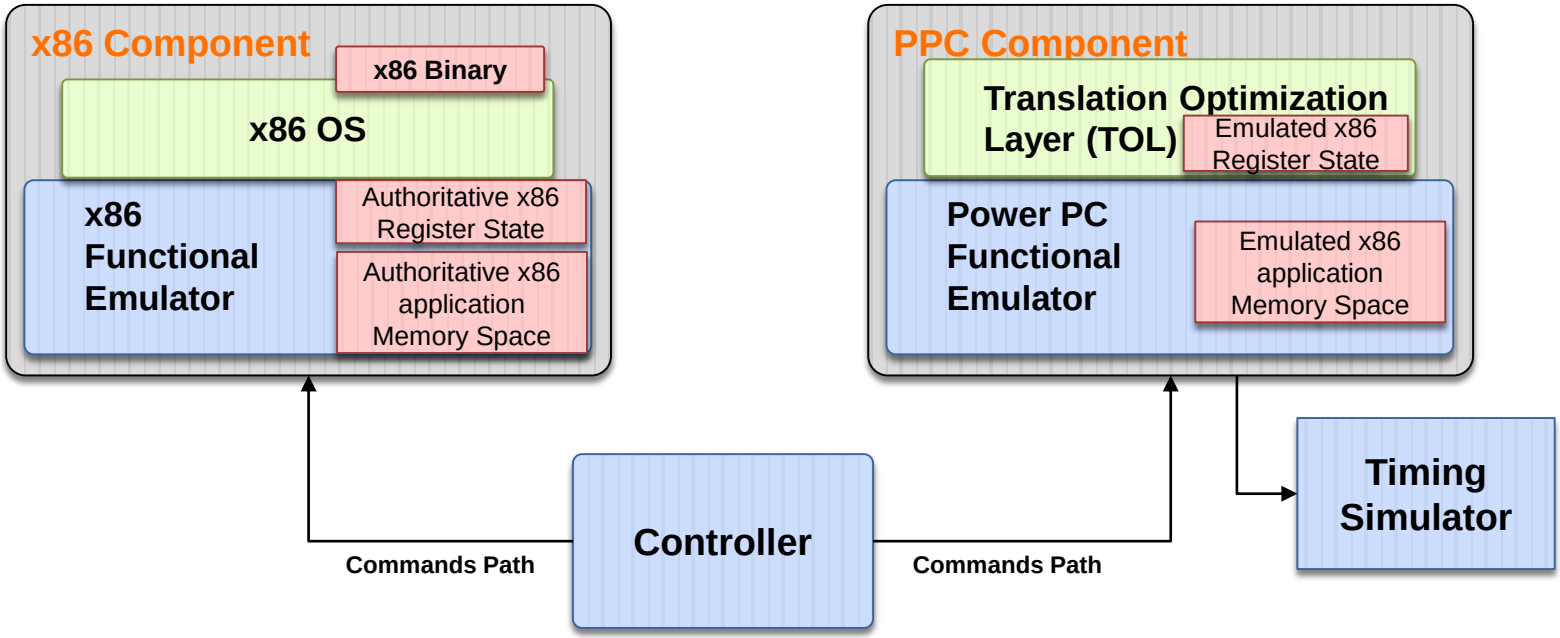
Why HW/SW Co-designed Environment

- Efficient Speculation and Recovery Support
  - Speculation uncovers more opportunities
- Bigger Optimization Regions
  - Vectorization scope increases to multiple basic blocks
- Decoupled ISA from Hardware Implementation
  - Portable vectorization
- Dynamic Optimizations
  - Legacy code vectorization

Outline

- Introduction
- Simulation Infrastructure
- Speculative Dynamic Vectorization
- Results
- Conclusion

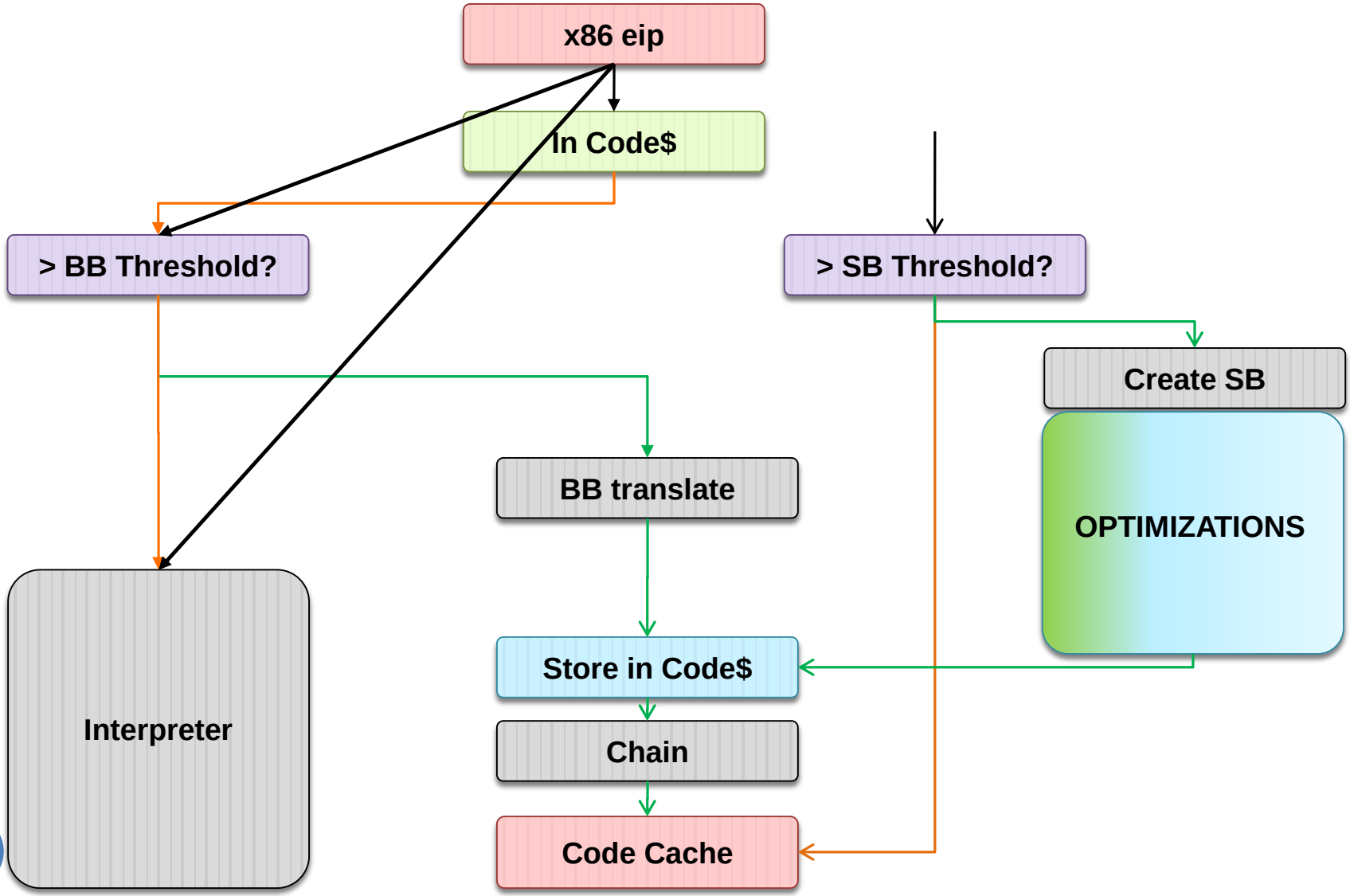
DARCO: The components



Principal Components

- **x86 Component** : System level QEMU of x86
  Executes the Operating System and the application
- **PPC Component** : Process level QEMU of PPC
  Executes the Translation Optimization Layer
- **Controller** : Synchronizes execution and handles communication of components

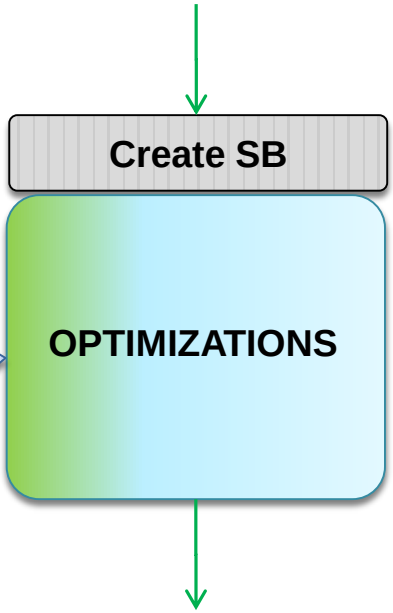
Translation Optimziation Layer



Optimizations

- x86 to IR Translation
  - Loop Unrolling
  - Control Speculation
- SSA
- Forward Pass
  - Const P, CP, CSE, CF
- Backward Pass
  - Dead Code Elimination
- DDG
  - Redundant Load Removal
  - Store Forwarding
  - Memory Alias Analysis
  - Consecutiveness Analysis

- Vectorization
- Dead Code Elimination
- Instruction Scheduling
  - Data Speculation
- Register Allocation
- Code Generation



Memory Speculation and Recovery

TOL

- Label instructions in the program order
- If a pair of memory references:
  - may alias and
  - is reordered

  convert original load/store instructions to speculative instructions

```
2       st_64_spec      v2, M[y]
1       ld_64_spec      v1, M[x]
2       st_64           v2, M[y]
```

Hardware

- Save Register and Memory State before executing speculative code

```
PC:   2    st_64_spec    v2, M[y]
PC:   1    ld_64_spec    v1, M[x]
```

Label	Address	Size
2	y	8

- If violation, restore saved state and restart execution without speculation

Similar to the speculation and recovery mechanism of BOA

Outline

- Introduction
- Simulation Infrastructure
- Speculative Dynamic Vectorization
- Results
- Conclusion

Static Vectorization

- Effective in vectorization of array based application.
- Able to apply complex and time consuming loop transformations.
- Loses vectorization opportunities:
    - Conservative memory disambiguation
    - Pointers further complicate the problem
    - Limited vectorization scope to basic blocks (some exceptions)

Static Vectorization

- Vectorization of Pointer vs Array based applications (UTDSP Kernels)

Benchmark	Type	Percent Packed
FFT	Array	49.90%
	Pointer	0.00%
FIR	Array	99.80%
	Pointer	0.00%
IIR	Array	0.00%
	Pointer	0.00%
LATNRM	Array	7.80%
	Pointer	8.20%
LMSFIR	Array	0.00%
	Pointer	0.00%
MULT	Array	50.40%
	Pointer	0.00%

J. Holewinski et al. “Dynamic trace-based analysis of vectorization potential of applications”. In Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '12). 371-382.

The Proposal

- Introduce a dynamic vectorizer to assist the static one
- Static vectorizer:
    - Apply complex loop transformations
    - Loop transformations are costly at runtime
-Dynamic vectorizer:
    - Speculatively reorders ambiguous memory references and vectorizes
    - Hardware checks for any memory dependence violation
    - In case of violation:
        - Execution reverts back to a previously saved checkpoint
        - Executes a non speculative version

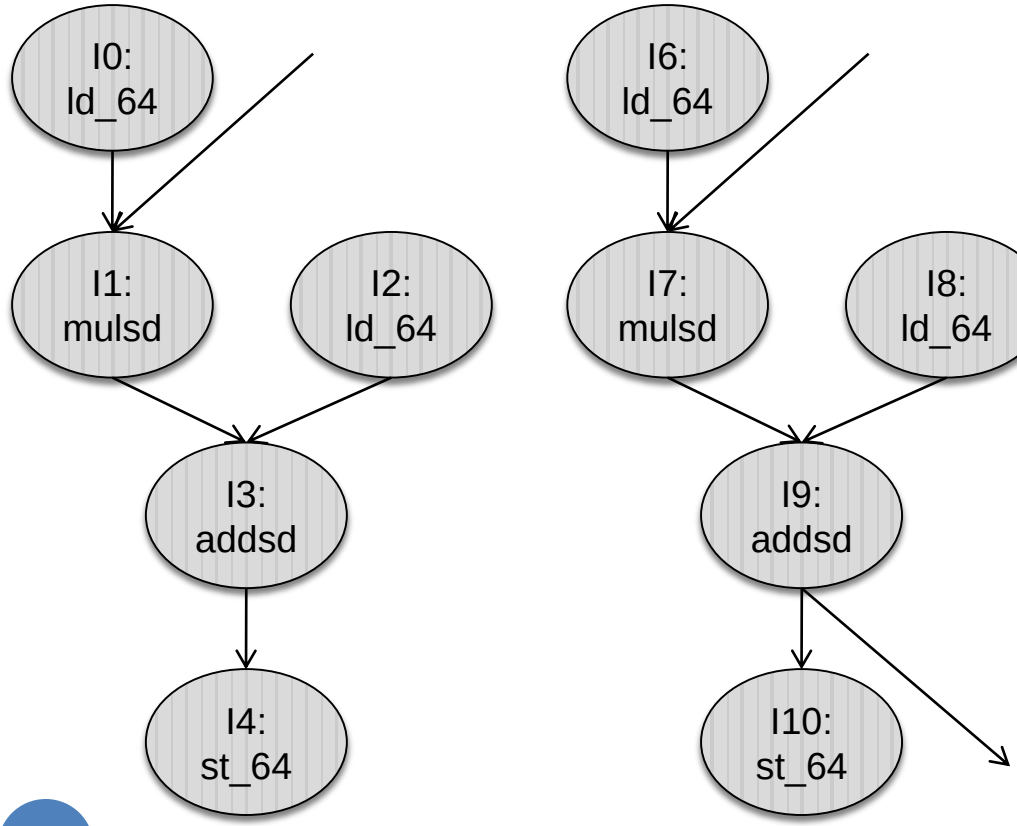
Vectorization (An Example)

```
void example(double *a, double *b)
{
   int i;
   for (i = 0; i < NUM_ITR; i++)
      a[i] += b[i] * CONST;
}
```

- Traditional Loop vectorization fails:
  - Due to complex interprocedural pointer analysis
- SLP fails:
  - Due to ambiguous memory references
- Possible Solution:
  - Multiple versions.
  - "__restrict annotation"

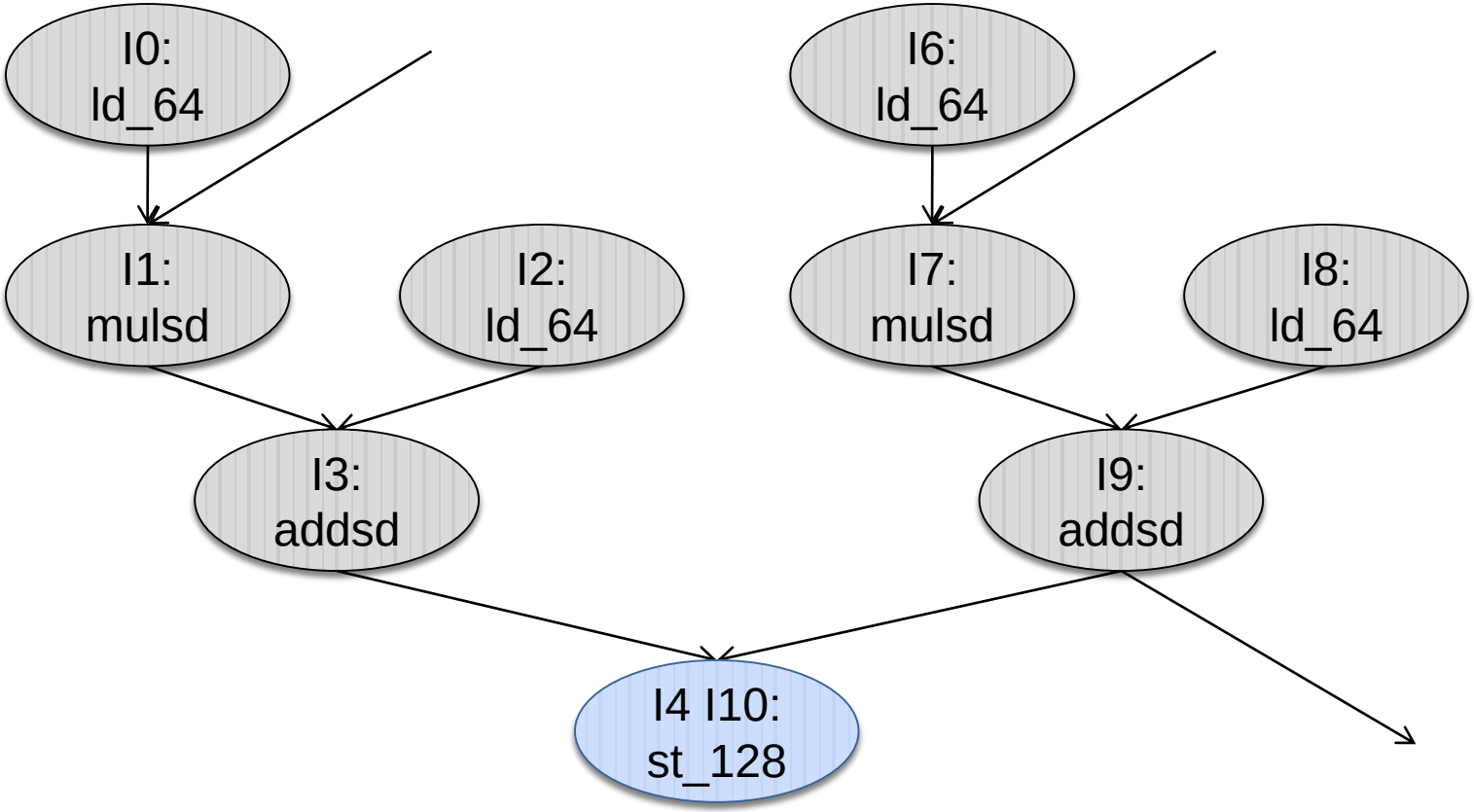
```
loop:   I0    ld_64   v2, M [r2 + r1 * 8]
        I1    mulsd   v3, v2, v1
        I2    ld_64   v4, M [r3 + r1 * 8]
        I3    addsd   v5, v4, v3
        I4    st_64   v5, M [r3 + r1 * 8]
        I5    add     r4, r1, 1
        I6    ld_64   v6, M [r2 + r4 * 8]
        I7    mulsd   v7, v6, v1
        I8    ld_64   v8, M [r3 + r4 * 8]
        I9    addsd   xmm0, v8, v7
        I10   st_64   xmm0, M [r3 + r4 * 8]
        I11   add     r1, r4, 1
        I12   cmp     r1, r0
        I13   jne     loop
```

Vectorization (An Example)

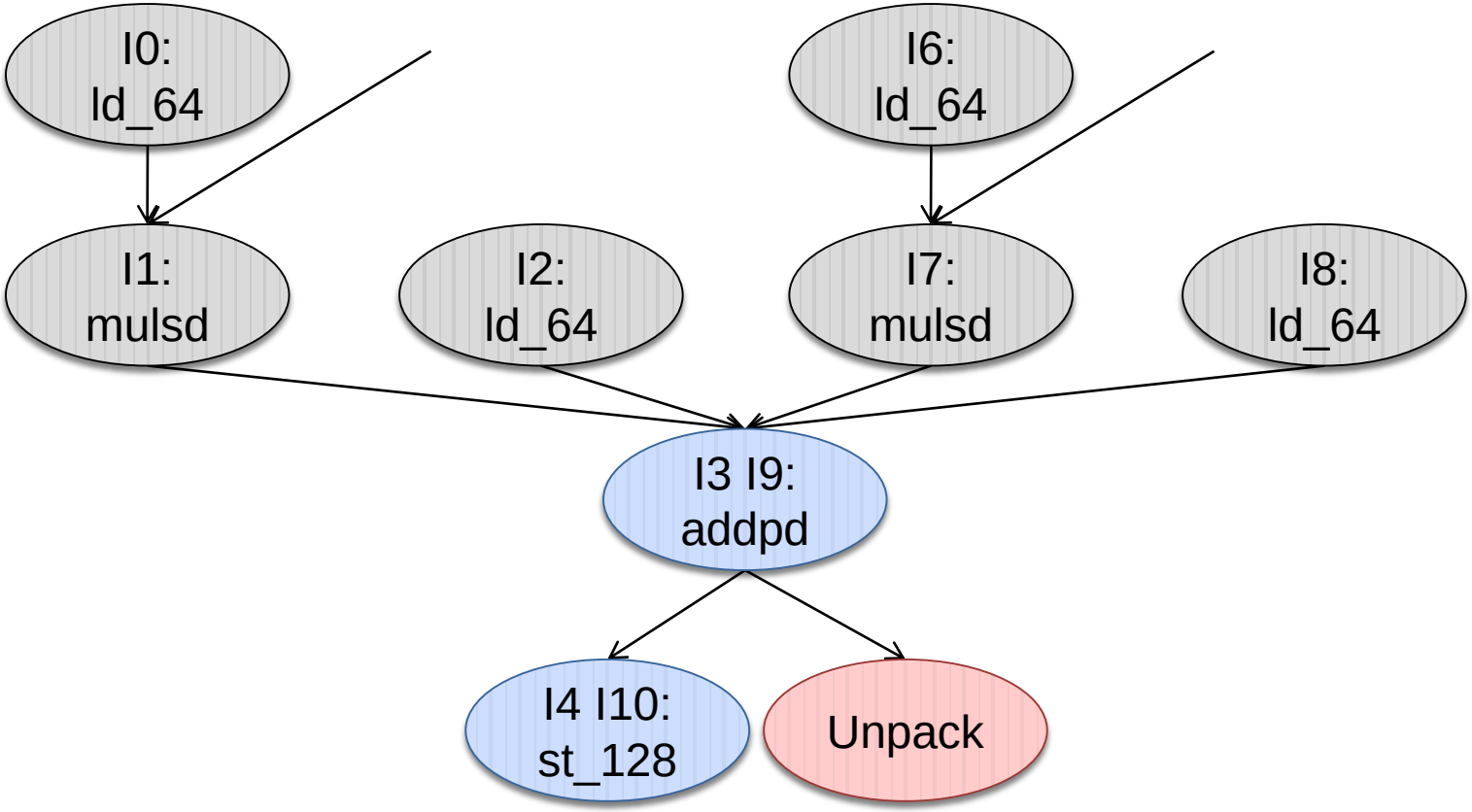



```
loop:   I0      ld_64    v2, M [r2 + r1 * 8]
        I1      mulsd    v3, v2, v1
        I2      ld_64    v4, M [r3 + r1 * 8]
        I3      addsd    v5, v4, v3
        I4      st_64    v5, M [r3 + r1 * 8]
        I5      add      r4, r1, 1
        I6      ld_64    v6, M [r2 + r4 * 8]
        I7      mulsd    v7, v6, v1
        I8      ld_64    v8, M [r3 + r4 * 8]
        I9      addsd    xmm0, v8, v7
        I10     st_64    xmm0, M [r3 + r4 * 8]
        I11     add      r1, r4, 1
        I12     cmp      r1, r0
        I13     jne      loop
```

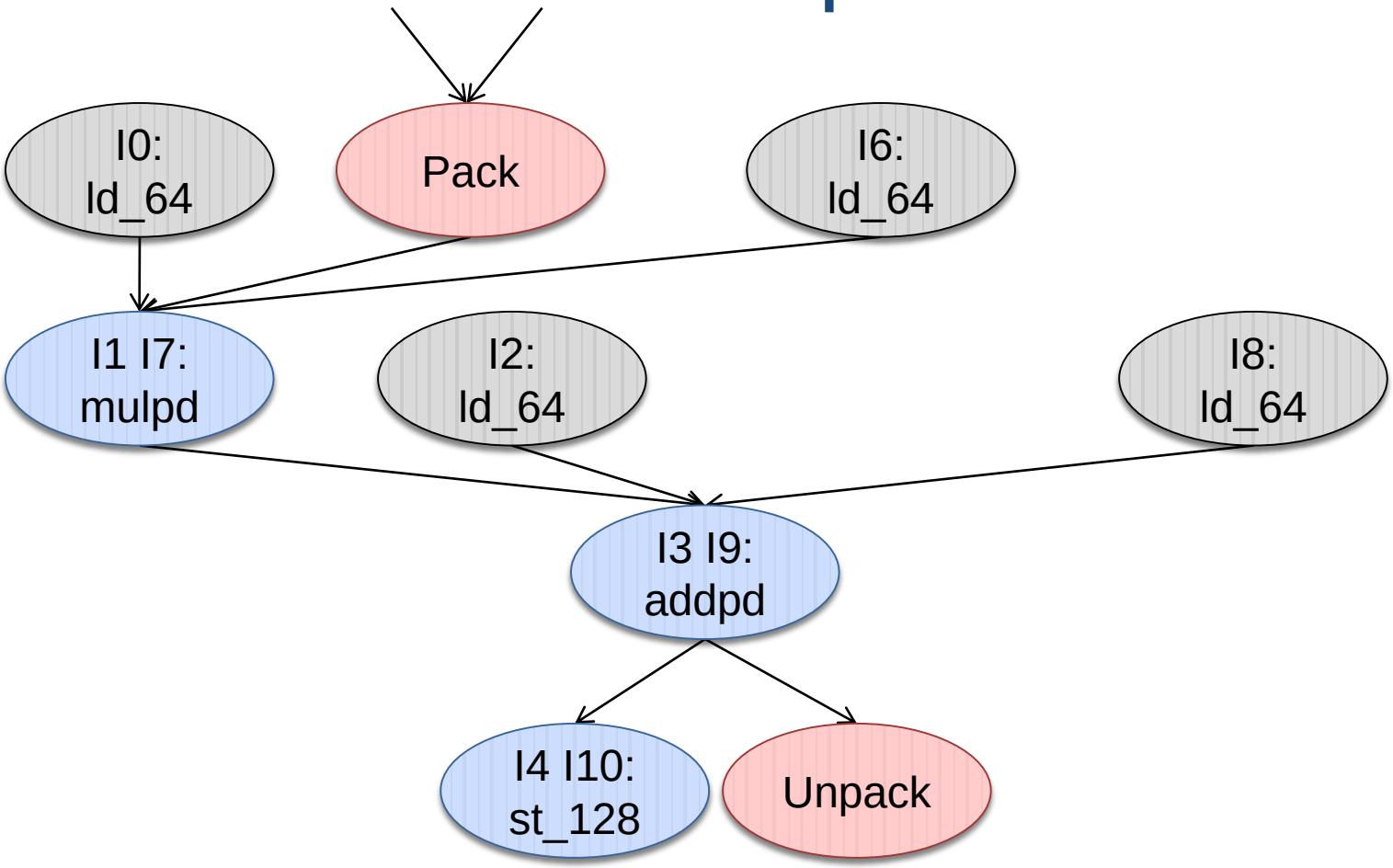
Vectorization Example



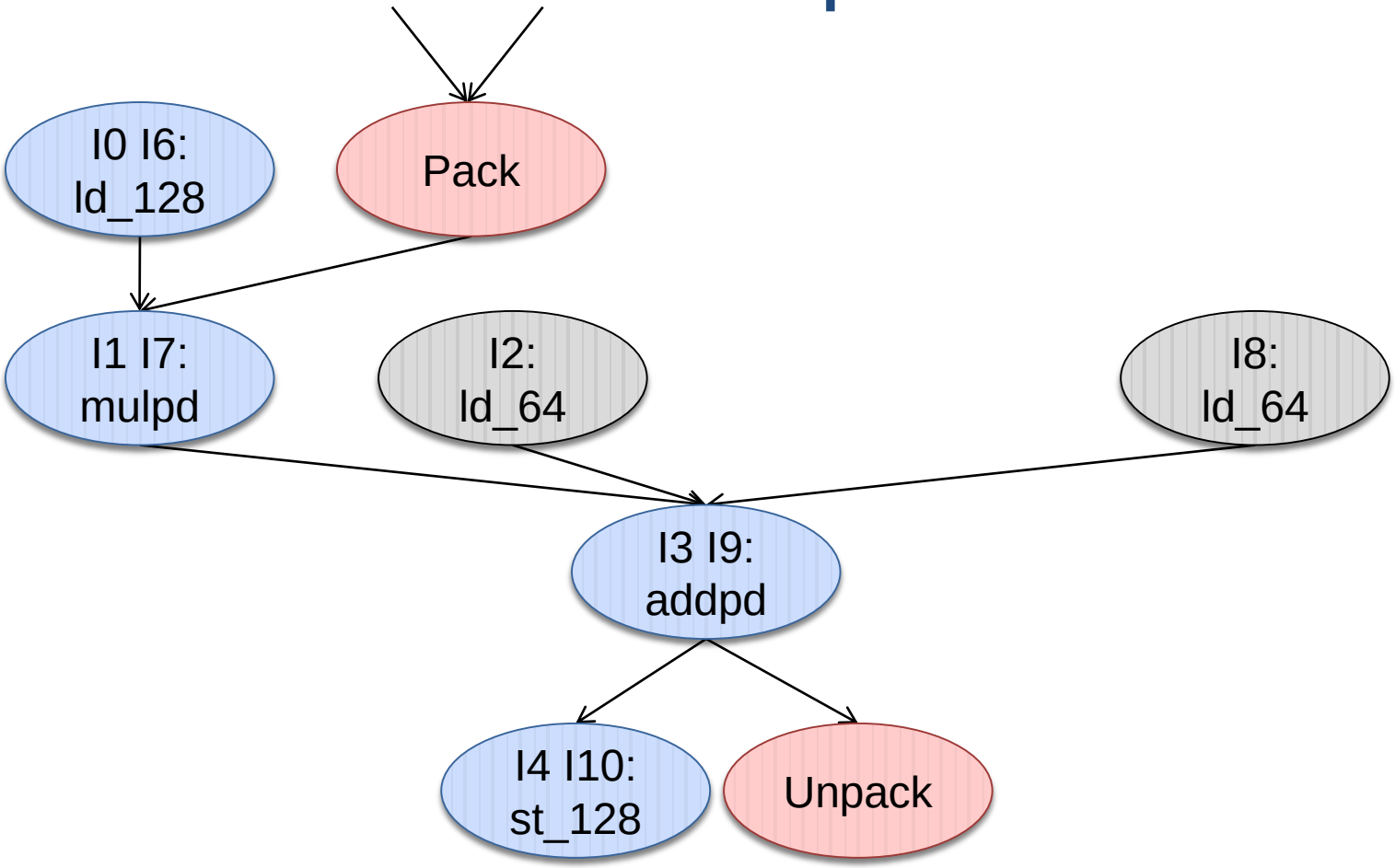
Vectorization Example



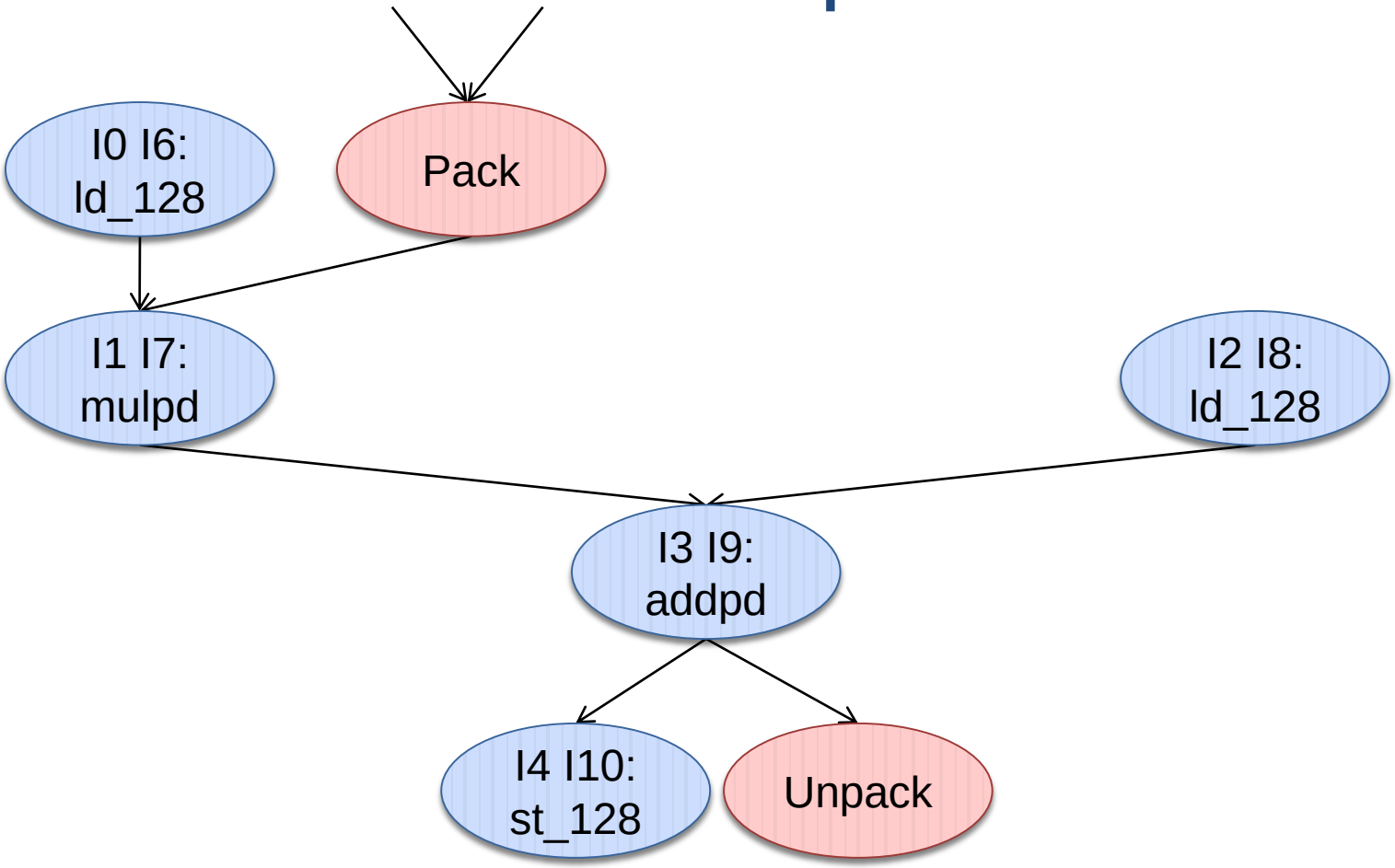
Vectorization Example



Vectorization Example



Vectorization Example



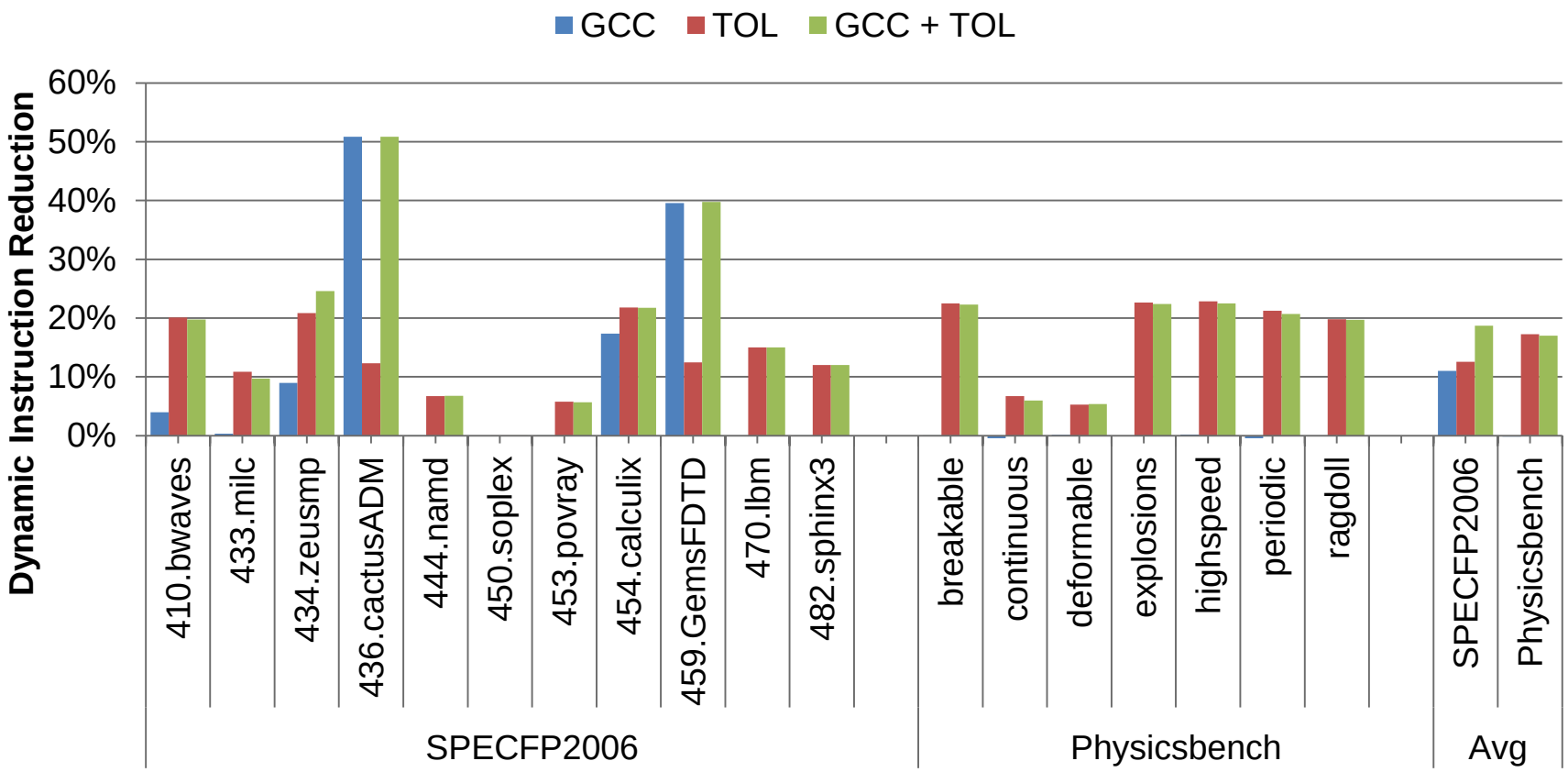
An Example (Loop)

```
loop:   Pack    v12, xmm1, xmm1
        ld_pd   v10, [ebx+eax*8]
        mulpd   v11, v12
        ld_pd   v13, [edx+eax*8]
        addpd   v14, v11, v13
        st_pd   [edx+eax*8], v14
        Unpack  xmm0, v14
        add     eax,0x2
        cmp     eax,ecx
        jne     loop
```

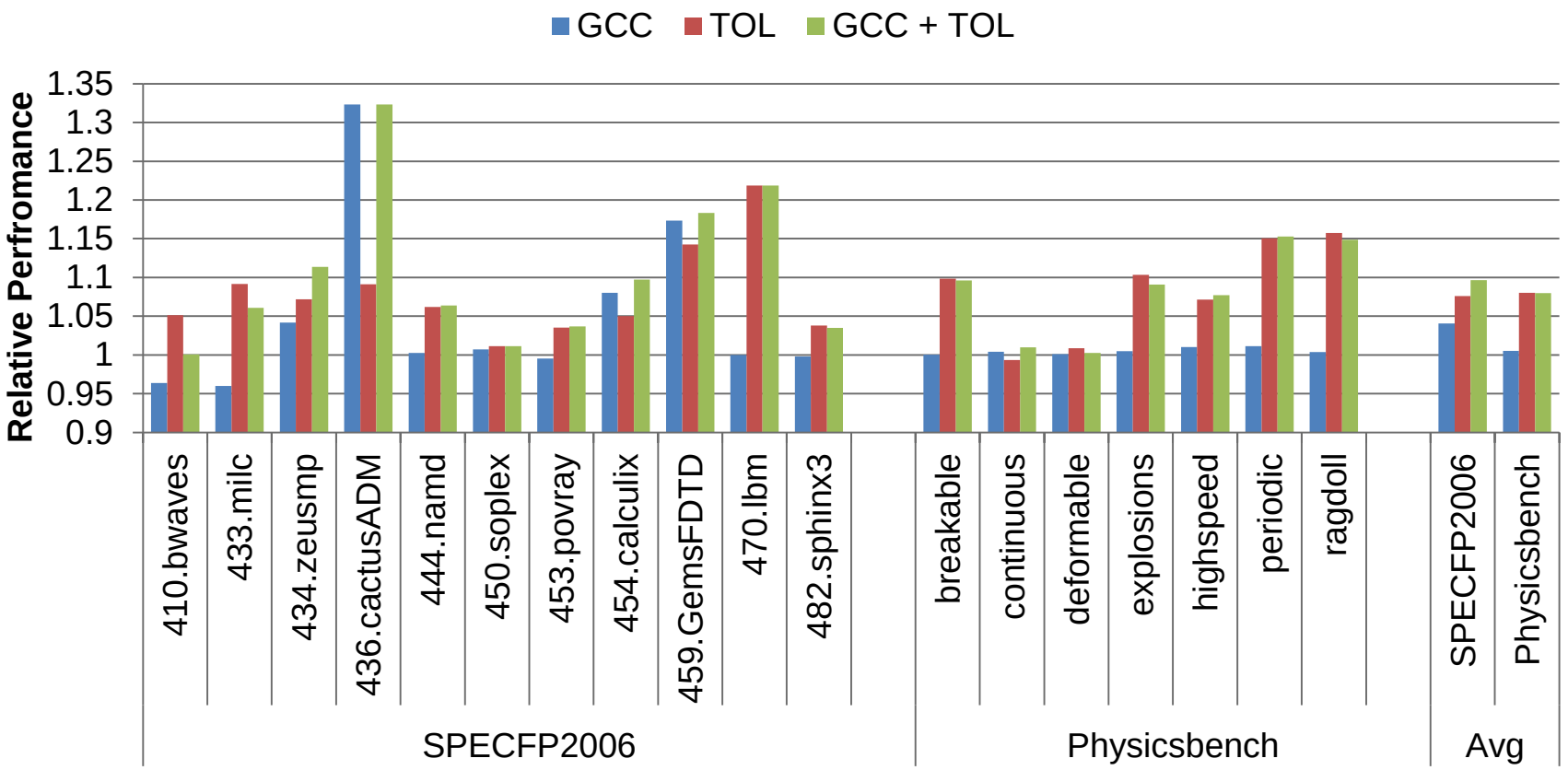
Outline

- Introduction
- Simulation Infrastructure
- Speculative Dynamic Vectorization
- Results
- Conclusion

Dynamic FP instruction reduction



Performance



Conclusion

- Combined vectorization extracts max opportunities
  - Compiler applies “costly” loop transformations
  - Runtime vectorizes aggressively/speculatively
  - Twice as good as static vectorization only
- Runtime vectorization
  - Speculation exposes more opportunities
  - Increases vectorization scope
  - Catches compiler missed opportunities
  - Portable vectorization
  - Legacy code vectorization
  - Outperforms static vectorization
- Negligible overhead (less than 0.5% on avg)

Thank you for your attention!