6.189 Multicore Programming Primer, January (IAP) 2007

6.189 IAP 2007

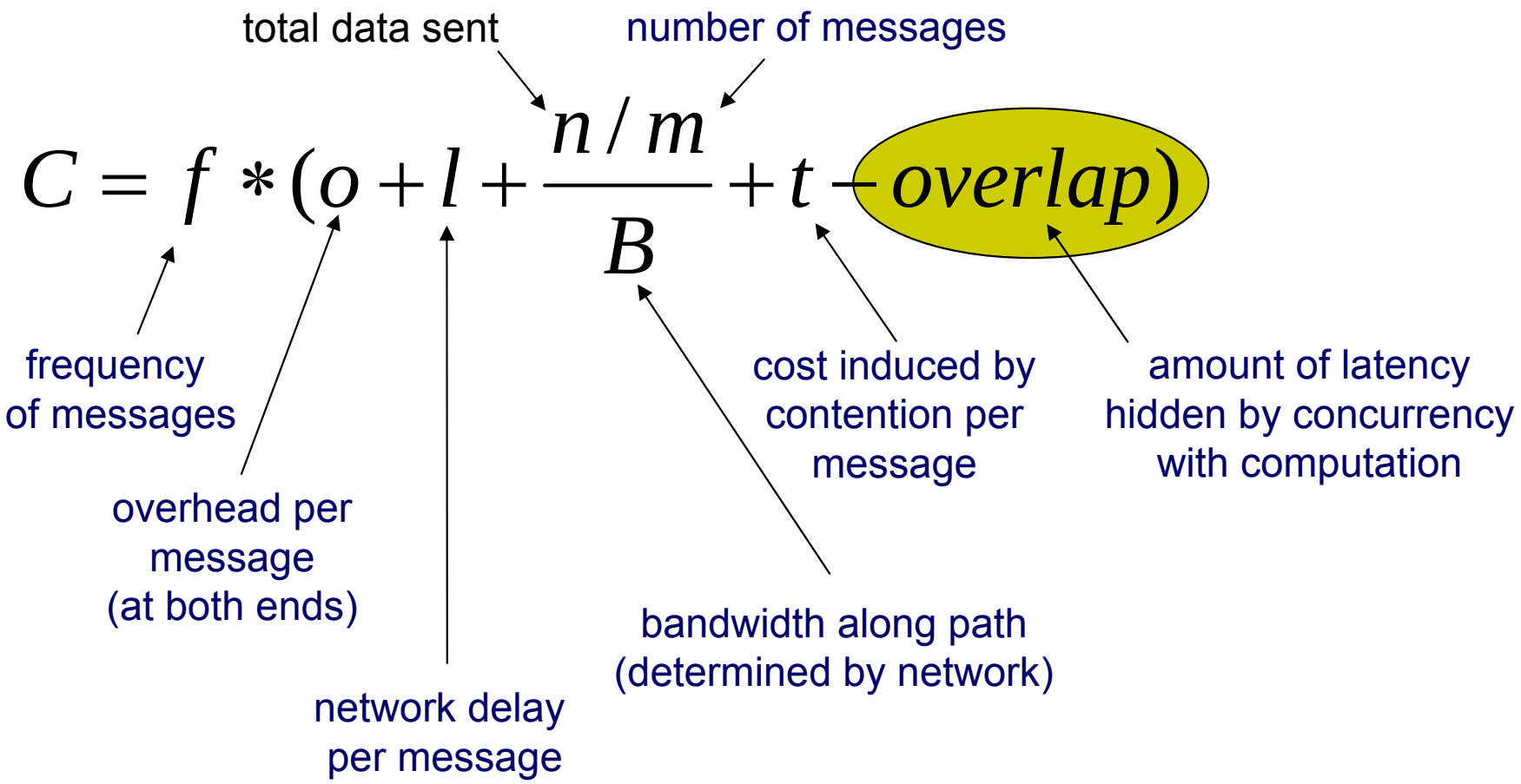
Lecture 10

Performance Monitoring and Optimizations

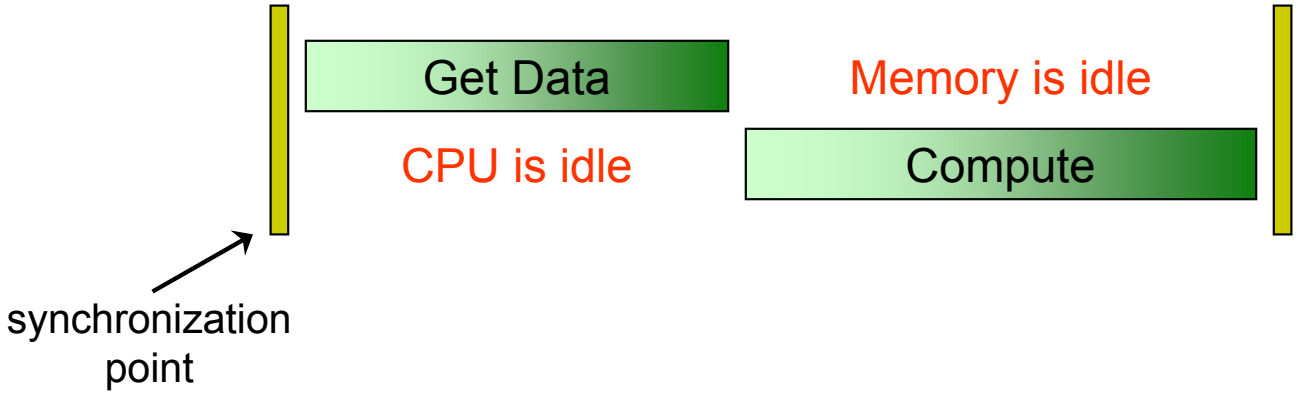
Review: Keys to Parallel Performance

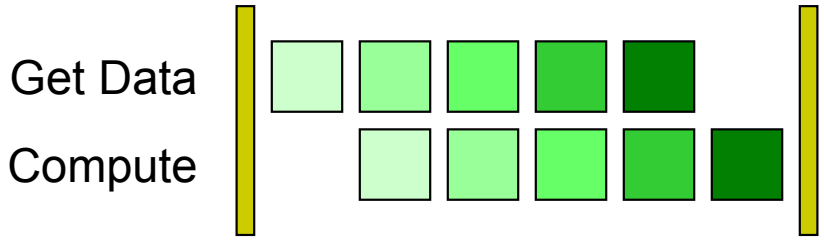
- **Coverage** or extent of parallelism in algorithm
  - Amdahl's Law
- **Granularity** of partitioning among processors
  - Communication cost and load balancing
- **Locality** of computation and communication
  - Communication between processors or between processors and their memories

Communication Cost Model



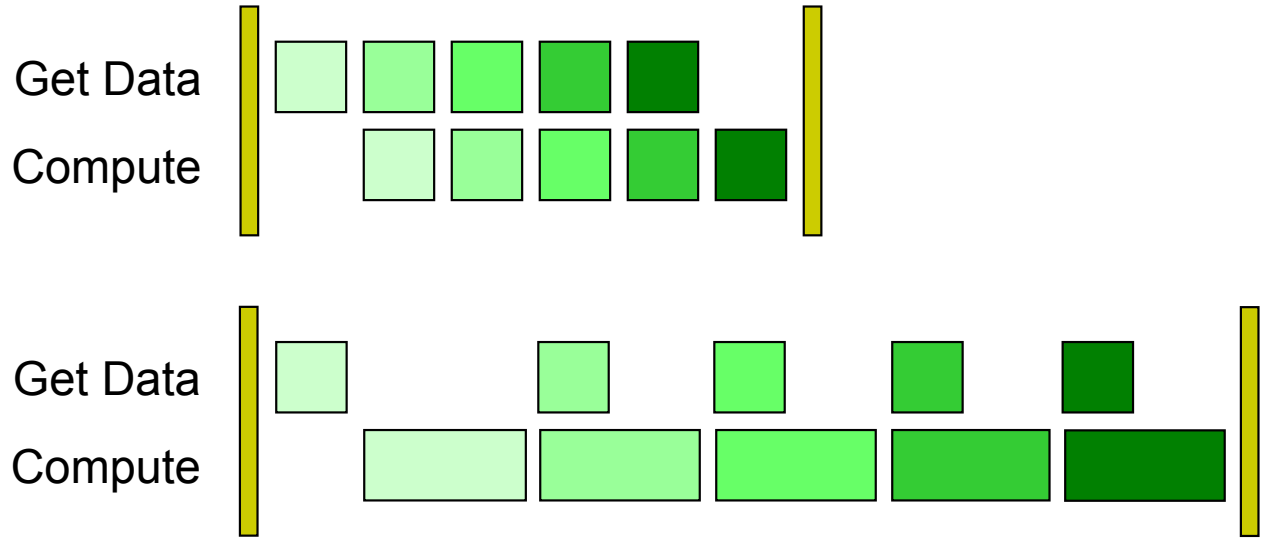
Overlapping Communication with Computation





Limits in Pipelining Communication

- Computation to communication ratio limits performance gains from pipelining



- Where else to look for performance?

Artifactual Communication

- Determined by program implementation and interactions with the architecture

- Examples:
  - Poor distribution of data across distributed memories
  - Unnecessarily fetching data that is not used
  - Redundant data fetches

Lessons From Uniprocessors

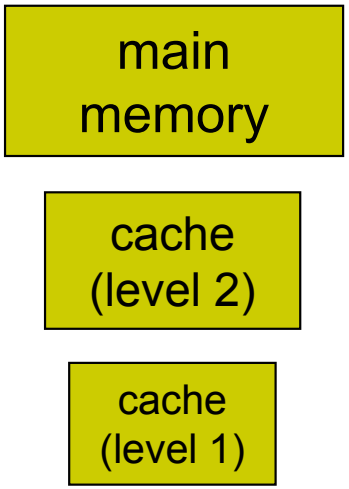
- In uniprocessors, CPU communicates with memory
- Loads and stores are to uniprocessors as "get" and "put" are to distributed memory multiprocessors
- How is communication overlap enhanced in uniprocessors?
  - Spatial locality
  - Temporal locality

Spatial Locality

- CPU asks for data at address 1000
- Memory sends data at address 1000 … 1064
  - Amount of data sent depends on architecture parameters such as the cache block size
- Works well if CPU actually ends up using data from 1001, 1002, …, 1064
- Otherwise wasted bandwidth and cache capacity

Temporal Locality

- Main memory access is expensive
- Memory hierarchy adds small but fast memories (caches) near the CPU
  - Memories get bigger as distance from CPU increases



- CPU asks for data at address 1000
- Memory hierarchy anticipates more accesses to same address and stores a local copy
- Works well if CPU actually ends up using data from 1000 over and over and over …
- Otherwise wasted cache capacity

Reducing Artifactual Costs in Distributed Memory Architectures

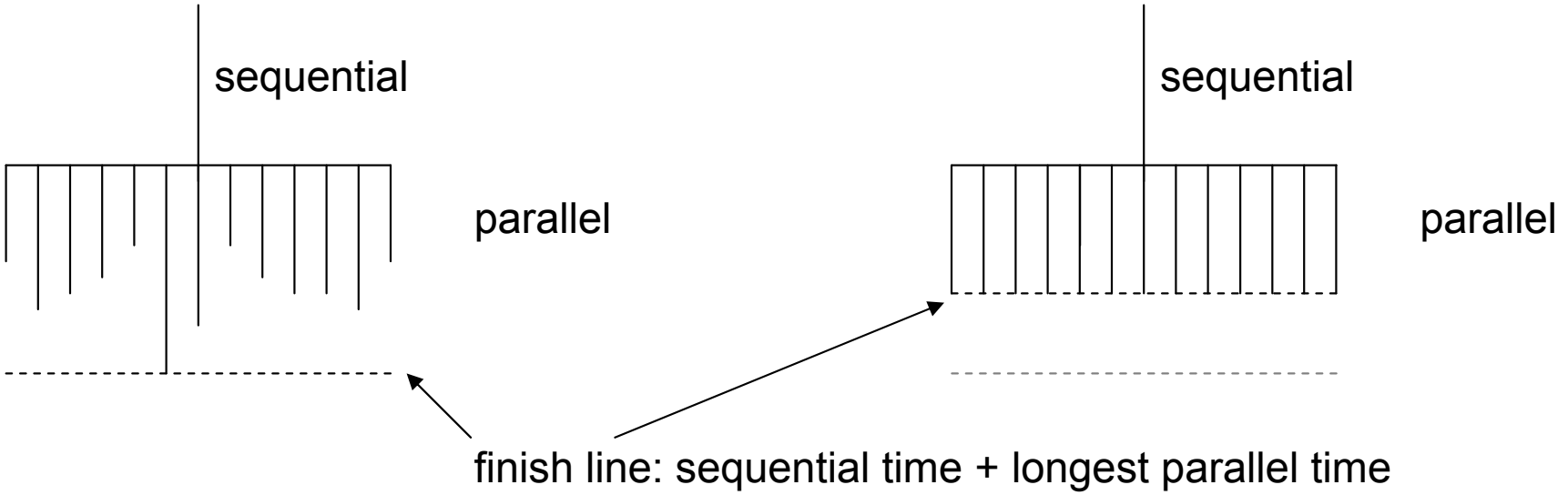
- Data is transferred in chunks to amortize communication cost
  - Cell: DMA gets up to 16K
  - Usually get a contiguous chunk of memory
- Spatial locality
  - Computation should exhibit good spatial locality characteristics
- Temporal locality
  - Reorder computation to maximize use of data fetched

6.189 IAP 2007

Single Thread Performance: the last frontier in the search for performance?

Single Thread Performance

- Tasks mapped to execution units (threads)
- Threads run on individual processors (cores)



- Two keys to faster execution
  - Load balance the work among the processors
  - Make execution on each processor faster

Understanding Performance

- Need some way of measuring performance
  - Coarse grained measurements

```
% gcc sample.c
% time a.out
2.312u 0.062s 0:02.50 94.8%
% gcc sample.c –O3
% time a.out
1.921u 0.093s 0:02.03 99.0%
```

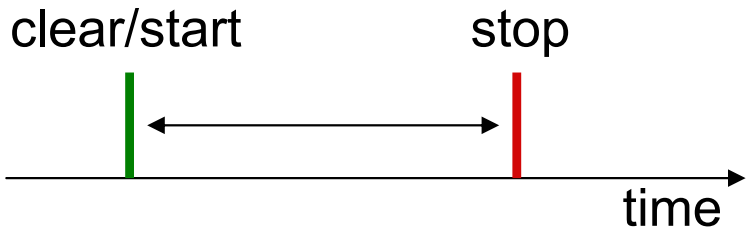
  - … but did we learn much about what's going on?

```
#define N (1 << 23)
#define T (10)
#include <string.h>
double a[N],b[N];

void cleara(double a[N]) {
  int i;
  for (i = 0; i < N; i++) {
    a[i] = 0;
  }
}
int main() {
  double s=0,s2=0; int i,j;
  for (j = 0; j < T; j++) {
    for (i = 0; i < N; i++) {
      b[i] = 0;
    }
    cleara(a);
    memset(a,0,sizeof(a));
                                  record start time
    for (i = 0; i < N; i++) {
      s += a[i] * b[i];
      s2 += a[i] * a[i] + b[i] * b[i];
    }
                                  record stop time
  }
  printf("s %f s2 %f\n",s,s2);
}
```

Measurements Using Counters

- Increasingly possible to get accurate measurements using performance counters
  - Special registers in the hardware to measure events
- Insert code to start, read, and stop counter
  - Measure exactly what you want, anywhere you want
  - Can measure communication and computation duration
  - But requires manual changes
  - Monitoring nested scopes is an issue
  - Heisenberg effect: counters can perturb execution time



Dynamic Profiling

- Event-based profiling
  - Interrupt execution when an event counter reaches a threshold
- Time-based profiling
  - Interrupt execution every *t* seconds
- Works without modifying your code
  - Does not require that you know where problem might be
  - Supports multiple languages and programming models
  - Quite efficient for appropriate sampling frequencies

Counter Examples

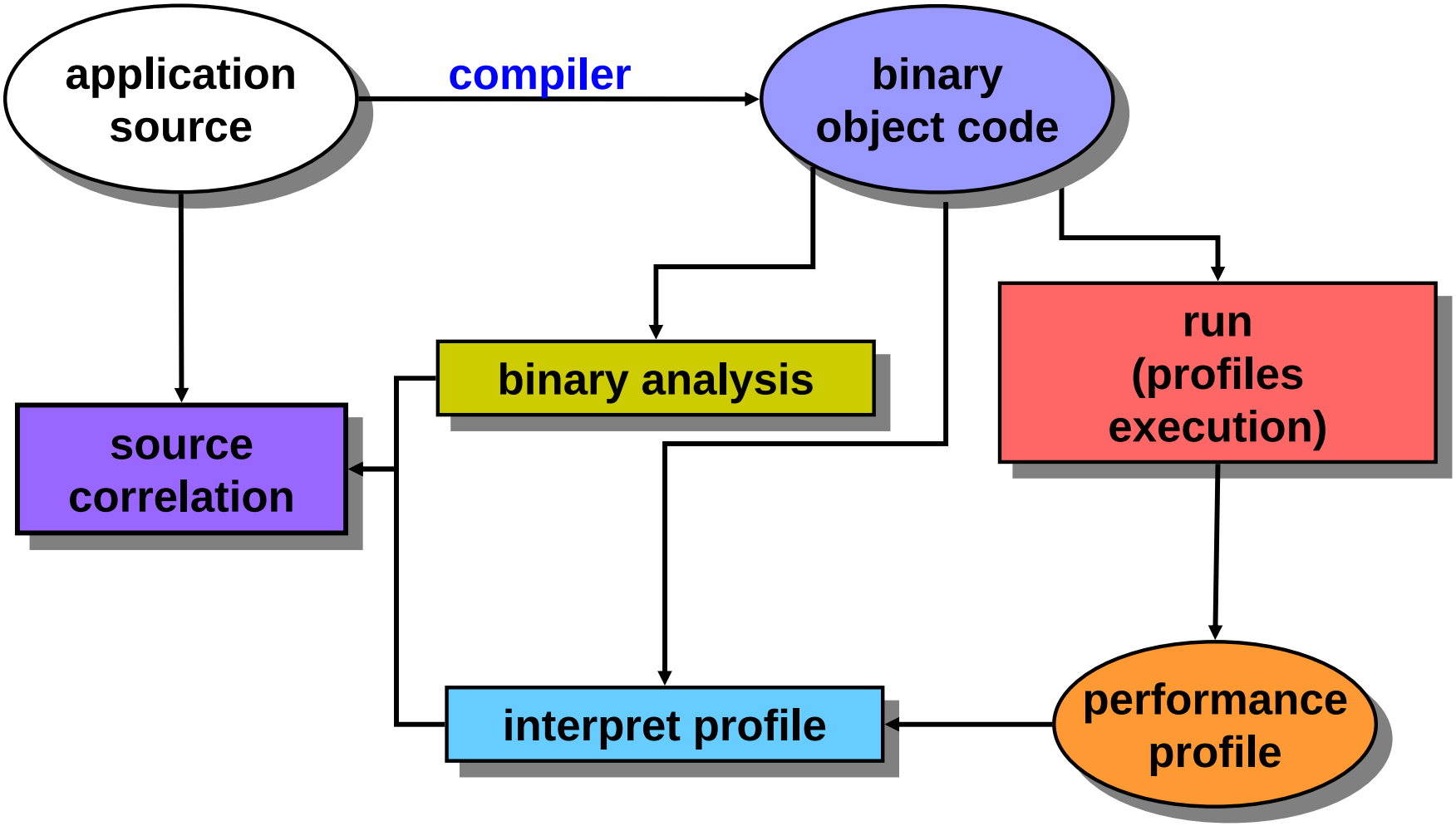
- Cycles (clock ticks)
- Pipeline stalls
- Cache hits
- Cache misses
- Number of instructions
- Number of loads
- Number of stores
- Number of floating point operations
- …

Useful Derived Measurements

- Processor utilization
  - Cycles / Wall Clock Time
- Instructions per cycle
  - Instructions / Cycles
- Instructions per memory operation
  - Instructions / Loads + Stores
- Average number of instructions per load miss
  - Instructions / L1 Load Misses
- Memory traffic
  - Loads + Stores * L_k Cache Line Size
- Bandwidth consumed
  - Loads + Stores * L_k Cache Line Size / Wall Clock Time

- Many others
  - Cache miss rate
  - Branch misprediction rate
  - …

Common Profiling Workflow



Popular Runtime Profiling Tools

- GNU gprof
  - Widely available with UNIX/Linux distributions

```
gcc –O2 –pg foo.c –o foo
./foo
gprof foo
```

- HPC Toolkit
  - http://www.hipersoft.rice.edu/hpctoolkit/
- PAPI
  - http://icl.cs.utk.edu/papi/
- VTune
  - http://www.intel.com/cd/software/products/asmo-na/eng/vtune/
- Many others

GNU gprof

- MPEG-2 decoder (reference implementation)

```
%./mpeg2decode -b mei16v2.m2v -f -r
```

-r uses double precision inverse DCT

% time	cumulative seconds	self seconds	calls	self ns/call	total ns/call	name
90.48	0.19	0.19	7920	23989.90	23989.90	Reference_IDCT
4.76	0.20	0.01	2148	4655.49	4655.49	Decode_MPEG1_Intra_Block

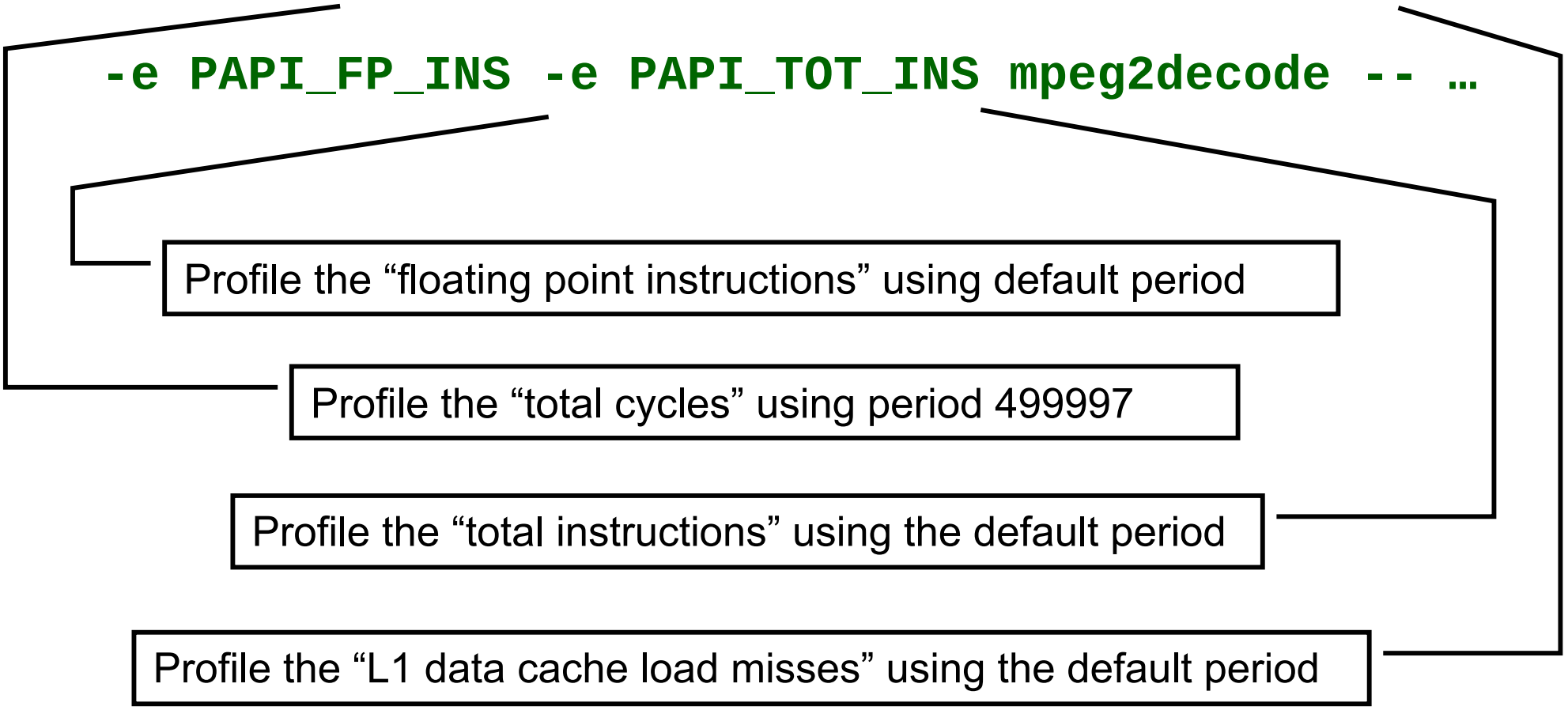
```
%./mpeg2decode -b mei16v2.m2v -f
```

uses fast integer based inverse DCT instead

66.67	0.02	0.02	8238	2427.77	2427.77	form_component_prediction
33.33	0.03	0.01	63360	157.83	157.83	idctcol

HPC Toolkit

```
% hpcrun -e PAPI_TOT_CYC:499997 -e PAPI_L1_LDM \
  -e PAPI_FP_INS -e PAPI_TOT_INS mpeg2decode -- …
```



Running this command on a machine “sloth” produced a data file **`mpeg2dec.PAPI_TOT_CYC-etc.sloth.1234`**

Interpreting a Profile

```
% hpcprof -e mpeg2dec mped2dec.PAPI_TOT_CYC-etc.sloth.1234
Columns correspond to the following events [event:period (events/sample)]
  PAPI_TOT_CYC:499997 - Total cycles (698 samples)
  PAPI_L1_LDM:32767 - Level 1 load misses (27 samples)
  PAPI_FP_INS:32767 - Floating point instructions (789 samples)
  PAPI_TOT_INS:32767 - Instructions completed (4018 samples)

Load Module Summary:
  91.7%  74.1% 100.0%  84.1% /home/guru/mpeg2decode
   8.3%  25.9%   0.0%  15.9% /lib/libc-2.2.4.so

Function Summary:
   ...

Line summary:
   ...
```

hpcprof Annotated Source File

```
 1                                        #define N (1 << 23)
 2                                        #define T (10)
 3                                        #include <string.h>
 4                                        double a[N],b[N];
 5                                        void cleara(double a[N]) {
 6                                          int i;
 7  14.5%  16.7%   0.0%   0.2%            for (i = 0; i < N; i++) {
 8                                            a[i] = 0;
 9                                          }
10                                        }
11                                        int main() {
12                                          double s=0,s2=0; int i,j;
13                                          for (j = 0; j < T; j++) {
14   9.0%  14.6%   0.0%   0.1%              for (i = 0; i < N; i++) {
15   5.6%   6.2%   0.0%   0.1%                b[i] = 0;
16                                          }
17                                          cleara(a);
18                                          memset(a,0,sizeof(a));
19   3.6%   5.8%  10.6%   9.2%              for (i = 0; i < N; i++) {
20  36.2%  32.9%  53.2%  49.1%                s += a[i]*b[i];
21  16.7%  23.8%  36.2%  41.2%                s2 += a[i]*a[i]+b[i]*b[i];
22                                          }
23                                          }
24                                          printf("s %d s2 %d\n",s,s2);
25                                        }
```

hpcviewer Screenshot

Performance in Uniprocessors
time = compute + wait

- Instruction level parallelism
  - Multiple functional units, deeply pipelined, speculation, ...
- Data level parallelism
  - SIMD: short vector instructions (multimedia extensions)
    - Hardware is simpler, no heavily ported register files
    - Instructions are more compact
    - Reduces instruction fetch bandwidth
- Complex memory hierarchies
  - Multiple level caches, may outstanding misses, prefetching, …

SIMD

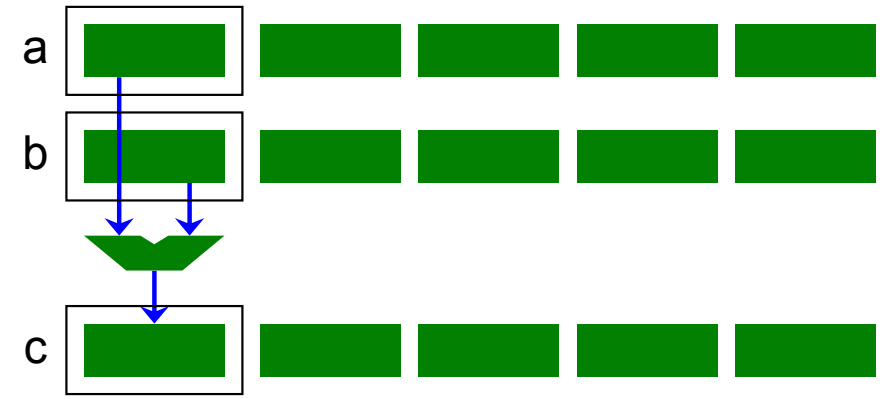
- Single Instruction, Multiple Data
- SIMD registers hold short vectors
- Instruction operates on all elements in SIMD register at once

Scalar code

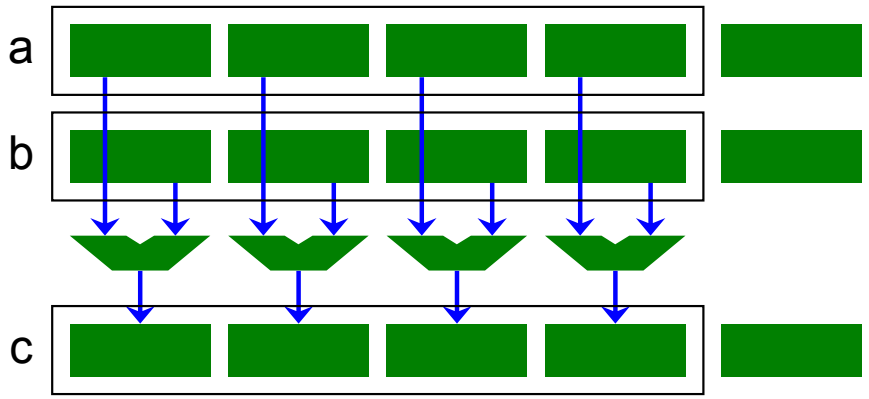
```
for (int i = 0; i < n; i+=1) {
  c[i] = a[i] + b[i]
}
```

Vector code

```
for (int i = 0; i < n; i += 4) {
  c[i:i+3] = a[i:i+3] + b[i:i+3]
}
```



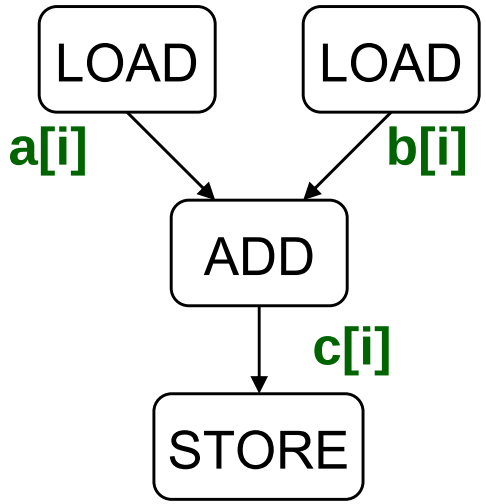
scalar register



SIMD register

SIMD Example

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



Cycle	Slot 1	Slot 2
1	LOAD	LOAD
2	ADD	
3	STORE	
…	…	…

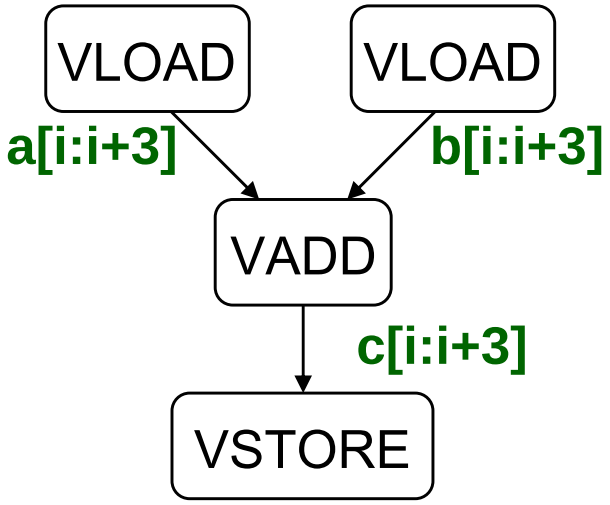
Estimated cycles for loop:

- Scalar loop

  n iterations ∗ 3 cycles/iteration

SIMD Example

```
for (int i = 0; i < n; i+=4) {
  c[i:i+3] = a[i:i+3] + b[i:i+3]
}
```



Cycle	Slot 1	Slot 2
1	VLOAD	VLOAD
2	VADD	
3	VSTORE	
…	…	…

Estimated cycles for loop:

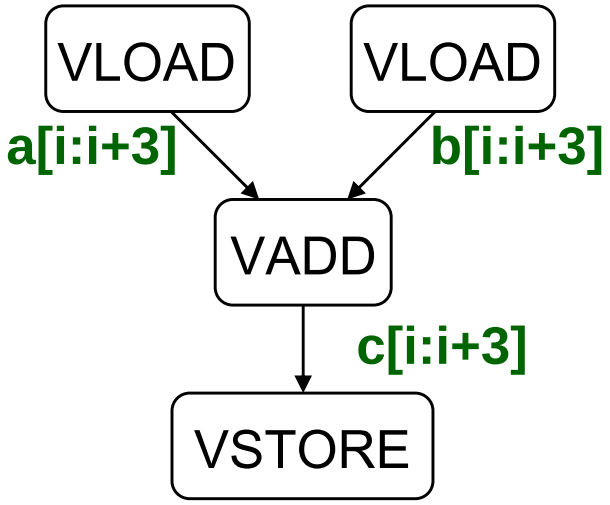
- Scalar loop

  n iterations ∗ 3 cycles/iteration
- SIMD loop

  n/4 iterations ∗ 3 cycles/iteration
- Speedup: ?

SIMD Example

```
for (int i = 0; i < n; i+=4) {
  c[i:i+3] = a[i:i+3] + b[i:i+3]
}
```



Cycle	Slot 1	Slot 2
1	VLOAD	VLOAD
2	VADD	
3	VSTORE	
…	…	…

Estimated cycles for loop:

- Scalar loop

  n iterations ∗ 3 cycles/iteration
- SIMD loop

  n/4 iterations ∗ 3 cycles/iteration
- Speedup: 4x

SIMD in Major ISAs

Instruction Set	Architecture	SIMD Width	Floating Point
AltiVec	PowerPC	128	yes
MMX/SSE	Intel	64/128	yes
3DNow!	AMD	64	yes
VIS	Sun	64	no
MAX2	HP	64	no
MVI	Alpha	64	no
MDMX	MIPS V	64	yes

- And of course Cell
  - SPU has 128 128-bit registers
  - All instructions are SIMD instructions
  - Registers are treated as short vectors of 8/16/32-bit integers or single/double-precision floats

Using SIMD Instructions

- Library calls and inline assembly
  - Difficult to program
  - Not portable
- Different extensions to the same ISA
  - MMX and SSE
  - SSE vs. 3DNow!
- You'll get first hand-experience experience with Cell

Superword Level Parallelism (SLP)

- Small amount of parallelism
  - Typically 2 to 8-way

  exists within basic blocks

- Uncovered with simple analysis

Samuel Larsen. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. Master's thesis, Massachusetts Institute of Technology, May 2000.

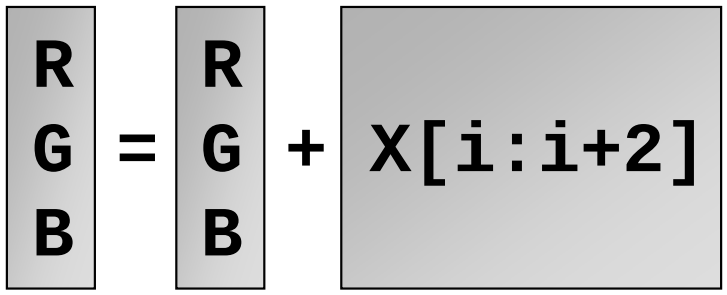
1. Independent ALU Ops

```
R = R + XR * 1.08327
G = G + XG * 1.89234
B = B + XB * 1.29835
```

```
R     R     XR     1.08327
G  =  G  +  XG  *  1.89234
B     B     XB     1.29835
```

2. Adjacent Memory References

```
R = R + X[i+0]
G = G + X[i+1]
B = B + X[i+2]
```



3. Vectorizable Loops

```
for (i=0; i<100; i+=1)
   A[i+0] = A[i+0] + B[i+0]
```

3. Vectorizable Loops

```
for (i=0; i<100; i+=4)
   A[i+0] = A[i+0] + B[i+0]
   A[i+1] = A[i+1] + B[i+1]
   A[i+2] = A[i+2] + B[i+2]
   A[i+3] = A[i+3] + B[i+3]
```

```
for (i=0; i<100; i+=4)
   A[i:i+3] = B[i:i+3] + C[i:i+3]
```

4. Partially Vectorizable Loops

```
for (i=0; i<16; i+=1)
   L = A[i+0] – B[i+0]
   D = D + abs(L)
```

4. Partially Vectorizable Loops

```
for (i=0; i<16; i+=2)
    L = A[i+0] – B[i+0]
    D = D + abs(L)
    L = A[i+1] – B[i+1]
    D = D + abs(L)
```

```
for (i=0; i<16; i+=2)
    L0 = A[i:i+1] – B[i:i+1]
    L1
    D = D + abs(L0)
    D = D + abs(L1)
```

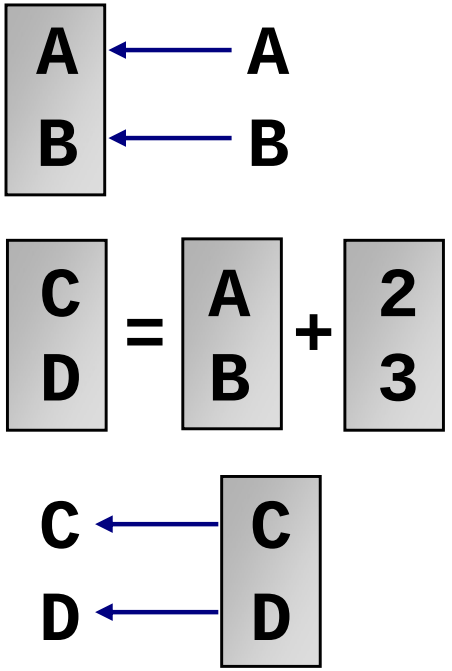
From SLP to SIMD Execution

- Benefits
  - Multiple ALU ops → One SIMD op
  - Multiple load and store ops → One wide memory op
- Cost
  - Packing and unpacking vector register
  - Reshuffling within a register

Packing and Unpacking Costs

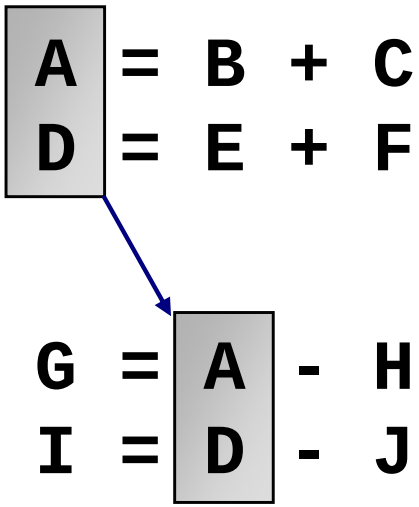
- Packing source operands
- Unpacking destination operands

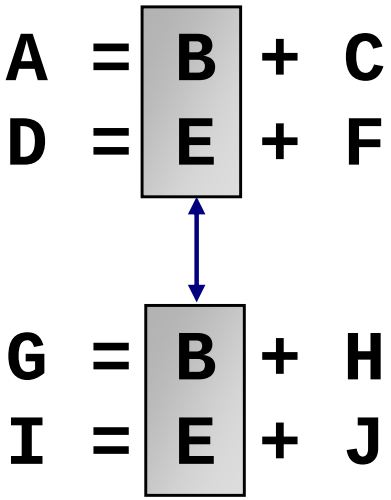
```
A = f()
B = g()
C = A + 2
D = B + 3
E = C / 5
F = D * 7
```



Packing Costs can be Amortized

- Use packed result operands
- Share packed source operands





Adjacent Memory is Key

- Large potential performance gains
  - Eliminate load/store instructions
  - Reduce memory bandwidth
- Few packing possibilities
  - Only one ordering exploits pre-packing

SLP Extraction Algorithm

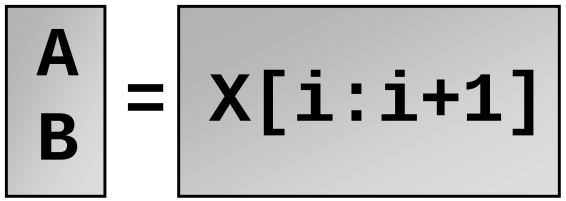
- Identify adjacent memory references

```
A = X[i+0]
C = E * 3
B = X[i+1]
H = C – A
D = F * 5
J = D - B
```

SLP Extraction Algorithm

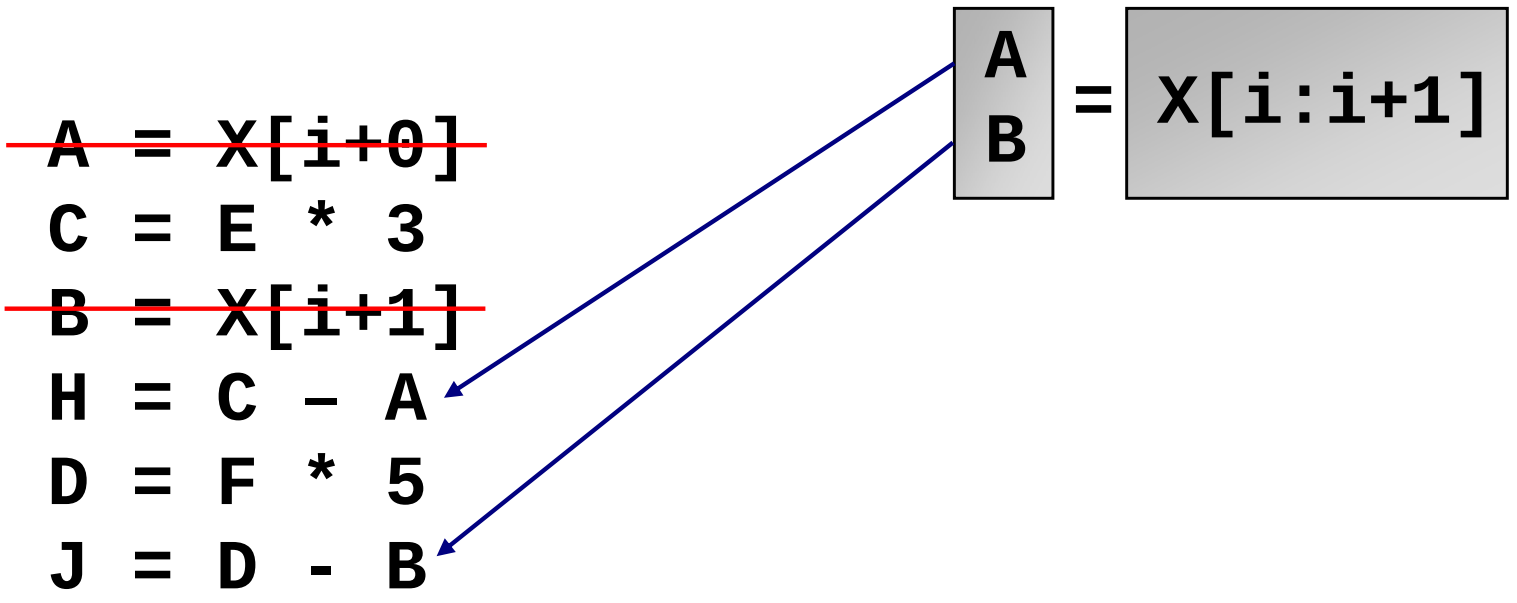
- Identify adjacent memory references

```
A = X[i+0]
C = E * 3
B = X[i+1]
H = C – A
D = F * 5
J = D - B
```



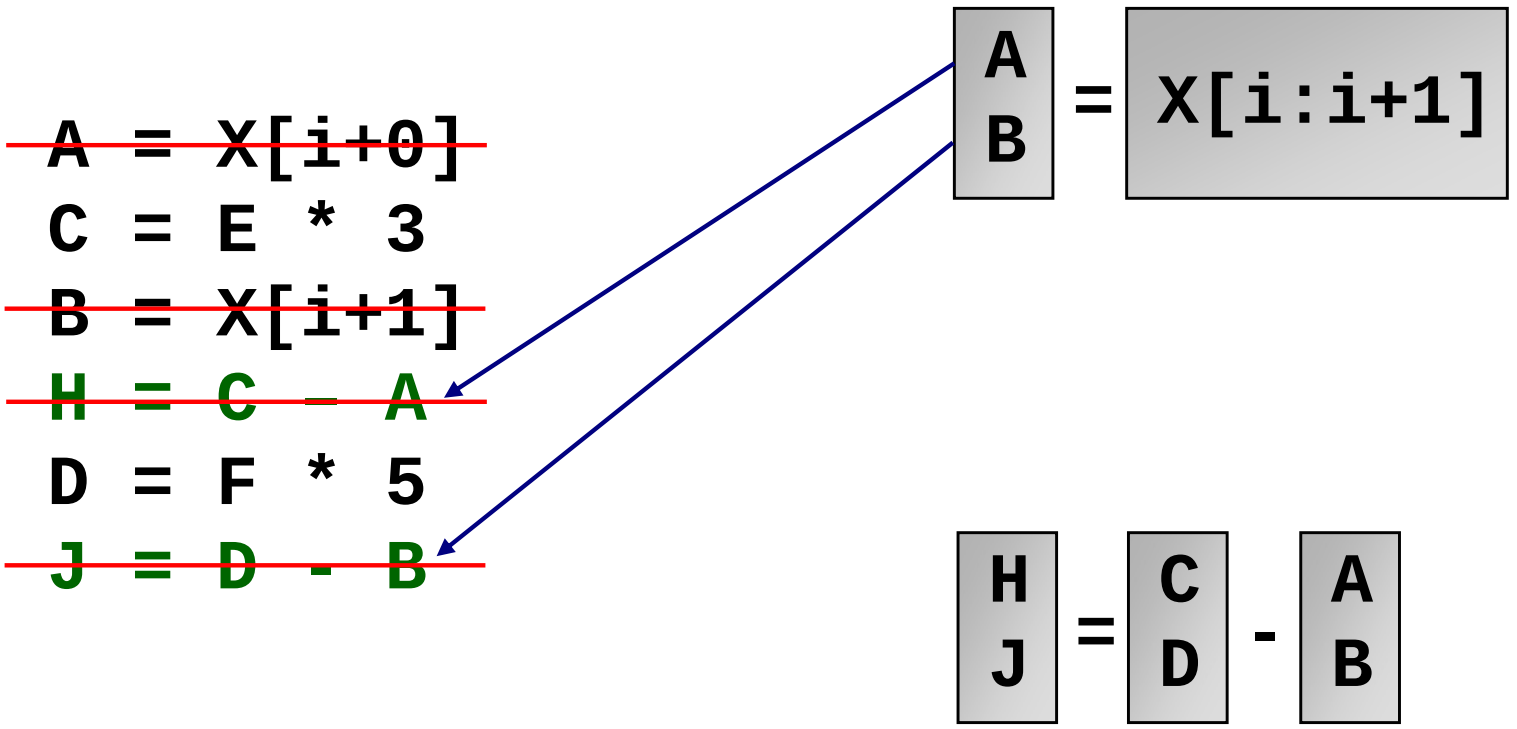
SLP Extraction Algorithm

- Follow operand use



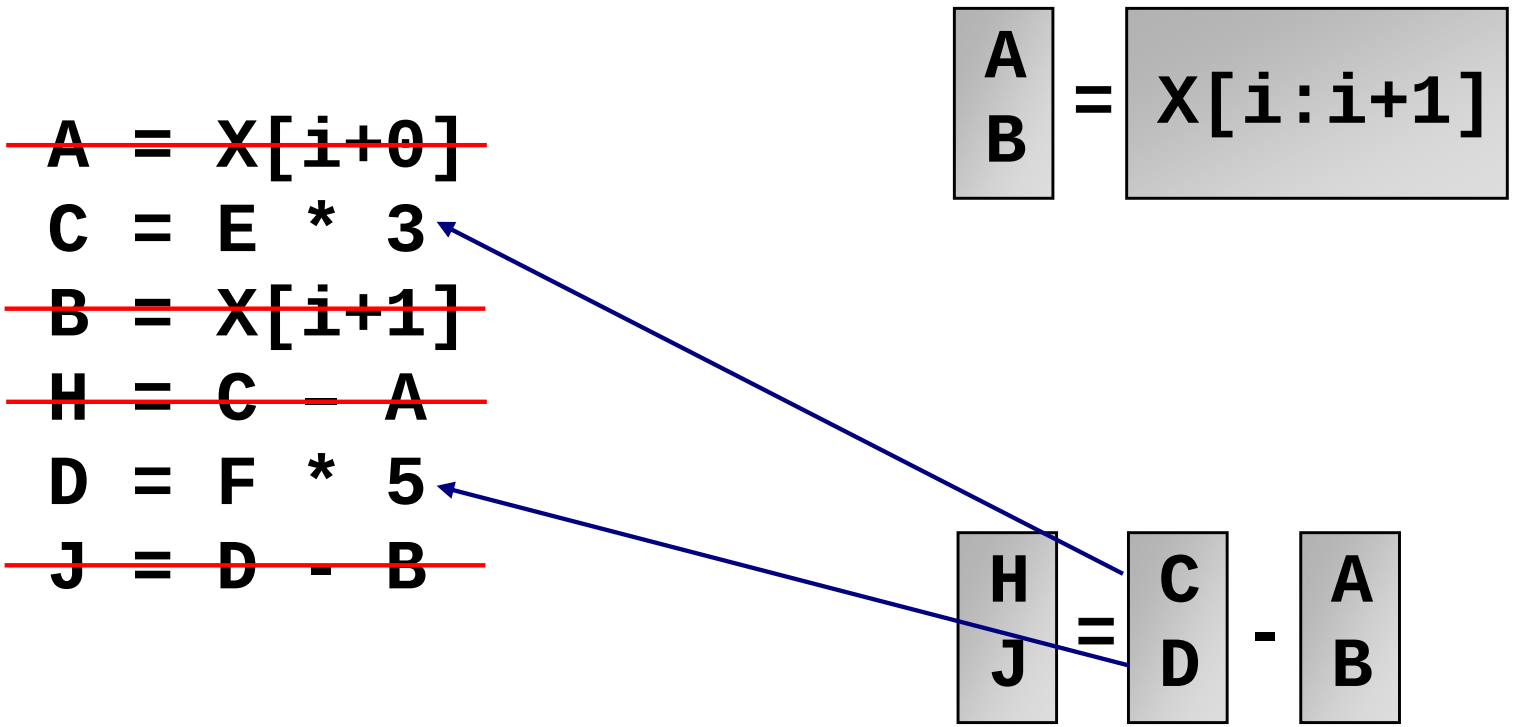
SLP Extraction Algorithm

- Follow operand use



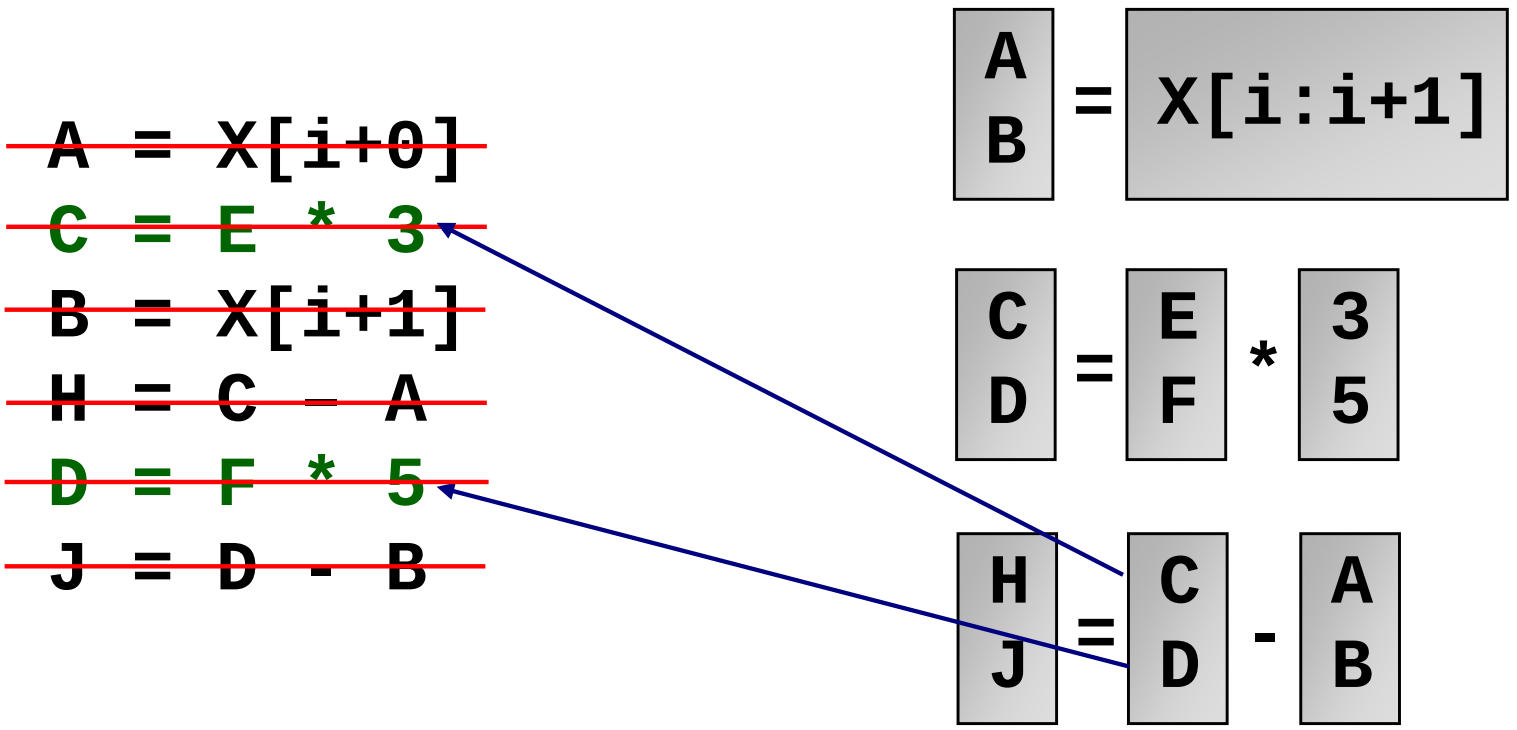
SLP Extraction Algorithm

- Follow operand use



SLP Extraction Algorithm

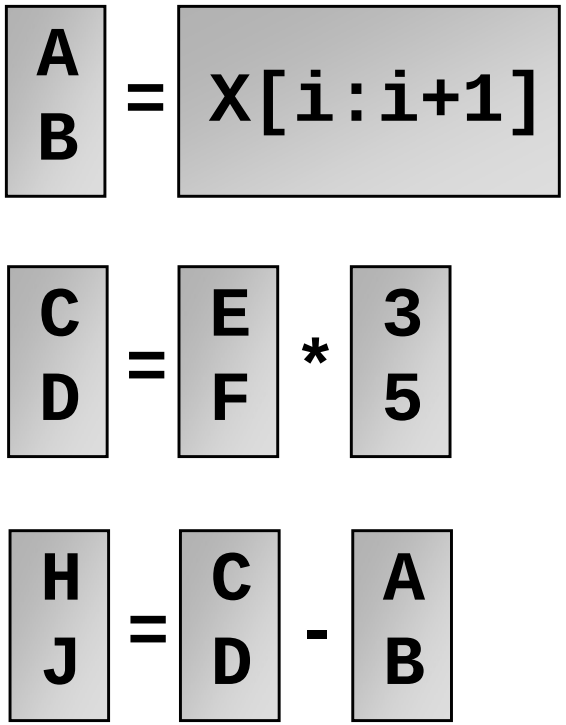
- Follow operand use



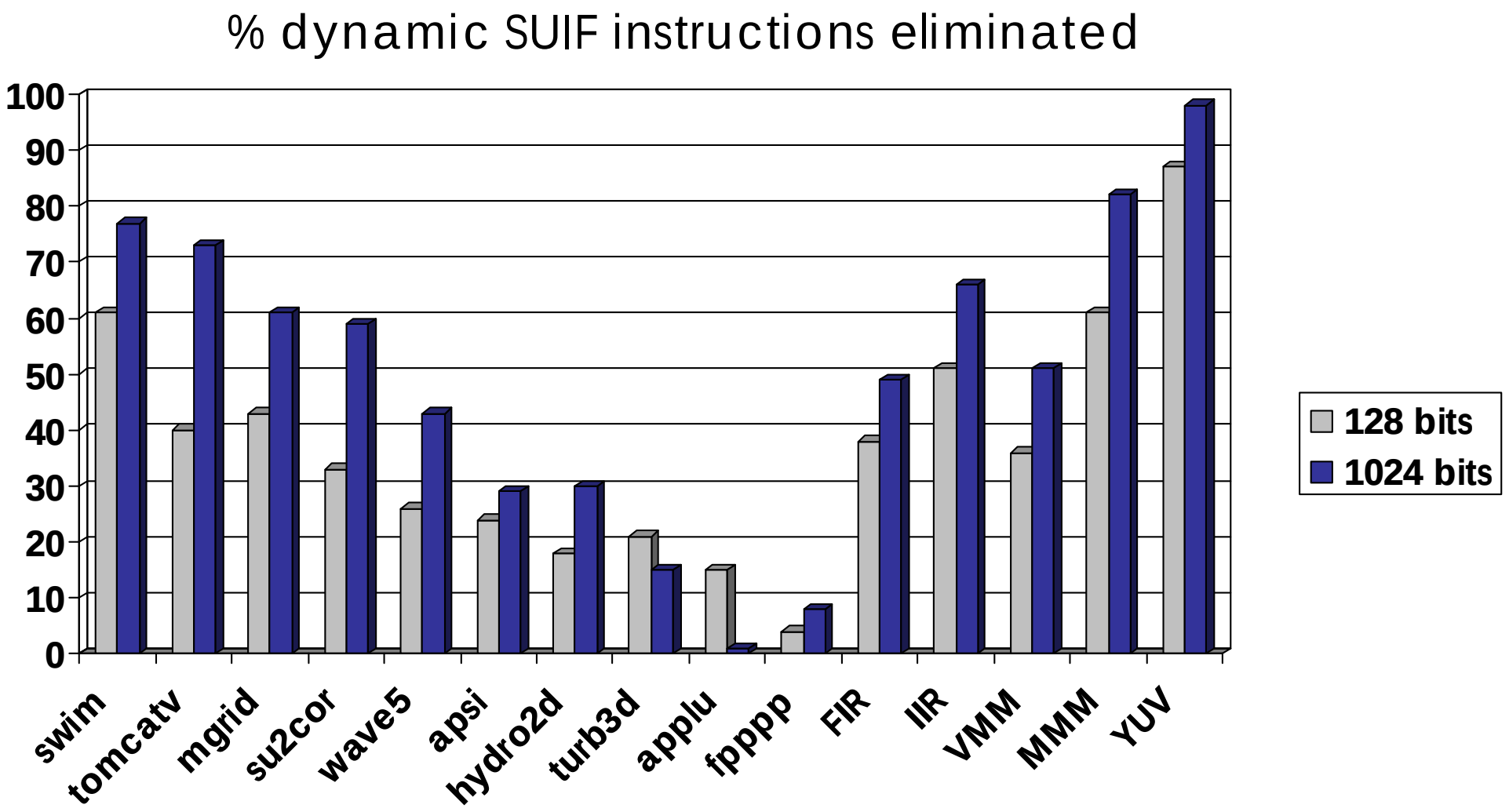
SLP Extraction Algorithm

- Follow operand use

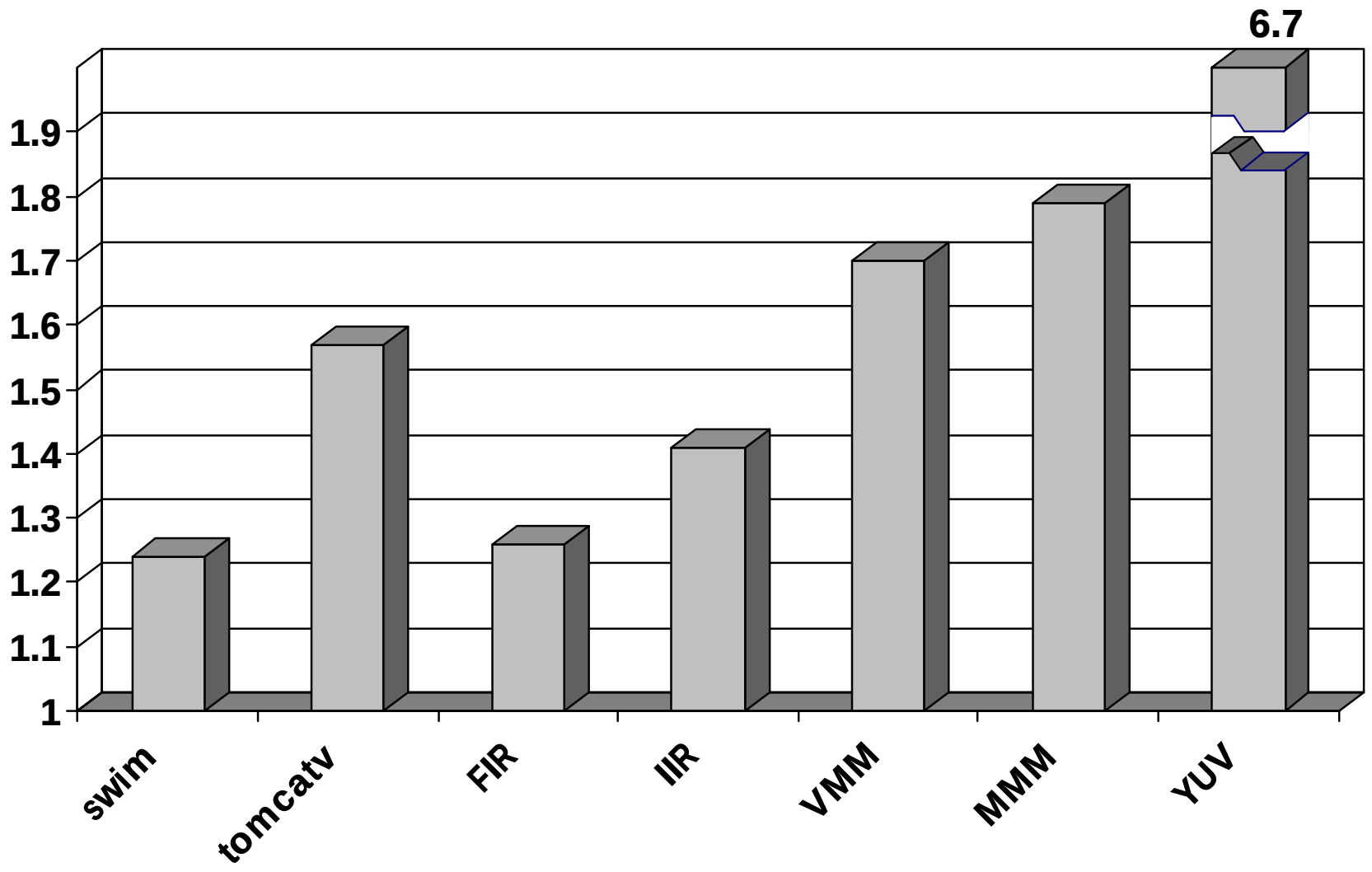
```
A = X[i+0]
C = E * 3
B = X[i+1]
H = C – A
D = F * 5
J = D - B
```



SLP Availability



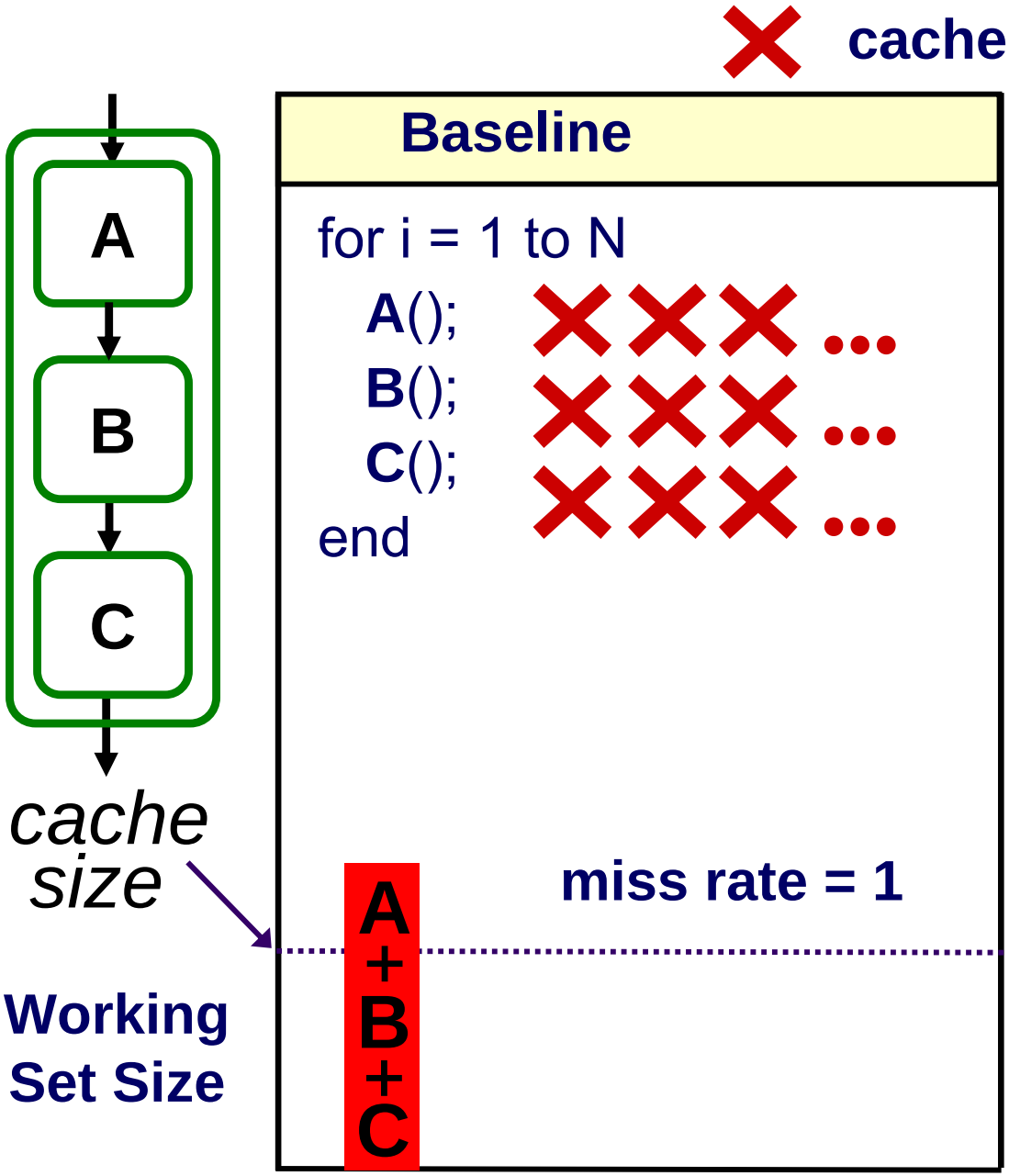
Speedup on AltiVec



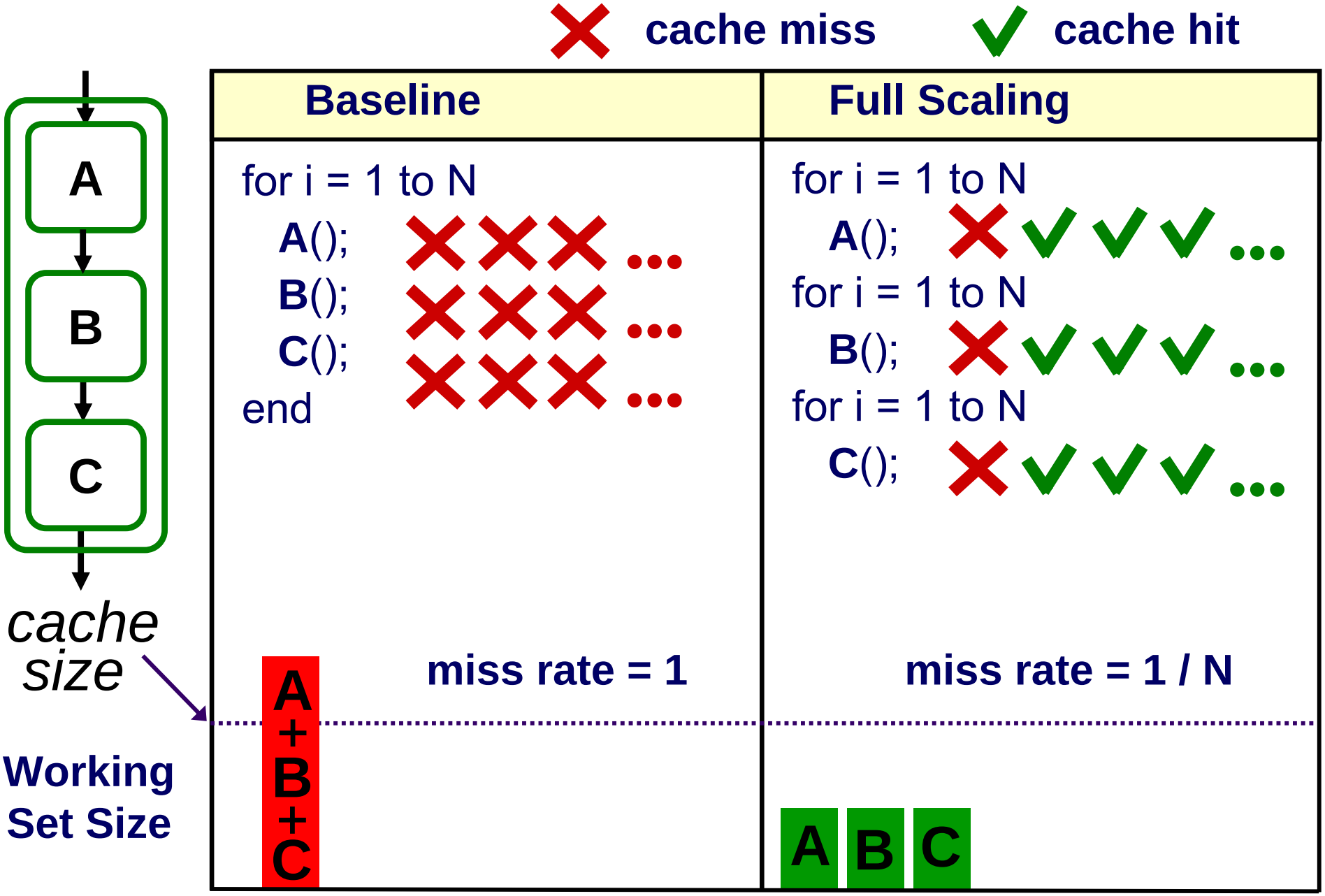
Performance in Uniprocessors
time = compute + wait

- Instruction level parallelism
  - Multiple functional units, deeply pipelined, speculation, ...
- Data level parallelism
  - SIMD: short vector instructions (multimedia extensions)
    - Hardware is simpler, no heavily ported register files
    - Instructions are more compact
    - Reduces instruction fetch bandwidth
- Complex memory hierarchies
  - Multiple level caches, may outstanding misses, prefetching, …
  - Exploiting locality is essential

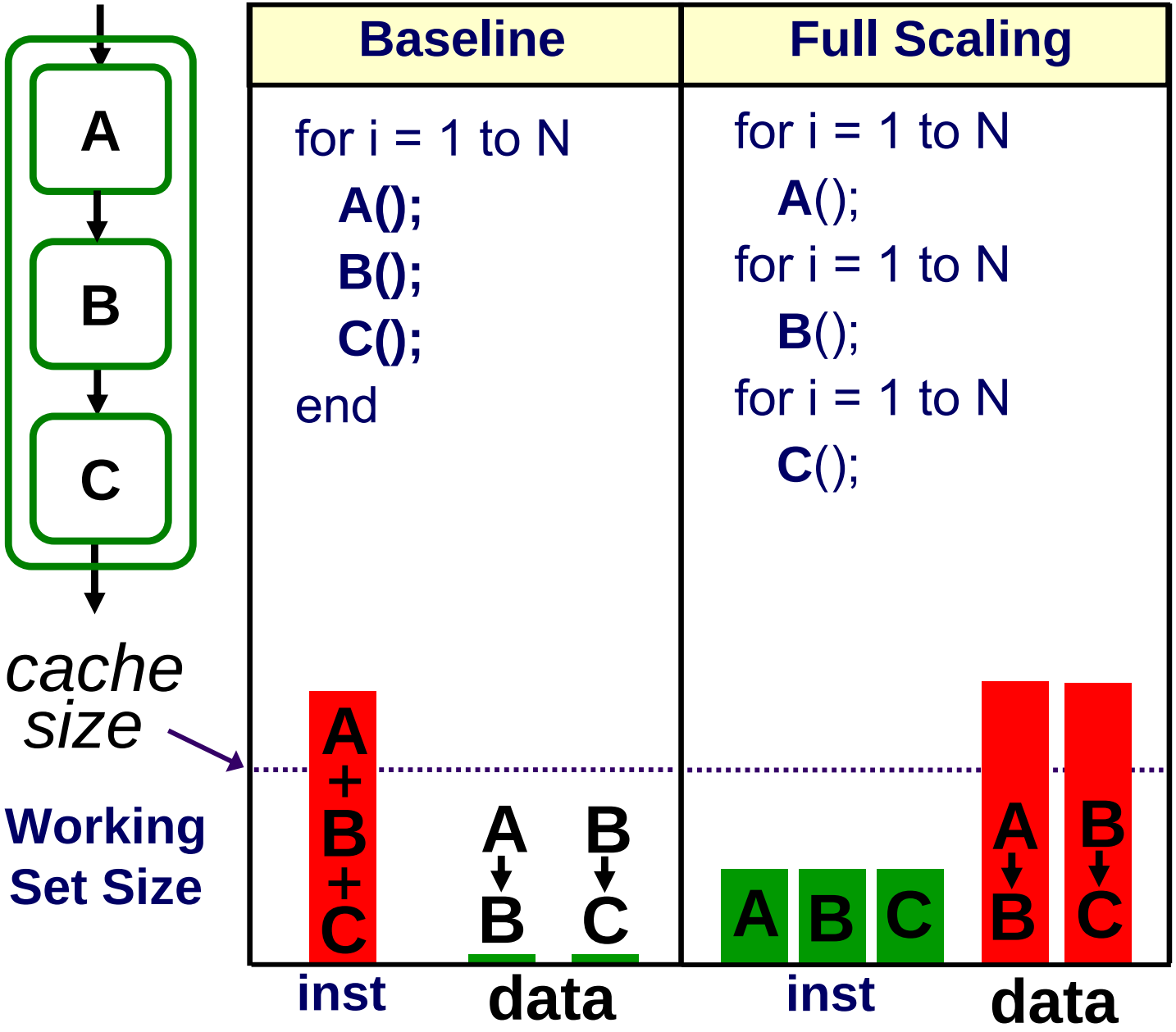
Instruction Locality



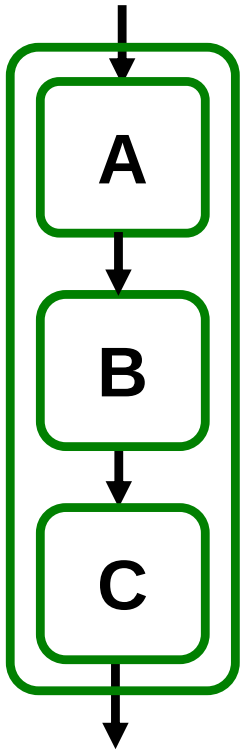
Instruction Locality



Example Memory (Cache) Optimization



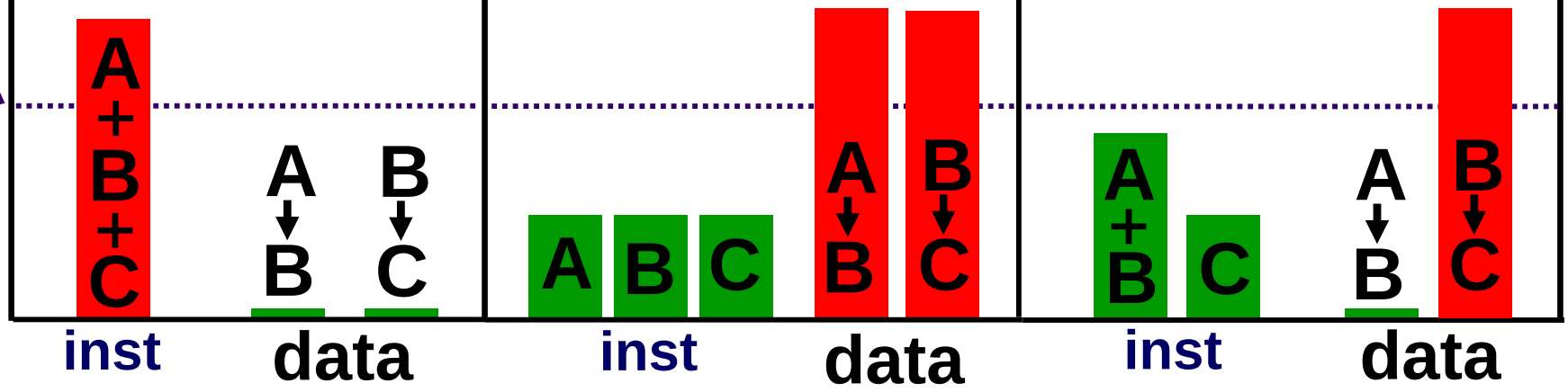
Example Memory (Cache) Optimization



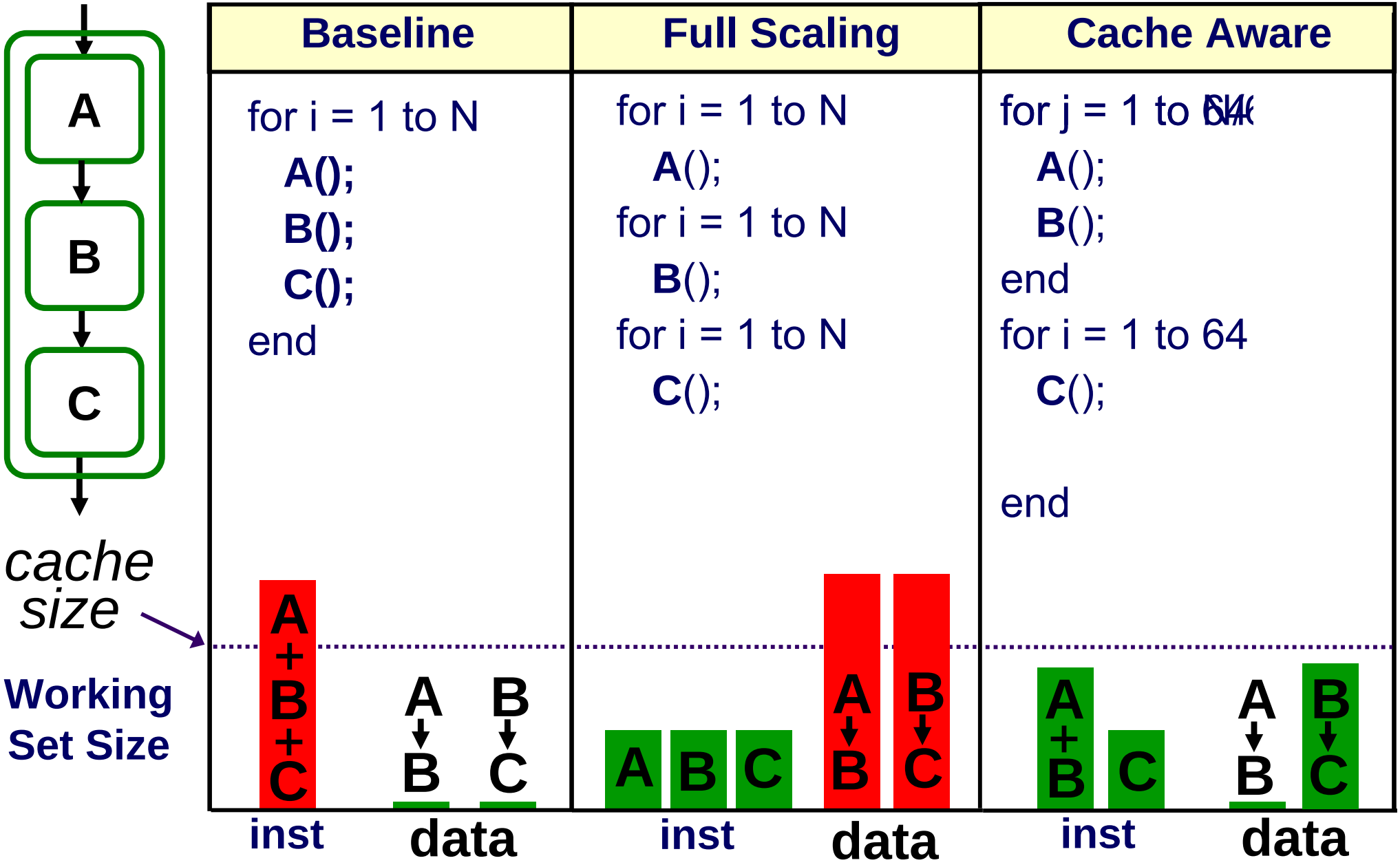
cache size

Working Set Size

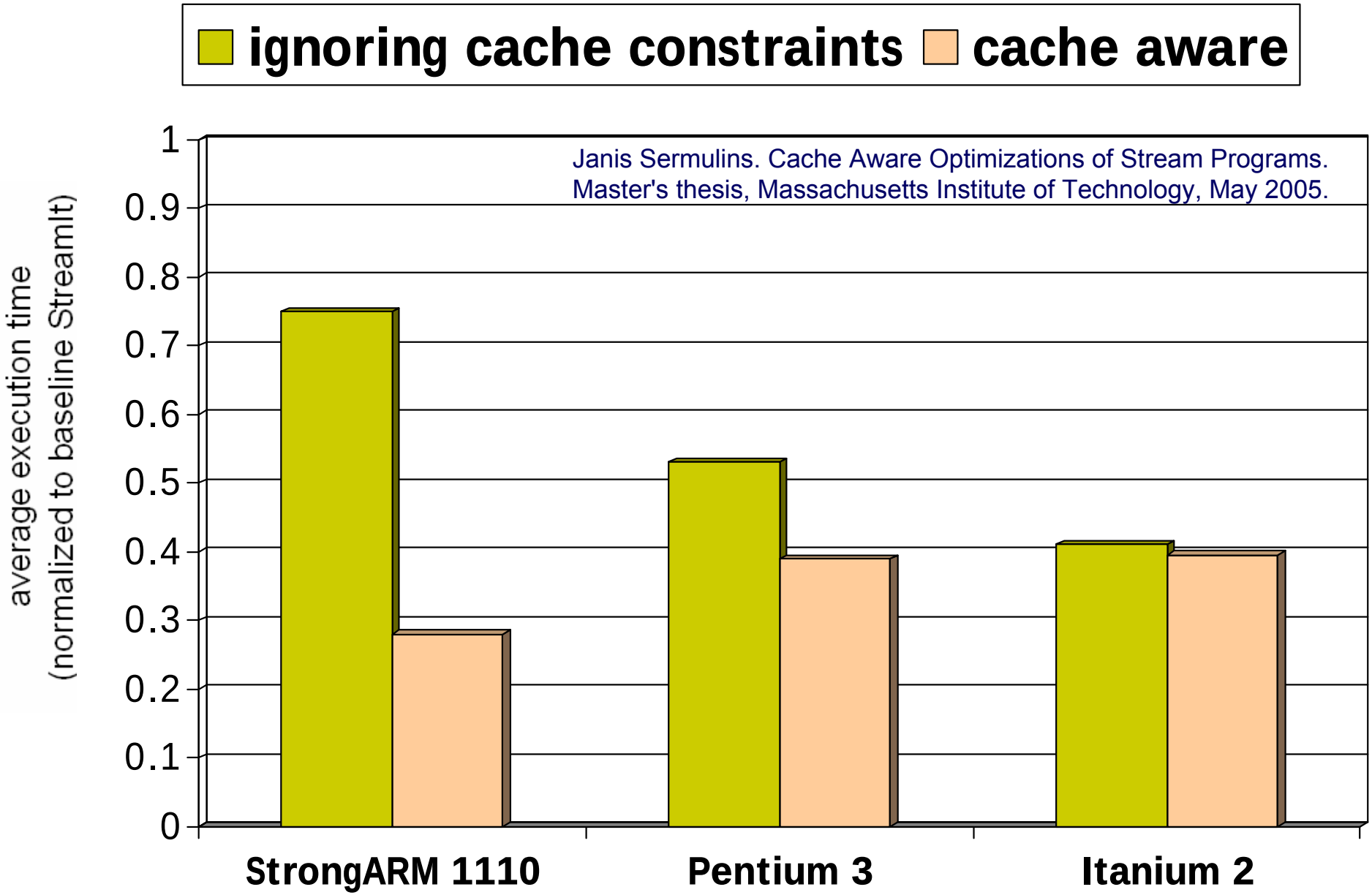
Baseline	Full Scaling	
for i = 1 to N **A();** **B();** **C();** end	for i = 1 to N **A**(); for i = 1 to N **B**(); for i = 1 to N **C**();	for i = 1 to N **A**(); **B**(); end for i = 1 to N **C**();



Example Memory (Cache) Optimization



Results from Cache Optimizations



6.189 IAP 2007

Summary

Programming for Performance

- Tune the parallelism first
- Then tune performance on individual processors
  - Modern processors are complex
  - Need instruction level parallelism for performance
  - Understanding performance requires a lot of probing
- Optimize for the memory hierarchy
  - Memory is much slower than processors
  - Multi-layer memory hierarchies try to hide the speed gap
  - Data locality is essential for performance

Programming for Performance

- May have to change everything!
  - Algorithms, data structures, program structure
- Focus on the biggest performance impediments
  - Too many issues to study everything
  - Remember the law of diminishing returns