Autotuning

John Cavazos

University of Delaware

What is Autotuning?

- Searching for the “best” code parameters, code transformations, system configuration settings, etc.
- Search can be
    - Quasi-intelligent: genetic algorithms, hill-climbing
    - Random (often quite good!)

Parameters to tune in all of these

application

compiler

runtime system

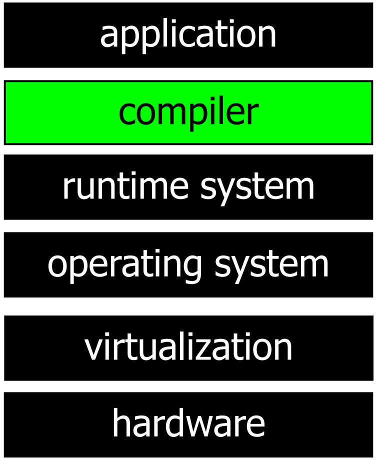
operating system

virtualization

hardware

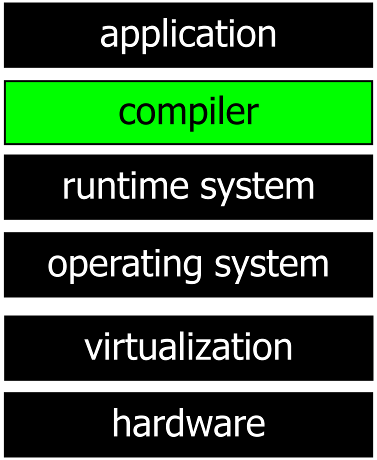
Traditional Compilers

- "One size fits all" approach
- Tuned for average performance
- Aggressive opts often turned **off**
- Target hard to model analytically



Solution : Random Search!

- Identify large set of optimizations to search over
  - Some optimizations require parameter values, search over those values also!
  - Out-performs state-of-the-art compiler

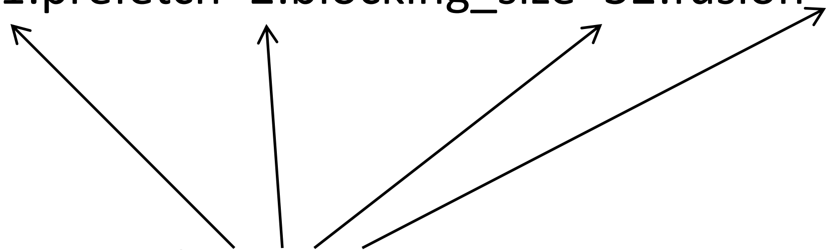

application
compiler
runtime system
operating system
virtualization
hardware

Optimization Sequence Representation

- Use random number generator to construct sequence

Example:

-LNO:interchange=1:prefetch=2:blocking_size=32:fusion=1:…



Generate each of these parameter values
using a random number generator
Note: need to define a range of interesting values a-priori

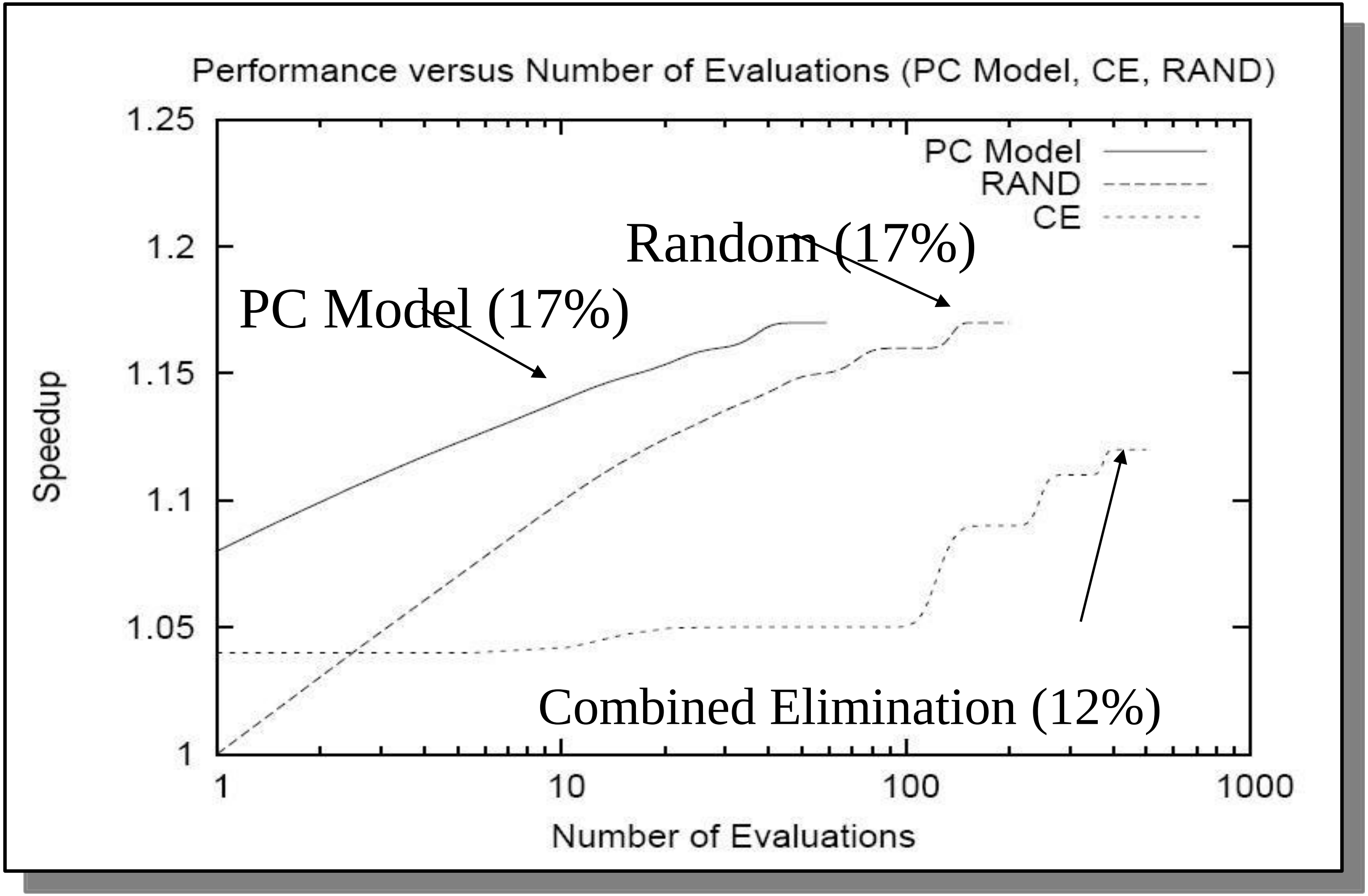
Case Study: Random vs State-of-the-Art

- PathScale compiler
  - Compare to highest optimization level
  - 121 compiler flags
- AMD Athlon processor
  - ***Real*** machine; Not simulation
- 57 benchmarks
  - SPEC (INT 95, INT/FP 2000), MiBench, Polyhedral

Evaluated Search Strategies

- RAND
  - Randomly select 500 optimization sequences
- Combined Elimination [CGO 2006]
  - Pure search technique
    - Evaluate optimizations one at a time
    - Eliminate negative optimizations in one go
  - Out-performed other pure search techniques
- PC Model [CGO 2007]
  - Machine learning model using performance counters
  - Mapping of performance counters to good optimizations

Performance vs Evaluations



PC Model (17%)

Random (17%)

Combined Elimination (12%)

Some recommendations

- Use small input sizes that are representative
  - Be careful as tuning on small inputs may not give you the best performance on regular (larger) inputs
- Reduce application to most important kernel(s) and tune those
- If kernels can be mapped to highly-tuned library implementations, use those!

Some recommendations

- No optimization search will help a bad algorithm!
  - **Chose the correct algorithm first!**

Using quasi-intelligent search

- Can easily setup a genetic algorithm or hill-climbing to perform search over optimization space
- We can help you set this up.
- Random often does as well!

Performance counters

- Can be used to narrow search to particular set of optimizations
- Lots of cache misses may require loop restructuring, e.g., blocking
- Lots of resource stalls may require instruction scheduling

Most Informative Perf Counters

1. L1 Cache Accesses
2. L1 Dcache Hits
3. TLB Data Misses
4. Branch Instructions
5. Resource Stalls
6. Total Cycles
7. L2 Icache Hits
8. Vector Instructions
9. L2 Dcache Hits
10. L2 Cache Accesses
11. L1 Dcache Accesses
12. Hardware Interrupts
13. L2 Cache Hits
14. L1 Cache Hits
15. Branch Misses

Dependence Analysis and *Loop Transformations*

John Cavazos

University of Delaware

Lecture Overview

- Very Brief Introduction to Dependences
- Loop Transformations

The Big Picture

What are our goals?

- Simple Goal: Make execution time as small as possible

Which leads to:

- Achieve execution of many (all, in the best case) instructions in parallel
- Find **independent** instructions

Dependences

- We will concentrate on data dependences
- Simple example of data dependence:

S_1 `PI = 3.14`

S_2 `R = 5.0`

S_3 `AREA = PI * R ** 2`

- Statement S_3 cannot be moved before either S_1 or S_2 without compromising correct results

Dependences

- Formally:

There is a data dependence from statement S_1 to statement S_2 (S_2 depends on S_1) if:

1. Both statements access the same memory location and at least one of them stores onto it, and
2. There is a feasible run-time execution path from S_1 to S_2

Load Store Classification

- Quick review of dependences classified in terms of load-store order:
  1. True dependence (RAW hazard)
  2. Antidependence (WAR hazard)
  3. Output dependence (WAW hazard)

Dependence in Loops

- Let us look at two different loops:

```
   DO I = 1, N
S1   A(I+1) = A(I)+ B(I)
   ENDDO
```

```
   DO I = 1, N
S1   A(I+2) = A(I)+B(I)
   ENDDO
```

- In both cases, statement S_1 depends on itself

Transformations

- We call a transformation safe if the transformed program has the same "meaning" as the original program

- But, what is the "meaning" of a program?

For our purposes:

- Two computations are equivalent if, on the same inputs:
    - They produce the same outputs in the same order

Reordering Transformations

- Is any program transformation that changes the order of execution of the code, without adding or deleting any executions of any statements

Properties of Reordering Transformations

- A reordering transformation does not eliminate dependences
- However, it can change the ordering of the dependence which will lead to incorrect behavior
- A reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink of that dependence.

Loop Transformations

- Compilers have always focused on loops
  - Higher execution counts
  - Repeated, related operations
- Much of real work takes place in loops

Several effects to attack

- Overhead
  - Decrease control-structure cost per iteration
- Locality
  - Spatial locality ⇒ use of co-resident data
  - Temporal locality ⇒ reuse of same data
- Parallelism
  - Execute independent iterations of loop in parallel

Eliminating Overhead

Loop unrolling (the oldest trick in the book)

- To reduce overhead, replicate the loop body

```
do i = 1 to 100 by 1
   a(i) = a(i) + b(i)
end
```

becomes

(unroll by 4)

```
do i = 1 to 100 by 4
   a(i)    = a(i) + b(i)
   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)
end
```

Sources of Improvement

- Less overhead per useful operation
- Longer basic blocks for local optimization

Eliminating Overhead

Loop unrolling with unknown bounds

- Generate guard loops

```
do i = 1 to n by 1
   a(i) = a(i) + b(i)
   end
```

becomes

(unroll by 4)

```
i = 1
do while (i+3 < n)
   a(i)   = a(i) + b(i)
   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)
   i = i + 4
   end

do while (i < n)
    a(i) = a(i) + b(i)
    i  = i + 1
    end
```

Eliminating Overhead

One other use for loop unrolling

- Eliminate copies at the end of a loop

```
$t_1 = b(0)$
do i = 1 to 100
    $t_2 = b(i)$
    $a(i) = a(i) + t_1 + t_2$
    $t_1 = t_2$
end
```

becomes

(unroll + rename)

```
$t_1 = b(0)$
do i = 1 to 100 by 2
    $t_2 = b(i)$
    $a(i) = a(i) + t_1 + t_2$
    $t_1 = b(i+1)$
    $a(i+1) = a(i+1) + t_2 + t_1$
end
```

Loop Unswitching

- Hoist invariant control-flow out of loop nest
- Replicate the loop & specialize it
- No tests, branches in loop body
- Longer segments of straight-line code

Loop Unswitching

```
loop
    statements
    if test then
        then part
    else
        else part
    endif
    more statements
endloop
```

becomes

(unswitch)

```
If test then
    loop
        statements
        then part
        more statements
    endloop
else
    loop
        statements
        else part
        more statements
    endloop
endif              *
```

Loop Unswitching

```
do i = 1 to 100
    a(i) = a(i) + b(i)
    if (expression) then
        d(i) = 0
end
```

becomes

(unswitch)

```
if (expression) then
    do i = 1 to 100
        a(i) = a(i) + b(i)
        d(i) = 0
    end
else
    do i = 1 to 100
        a(i) = a(i) + b(i)
    end
```

Loop Fusion

- Two loops over same iteration space ⇒ one loop
- Safe if does not change the values used or defined by any statement in either loop (i.e., does not violate deps)

```
do i = 1 to n
    c(i) = a(i) + b(i)
end

do j = 1 to n
    d(j) = a(j) * e(j)
end
```

becomes

(fuse)

```
do i = 1 to n
    c(i) = a(i) + b(i)
    d(i) = a(i) * e(i)
end
```

For big arrays, a(i) may not be in the cache

a(i) will be found in the cache

Loop Fusion Advantages

- Enhance temporal locality
- Reduce control overhead
- Longer blocks for local optimization & scheduling
- Can convert inter-loop reuse to intra-loop reuse

Loop Fusion of Parallel Loops

- Parallel loop fusion legal if dependences loop independent
  - Source and target of flow dependence map to same loop iteration

Loop distribution (fission)

- Single loop with independent statements ⇒ multiple loops
- Starts by constructing statement level dependence graph
- Safe to perform distribution if:
    - No cycles in the dependence graph
    - Statements forming cycle in dependence graph put in same loop

Loop distribution (fission)

Reads b, c, e, f, h, & k

Writes a, d, & g

```
do i = 1 to n
    a(i) = b(i) + c(i)
    d(i) = e(i) * f(i)
    g(i) = h(i) - k(i)
    end
```

becomes

(fission)

```
do i = 1 to n
    a(i) = b(i) + c(i)
    end
```

Reads b & c

Writes a

```
do i = 1 to n
    d(i) = e(i) * f(i)
    end
```

Reads e & f

Writes d

```
do i = 1 to n
    g(i) = h(i) - k(i)
    end
```

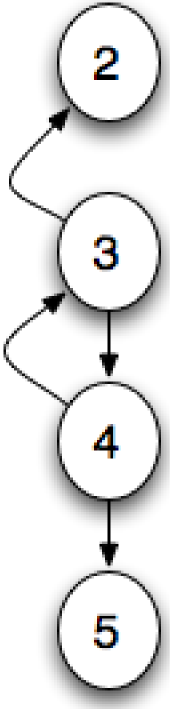
Reads h & k

Writes g

Loop distribution (fission)

```
(1) for I = 1 to N do
(2)    A[I] = A[i] + B[i-1]
(3)    B[I] = C[I-1]*X+C
(4)    C[I] = 1/B[I]
(5)    D[I] = sqrt(C[I])
(6) endfor
```

Has the following dependence graph



Loop distribution (fission)

```
for I = 1 to N do
    A[I] = A[i] + B[i-1]
    B[I] = C[I-1]*X+C
    C[I] = 1/B[I]
    D[I] = sqrt(C[I])
endfor
```

becomes

(fission)

```
for I = 1 to N do
    A[I] = A[i] + B[i-1]
endfor
for
    B[I] = C[I-1]*X+C
    C[I] = 1/B[I]
endfor
for
    D[I] = sqrt(C[I])
endfor
```

Loop Fission Advantages

- Enables other transformations
  - E.g., Vectorization
- Resulting loops have smaller cache footprints
  - More reuse hits in the cache

Loop Interchange

```
do i = 1 to 50
   do j = 1 to 100
      a(i,j) = b(i,j) * c(i,j)
    end
 end
```

becomes

(interchange)

```
do j = 1 to 100
    do i = 1 to 50
       a(i,j) = b(i,j) * c(i,j)
    end
 end
```

- Swap inner & outer loops to rearrange iteration space

Effect

- Improves reuse by using more elements per cache line
- Goal is to get as much reuse into inner loop as possible

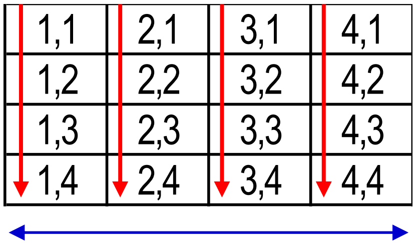
Loop Interchange Effect

- If one loop carries all dependence relations
  - Swap to outermost loop and all inner loops executed in parallel
- If outer loops iterates many times and inner only a few
  - Swap outer and inner loops to reduce startup overhead
- Improves reuse by using more elements per cache line
- Goal is to get as much reuse into inner loop as possible

Reordering Loops for Locality

In row-major order, the opposite loop ordering causes the same effects

In Fortran's column-major order, a(4,4) would lay out as

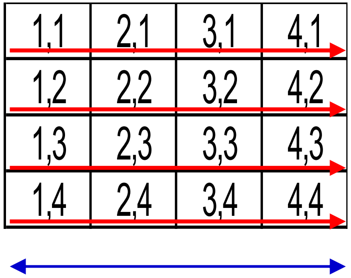


1,1	2,1	3,1	4,1
1,2	2,2	3,2	4,2
1,3	2,3	3,3	4,3
1,4	2,4	3,4	4,4

cache line

As little as 1 used element per line

After interchange, direction of Iteration is changed



1,1	2,1	3,1	4,1
1,2	2,2	3,2	4,2
1,3	2,3	3,3	4,3
1,4	2,4	3,4	4,4

cache line

Runs down cache line

Loop permutation

- Interchange is degenerate case
    - Two perfectly nested loops
- More general problem is called permutation

Safety

- Permutation is safe iff no data dependences are reversed
    - The flow of data from definitions to uses is preserved

Loop Permutation Effects

- Change order of access & order of computation
- Move accesses closer in time ⇒ increase temporal locality
- Move computations farther apart ⇒ cover pipeline latencies

Strip Mining

- Splits a loop into two loops

```
do j = 1 to 100
   do i = 1 to 50
      a(i,j) = b(i,j) *
c(i,j)
   endend
```

becomes

(strip mine)

```
do j = 1 to 100
  do ii = 1 to 50 by 8
     do i = ii to min(ii+7,50)
         a(i,j) = b(i,j) * c(i,j)
     end
  end
end
```

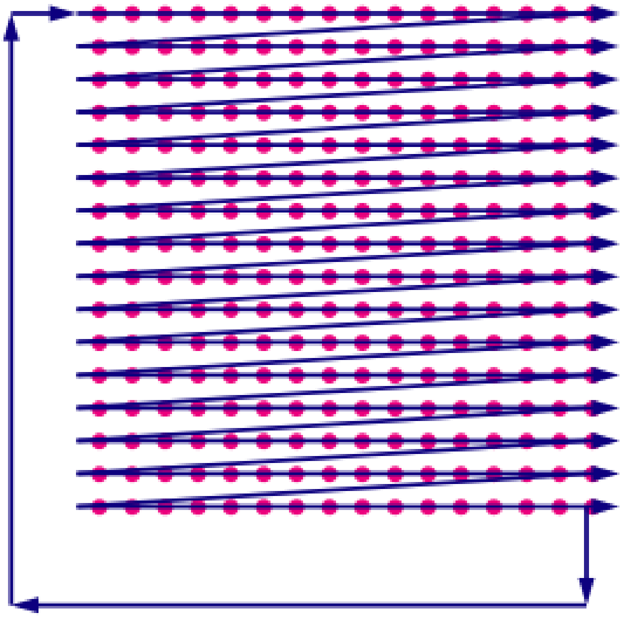
Note: This is always safe, but used by itself not profitable!

Strip Mining Effects

- May slow down the code (extra loop)
- Enables vectorization

Loop Tiling (blocking)

```
do t = 1,T
  do i = 1,n
    do j = 1,n
       … a(i,j) …
    end do
  end do
end do
```

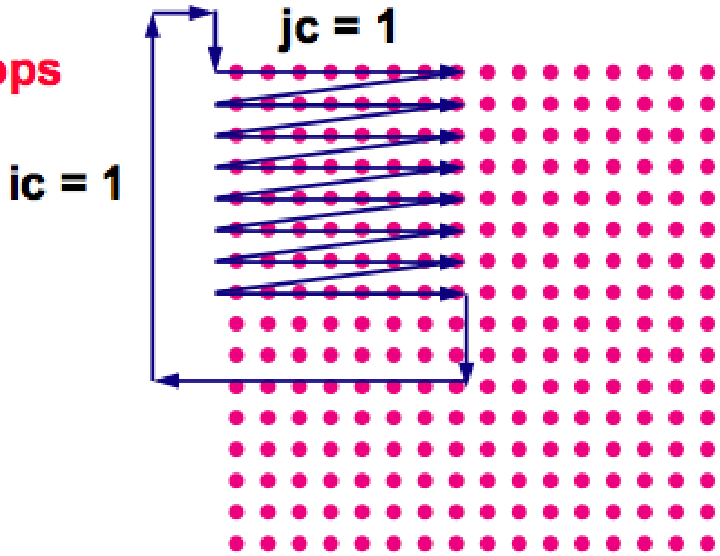


Want to exploit temporal locality
in loop nest.

Loop Tiling (blocking)

```
do ic = 1, n, B
 do jc = 1, n, B
  do t = 1,T
   do i = ic, min(n,ic+B-1), 1
    do j = jc, min(n, jc+B-1), 1
     … a(i,j) …
    end do
   end do
  end do
 end do
end do
```

control loops

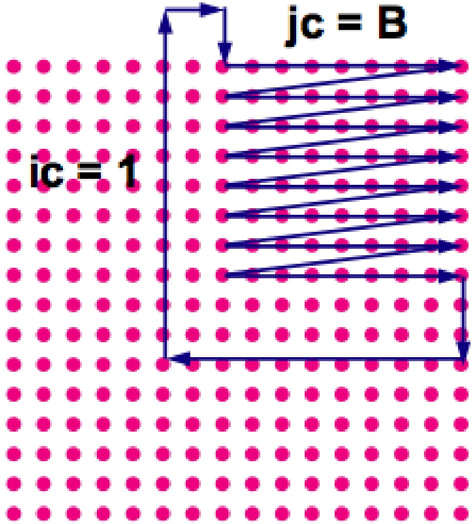


B: Block Size

Loop Tiling (blocking)

control loops

```
do ic = 1, n, B
  do jc = 1, n, B
    do t = 1,T
      do i = ic, min(n,ic+B-1), 1
        do j = jc, min(n, jc+B-1), 1
          … a(i,j) …
        end do
      end do
    end do
  end do
end do
```

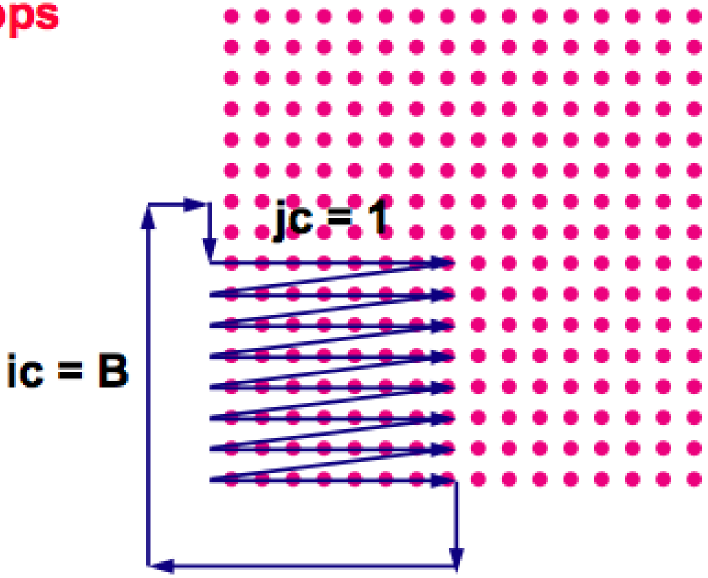


B: Block Size

Loop Tiling (blocking)

control loops

```
do ic = 1, n, B
  do jc = 1, n, B
    do t = 1,T
      do i = ic, min(n,ic+B-1), 1
        do j = jc, min(n, jc+B-1), 1
          … a(i,j) …
        end do
      end do
    end do
  end do
end do
```

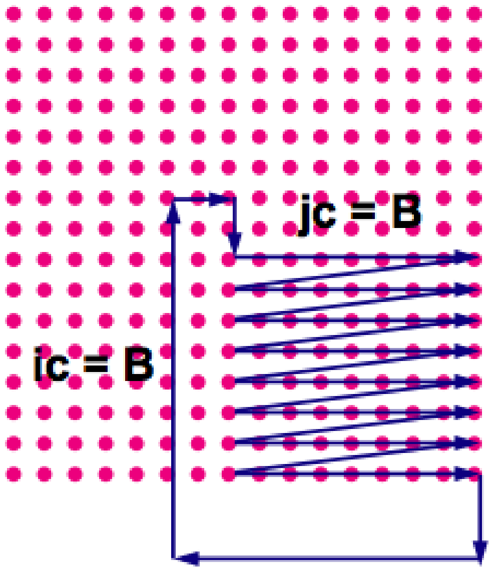


B: Block Size

Loop Tiling (blocking)

```
do ic = 1, n, B
  do jc = 1, n, B
    do t = 1,T
      do i = ic, min(n,ic+B-1), 1
        do j = jc, min(n, jc+B-1), 1
          … a(i,j) …
        end do
      end do
    end do
  end do
end do
```

control loops



B: Block Size

When is this legal?

Loop Tiling Effects

- Reduces volume of data between reuses
  - Works on one “tile” at a time (*tile size is* B *by* B)
- Choice of tile size is crucial

Scalar Replacement

- Allocators never keep c(i) in a register
- We can trick the allocator by rewriting the references

The plan

- Locate patterns of consistent reuse
- Make loads and stores use temporary scalar variable
- Replace references with temporary's name

Scalar Replacement

```
do i = 1 to n
    do j = 1 to n
       a(i) = a(i) + b(j)
    end
end
```

becomes

(scalar replacement)

```
do i = 1 to n
    t = a(i)
    do j = 1 to n
         t =  t + b(j)
    end
    a(i) = t
end
```

Almost any register allocator can get `t` into a register

Scalar Replacement Effects

- Decreases number of loads and stores
- Keeps reused values in names that can be allocated to registers
- In essence, this exposes the reuse of a(i) to subsequent passes