COMP 515: Advanced Compilation for Vector and Parallel Processors

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP515

Managing Cache (Recap)

Allen and Kennedy, Chapter 9

Spatial Reuse

- Permits high reuse when accessing closely located data
-
```
DO I = 1, M
   DO J = 1, N
      A(I, J) = A(I, J) + B(I, J)
   ENDDO
ENDDO
```

No reuse/locality for Fortran's column-major layout

Spatial Reuse (after loop interchange)

- ```
  DO J = 1, N
     DO I = 1, M
        A(I, J) = A(I, J) + B(I, J)
     ENDDO
  ENDDO
  ```

  Iterates over columns instead
  ```

Temporal Reuse

- Reuse limited by cache size, LRU replacement strategy
-
```
DO I = 1, M
   DO J = 1, N
      A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

Temporal Reuse

- Strip mining + Interchange (or Tiling) can improve temporal reuse when tile size S is chosen so that inner loops can fit in cache

- ```
  DO J = 1, N, S
     DO I = 1, M
        DO jj = J, MIN(N, J+S-1)
           A(I) = A(I) + B(jj)
        ENDDO
     ENDDO
  ENDDO
  ```
  ```

Selection of Transformations

- Open problem: how to select best combination of transformations to optimize locality?

- Today's lecture: Automatic Selection of High-Order Transformations in the IBM XL Fortran Compiler

- Thursday's lecture: Polyhedral Compilation Framework (guest lecturer: Uday Reddy)

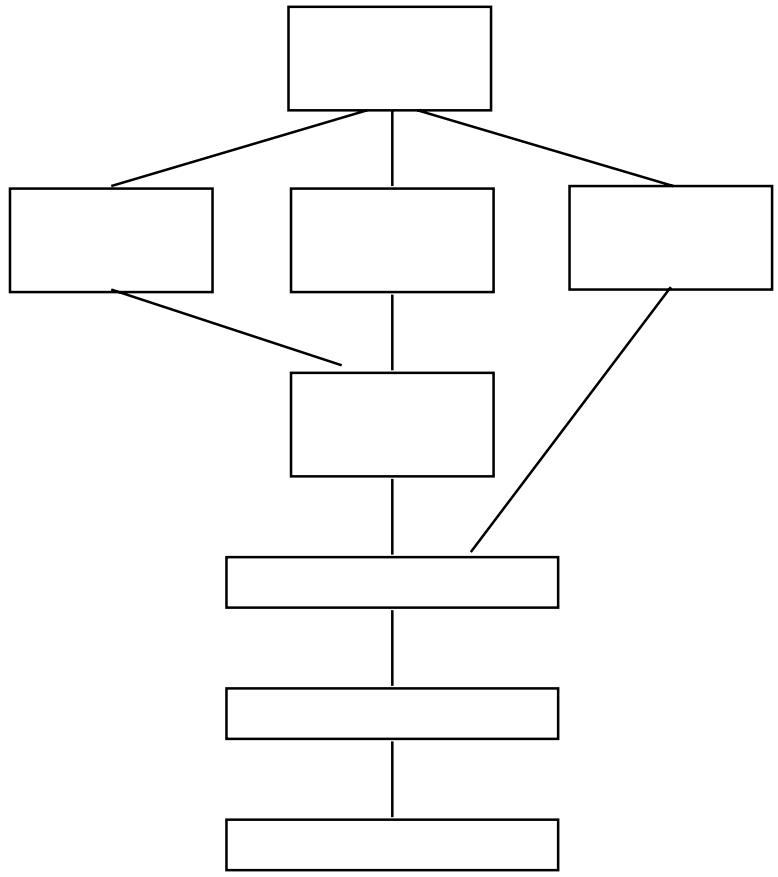
Acknowledgments

- Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. Vivek Sarkar. IBM Journal of Research and Development, 41(3), May 1997.
  - Special thanks to Ray Ellersick, Roy Ju, Paula Newman, John Ng, Khoa Nguyen, Jin-Fan Shaw, Radhika Thekkath and other contributors to the design and implementation of the ASTI transformer at IBM Santa Teresa Laboratory.
- POPL 1996 tutorial on "Code Optimization in Modern Compilers", K.V. Palem, V. Sarkar, Jan 1996

Memory Hierarchies in Computer Systems

Ideally one would desire an indefinitely large memory capacity ... We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

– A. W. Burks, H. H. Goldstine, and J. von Neumann (1946).



Principle of Locality

Programs tend to reuse data and instructions they have used recently:

1. *temporal locality* — if an item is referenced, it will tend to be referenced again soon
2. *spatial locality* — if an item is referenced, nearby items will tend to be referenced soon

90/10 rule for instruction locality — an average program spends 90% of its execution in only 10% of the code

Caches

- *cache* = level of memory hierarchy between CPU and main memory
- *block* = unit of data that can be stored in cache (also called a cache *line*)
- 2^n = number of blocks in cache
- 2^d = degree of associativity = number of blocks in a set
- 2^s = number of sets in cache (note that $2^s \times 2^d = 2^n$)

  $d = 0, s = n \Rightarrow$ cache is *direct mapped*

  $d = n, s = 0 \Rightarrow$ cache is *fully associative*
- 2^b = number of words (data elements) in a block

How do caches work?

- *hit* = memory access that is found in cache
- *miss* = memory access that is not found in cache
- *replacement policy* = strategy for determining which (valid) block in the set should be replaced

Sources of Cache Misses

Intuitively, a cache miss can be classified as follows [Hill '87]:

1. *compulsory miss* = first access to a block during program execution
2. *capacity miss* = subsequent access to a block, after the block had been replaced due to cache size limitation (can be avoided by increasing number of sets)
3. *collision miss* = subsequent access to a block, after the block had been replaced due to set size limitation (can be avoided by increasing degree of associativity)

An optimizing compiler can reduce all three kinds of cache misses!

Caching the Page Table: Translation-Lookaside Buffers (TLBs)

- Page table for virtual→real address translation is stored in main memory
- TLB is like a cache for the page table:
  - hit = virtual→real address translation for memory access was found in TLB
  - miss = translation was not found in TLB, and had to be retrieved from page table
- TLB miss penalties are larger than cache miss penalities
  ⇒ it is important to take TLB into account in locality optimizations performed by the compiler

Goals of Compile-time Cache Usage Estimation

- Estimate cache effectiveness so as to guide compile-time selection of program transformations
- Consider realistic cache models: block size > 1, set associativity
- Solution should be efficient: estimation time should be independent of number of loop iterations, array dimension sizes, cache size

Exact solution is too hard
$\Rightarrow$ seek good approximations/bounds

Identifying Loops that carry Cache Block Reuse

Two approaches:

1. *Count number of blocks accessed*
   [Ferrante, Sarkar, Thrash '91] :
   Let $DB(L, n)$ = number of distinct cache blocks accessed by n consecutive iterations of loop L
   Loop L carries cache block reuse if $DB(L, n) < n \times DB(L, 1)$ for some $1 < n \leq$ # iterations of L

2. *Compute reuse-vectors* [Wolf, Lam '91]:
   Loop L carries reuse if its basis vector $\vec{r}$ is included in the reuse vector space for the loop body

In this tutorial, we will focus on approach # 1.

General Approach for Compile-time Cache Usage Estimation

- Estimate # *distinct words* (DW) accessed by a *single* array reference
- Estimate # *distinct blocks* (DB) accessed by a *single* array reference
- Estimate # *distinct words* (DW) accessed by *multiple* array references
- Estimate # *distinct blocks* (DB) accessed by *multiple* array references
- Adjust estimates to account for collision misses that occur due to *limited associativity*

Assumptions

- Loops are normalized to step $= +1$; we define LB_i and UB_i to be the lower and upper bound expressions of loop i
- Array subscript expressions are affine functions of loop index variables
- Only consider data accesses (for these cost functions)

Estimating DW for a single array reference

Sometimes it is obvious:

```
   DO 10 i = 1, 100
10   A(i) = A(i) + 5   ==>  DW = 100
```

Sometimes it is not so obvious:

```
   DO 10 i = 1, 8
     DO 10 j = 1, 5
10     A(6*i+9*j-7) = 5  ==>  DW = 25
```

e.g. iteration ($i = 1, j = 3$) and ($i = 4, j = 1$) both access the same word, `A(26)`

Estimating DW for a single unidimensional array reference

Upper bound analysis:

- Consider array reference $A(f(i_1, \ldots, i_h))$ in loops $i_1, \ldots, i_h$, such that $f(i_1, \ldots, i_h) = a_0 + \sum_{k=1}^{h} a_k i_k$
- Compute f^{lo} and f^{hi}, lower and upper bounds for f, using Banerjee's inequality [Banerjee '88] (for example)
- Compute $g = gcd(|a_1|, \ldots, |a_h|)$

$$\boxed{\Rightarrow DW(f) \leq \frac{(f^{hi} - f^{lo})}{g} + 1}$$

Proof: values taken on by $f()$ must be a subset of $\{f^{lo}, f^{lo} + g, f^{lo} + 2 \times g, \ldots, f^{hi}\}$

Upper bound for not-so-obvious example

```
   DO 10 i = 1, 8
     DO 10 j = 1, 5
10     A(6*i+9*j-7) = 5
```

$$f = 6i + 9j - 7$$

$$\Rightarrow \quad f^{lo} = 8, \quad f^{hi} = 86 \quad , g = gcd(6, 9) = 3$$

$$\Rightarrow \quad DW(f) \leq \frac{(86-8)}{3} + 1 = 27$$

Estimating DW for a single multidimensional array reference

Consider array reference $A(f_1, \ldots, f_m)$

$$DW(f_1, \ldots, f_m) \leq \prod_{j=1}^{m} DW(f_j) \leq \prod_{j=1}^{m} \left(\frac{(f_j^{hi} - f_j^{lo})}{g_j} + 1 \right)$$

NOTE: above bound is too conservative for coupled subscripts
e.g. for `A(i,i)`, we get $DW \leq (UB_i - LB_i + 1) \times (UB_i - LB_i + 1)$

SOLUTION: linearize subscript expressions for all coupled dimensions
e.g. if `A` has shape `A(100,100)`, linearize `A(i,i)` to obtain `*(addr(A) + 101*i)`, which results in the bound
$DW \leq (UB_i - LB_i + 1)$

Estimating DB for a single array reference, A(f)

1. **Sparse stride bound:**

$$DB(f) \leq DW(f)$$

Example: $f(i) = 100i \Rightarrow DW(f) \leq (UB_i - LB_i + 1)$

2. **Dense stride bound:**

$$DB(f) \leq \left\lceil \frac{(f^{hi} - f^{lo})}{2^b} \right\rceil + 1$$

Example: $f(i) = 2i \Rightarrow DW(f) \leq \left\lceil \frac{(2*UB_i - 2*LB_i)}{2^b} \right\rceil + 1$

Putting both upper bounds together yields

$$\begin{aligned} DB(f) &\leq \min\left(DW(f), \left\lceil \frac{(f^{hi} - f^{lo})}{2^b} \right\rceil + 1\right) \\ &\leq \min\left(\frac{(f^{hi} - f^{lo})}{g} + 1, \left\lceil \frac{(f^{hi} - f^{lo})}{2^b} \right\rceil + 1\right) \end{aligned}$$

Estimating DB for a single multidimensional array reference (Example)

Assume block size $2^b = 16$ words:

```
real*8 c(201,301)
DO j = 1, 100
  DO i = 1, 100
    c(2*i+1,3*j-2) ...
  ENDDDO
ENDDO
```

$$DB(2i+1) \leq$$
$$\min(100, \left\lceil \frac{(201-3)}{16} \right\rceil + 1) = 14$$
$$DW(3j-2) = 100$$
$$\Rightarrow DB(2i+1, 3j-2) \leq 1400$$

(exact value of $DB(2i+1, 3j-2)$ is 1337 or 1338 for above example)

Estimating DW for multiple array references (simple solutions)

1. ignore group reuse
   $\Rightarrow DW(\{f1, \ldots, fk\}) \leq DW(f1) + \ldots + DW(fk)$

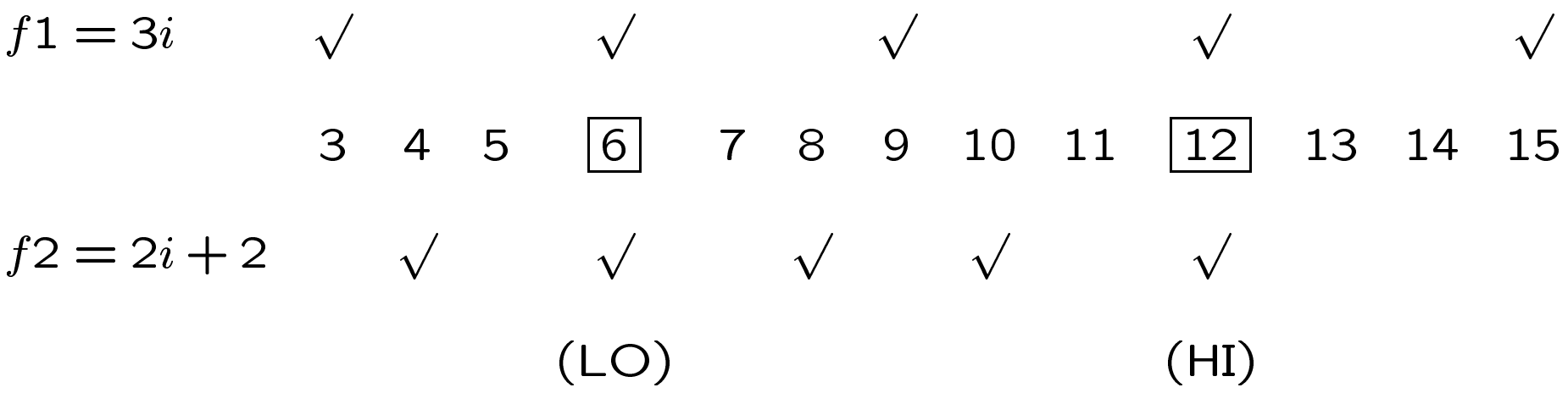
2. equivalence class approach:

   - partition $\{f1, \ldots, fk\}$ into equivalence classes $\{C_1, \ldots, C_l\}$, such that $C_i = \{f_1^{C_i}, \ldots, f_{|C_i|}^{C_i}\}$ (a common approach is to partition according to uniformly generated array references [Gallivan et al '88])

   - assume 100% reuse within a class and 0% reuse among classes
     $\Rightarrow DW(\{f1, \ldots, fk\}) \simeq \sum_{i=1}^{l} DW(f_1^{C_i})$

Estimating DW for multiple array references (Example)

```
DO i = 1, 5
  ... A(3*i) ... A(2*i+2) ...
ENDDO
```



$$\begin{aligned} DW(\{f1, f2\}) &= DW(f1) + DW(f2) - overlap(f1, f2) \\ &= 5 + 5 - 2 = 8 \end{aligned}$$

Estimating DW for multiple array references

Define $test(f1, f2) = 1$ if and only if subscript expressions $f1$ and $f2$ can overlap (regardless of loop bounds); $test$ can be computed using the GCD data dependence test.

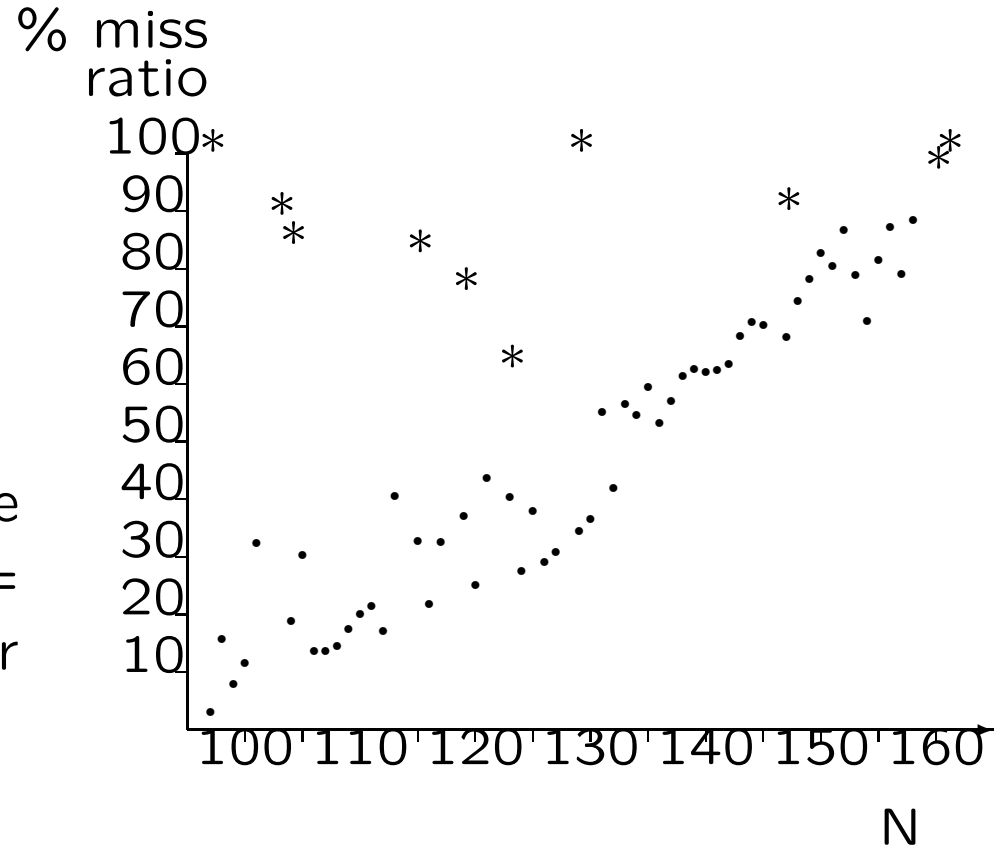
Upper bound for $DW(\{f1, f2\})$:

$$\begin{aligned} DW(\{f1, f2\}) &= DW(f1) + DW(f2) - overlap(f1, f2) \\ &\leq \left(\frac{(f1^{hi} - f1^{lo})}{g1} + 1\right) + \left(\frac{(f2^{hi} - f2^{lo})}{g2} + 1\right) \\ &\quad - test(f1, f2) \times \left(\frac{(HI - LO)}{lcm(g1, g2)} + 1\right) \end{aligned}$$

Example of Set Conflicts

```
    real*8 A(N,N)
    do 10 i = 1, N
      do 10 j = 1, N
        do 10 k = 1, N
10        . . . A(i,k)
```

Simulate `A(i,k)` reference for cache parameters, $2^b = 16$, $2^s = 32$, $2^d = 4$, and for $96 \leq N \geq 160$



Set conflict analysis identifies the main outlying points,
$N = 96$, 102, 103, 114, 118, 122, 128, 146, 159, 160

Dealing with Low Cache Utilization Efficiency

What should we do when the cache utilization efficiency is low?

Possible solutions:

1. Pad array dimension size to improve efficiency, if legal to do so

2. Copy into temporary array with larger dimension size, if legal and efficient to do so

3. Adjust nominal (effective) cache size to reflect actual utilization efficiency in compiler cost functions

Using Effective Cache Size to Estimate # Misses for a Direct-Mapped (or Set-Associative) Cache

Summary of approach:

- Compute m = # innermost loops in locality group assuming fully-associative cache
- For each array variable, A, set $\eta_{min}(A)$ = *minimum* stride efficiency value across m innermost loops
- Set *effective cache size* $S' = \lfloor \eta_{avg/min} S \rfloor$, where $\eta_{avg/min}$is the *average* of all $\eta_{min}(A)$
- Do locality analysis assuming a *fully associative cache* of size S'

Selection of Tile Sizes — a constrained optimization problem

Objective function: Select tile sizes $t_1, \ldots, t_h$ so as to minimize $F(t_1, \ \ldots \ , t_h) = \frac{COST_{total}}{t_1 \times \ \ldots \ \times t_h}$

Constraints:

- Each tile size must be in the range $1 \leq t_k \leq Ubound_k$.
- $DL_{total}(t_1, \ \ldots \ , t_h) \leq ECS$. The number of distinct cache lines accessed in a tile must not exceed the effective cache size.
- $DP_{total}(t_1, \ \ldots \ , t_h) \leq ETS$. The number of distinct virtual pages accessed in a tile must not exceed the effective TLB size.

Constant-time solution for two loops. For $N > 2$ loops with negative slope, search on t_k values for $(N - 2)$ loops.

Selection of Tile Sizes for Matrix Multiply-Transpose Example

$$DL_{total}(t_1, t_2, t_3) = (0.25t_1 + 0.75)\, t_2 + (0.25t_2 + 0.75)\, t_3 + (0.25t_3 + 0.75)\, t_1$$

- $DL_{total}(t_1, t_2, t_3) \leq ECS = 2048$ is the active constraint
- Solution returned by algorithm is $t_1 = 50$, $t_2 = 51$, $t_3 = 51$

  (Note that $DL_{total}(50, 51, 51) = 2039.25$ and
  $DL_{total}(51, 51, 51) = 2065.50$)

NOTE: in general, tile sizes need not be equal.

Transformed Code after Tiling

```
do bb$_12=1,n,50
  do bb$_13=1,n,51
    do bb$_14=1,n,51
      do i1=MAX0(1,bb$_12),MIN0(n,49 + bb$_12)
        do i2=MAX0(1,bb$_13),MIN0(n,50 + bb$_13)
          do i3=MAX0(1,bb$_14),MIN0(n,50 + bb$_14),1
            a(i1,i2) = a(i1,i2) + b(i2,i3) * c(i3,i1)
          end do
        end do
      end do
    end do
  end do
end do
```

Selection of Unroll Factors

Objective function: Select unroll factors $u_1, \ldots$ so as to minimize *amortized* execution time per original iteration

Constraints:

- $DFR(u_1, \ldots) \leq EFR$. The number of distinct floating-point references in the unrolled loop body must not exceed the effective number of floating-point registers available.
- $DXR(u_1, \ldots) \leq EXR$. The number of distinct fixed-point references in the unrolled loop body must not exceed the effective number of fixed-point registers available.

Objective function may not be monotonically nonincreasing
$\Rightarrow$ do an exhaustive enumeration of feasible unroll factors

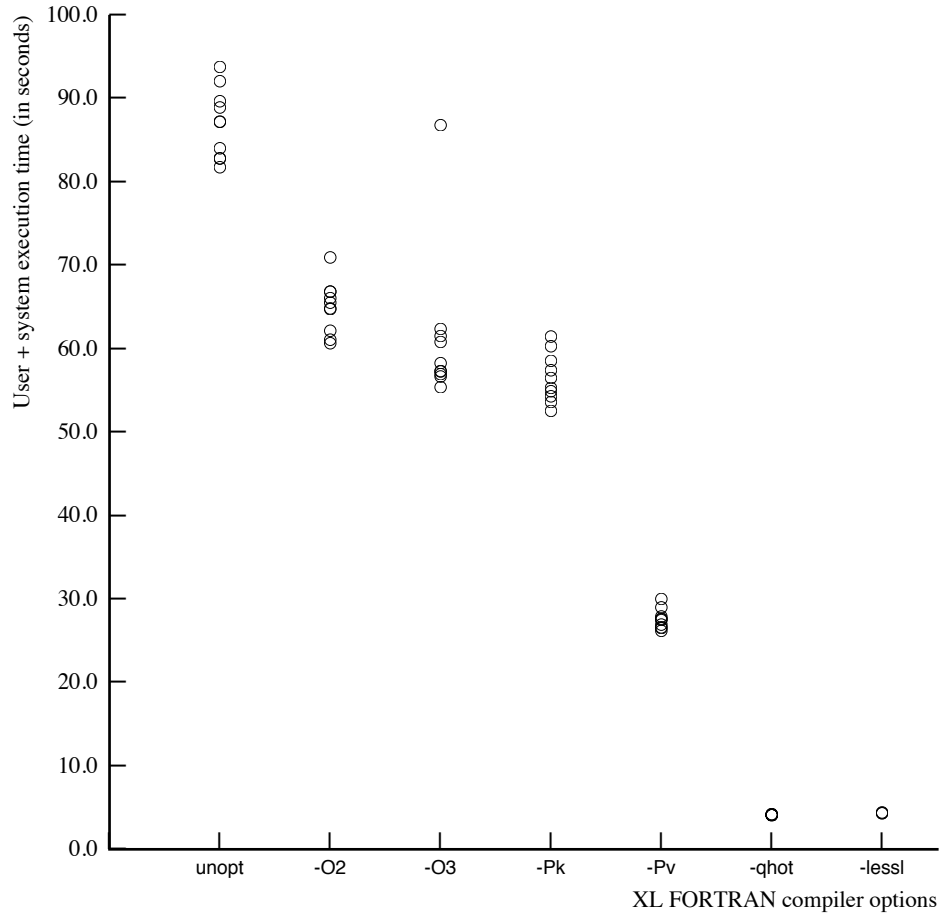
Selection of Unroll Factors for Matrix Multiply-Transpose Example

To simplify discussion, assume that only benefit of unrolling is savings of loads of `b(i2,i3)` and `c(i3,i1)`:

$$\begin{aligned} \text{Amortized \# loads, } F(u_1,u_2,u_3) &= \frac{u_1u_3+u_2u_3}{u_1u_2u_3} = \frac{1}{u_2}+\frac{1}{u_1} \\ DFR(u_1,u_2,u_3) &= u_1u_2+u_1u_3+u_2u_3 \end{aligned}$$

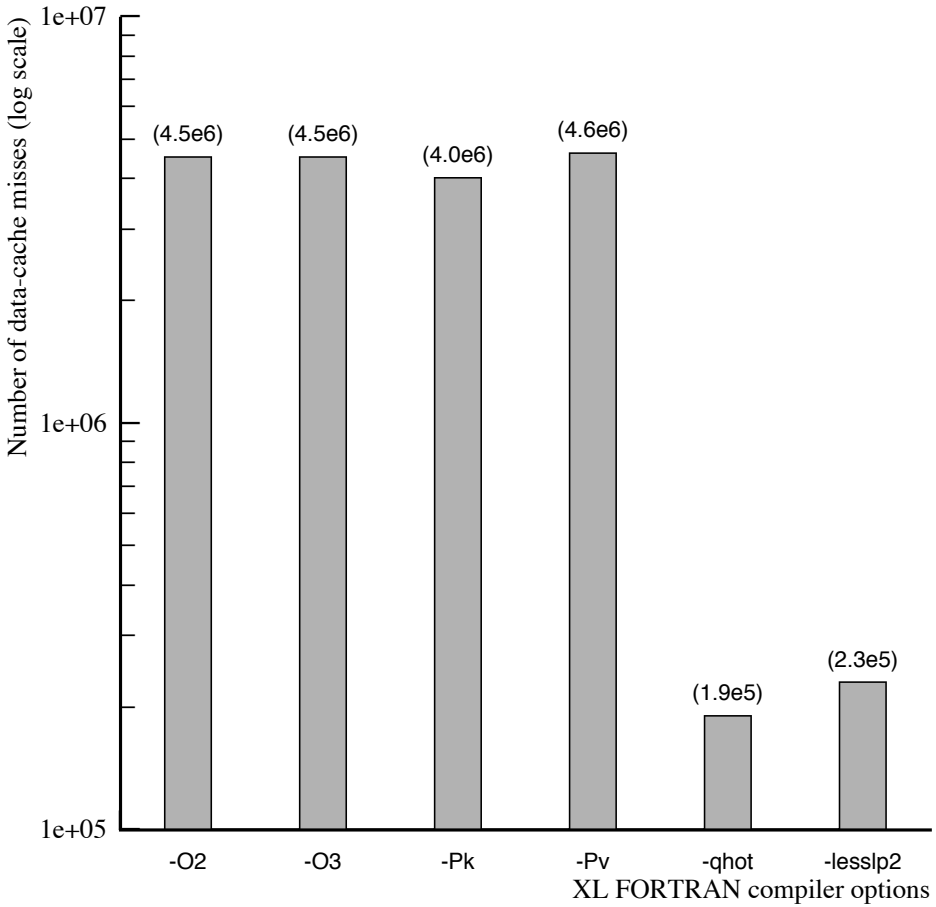
Setting $DFR(u_1,u_2,u_3) \leq 28$ yields $u_1 = 4$, $u_2 = 4$, $u_3 = 1$ as the best solution with $DFR(4,4,1) = 24$ and $F(4,4,1) = 0.5$ loads/iteration.

Preliminary Experimental Results (Multiply-Transpose)



Performance measurements on a 133MHz PowerPC 604 processor for matrix multiply-transpose example.

Data Cache Misses



Conclusions

- We described how the ASTI transformer automatically selects high-order transformations for a given target uniprocessor.

- Quantitative approach to program optimization is critrical for delivering robust optimizations across different programs and target parameters.

- To the best of our knowledge, the ASTI transformer is the first system to support automatic selection of the wide range of transformations described in this paper, using a cost-based framework.