CS 380C:
Advanced Topics in Compilers

Administration

- Instructor: Keshav Pingali
  - Professor (CS, ICES)
  - Office: POB 4.126A
  - Email: pingali@cs.utexas.edu
- TA: Sridhar Gopinath
  - PhD student in CS
  - Email:sridhar@cs.utexas.edu

Meeting times

- Lecture:
  - TTh 12:30-2:00PM, GDC 2.210
- Office hours:
  - Keshav Pingali: Tuesday 3-4 PM, POB 4.126

Prerequisites

- Compilers and architecture
  - Some background in compilers
  - Basic computer architecture
- Machine learning
  - Basic knowledge of machine learning
- Software and math maturity
  - Able to implement large programs in C/C++
  - Comfortable with abstractions like graph theory
- Ability to read research papers and understand content

Course material

- Website for course
  - http://www.cs.utexas.edu/users/pingali/CS380C/2019/index.html
- All lecture notes, announcements, papers, assignments, etc. will be posted there
- No assigned book for the course
  - post papers and other material as appropriate

Coursework

- 4-5 programming assignments and problem sets
- Mid-semester exam
- Paper presentations
  - Second half of semester
- Term project
  - Substantial implementation project in area of compilers
- Final exam (at my discretion)

Why do we need compilation technology?

- Traditional view:
  - Translation: high-level language (HLL) programs to low-level machine code
  - Optimization: reduce number of arithmetic operations by optimizations like common subexpression elimination
  - Ignore data structures: too complex to analyze
- Modern view:
  - Collection of automatic techniques for extracting meaning from and transforming programs
  - Useful for debugging, optimization, verification, detecting malware, translation, …..
  - Optimization:
    - Restructure (reorganize) computation to optimize locality and parallelism
    - Reducing amount of computation is useful but not critical
    - Optimizing data structure accesses is critical

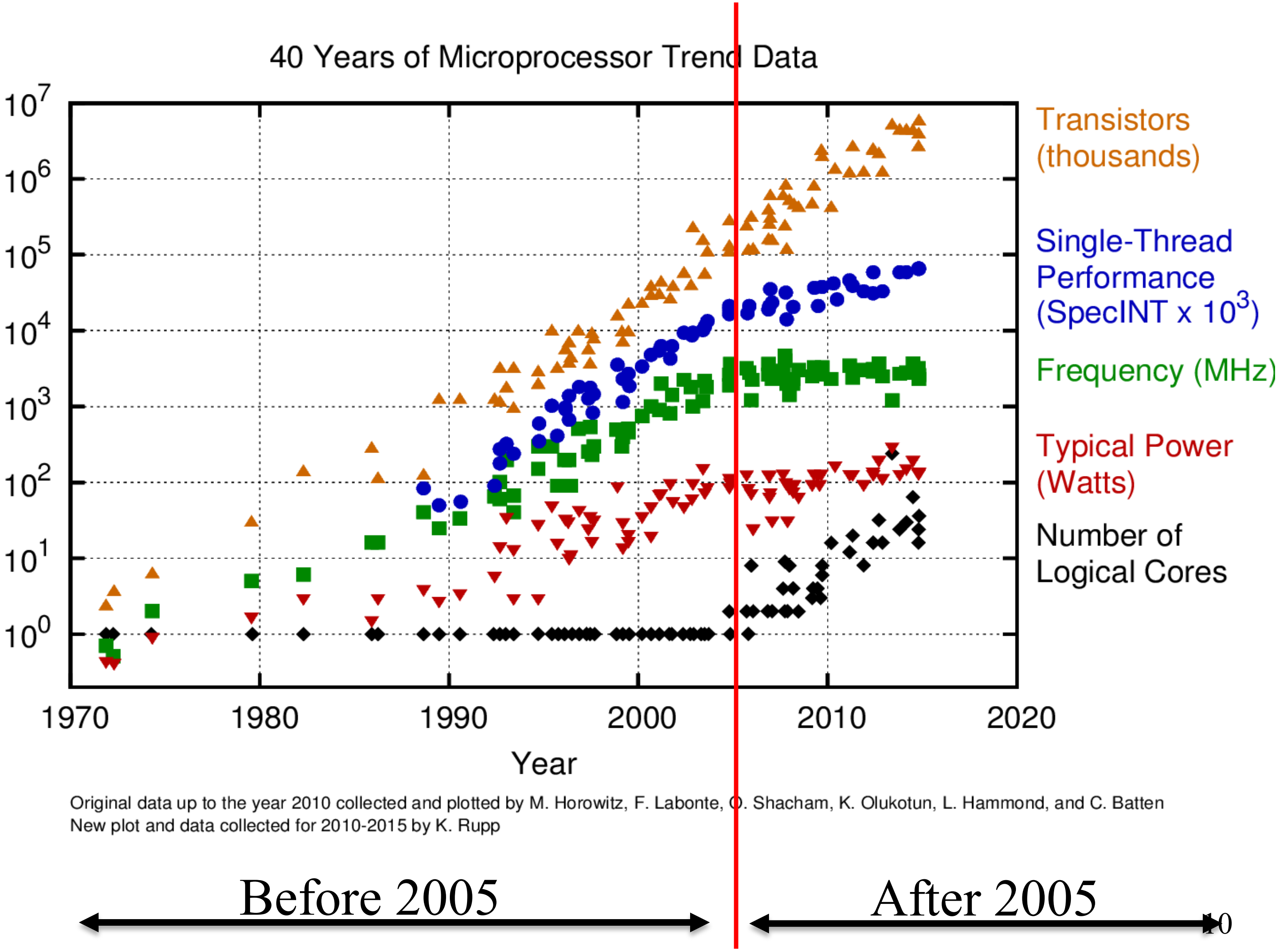
Why do we need translators?

- Bridge the "semantic gap"
  - Programmers prefer to write programs at a high level of abstraction
  - Modern architectures are very complex, so to get good performance, we have to worry about a lot of low-level details
  - Compilers let programmers write high-level programs and still get good performance on complex machine architectures
- Application portability
  - When a new ISA or architecture comes out, you only need to reimplement the compiler on that machine
  - Application programs should run without (substantial) modification
  - Saves programming effort
- Summary: performance + portability of HLL programs

Getting performance

- Programs must exploit
  - coarse-grain (thread-level) parallelism
  - memory hierarchy (L1,L2,L3,..)
  - instruction-level parallelism (ILP)
  - registers
  - ….
- How important is it to exploit these hardware features?
  - If you have n cores and you run on only one, you get at most 1/n of peak performance, so this is obvious
  - Memory hierarchy: typical latencies
    - L1 cache: ~ 1 cycle
    - L2 cache: ~ 10 cycles
    - Memory: ~ 500-1000 cycles
    - If most memory accesses hit in L1/L2 cache, performance is much better than if most of accesses go to memory

Microprocessor trend data



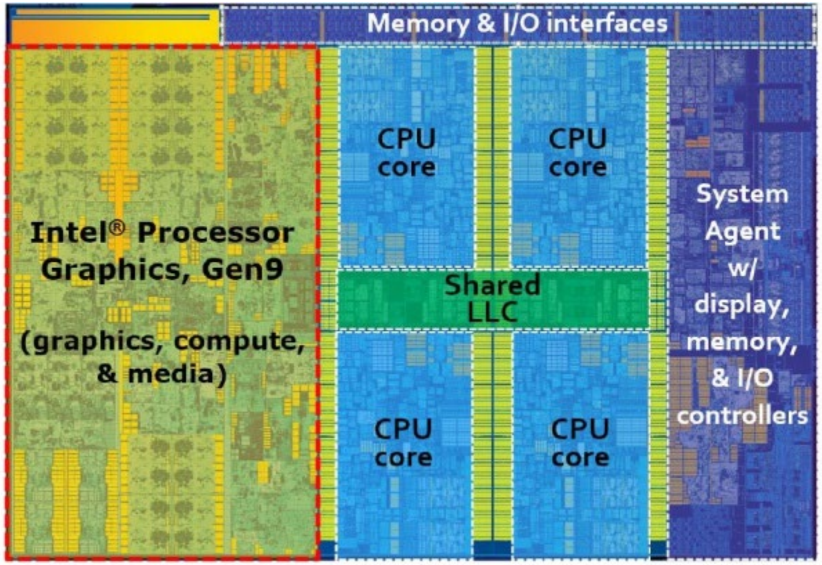
40 Years of Microprocessor Trend Data

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Before 2005

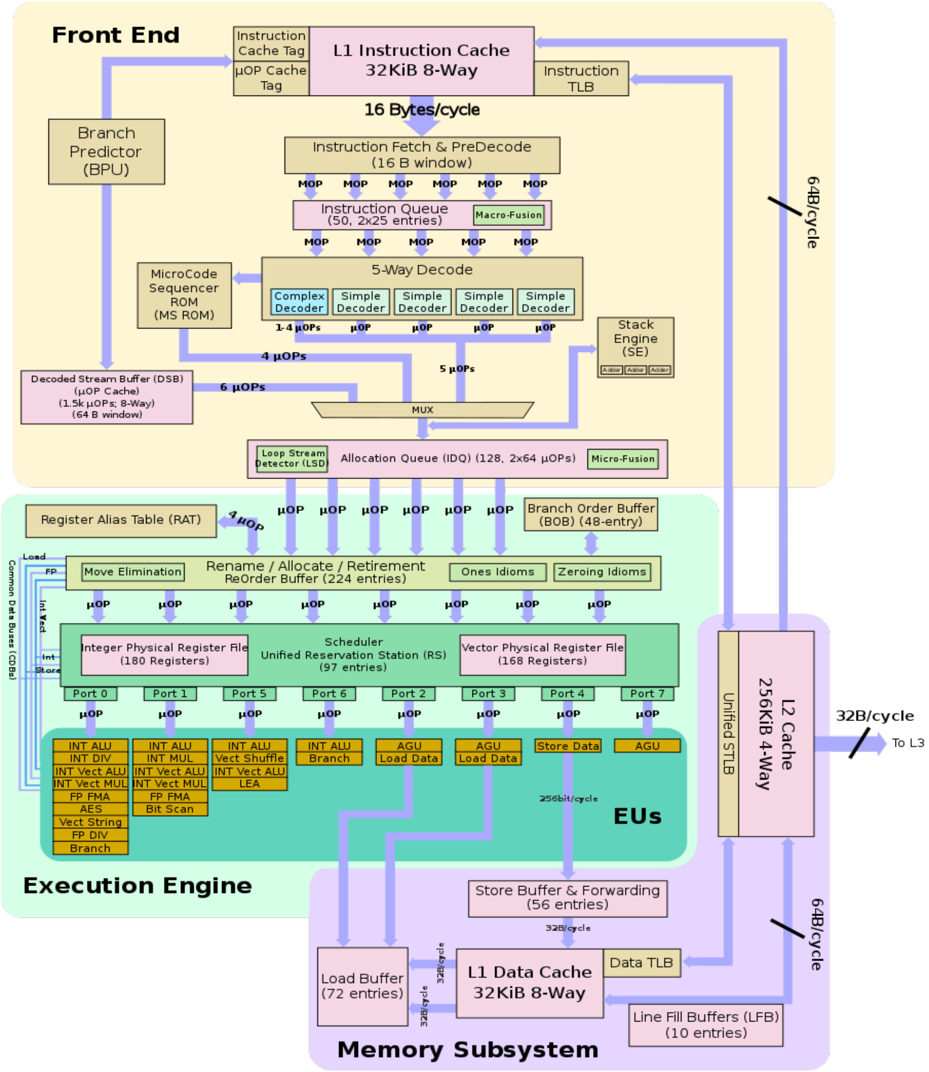
After 2005

Intel Skylake chip



Memory & I/O interfaces
Intel® Processor Graphics, Gen9
(graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers

Chip



Block diagram of each core

Software problem

- Problem:
  - Programs obtained by expressing most algorithms in the straight-forward way perform poorly
  - Worrying about performance when coding algorithms complicates the software process greatly
- Let us study cache optimization to understand this
- Caches are useful only if programs have locality of reference
  - temporal locality: program references to given memory address are clustered together in time
  - spatial locality: program references clustered in address space are clustered in time

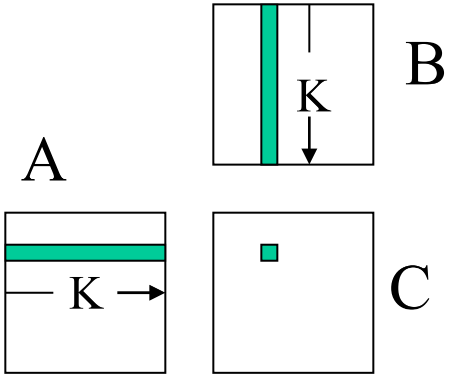
Example: matrix multiplication

```
for I = 1, N    //assume arrays stored in row-major order
  for J = 1, N
    for K = 1, N
      C(I,J) = C(I,J) + A(I,K)*B(K,J)
```

- All six loop permutations are computationally equivalent (even modulo round-off error).
- Great algorithmic data reuse: each array element is touched O(N) times!
- However, execution times of the six versions can be very different if machine has a cache.

IJK version (large cache)

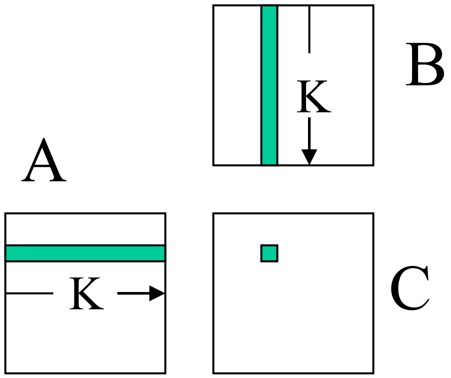
```
for I = 1, N
   for J = 1, N
      for K = 1, N
        C(I,J) = C(I,J) + A(I,K)*B(K,J)
```



- Large cache scenario: matrices are small enough to fit into cache
  - Assume only cold misses, no capacity or conflict misses
  - Miss ratio:
    - Data size = 3 N^2
    - Assume line size = b floating-point numbers
    - Miss ratio = 3 N^2 /b*4N^3 = 0.75/bN = 0.019 (b = 4,N=10)

IJK version (small cache)

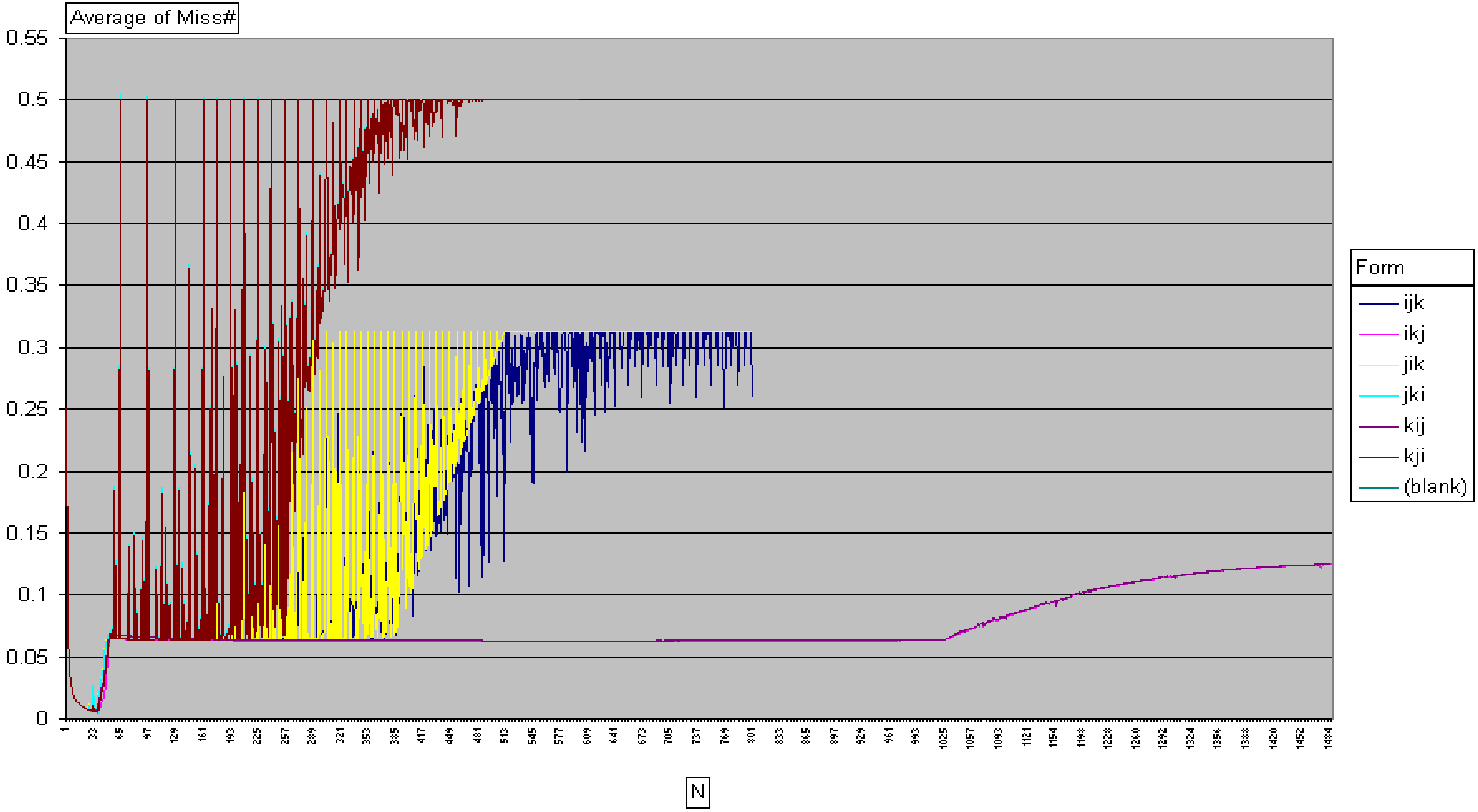
```
for I = 1, N
   for J = 1, N
      for K = 1, N
         C(I,J) = C(I,J) + A(I,K)*B(K,J)
```



- Small cache scenario: matrices are large compared to cache/row-major storage
  - Cold and capacity misses (ignore conflict misses)
  - Miss ratio:
    - C: N^2/b misses (good temporal locality)
    - A: N^3/b misses (good spatial locality)
    - B: N^3 misses (poor temporal and spatial locality)
    - Miss ratio → 0.25 (b+1)/b = 0.3125 (for b = 4)

MMM Experiments

- Simulated L1 Cache Miss Ratio for Intel Pentium III
  - MMM with N = 1…1300
  - 16KB 32B/Block 4-way 8-byte elements



Quantifying performance differences

```
for I = 1, N    //assume arrays stored in row-major order
  for J = 1, N
    for K = 1, N
      C(I,J) = C(I,J) + A(I,K)*B(K,J)
```

- Typical cache parameters:
  - L2 cache hit: 10 cycles, cache miss 70 cycles
- Time to execute IKJ version:

  $2N^3 + 70*0.13*4N^3 + 10*0.87*4N^3 = 73.2\ N^3$
- Time to execute JKI version:

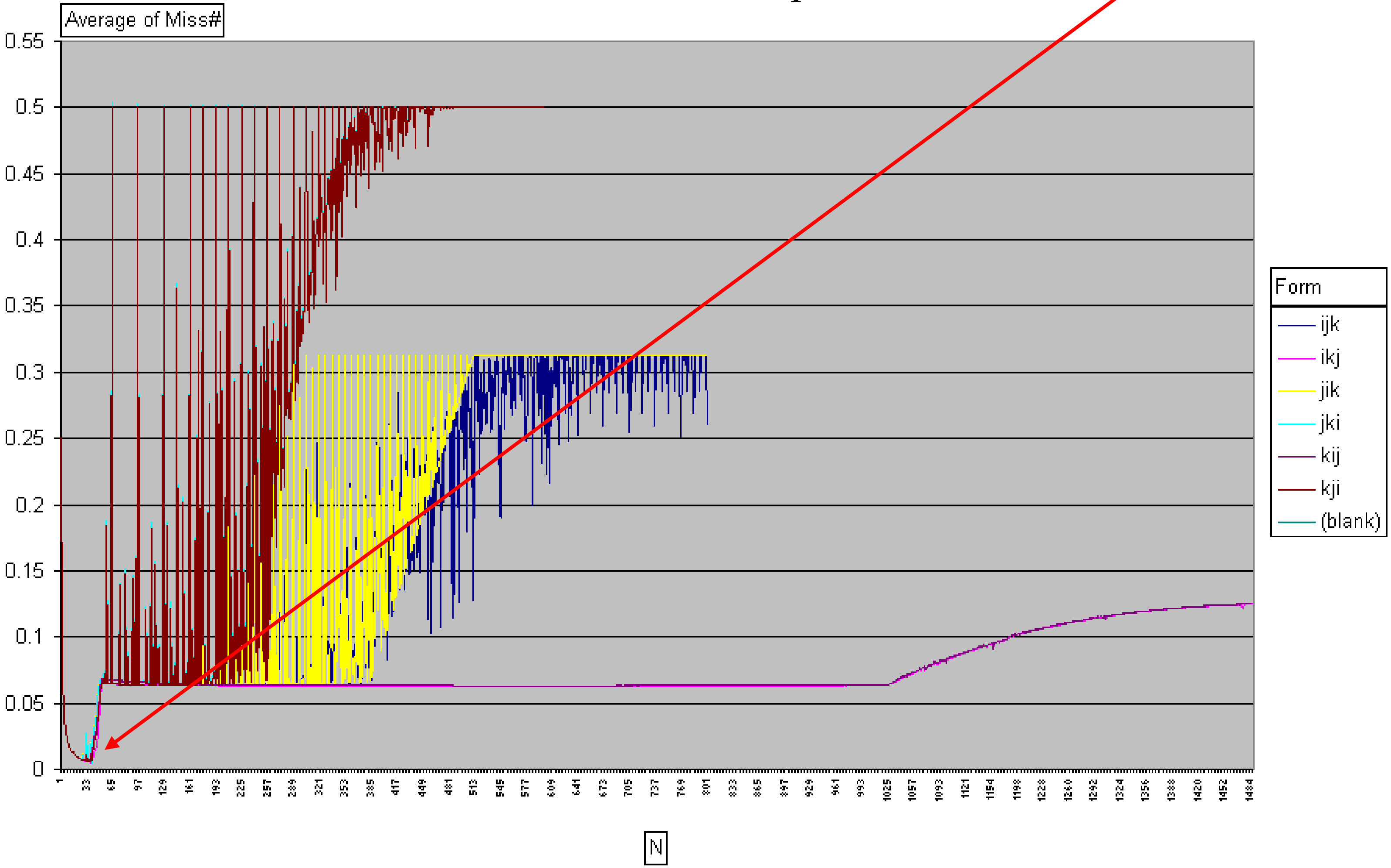
  $2N^3 + 70*0.5*4N^3 + 10*0.5*4N^3 = 162\ N^3$
- Speed-up = 2.2
- Key transformation: loop permutation

Even better…..

- Break MMM into a bunch of smaller MMMs so that large cache model is true for each small MMM

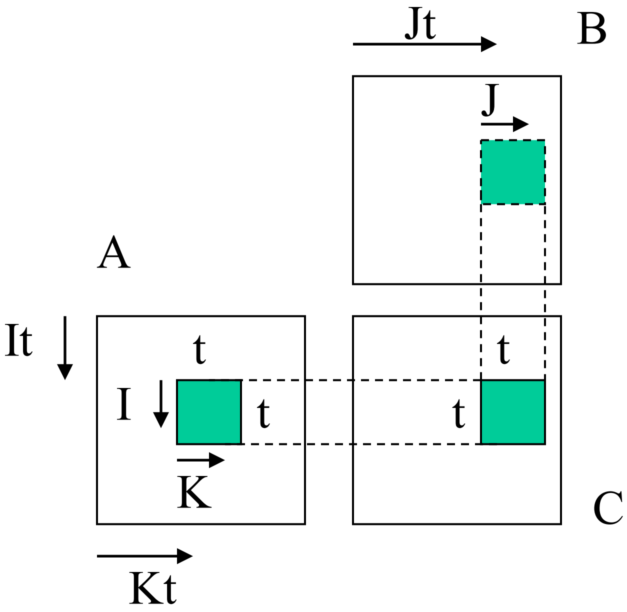
  ➔ large cache model is valid for entire computation

  ➔ miss ratio will be 0.75/bt for entire computation where t is



Loop tiling/blocking

```
for It = 1,N, t
  for Jt = 1,N,t
    for Kt = 1,N,t
      for I = It,It+t-1
        for J = Jt,Jt+t-1
          for K = Kt,Kt+t-1
            C(I,J) = C(I,J)+A(I,K)*B(K,J)
```



- Break big MMM into sequence of smaller MMMs where each smaller MMM multiplies sub-matrices of size txt.
- Parameter t (tile size) must be chosen carefully
  - as large as possible
  - working set of small matrix multiplication must fit in cache

Speed-up from tiling/blocking

- Miss ratio for block computation

  = miss ratio for large cache model

  = 0.75/bt

  = 0.001 (b = 4, t = 200)

- Time to execute tiled version =

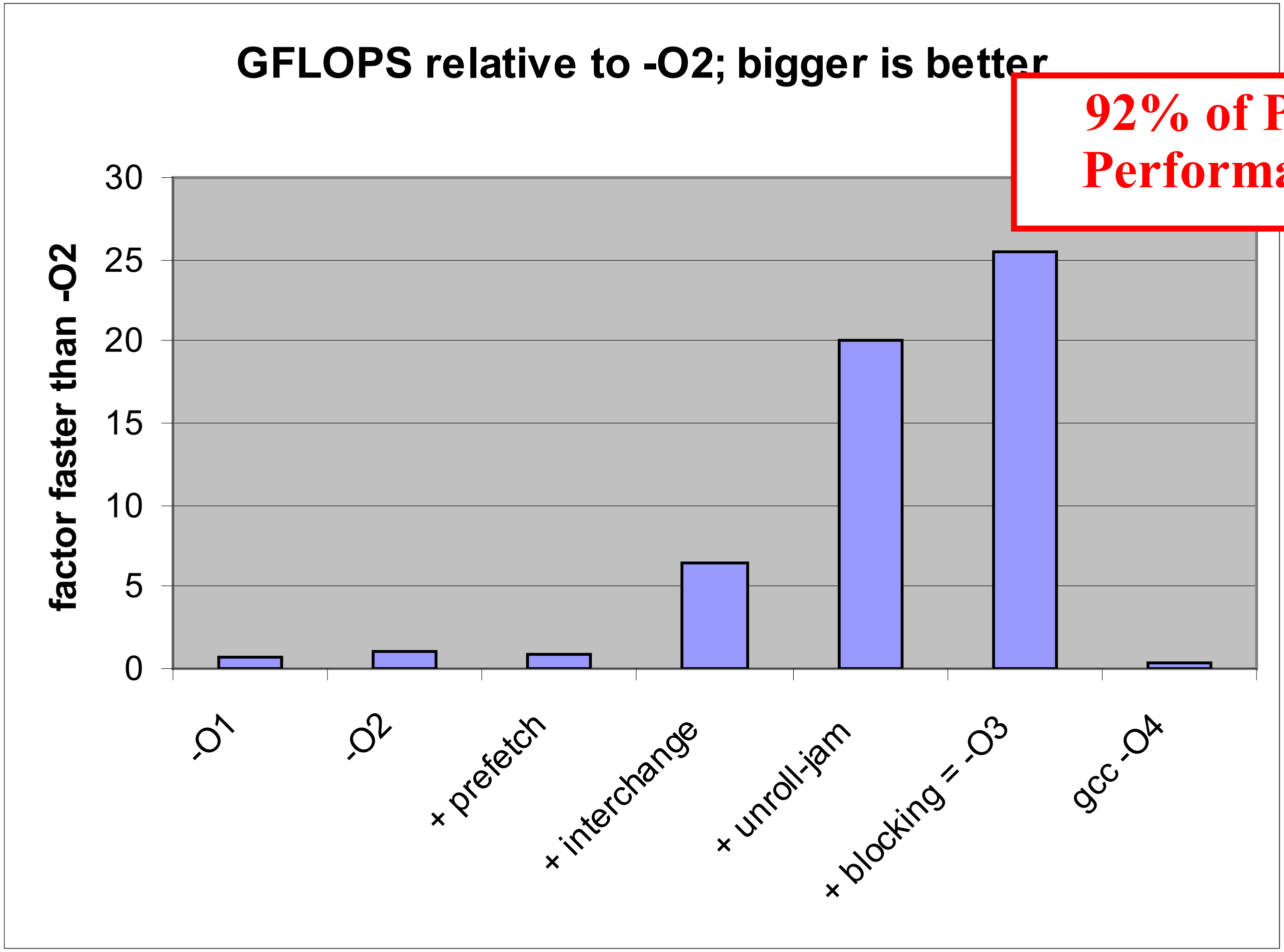
  $2N^3 + 70*0.001*4N^3 + 10*0.999*4N^3 = 42.3N^3$

- Speed-up over JKI version = 4

Observations

- Locality optimized code is more complex than high-level algorithm.
- Locality optimization changed the order in which operations were done, not the number of operations
- “Fine-grain” view of data structures (arrays) is critical
- Loop orders and tile size must be chosen carefully
  - cache size is key parameter
  - associativity matters
- Actual code is even more complex: must optimize for processor resources
  - registers: register tiling
  - pipeline: loop unrolling
  - Optimized MMM code can be ~1000’s of lines of C code
- Wouldn’t it be nice to have all this be done automatically by a compiler?
  - Actually, it is done automatically nowadays…

Performance of MMM code produced by Intel's Itanium compiler (-O3)



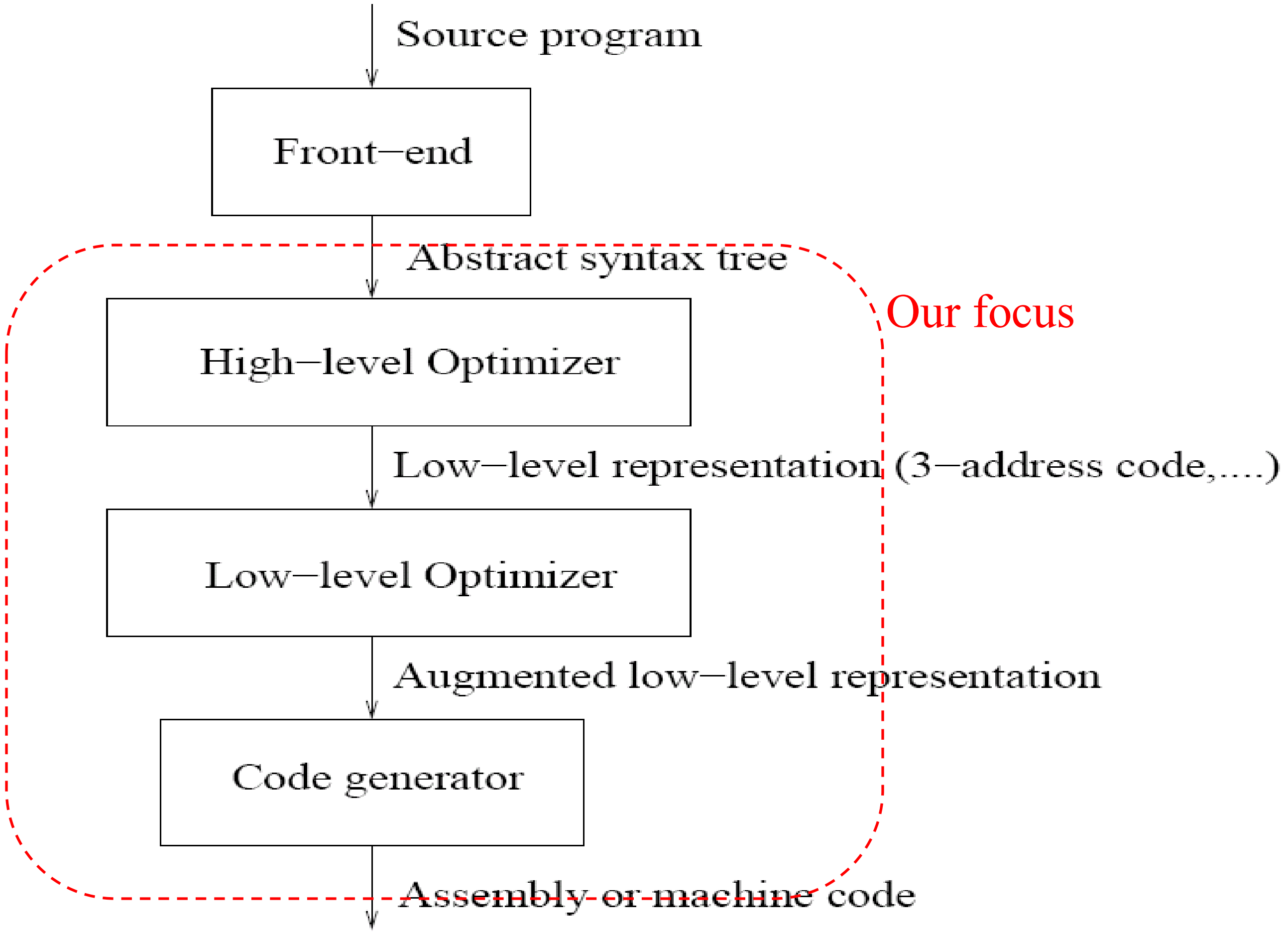
92% of Peak Performance

Goto BLAS obtains close to 99% of peak, so compiler is pretty good!

Discussion

- Exploiting parallelism, memory hierarchies etc. is very important
- If program uses only one core out of n cores in processors, you get at most 1/n of peak performance
- Memory hierarchy optimizations are very important
  - can improve performance by 10X or more
- Key points:
  - need to focus on data structure manipulation
  - reorganization of computations and data structure layout are key
  - few opportunities usually to reduce the number of computations except in address arithmetic

Organization of modern compiler



Front-end

- Goal: convert linear representation of program to hierarchical representation
  - Input: text file
  - Output: abstract syntax tree + symbol table
- Key modules:
  - Lexical analyzer: converts sequence of characters in text file into sequence of tokens
  - Parser: converts sequence of tokens into abstract syntax tree + symbol table
  - Semantic checker: (eg) perform type checking

High-level optimizer

- Goal: perform high-level analysis and optimization of program
- Input: AST + symbol table from front-end
- Output: Low-level program representation such as 3-address code
- Tasks:
  - Procedure/method inlining
  - Array/pointer dependence analysis
  - Loop transformations: unrolling, permutation, tiling, jamming,….

Low-level optimizer

- Goal: perform scalar optimizations on low-level representation of program
- Input: low-level representation of program such as 3-address code
- Output: optimized low-level representation + additional information such as def-use chains
- Tasks:
  - Dataflow analysis: live variables, reaching definitions, …
  - Scalar optimizations: constant propagation, partial redundancy elimination, strength reduction, ….

Code generator

- Goal: produce assembly/machine code from optimized low-level representation of program
- Input: optimized low-level representation of program from low-level optimizer
- Output: assembly/machine code for real or virtual machine
- Tasks:
  - Register allocation
  - Instruction selection

JIT compilation

- Traditionally, all phases of compilation were completed before program was executed
- New twist: virtual machines
  - Offline compiler:
    - Generates code for virtual machine like JVM
  - Just-in-time compiler:
    - Generates code for real machine from VM code while program is executing
- Advantages:
  - Portability
  - JIT compiler can perform optimizations for particular input

My lectures (scalar stuff)

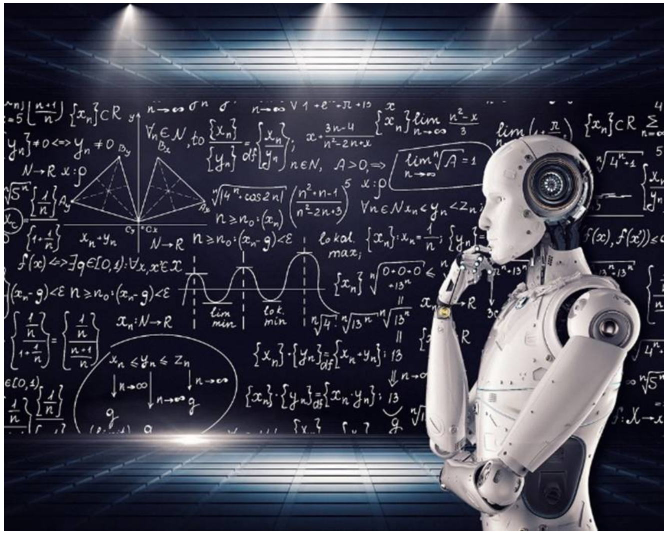
- Introduction
  - compiler structure, architecture and compilation, sources of improvement
- Control flow analysis
  - basic blocks & loops, dominators, postdominators, control dependence
- Data flow analysis
  - lattice theory, iterative frameworks, reaching definitions, liveness
- Static-single assignment form (SSA)
  - static-single assignment, constant propagation.
- Global optimizations
  - loop invariant code motion, common subexpression elimination, strength reduction.
- Register allocation
  - coloring, allocation, live range splitting.
- Instruction scheduling (depending on schedule)
  - pipelined and VLIW architectures, list scheduling.

My lectures (data structure stuff)

- Array dependence analysis
  - integer linear programming, dependence abstractions.
- Loop transformations for array programs
  - linear loop transformations, loop fusion/fission, enhancing parallelism and locality
- Self-optimizing programs
  - empirical search, ATLAS, FFTW
- Analysis of pointer-based programs
  - points-to and shape analysis
- Parallelizing graph programs
  - amorphous data parallelism, exploiting amorphous data-parallelism

Advanced topics for CS 380C

- Optimizing machine learning programs
  - Training and testing times can be large
    - Models are getting more complex
    - Lot of training data
  - How for we optimize training and testing times on modern architectures?
- Exploiting machine learning in compilers
  - Some work in this area but no major breakthroughs yet
  - Active research topic
- Course
  - See website for partial list of papers we will study in this area
  - Papers will be presented by students
  - Ideally, your paper presentation and course project will be linked



Schedule for lectures

- See
  - http://www.cs.utexas.edu/users/pingali/CS380C/2019/index.html
- Some lectures will be given by guest lecturers from my group and from industry

Reading assignments

- My SIGARCH blogpost:
  - Why has machine learning not had more impact on systems?
- Mike O'Boyle's survey article on using machine learning in compilers
  - Machine learning in compiler optimization Wang and O'Boyle, arXiv:1805.03441
- Eran Yahav's SIGPLAN blog post on machine learning in compilers
  - From-programs-to-deep-models-part-1
- Lecture slides on SAM
  - Simple stack machine