Higher Level Programming on Parallel Computers: Sweetening the Deal for Programmers
(and Making Compilers Work Harder)

Arun Chauhan, Indiana University

University of Rochester, Nov 30, 2009

Collaborators

Blake Barker

Torsten Hoefler

Eric Holk

William Holmes

Andrew Keep

Andrew Lumsdaine

Daniel McFarlin

Pushkar Ratnalikar

Koby Rubinstein

Sidney Shaw

Chun-Yu Shei

Jeremiah Willcock

Kevin Zumbrun

"A New Kind of Science"

Stephen Wolfram

"Computing is as fundamental as the physical, life, and social sciences."

Peter J. Denning and Paul S. Rosenbloom
Communications of the ACM, Sep 2009

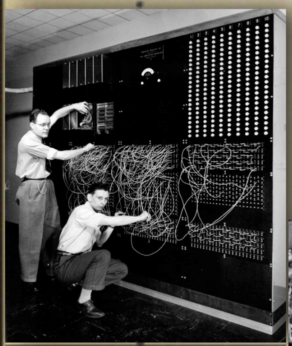
"What our community should really aim for is the development of a curriculum that turns our subject into the fourth R—as in 'rogramming—of our education systems.

…

A form of mathematics can be used as a full-fledged programming language, just like Turing Machines."

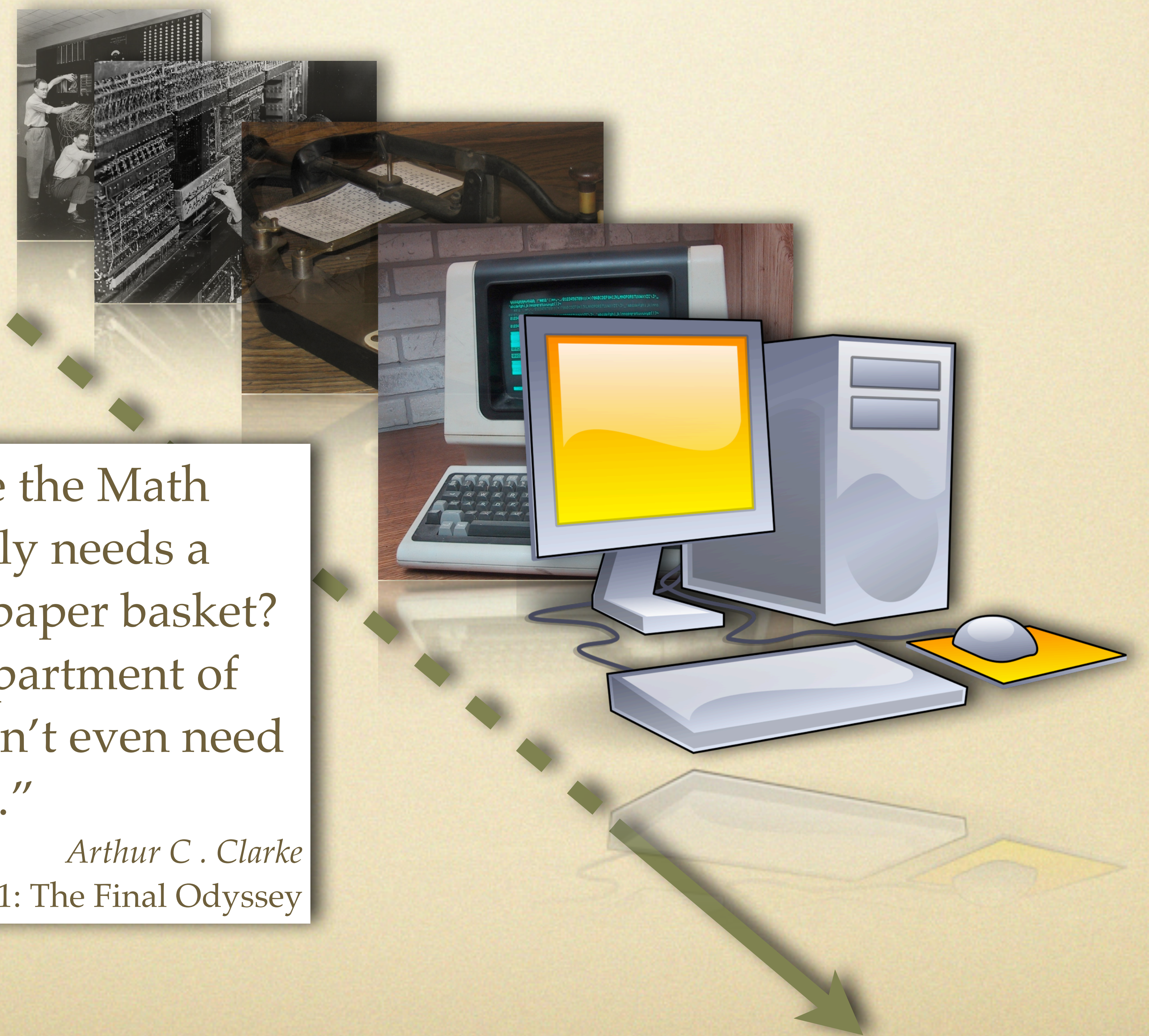
Matthias Felleisen and Shriram Krishnamurthy
Communications of the ACM, Jul 2009

Programming





Programming



"Why can't you be like the Math Department, which only needs a blackboard and wastepaper basket? Better still, like the Department of Philosophy. That doesn't even need a wastepaper basket …"

Arthur C . Clarke

3001: The Final Odyssey

Computers are for Computing and ...

- Computers as general-purpose tools
  - communication, navigation, data collection, entertainment, etc.
- Computers as computing tools
  - problem solving
  - data processing and analysis

Overview

- **Motivation**
- Rethinking program analysis
- Rescuing parallel programmers
- Concluding remarks

Rethinking Program Analysis

Problem

- Nice programming languages
  - domain-specific
  - often dynamically typed and interpreted
- Poor performance
  - inefficient use of computing resources
  - inefficient use of energy

"It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts."

Sir Arthur Conon Doyle
A Scandal in Bohemia

Example 1: BLAS

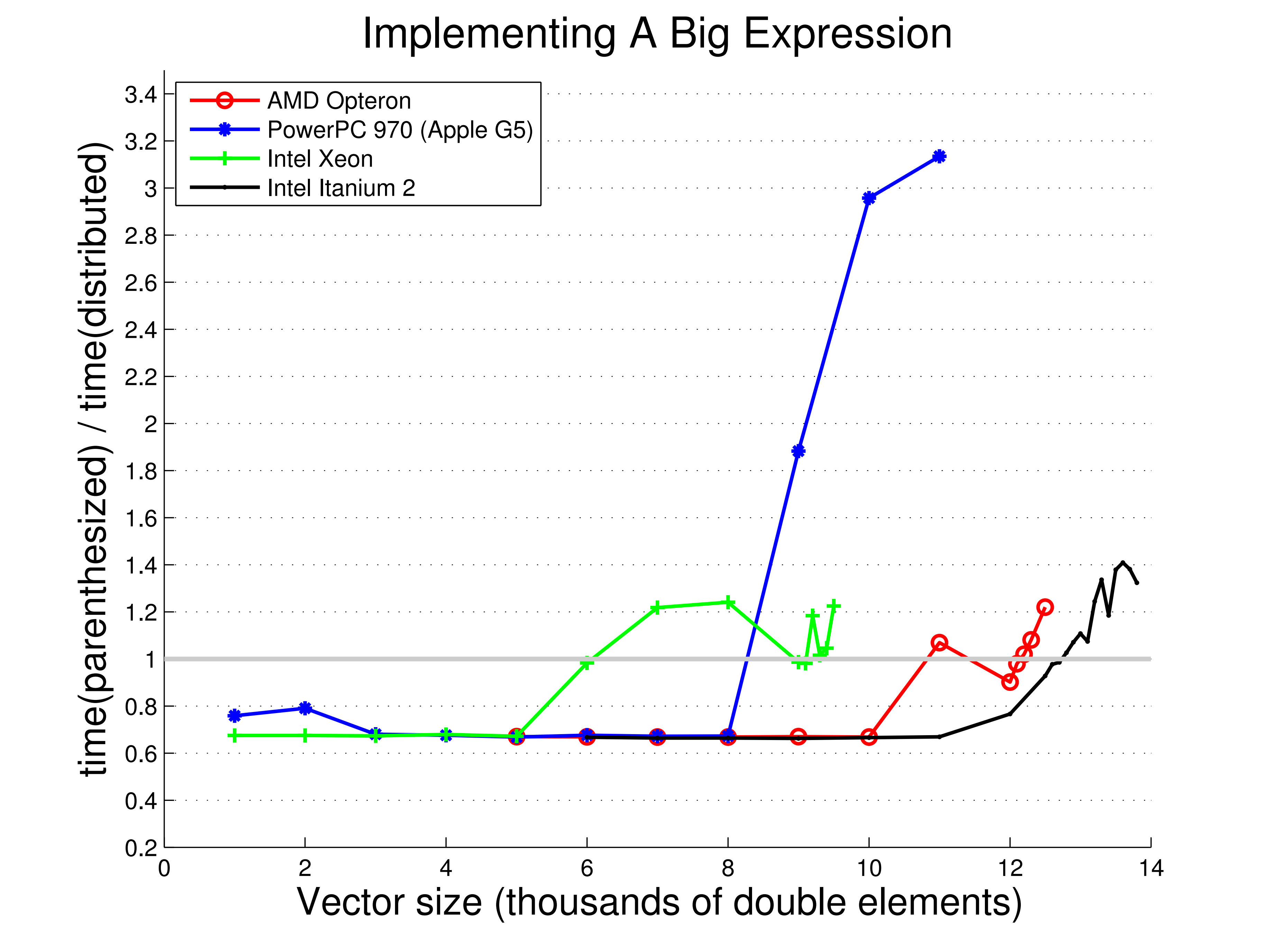
A+A*B′ + 2*(A+B)′*A + (x+y)*x′

```
copy(A,tmp0);
gemm(1,A,B,1,tmp0);
copy(A,tmp1);
axpy(1,B,1,tmp1);
gemm(2,tmp1,A,1,tmp0);
copy(x,tmp1);
axpy(1,y,1,tmp1);
ger(1,tmp1,x,tmp0);
```

A+A*B′ + 2*A′*A + 2*B′*A + x*x′ + y*x′

```
gemm(1,A,B,1,tmp0);
ger(1,x,x,tmp0);
ger(1,y,x,tmp0);
gemm(2,A,A,1,tmp0);
gemm(2,B,A,1,tmp0);
```

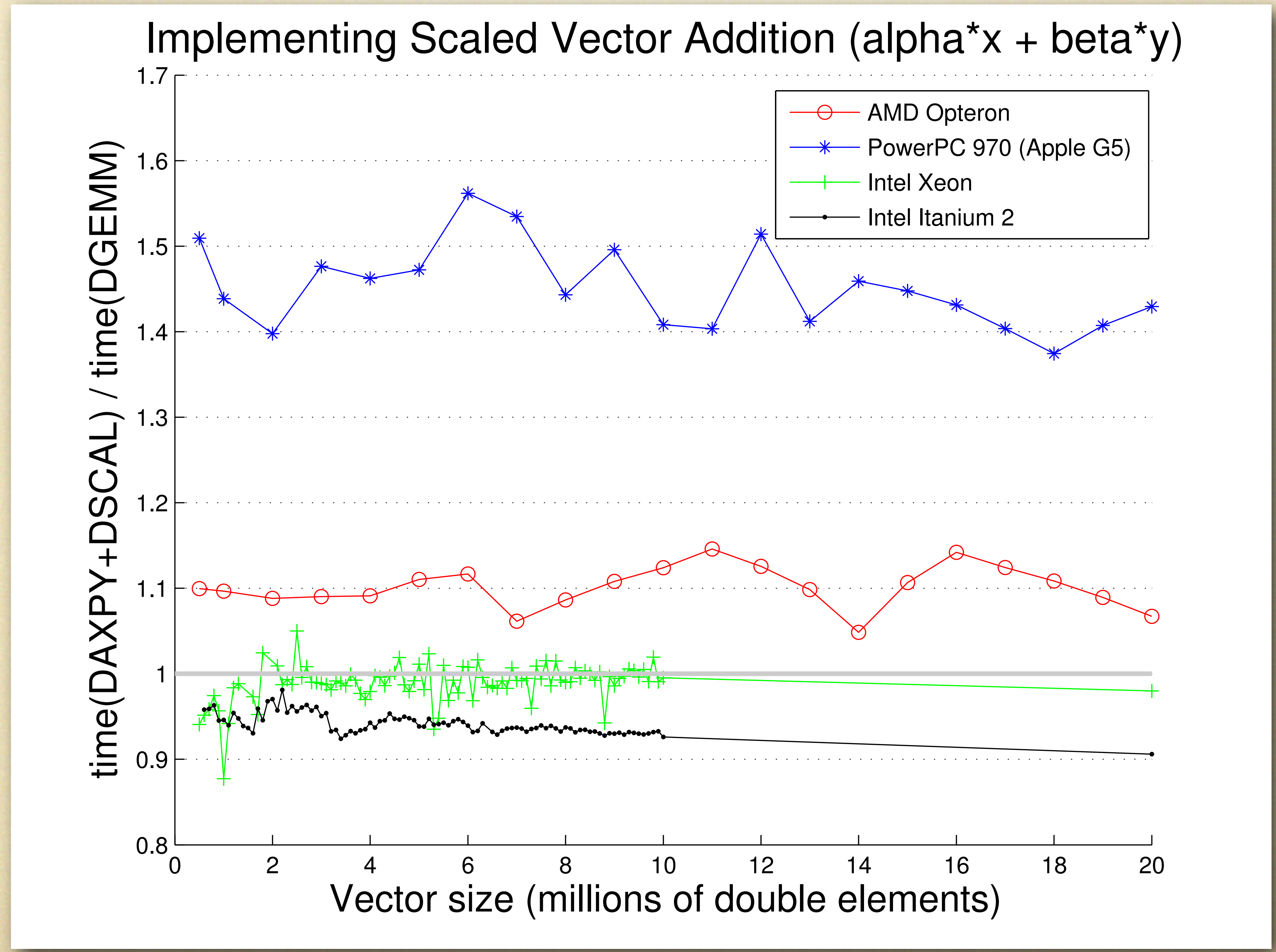
Example 1: BLAS



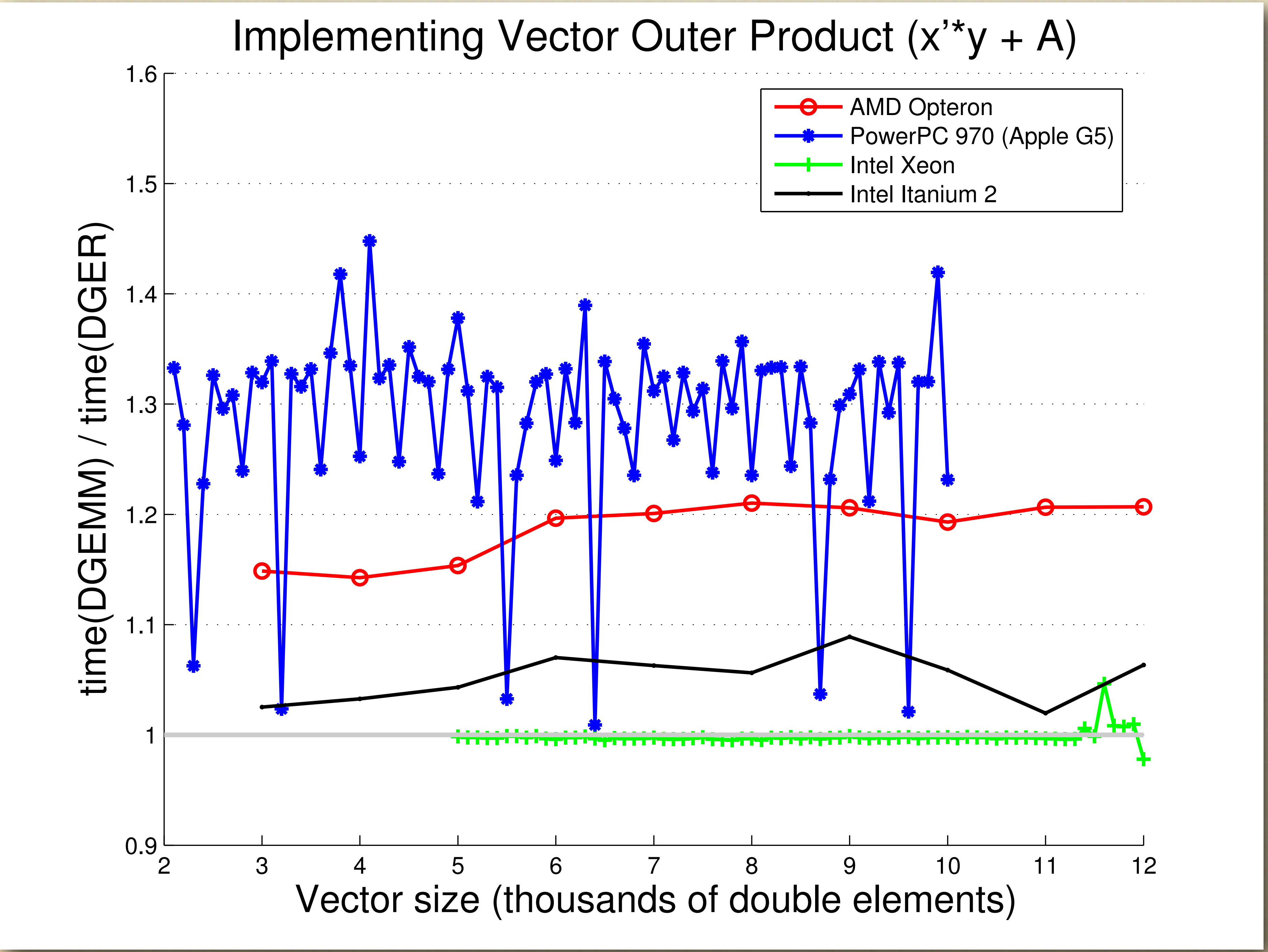
Lessons

- Minimize buffer copies
- Combine as many simple operations as possible into a single BLAS call
- Work on data-flow graph
  - simple algorithm within basic blocks
  - expanded to work globally (intra-procedurally)

Example 1: BLAS

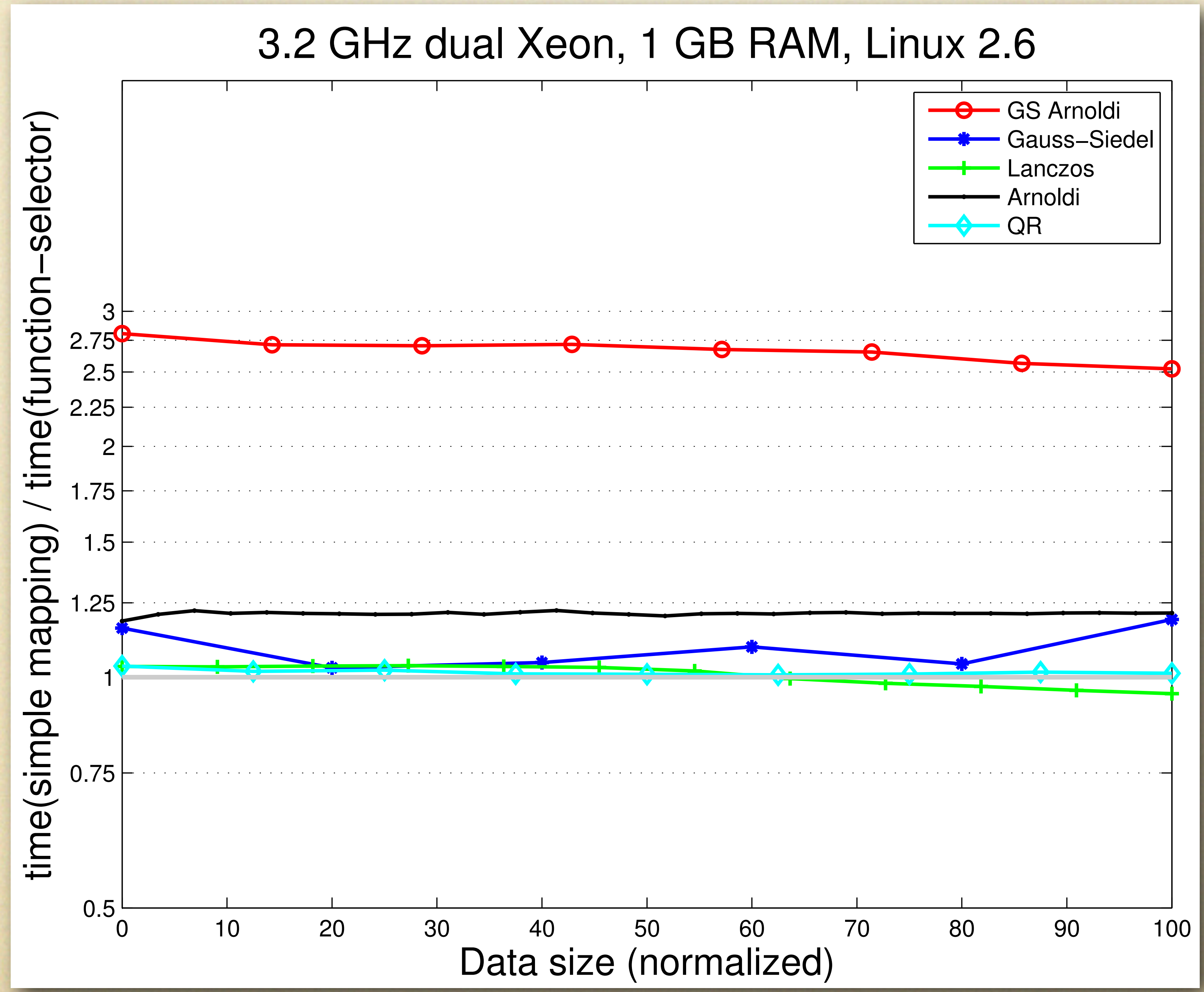


Example 1: BLAS



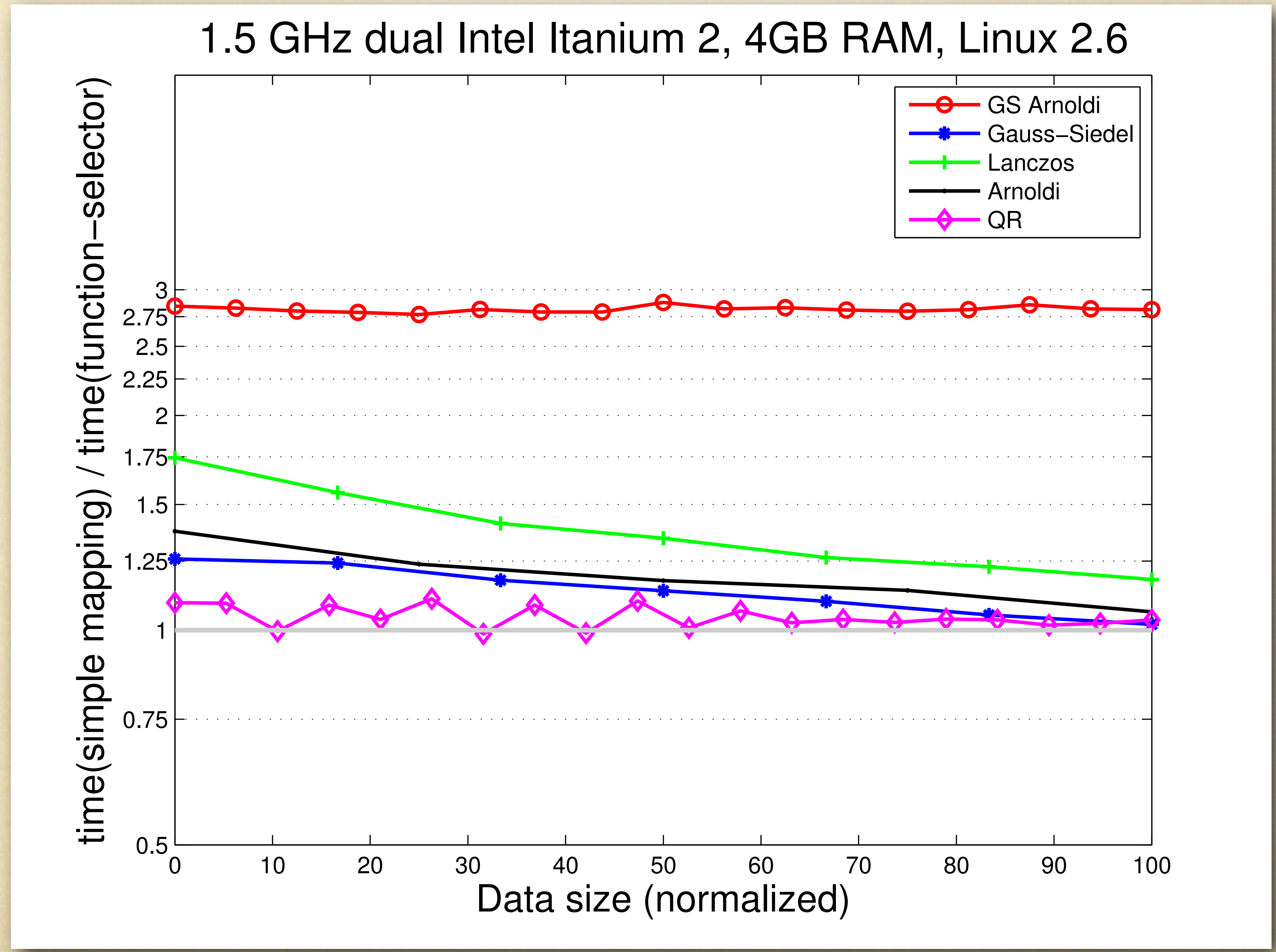
Results

POHLL (IPDPS) 2007, McFarlin and Chauhan



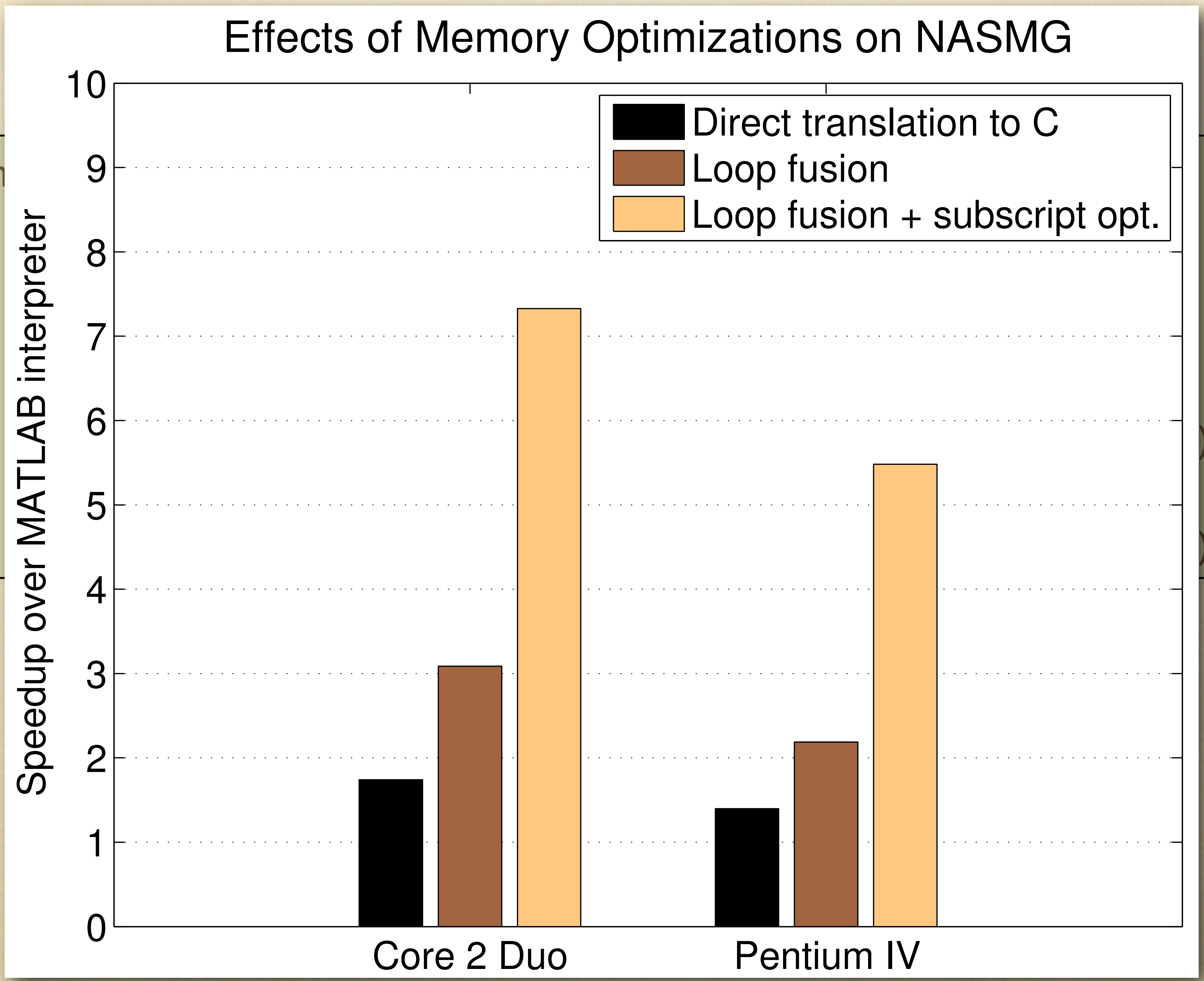
Results

POHLL (IPDPS) 2007, McFarlin and Chauhan



Example 2: Subscripts

HiPC 2009, Shei, Chauhan, and Shaw



Enabling Technology

HiPC 2009, Shei, Chauhan, and Shaw

- Type Inference
  - infer base types, and array sizes
- Leverage MATLAB / Octave interpreter
  - “concretely interpreted partial evaluation” to combine type inference and constant propagation+folding
  - type transfer functions encoded within MATLAB
- Potential for spectacular improvements
  - 100x on biology code (electron μ-scope image-proc.)
  - 1.5x on math code (ODE solver)

Type Inference Through Concrete Interpretation

```
x = 10.5;
y = [1, 2; 3, 4];
y = x * y + a;
```

Example Code

⇓

```
x$1 = 10.5;
y$1 = [1, 2; 3, 4];
t$1 = x$1 * y$1;
y$2 = t$1 + a$1;
```

After SSA & flattening

⇒

```
BT_x$1 = 'd';
x$1 = 10.5;
BT_y$1 = BXF_vertcat( ...
    BXF_horzcat('i', 'i'), ...
    BXF_horzcat('i', 'i') ...
  );
y$1 = [1, 2; 3, 4];
BT_t$1 = ...
  BXF_product(BT_x$1, BT_y$1);
t$1 = x$1 * y$1;
BT_y$2 = ...
  BXF_sum(BT_t$1, BT_a$1);
y$2 = t$1 + a$1;
```

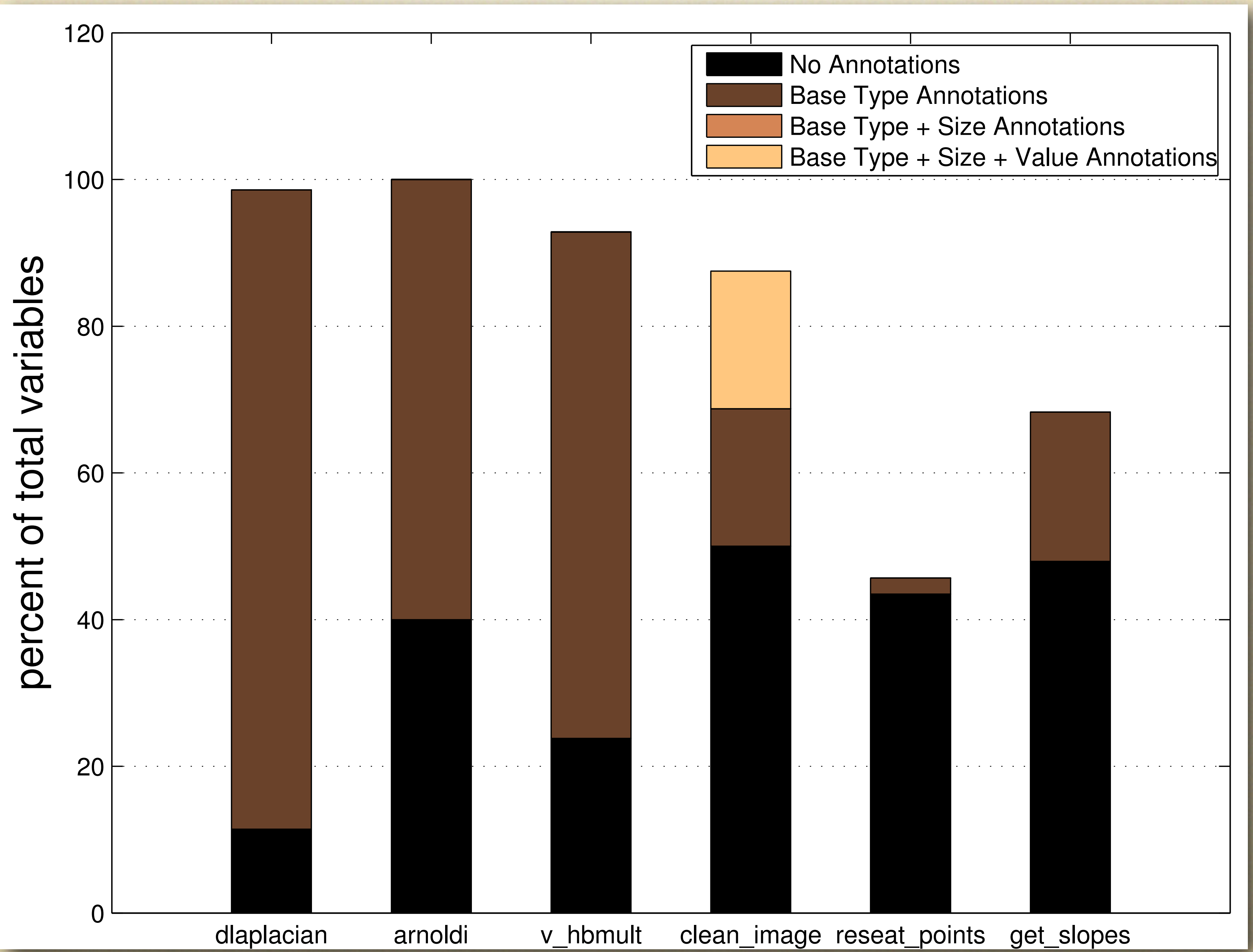
With type disambiguation code

⇒

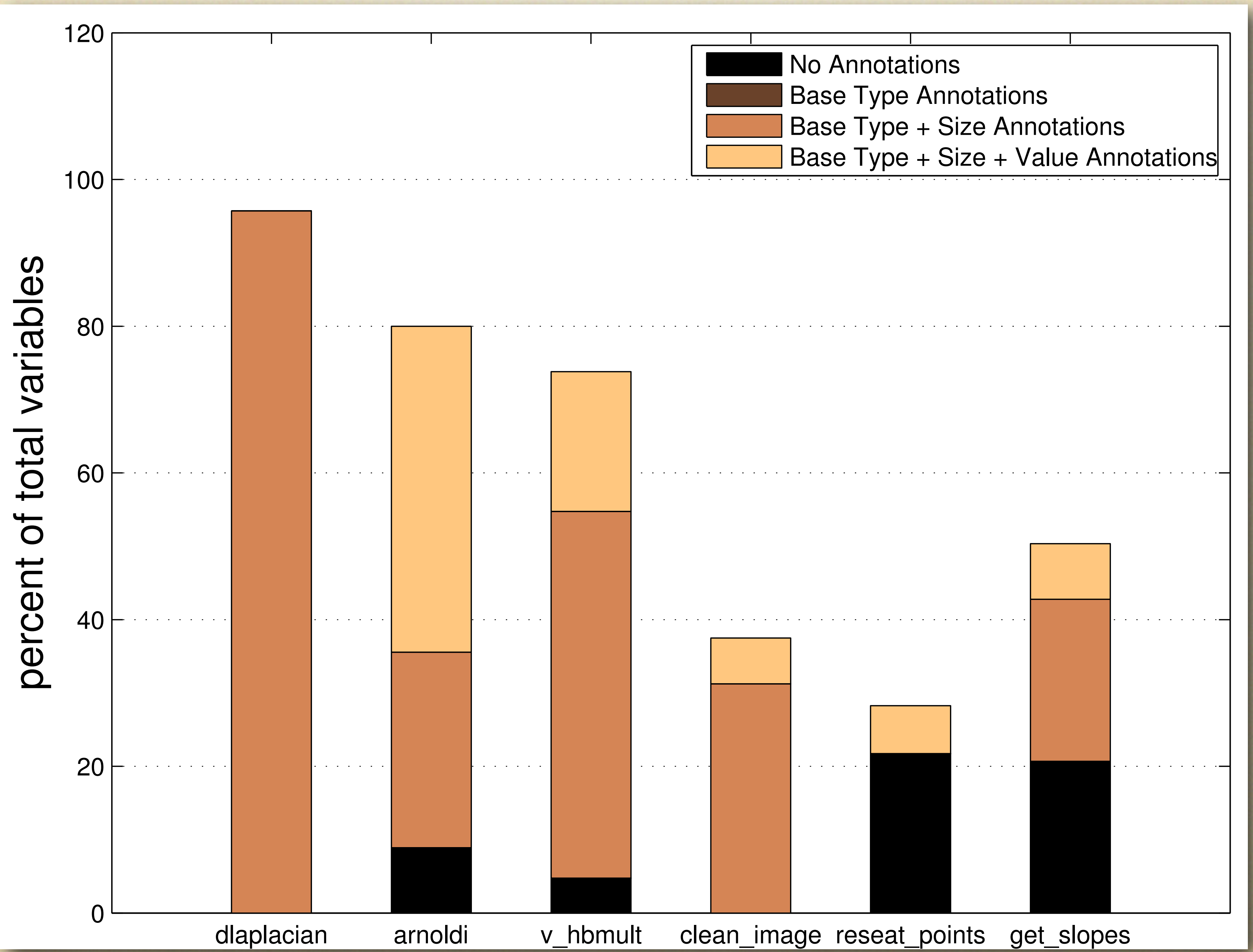
```
x$1 = 10.5;
y$1 = [1, 2; 3, 4];
t$1 = x$1 * y$1;
BT_y$2 =...
  BXF_sum(BT_t$1, BT_a$1);
y$2 = t$1 + a$1;
```

After partial evaluation

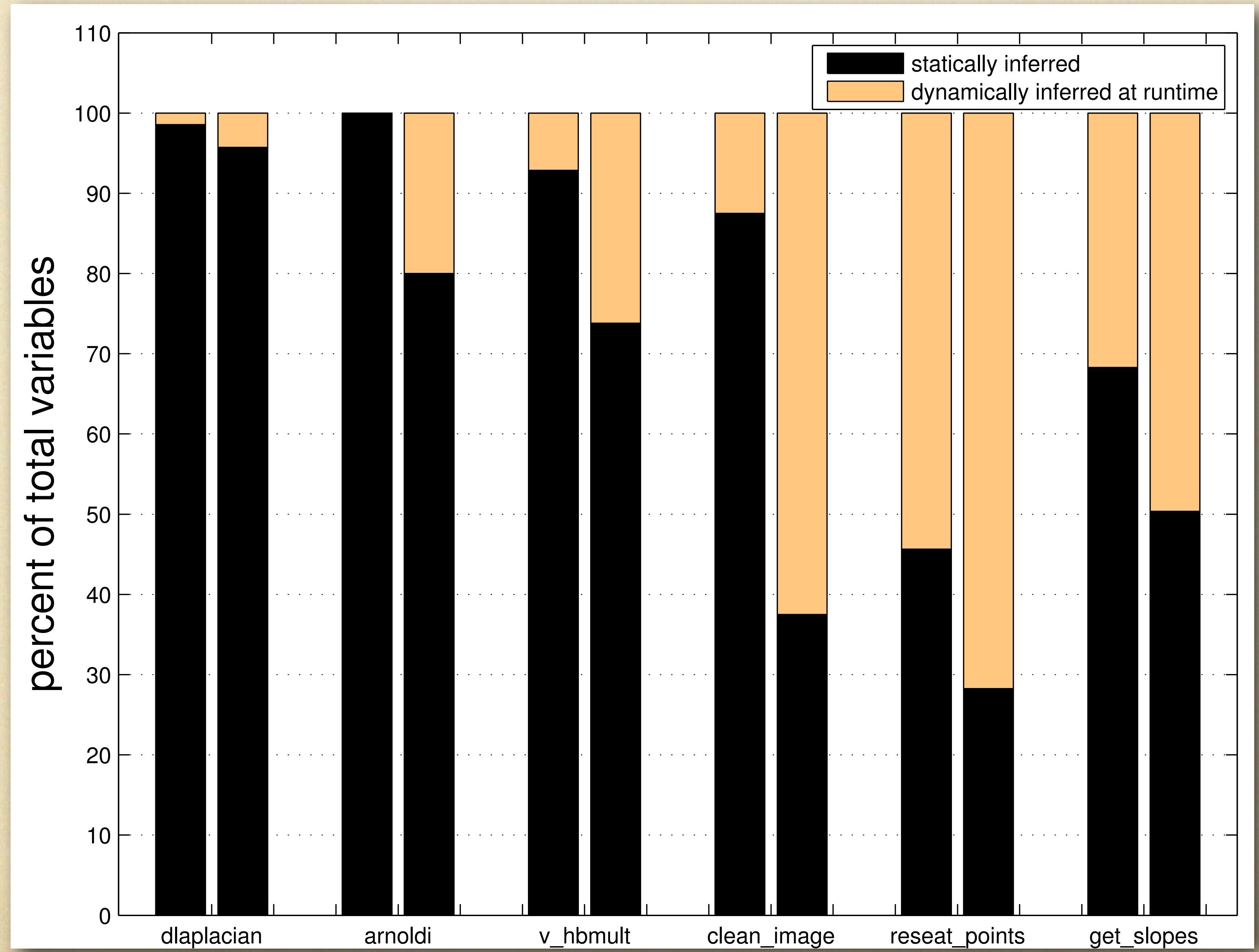
Inferring Base Types



Inferring Array Sizes



Static vs Dynamic Inference



Observations

- Memory seems to play a key role in performance of high-level dynamically type languages (studied MATLAB and Ruby)
- Lack of general-purpose analytical models to guide the compiler toward generating programs with better memory locality
  - need inter-procedural methods
  - need a way to incorporate separately-compiled libraries

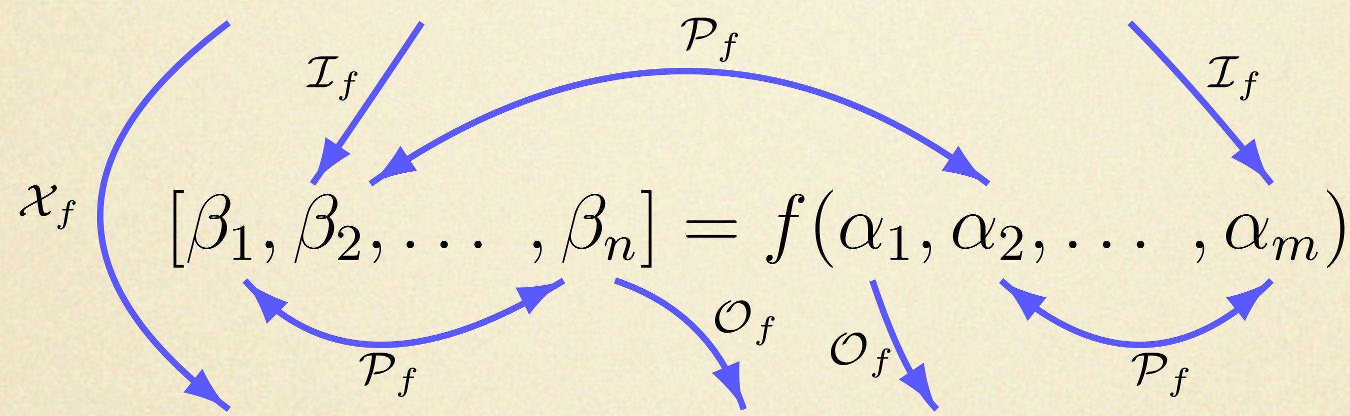
Static Reuse Distances

```
x = a + b;
c = a + d[i]*100;
y =  x * 10;
```

Static Reuse Distance = 6 (a, b, c, d, i, 100)

Definition: A reference point, *p*, is the unique syntactic reference that is either an lvalue or an rvalue. When the point is inside a loop nest a superscripted reference point $p^{\underline{i}}$ refers to the dynamic instance of *p* at the iteration vector $\underline{i}$.

SRD Across Function Calls



$\mathcal{X}_f : A \rightarrow Z$

$\mathcal{X}_f(a)$ is the volume of data accessed within f.

$\mathcal{I}_f : P \times A \rightarrow Z$

$\mathcal{I}_f(p, a)$ is the volume of data accessed within f before the first access to ρ.

$\mathcal{O}_f : P \times A \rightarrow Z$

$\mathcal{O}_f(p, a)$ is the volume of data accessed within f after the last access to ρ.

$\mathcal{P}_f : P \times P \times A \rightarrow Z$

$\mathcal{P}_f(p_1, p_2, a)$ is the volume of data accessed between the last use of ρ_1 and the first use of ρ_2 within f. It is 0 if $a(\rho_1, \rho_2)$ is `true`.

P: set of all possible reference points in a program
A: domain of alias functions such that an alias function a(p1,p2) returns true iff p1 and p2 overlap in their memory references
Z: set of integers

SRD For Regions of Code

$\mathcal{X}_f : A \rightarrow Z$
: $\mathcal{X}_f(a)$ is the volume of data accessed within f.

$\mathcal{I}_f : P \times A \rightarrow Z$
: $\mathcal{I}_f(p, a)$ is the volume of data accessed within f before the first access to ρ.

$\mathcal{O}_f : P \times A \rightarrow Z$
: $\mathcal{O}_f(p, a)$ is the volume of data accessed within f after the last access to ρ.

$\mathcal{P}_f : P \times P \times A \rightarrow Z$
: $\mathcal{P}_f(p_1, p_2, a)$ is the volume of data accessed between the last use of ρ_1 and the first use of ρ_2 within f. It is 0 if $a(\rho_1, \rho_2)$ is `true`.

$\mathcal{X}_c : A \rightarrow Z$
: $\mathcal{X}_c(a)$ is the volume of data accessed within c.

$\mathcal{I}_c : P \times A \rightarrow Z$
: $\mathcal{I}_c(p, a)$ is the volume of data accessed within c before the first execution of p.

$\mathcal{O}_c : P \times A \rightarrow Z$
: $\mathcal{O}_c(p, a)$ is the volume of data accessed within c after the last execution of p.

Algorithm to Compute I_c

Algorithm: COMPUTE $\mathcal{I}_c$

Input: code region c; reference point p; alias function a that is valid over c

Output: $\mathcal{I}_c(p, a)$

---

if $c = \boxed{c_1; c_2}$ **then**
  if $p \in \boxed{c_1}$ **then**
    return $\mathcal{I}_{c_1}(p, a)$
  else
    return $\mathcal{X}_{c_1}(a) + \mathcal{I}_{c_2}(p, a)$
else if $c = \boxed{\text{if } e \text{ then } c_1 \text{ else } c_2}$ **then**
  if $p = \boxed{e}$ **then**
    return 0
  else if $p \in \boxed{c_1}$ **then**
    return $|e| + \mathcal{I}_{c_1}(p, a)$
  else
    return $|e| + \mathcal{I}_{c_2}(p, a)$
else if $c = \boxed{\text{for i} = \text{L : S : U begin } c_1 \text{ end}}$ **then**
  if $p \in \boxed{\text{for i} = \text{L : S : U}}$ **then**
    return 0
  else
    $\bar{k} \leftarrow$ first iteration vector in which p is reached
    $\bar{k}' \leftarrow$ largest iteration vector smaller than $\bar{k}$
    $r \leftarrow \Sigma_{p_1, p_2 \in c_1} |p_1 \xrightarrow{c} p_2|,\ p_1 \xrightarrow{c} p_2 \in \text{polytope}(c, \bar{k}')$
    `/*` $\xrightarrow{c}$ `denotes loop carried dependence */`
    return $\Sigma_{\bar{\imath} < \bar{k}} \mathcal{X}^{\bar{\imath}}_{c_1}(a) + \mathcal{I}^{\bar{k}}_{c_1}(p, a) - r$
else
  ERROR

Algorithm to Compute X_c

Algorithm: COMPUTE $\mathcal{X}_c$

Input: code region c; alias function a that is valid over c; probability weights, π, on CFG edges

Output: $\mathcal{X}_c(a)$

---

if $c = \boxed{c_1; c_2}$ **then**

  return $\mathcal{X}_{c_1}(a) + \mathcal{X}_{c_2}(a)$

else if $c = \boxed{\text{if } e \text{ then } c_1 \text{ else } c_2}$ **then**

  return $|e| + \pi(\text{true}) \times \mathcal{X}_{c_1}(a) + \pi(\text{false}) \times \mathcal{X}_{c_2}(a)$

else if $c = \boxed{\text{for i} = \text{L} : \text{S} : \text{U begin } c_1 \text{ end}}$ **then**

  $\bar{n} \leftarrow$ iteration vector for the last iteration

  $r = \Sigma_{p_1, p_2 \in c_1} |p_1 \xrightarrow{c} p_2|$, $p_1 \xrightarrow{c} p_2 \in \text{polytope}(c, \bar{n})$

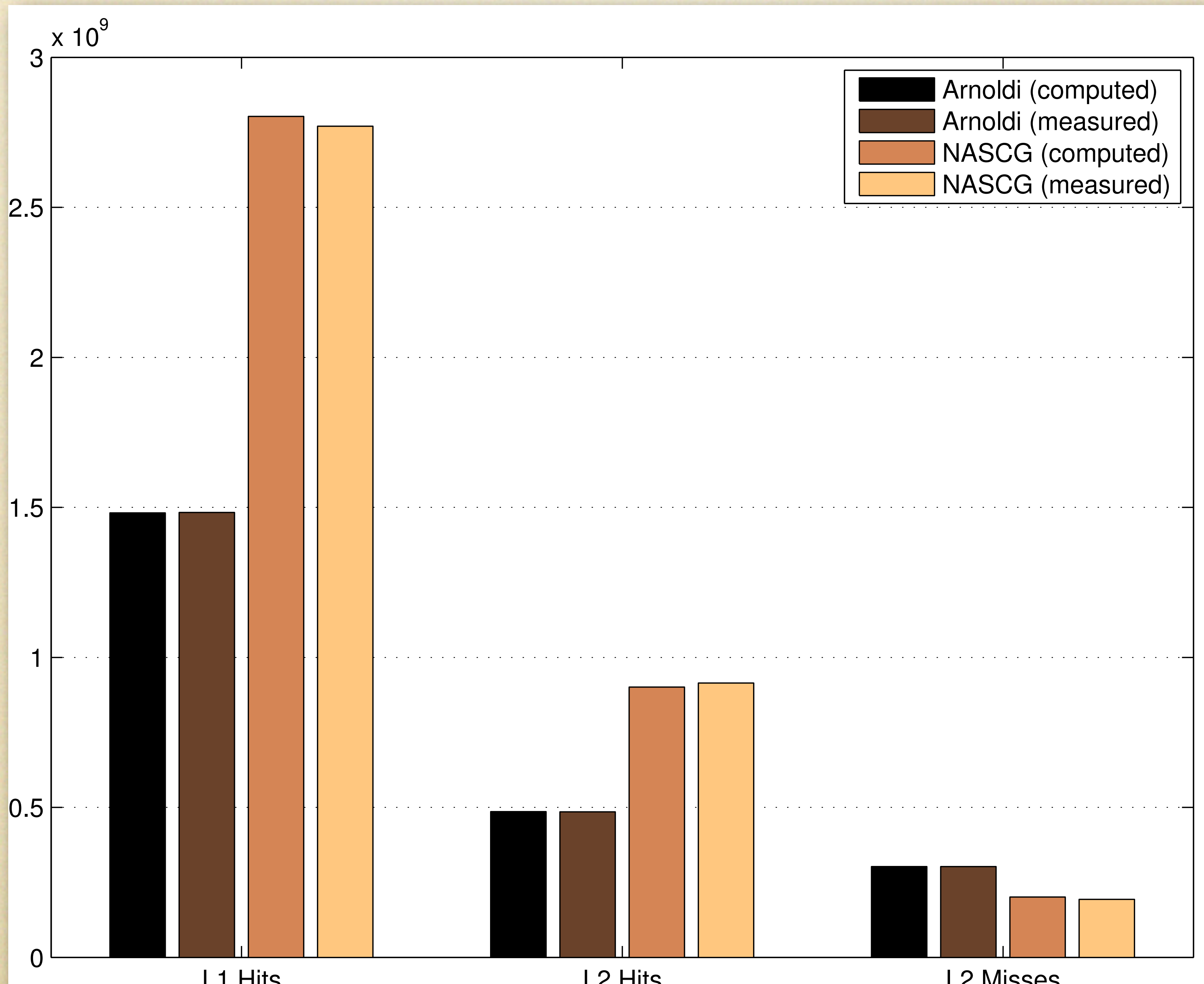
  `/*` $\xrightarrow{c}$ `denotes loop carried dependence */`

  return $\Sigma_{\bar{\imath} \leq \bar{n}} \mathcal{X}^{\bar{\imath}}_{c_1}(a) - r$

else

  ERROR

Accuracy


x 10^9
3
2.5
2
1.5
1
0.5
0
Arnoldi (computed)
Arnoldi (measured)
NASCG (computed)
NASCG (measured)
L1 Hits
L2 Hits
L2 Misses

Empirical Data for Library Functions

$\mathcal{I}_f$ $\mathcal{P}_f$ $\mathcal{I}_f$

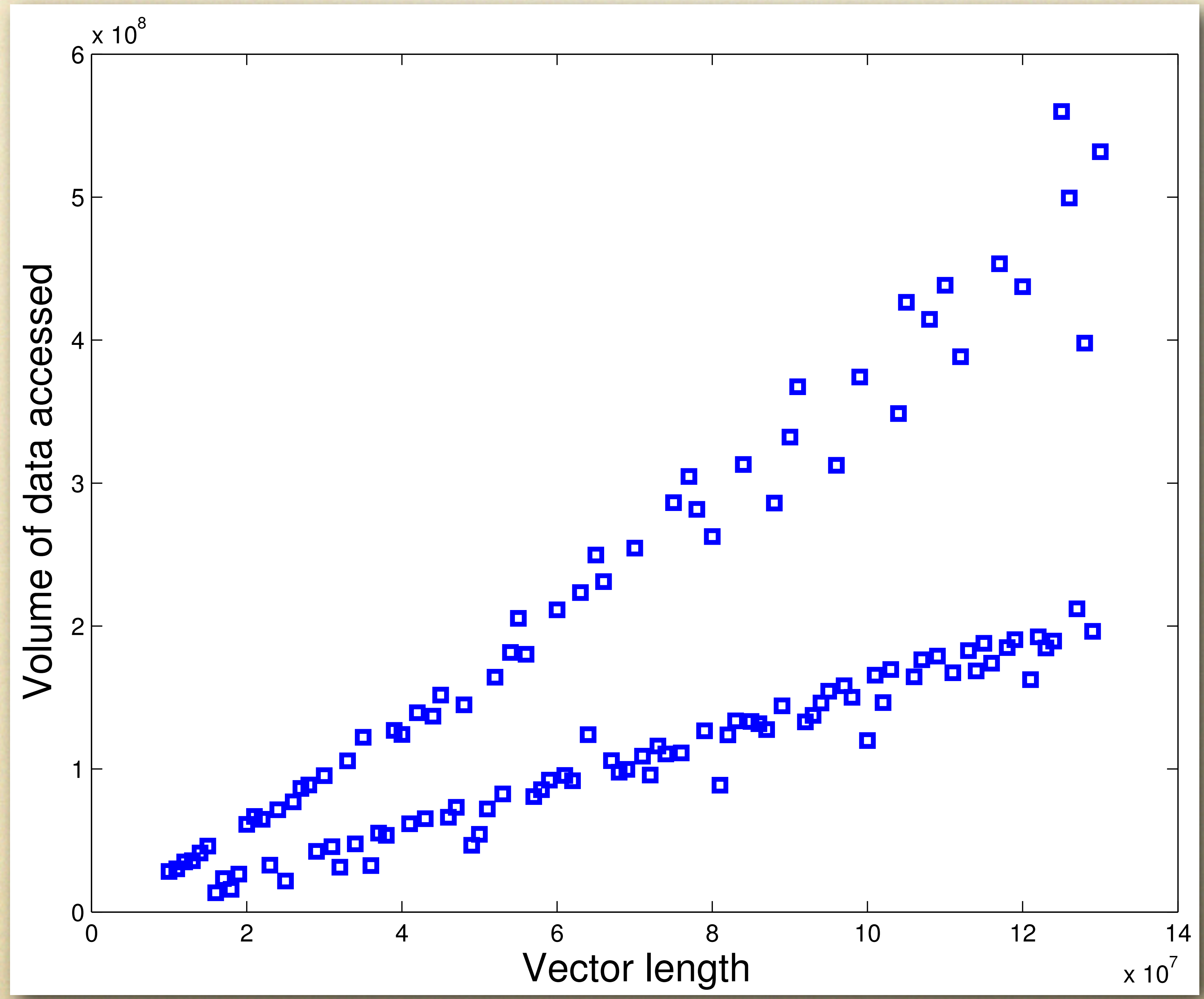
$$\mathcal{X}_f \quad [\beta_1, \beta_2, \ldots\ , \beta_n] = f(\alpha_1, \alpha_2, \ldots\ , \alpha_m)$$

$\mathcal{P}_f$ $\mathcal{O}_f$ $\mathcal{O}_f$ $\mathcal{P}_f$

$$X_f \approx \text{L2 cache misses}$$

$$\mathcal{I}_f \approx \mathcal{O}_f \approx \mathcal{X}_f$$

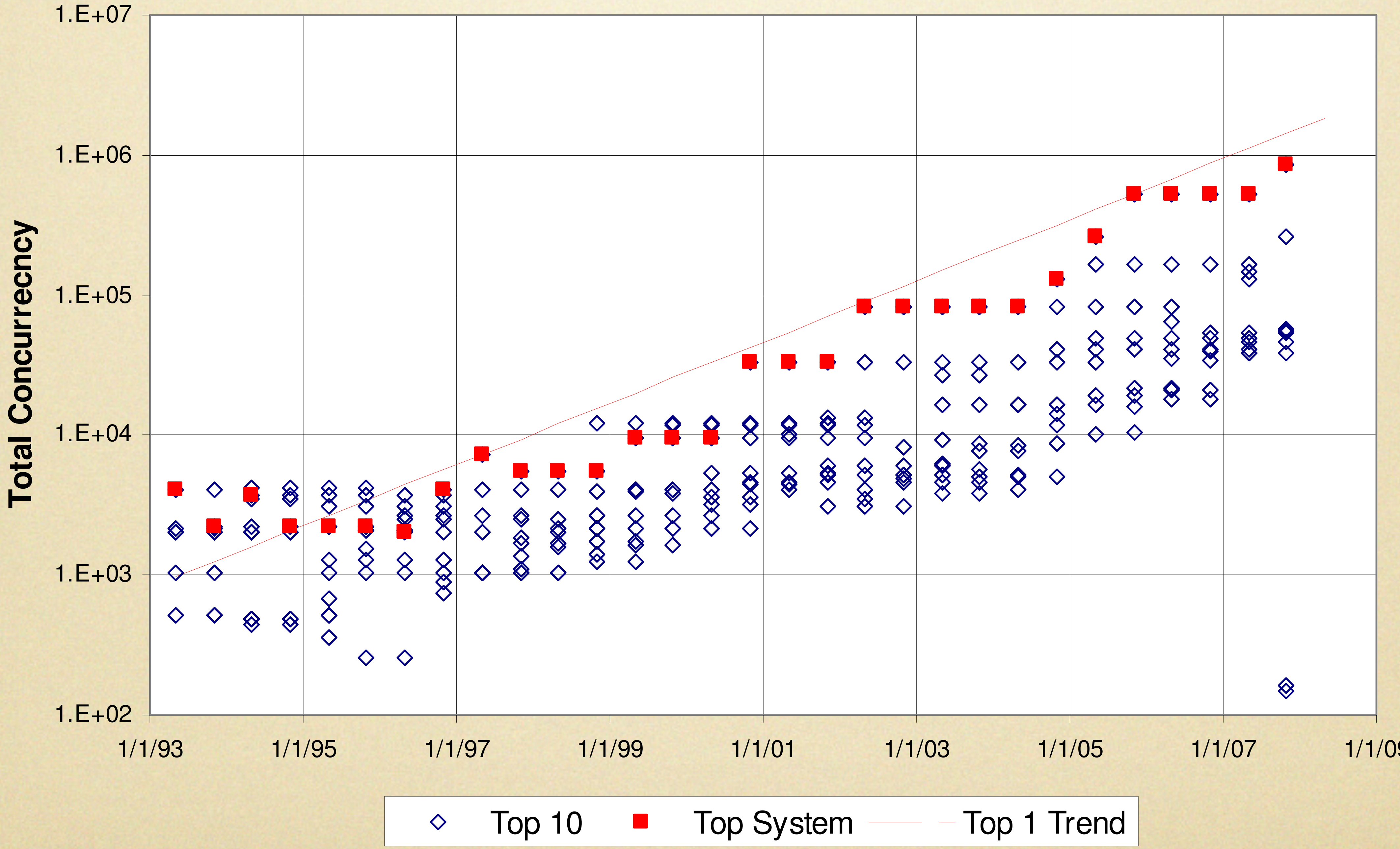
Challenges Remain: FFT



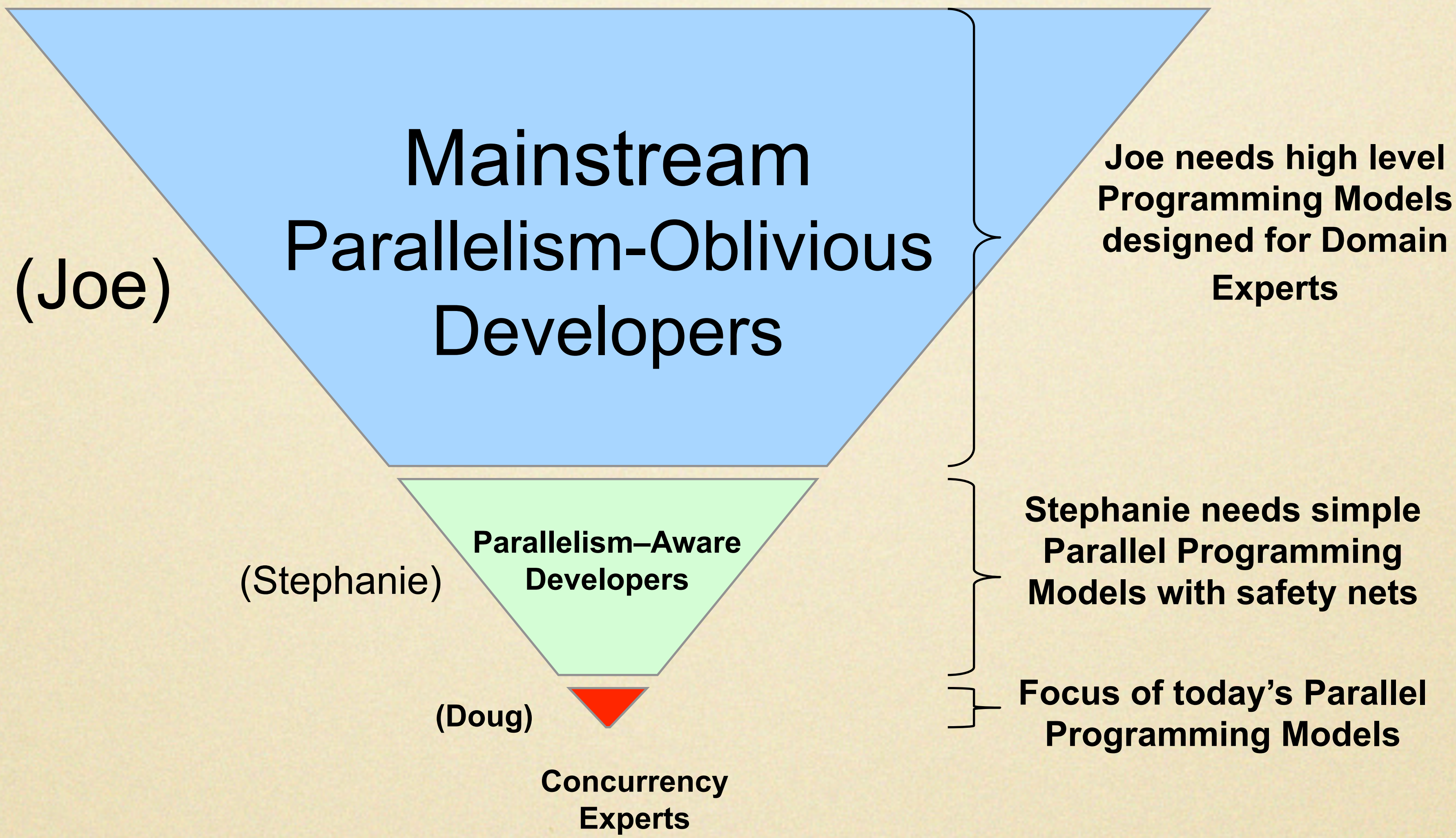
Rescuing Parallel Programmers

Concurrency Trends

(ExaScale Computing Study, Peter Kogge et al.)



Types of (Parallel) Programmers



Courtesy: Vivek Sarkar, Rice University

Parallelism Oblivious Users

- Programming languages-driven
  - implicit parallelism, compiler support
- OS-driven
  - innovative solutions to leverage extra cores
- Architecture-driven
  - ILP, hyper-threading

Observations for Parallelism-Aware and Expert Users

- Completely automatic parallelization has had limited success
- Writing parallel programs is hard; optimizing and maintaining them is harder!
- Compilation technology has worked well in communication optimization

Declarative Parallel Programming

- Let users write parallel programs
- Let compilers optimize parallel programs
- Separate computation and communication specification, using a domain-specific language to specify communication
- Key insight: most parallel applications have predictable (but not necessarily static) communication patterns

Declarative Specification of Communication

ICPP 2009, Hoefler et al.

```
@collective cshift (A)
{
  foreach processor i
  {
    A@i := A@(i+1)
  }
}
```

Compiler converts collectives to MPI calls and optimizes communication by coalescing and overlapping with computation

```
@collective stencil (A, B)
{
  foreach processor i in Mesh2D
  {
    B@i := 0.25*(A@i.N + A@i.S + A@i.W + A@i.E)
  }
}
```

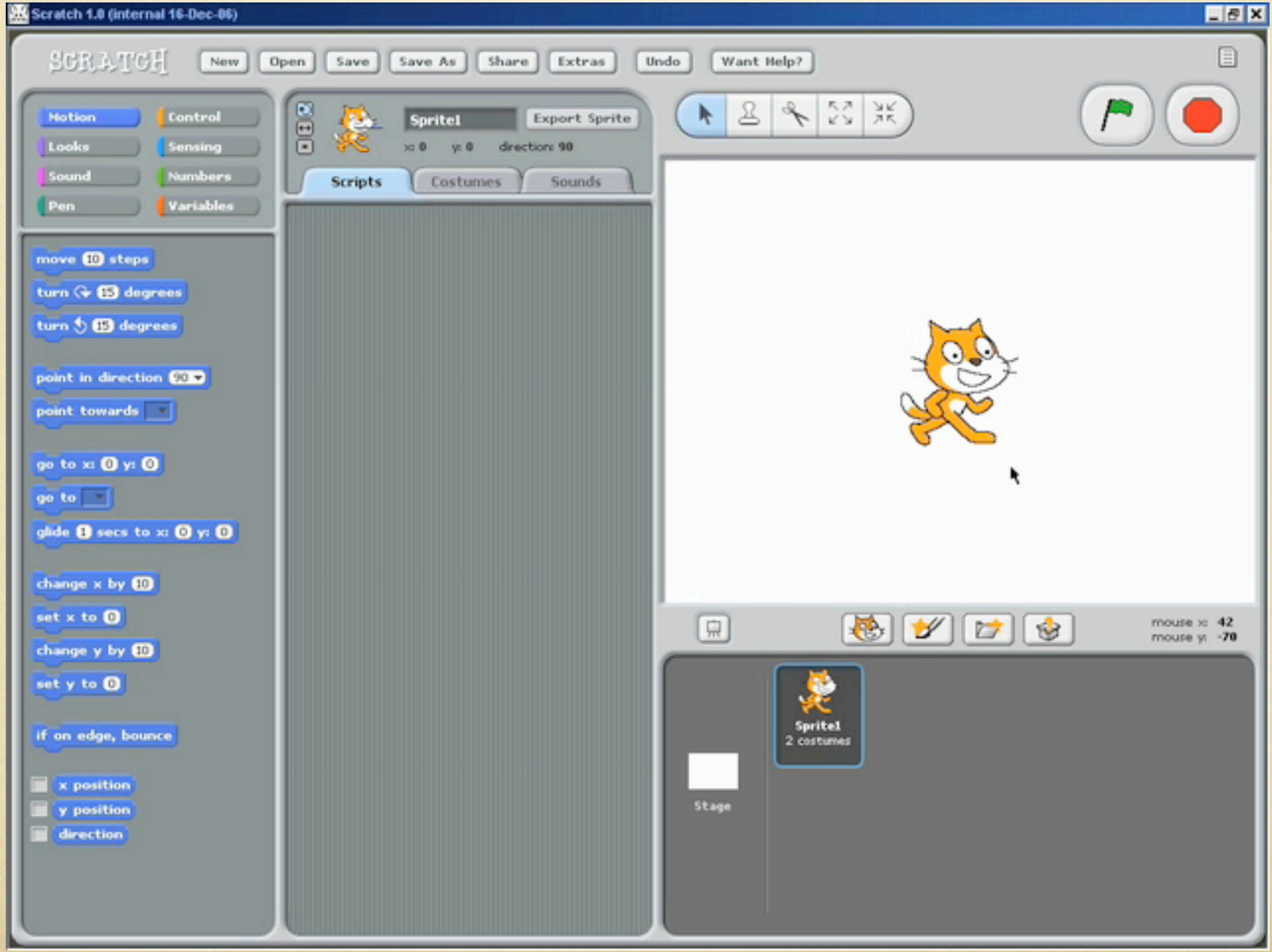
Concluding Remarks

- Computing is a core technique in an increasing number of fields
  - programming is no longer restricted to scientists and engineers
  - conventional programming models are inadequate
- Parallelism is no longer restricted to scientific and engineering applications
  - need to address the needs of different types of users and applications
- Traditional program analysis is inadequate on modern machines

Other Interests

- High-level Languages
  - Ruby
- Heterogeneous parallel computing
- Large memory-footprint applications
- Automatic parallelization

Scratch
http://scratch.mit.edu/



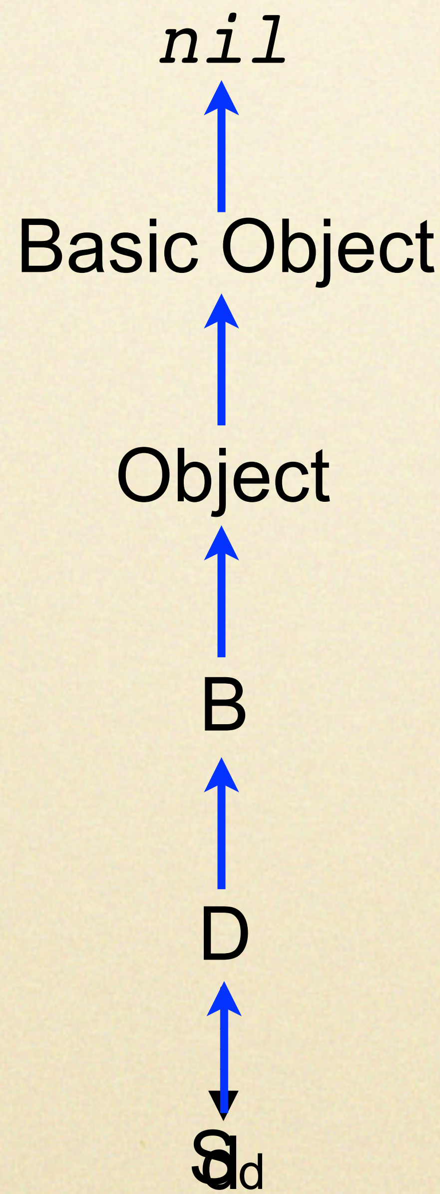
http://www.cs.indiana.edu/~achauhan

http://phi.cs.indiana.edu/

Bonus Material

Ruby Class Hierarchy

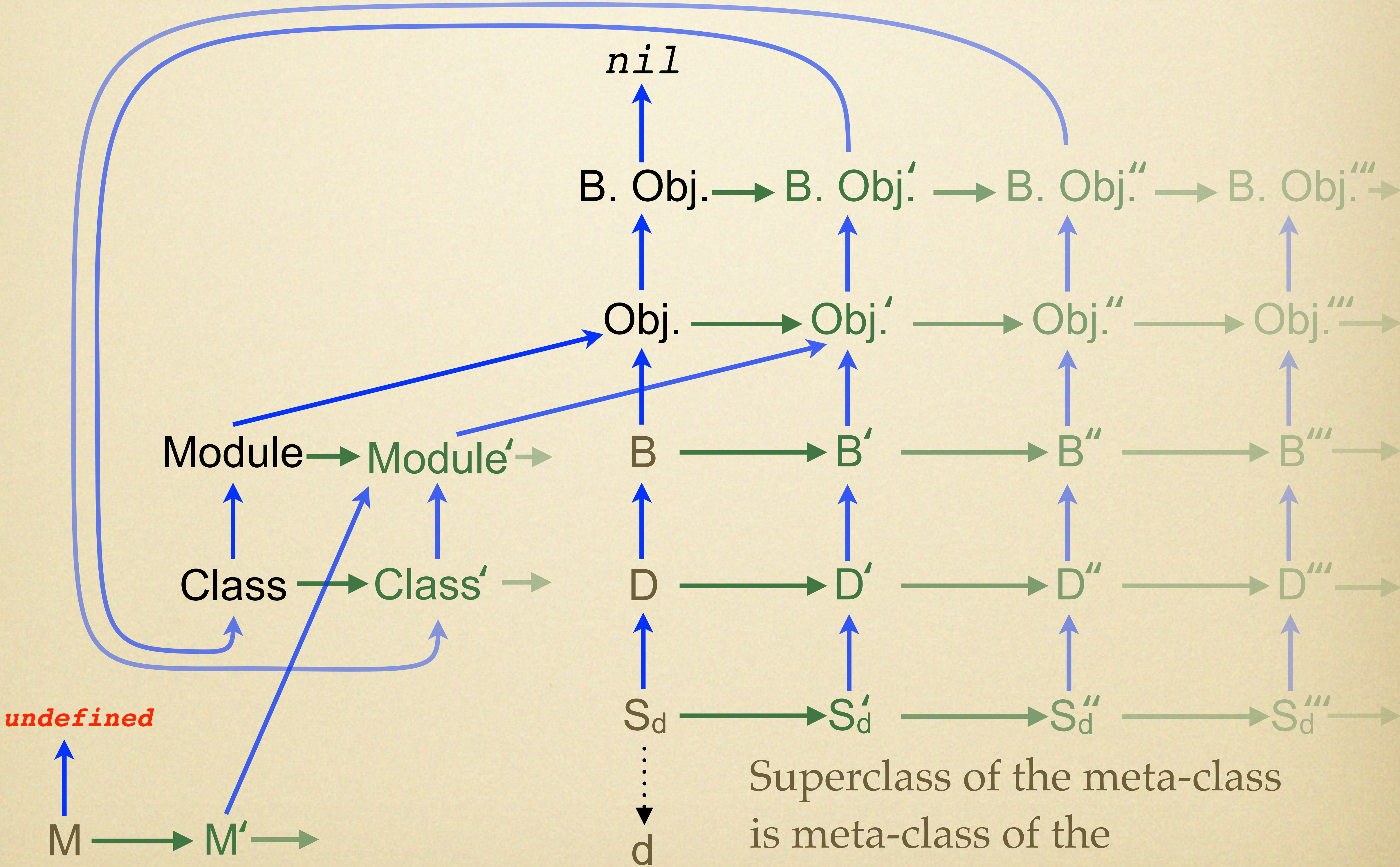
```
class B < Object
   ...
end

class D < B
   ...
end

d = D.new
def d.newMeth
   ...
end
```



Objects store values

Classes store methods

Ruby Classes as Objects



Superclass of the meta-class is meta-class of the superclass