Beating Hand-Tuned Assembly

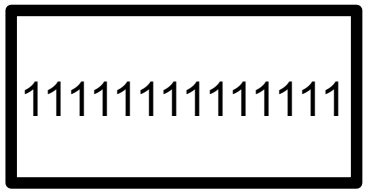
David Richardson

d.g.richardson@gmail.com

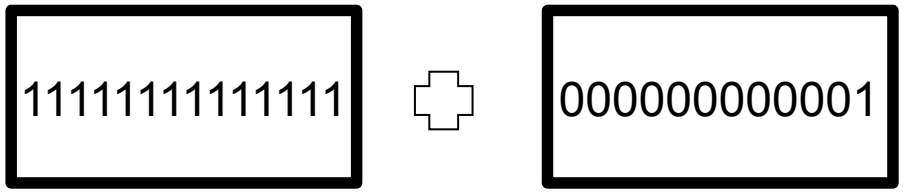
Outline

- Problem
- Hand tuned assembly
- Faster than assembly
- All optimization is like this

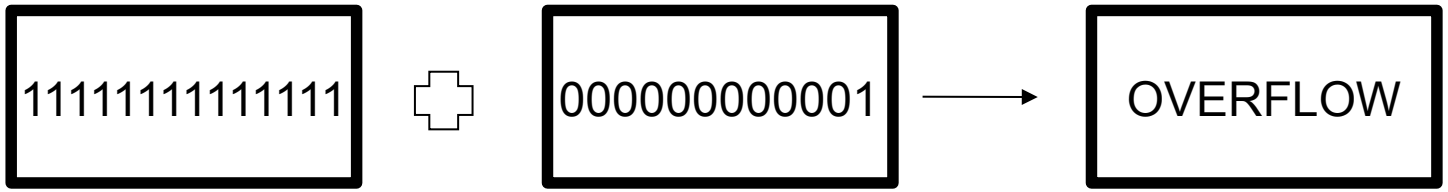
Multi Precision Arithmetic



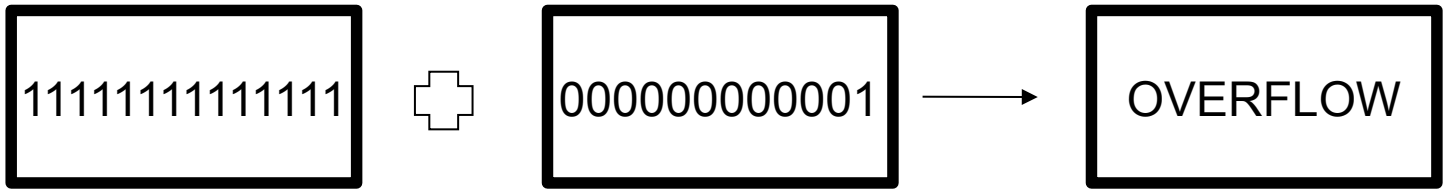
Multi Precision Arithmetic

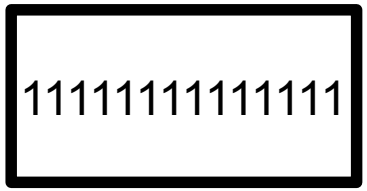


Multi Precision Arithmetic

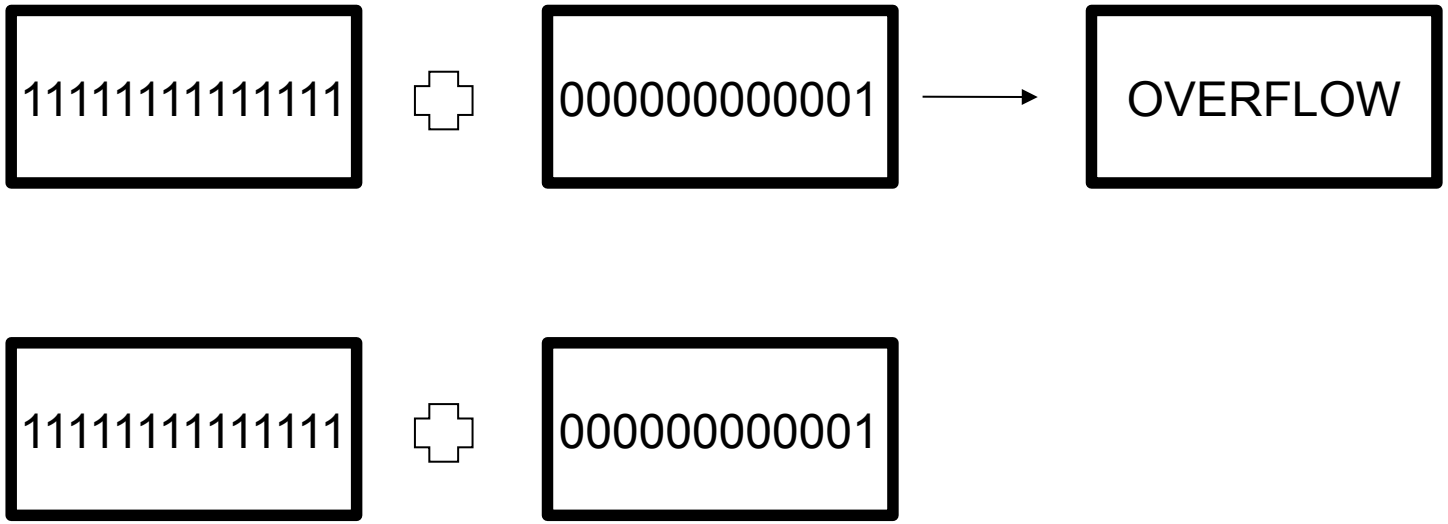


Multi Precision Arithmetic

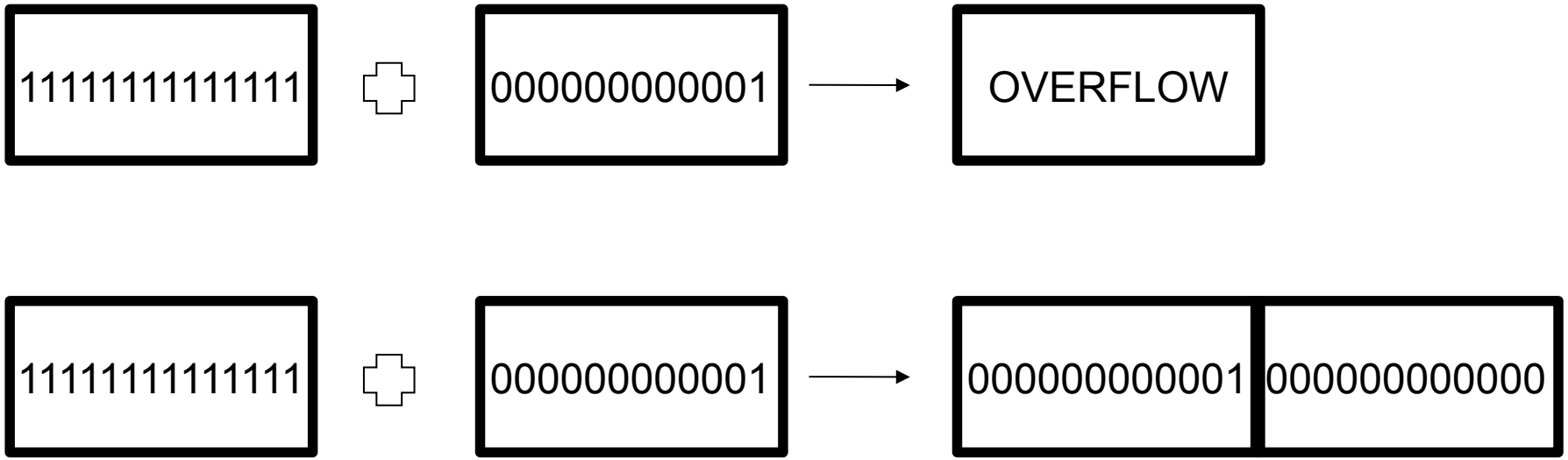




Multi Precision Arithmetic



Multi Precision Arithmetic



Multi Precision Arithmetic

- Core primitive of
  - Computer algebra
  - Cryptography
  - Computational Geometry
  - Computing billions of digits of Pi
- Plays a huge role in
  - Speed
  - Memory consumption
  - Algorithm cross over points

The GNU Multiple Precision Arithmetic Library (GMP)

GMP is carefully **designed to be as fast as possible**, both for small operands and for huge operands. The speed is achieved by using fullwords as the basic arithmetic type, by using fast algorithms, with highly optimised assembly code for the most common inner loops for a lot of CPUs, and by a general emphasis on speed.

https://gmplib.org/

GMP

- Performance claims are actually true
- Actively maintained for new CPUs
- Dual licensed under LGPLv3 and GPLv2
- “Gold Standard” of multi precision
  - Especially on unix
  - Although it is “known to work” on Windows

(some) GMP users

- Maple
- Mathematica
- Sage
  - Python computer algebra system
  - Asked to use our “faster than GMP” implementation
- Magma
- Gnu Compiler Collection
- Glasgow Haskell Compiler

What does optimized mean?

This was generated by the Sun C compiler. It runs at 22 cycles/limb on the UltraSPARC-1/2, **three cycles slower than theoretically possible** for optimal code using the same algorithm. For 1-3 limbs, a special loop was generated, which causes performance problems in particular for 2 and 3 limbs. Ultimately, this should be replaced by hand-written code in the same software pipeline style as e.g., addmul_1.asm.

sqr_diagonal.asm

What should you really do?

Instruction classification (as per UltraSPARC-1/2 functional units):

- 8 FM
- 10 FA
- 12 MEM
- 10 ISHIFT + 14 IADDLOG
- 1 BRANCH
- 55 insns totally (plus one mov insn that should be optimized out)

The loop executes 56 instructions in 14 cycles on UltraSPARC-1/2, i.e **we sustain the peak execution rate of 4 instructions/cycle**.

addmul_1.asm

What is the algorithm?

Algorithm: We use **eight floating-point multiplies** per limb product, with the invariant v operand split into four 16-bit pieces, and the up operand split into 32-bit pieces. We sum pairs of 48-bit partial products using floating-point add, then convert the four 49-bit product-sums and **transfer them to the integer unit**.

addmul_1.asm

Can you beat that?

- They know
  - The hardware
  - The best published algorithms
  - Some algorithms they invented
- They been at this since at least 1991
- They think they are ~10-20% from optimal

A weakness?

- They are general
  - Good for large or small inputs
- Maybe they gave something up

Cross Over Points

- Frequently, there are multiple algorithms that do the same thing…
- But with different time complexity
- In real life, constants matter
- Optimization: pick your algorithm for your input

Example: Integer Multiplication

- Classical — n^2
- (1960) Karatsuba — n^1.585
- (1966) Toom–Cook — n^1.465
- (1971) Schönhage–Strassen — n log n log log n
  - GMP: 33,000 to 150,000 decimal digits
- (2007) Fürer's algorithm — n log n 2^O(log*n)
  - Watch out at conferences

GMP is good at cross over

- They didn't leave much on the table
  - Length check
  - Jump
- We're not going to beat them here

What problem is GMP solving?

- They only do a single multi precision arithmetic operation
  - +, -, *, /
  - two operands
- They are really good at this

What do people do with GMP?

- Implement algorithms that perform many arithmetic operations.

- They care about the entire set of operations, not any one of them.

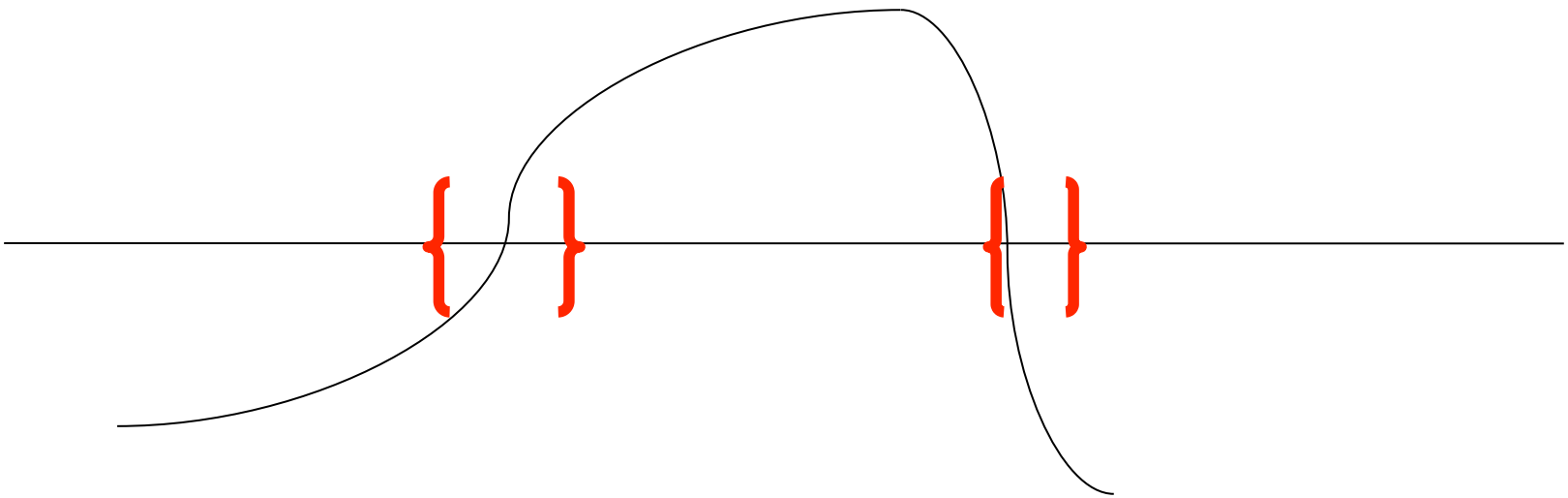
An opening?

- What if we try to optimize an algorithm?
- One that can use single arithmetic operations as primitives

Root Isolation

Input: a polynomial

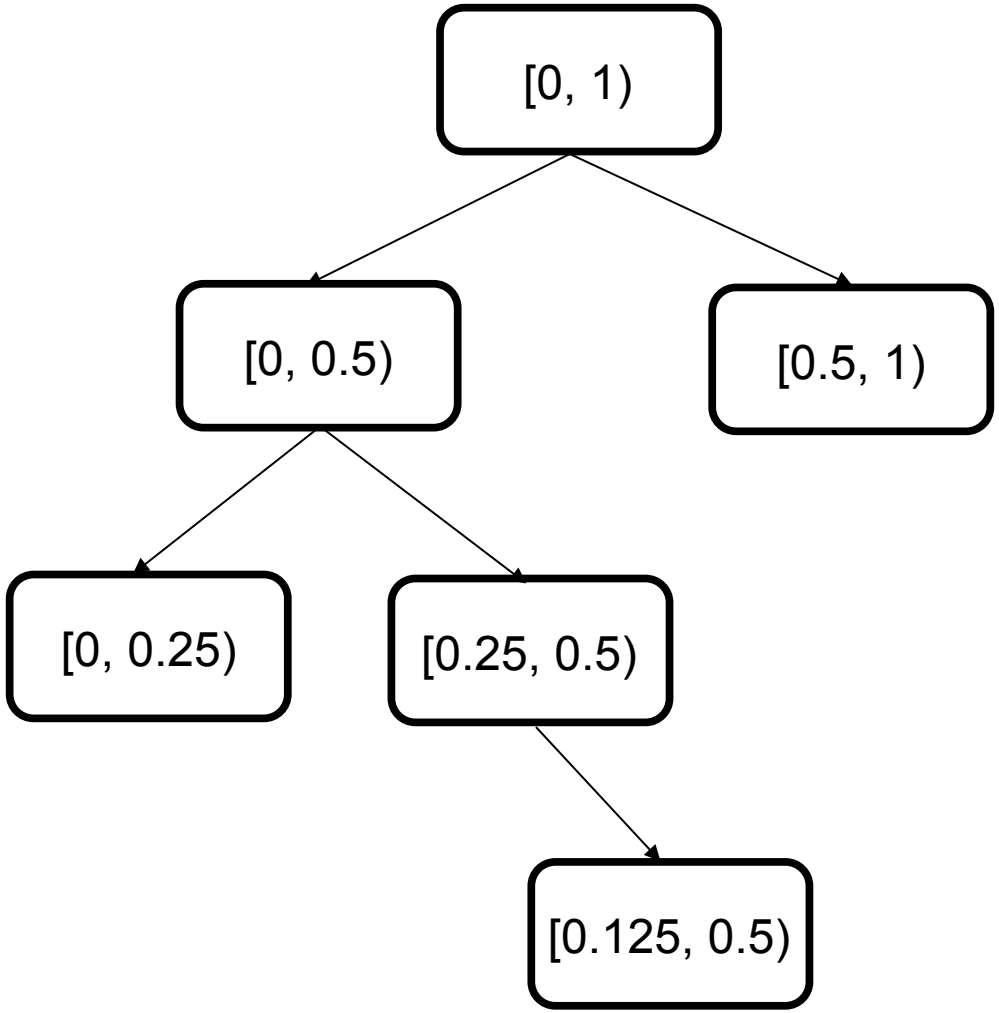
Output: set of isolating intervals



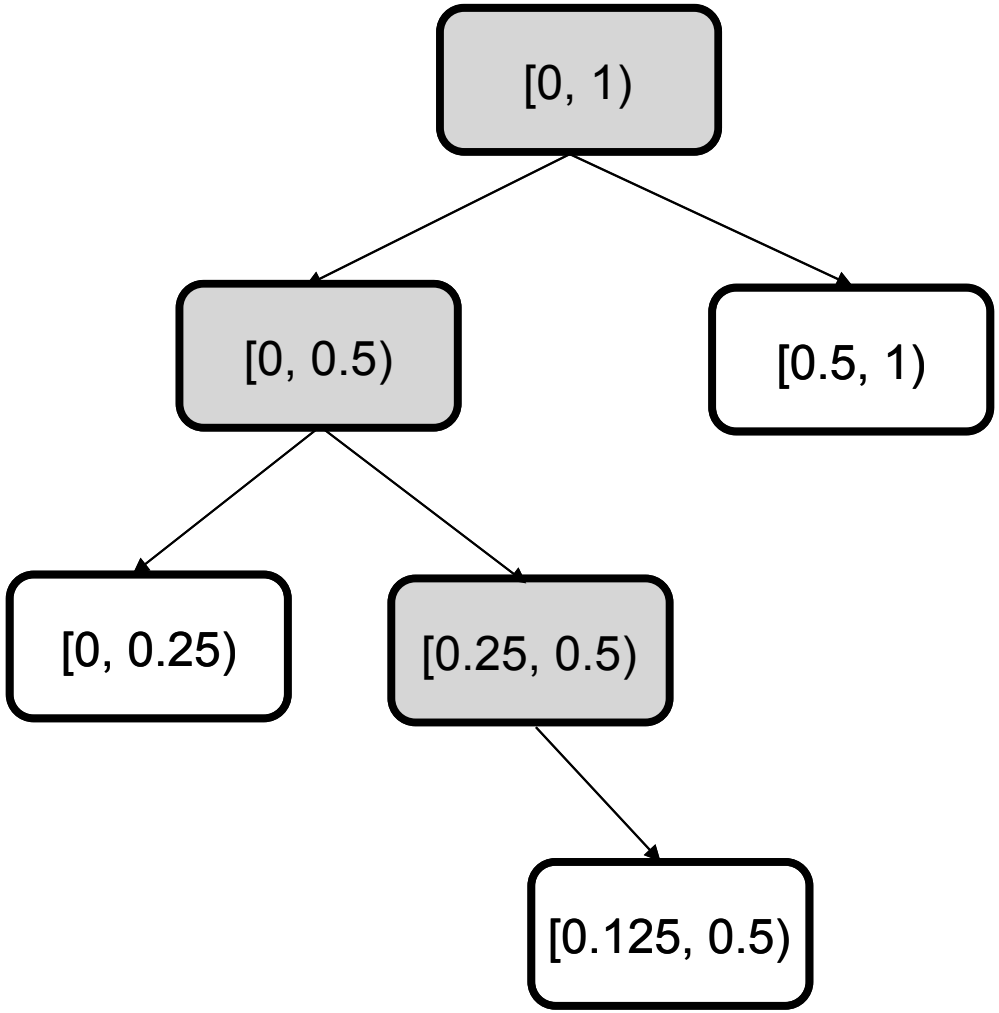
Root Isolation

- Common in science and computer algebra
- Important enough to be worth optimizing
- Exact integer coefficients give lots of multi precision arithmetic
- People usually use GMP to implement it

The Descartes Method

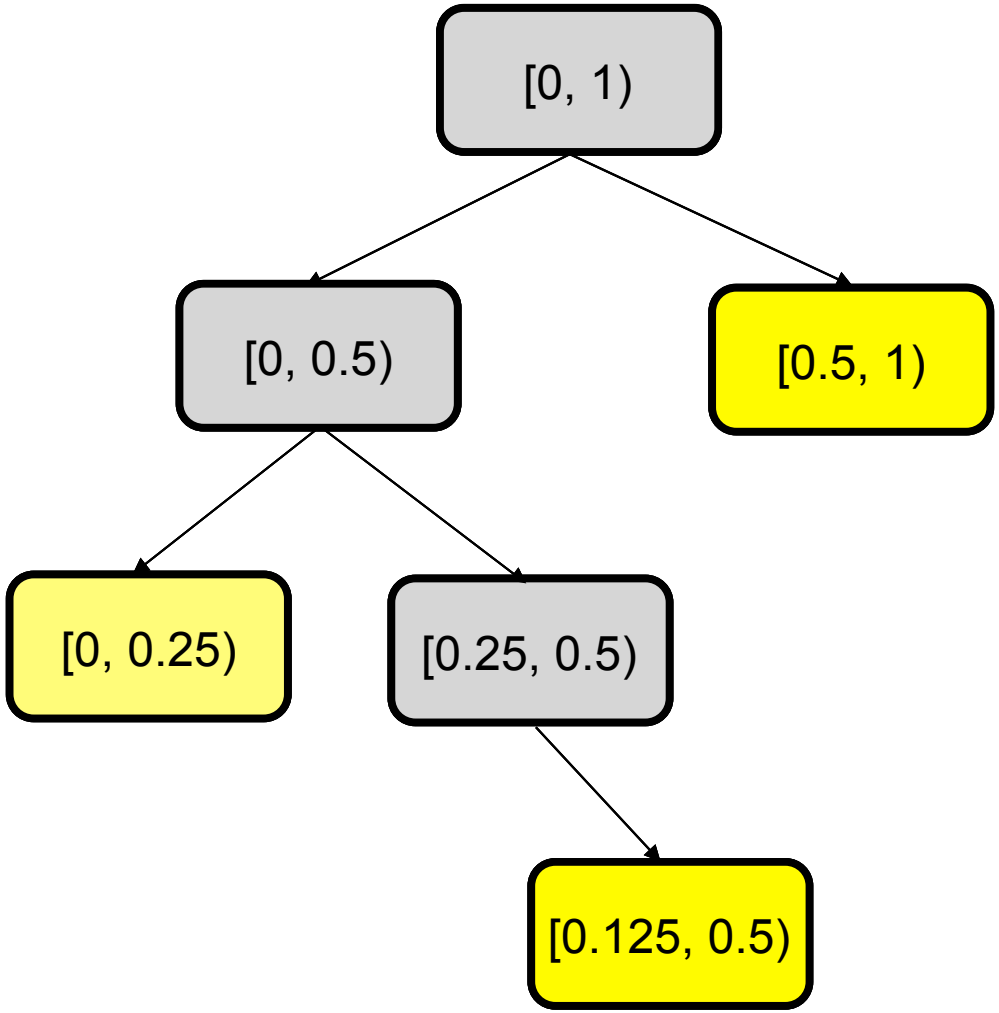


The Descartes Method



Many roots

The Descartes Method

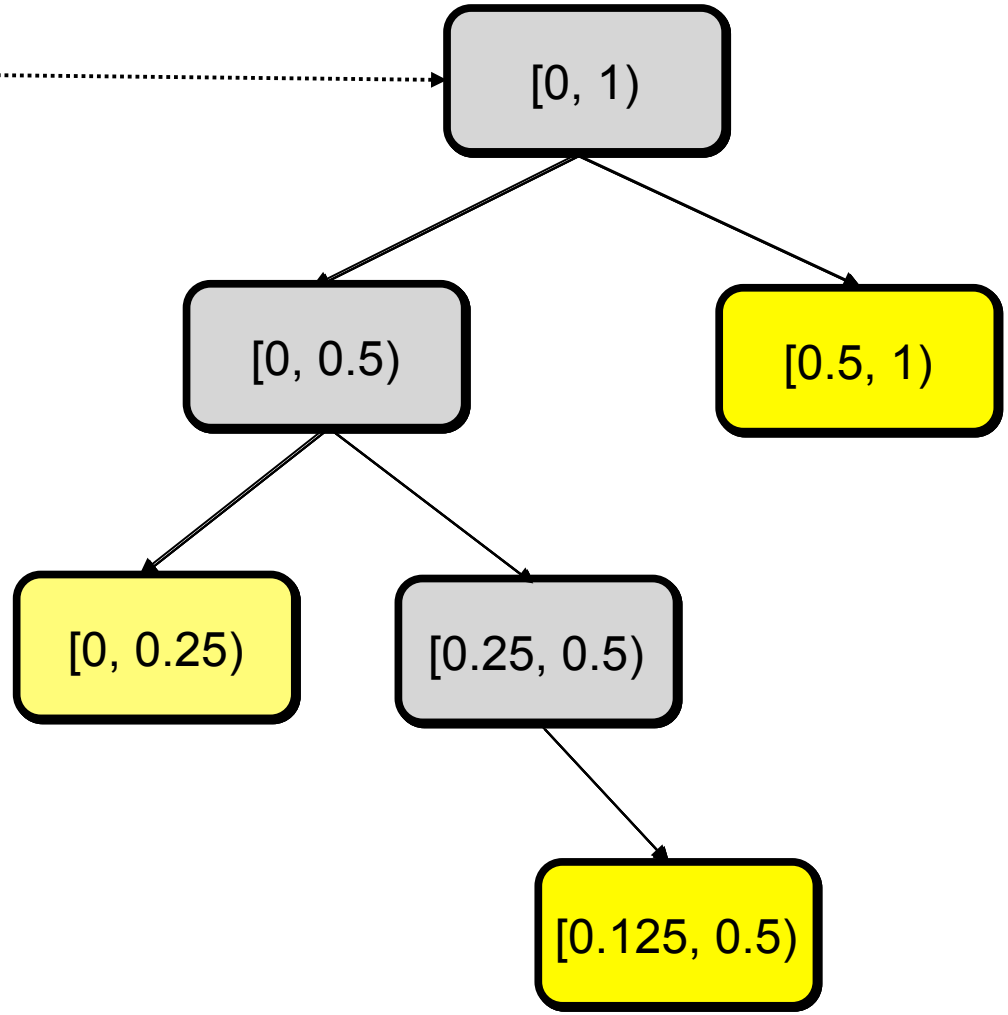


Many roots

One root

The Descartes Method

Three Taylor Shifts
or
One de Casteljau



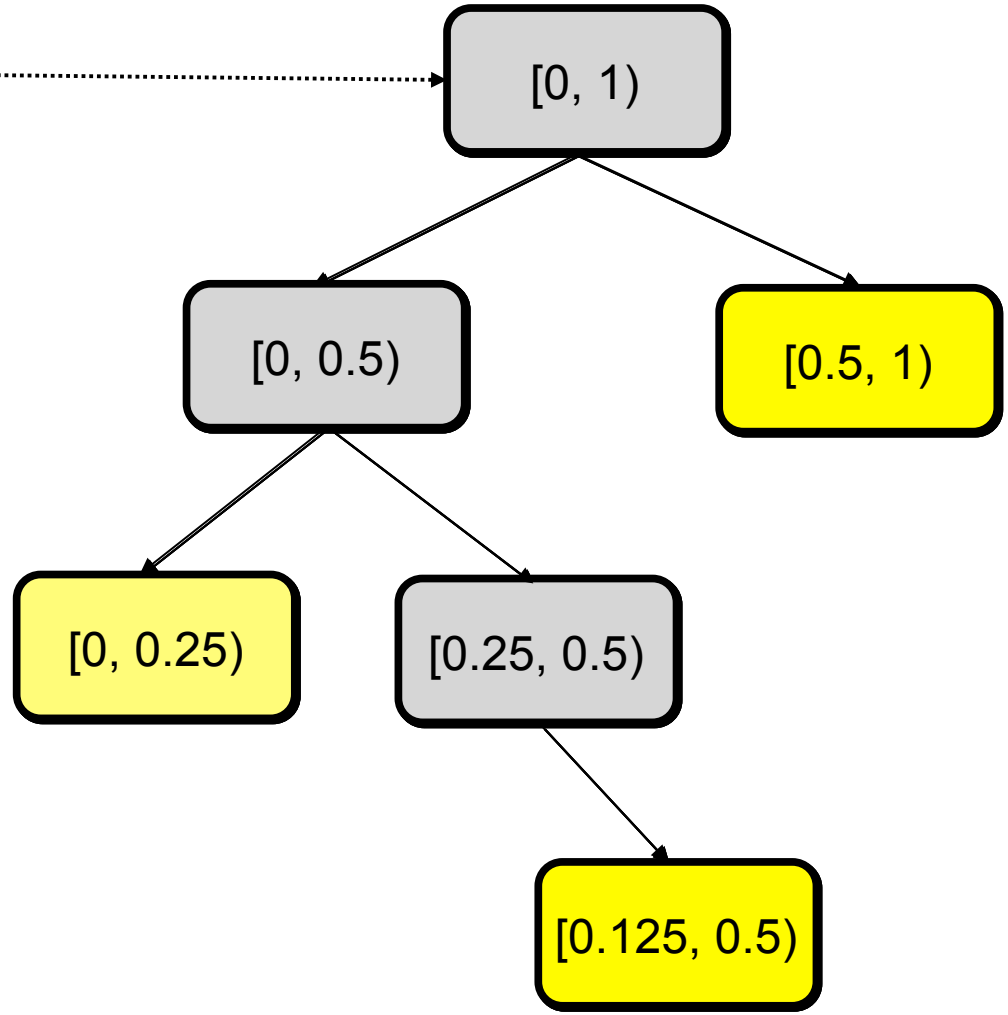
Many roots

One root

The Descartes Method

Three Taylor Shifts
or
One de Casteljau

(GMP)



Many roots

One root

Two choices

- Use the structure of root isolation to
  - Reduce the number of arithmetic operations
    - It worked for multiplication
  - Reduce the cost of each arithmetic operation
    - It worked for GMP

Reduce number of operations

- This is an algorithm problem
- I'm a systems person
- So were all of my collaborators
- It might be possible
  - We were not going to figure it out
  - I don't even know the lower bound

Reduce the cost of each operation

- This is an implementation problem
- Maybe we can work this one out?

Cost of GMP arithmetic

- Memory access to get the words
- Arithmetic on the words
- GMP does not reuse a word across multiple operations
- Compilers are unlikely to figure out how to reuse them across operations
  - GMP made sure to saturate the CPU

Arithmetic vs Memory

- Memory access is more expensive than arithmetic
  - Pretty much on any CPU you'll get to use
  - Especially true on the x86
    - .33 ns for an integer operation
    - 100 ns for a cache miss
- Not reducing number of arithmetic operations
- Reducing memory access reduces cost per arithmetic operation
- Only counting operations is for text books

Memory Trade Off

- Assuming a fixed budget
  - Money
  - Power
  - Area
- You can have
  - Fast memory or
  - High capacity memory
- It won't all be fast

x86 Memory Hierarchy

- Registers
- Cache
  - Per core cache
  - Per socket cache
- Memory
  - Local numa node
  - Remote numa node
- Hard Drive
  - Volatile cache
  - Persistent storage

Tiling

- Lots of people know memory access is expensive
- Once you load something, do as many computations on it as possible
- For regularly structure problems, this is called tiling

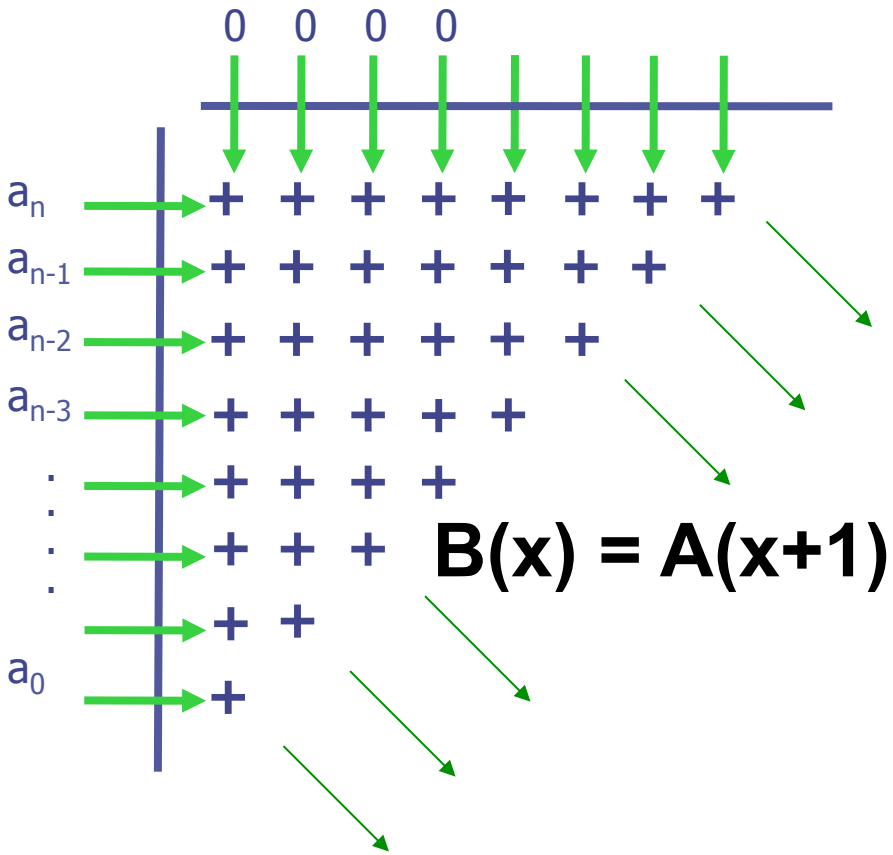
Computational Structure

- You don't want to look at a specific algorithm
  - Detail obscure what needs to be done
  - Makes it hard to reuse optimizations
- You are optimizing a dependency graph
  - Some nodes might have different costs
- Instead of more math, we get to just look at graphs
  - I don't understand most of the math
  - I did a lot of the optimization

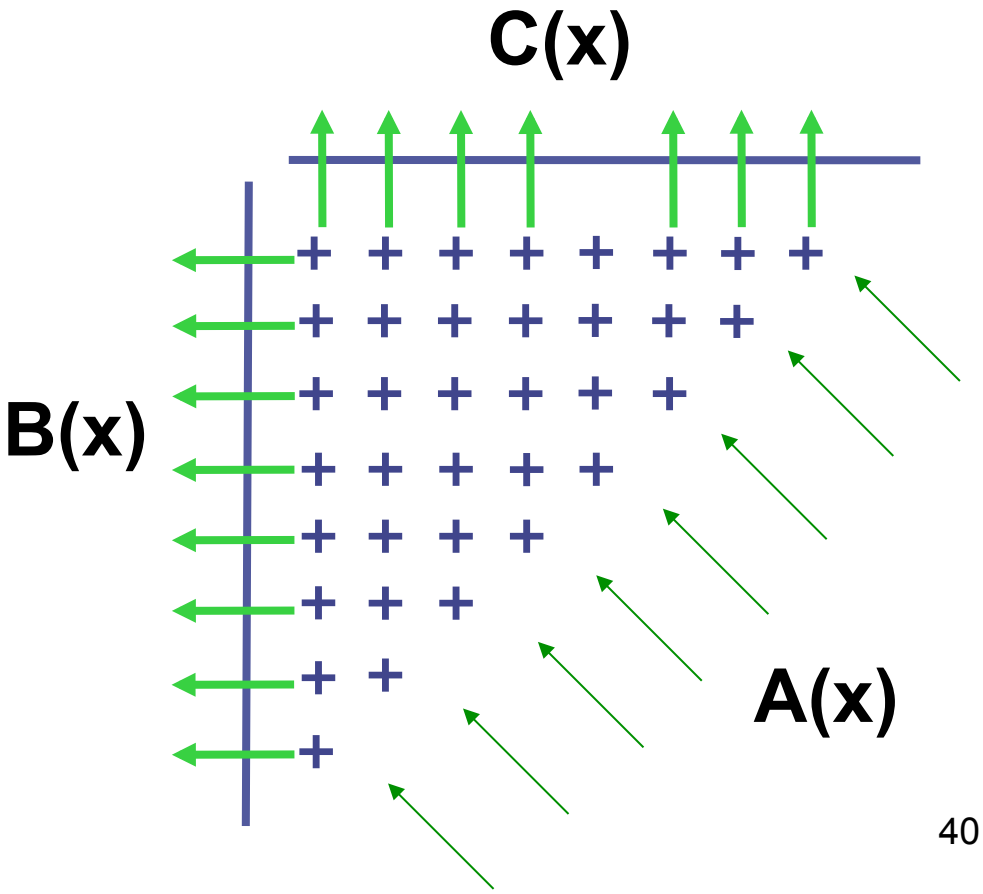
Major sub-algorithms

Let $A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$

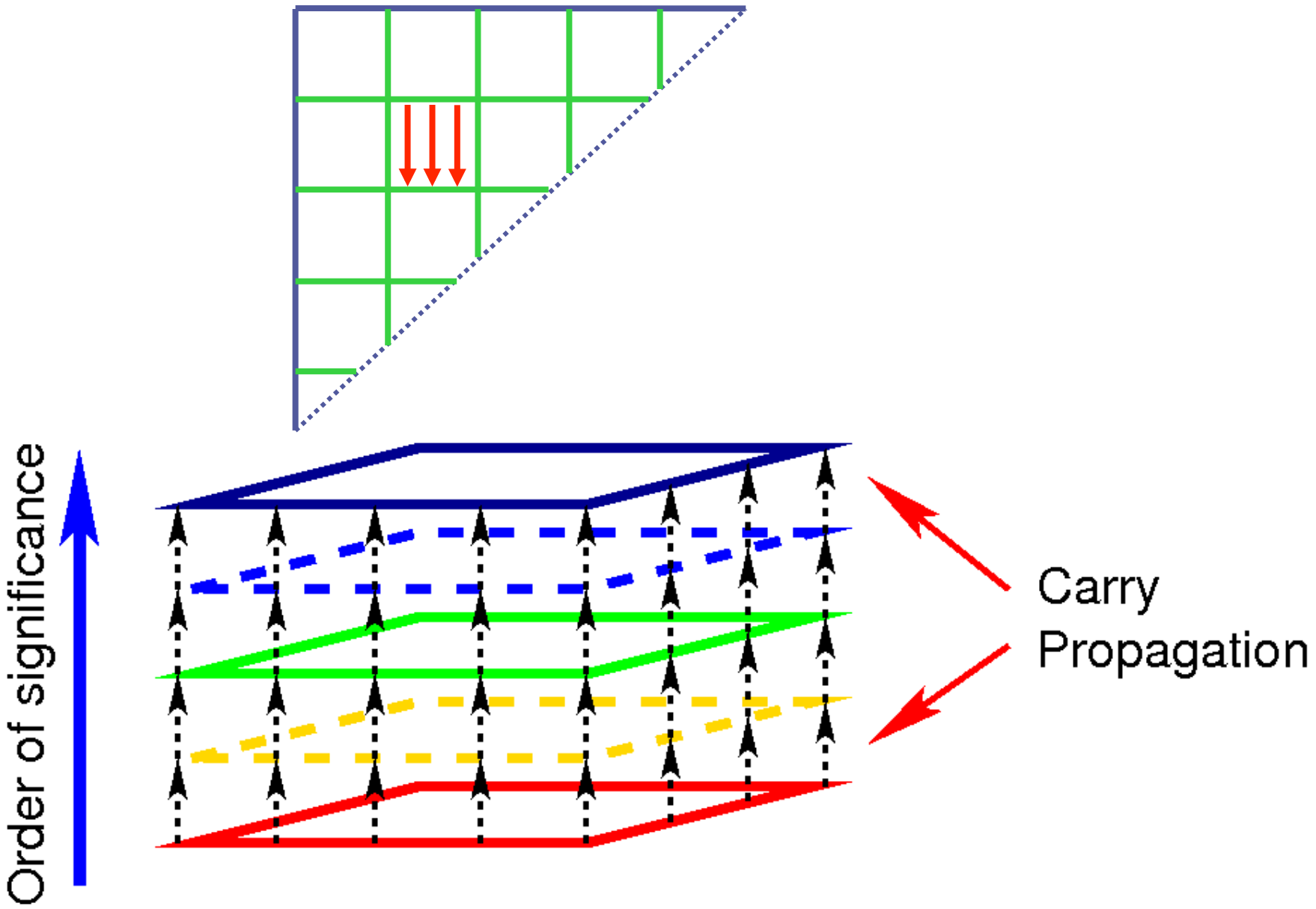
Taylor shift by 1



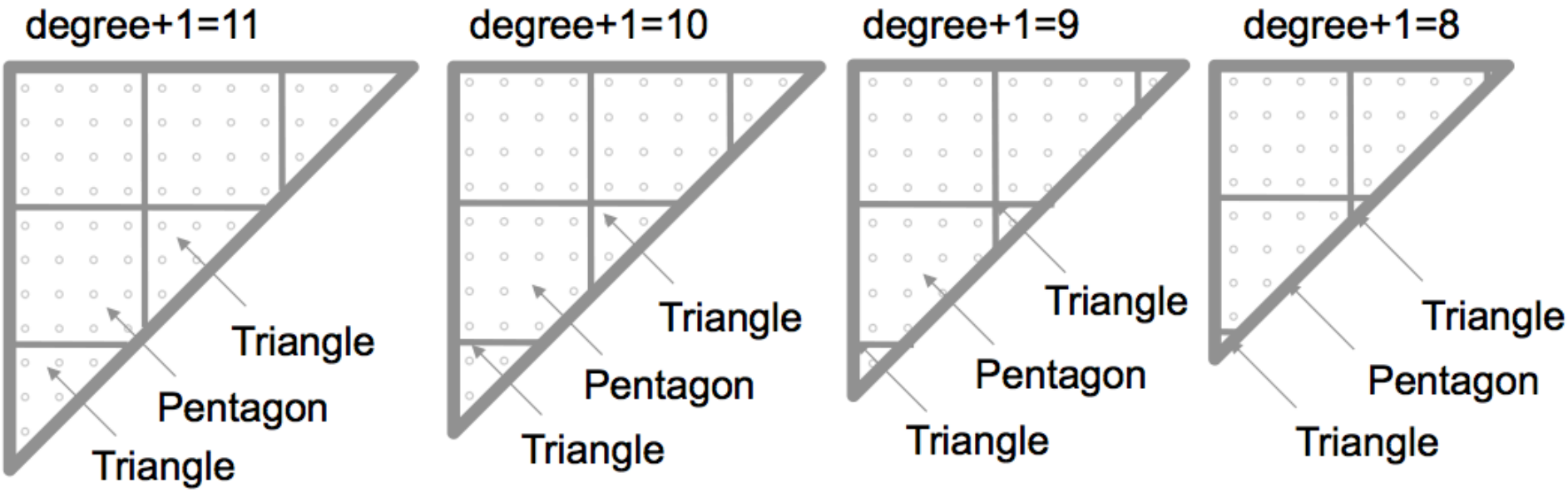
de Casteljau's algorithm



Tiling: Multi Precision



Tile Pairings



square size e=4
diagonals d=3

square size e=4
diagonals d=2

square size e=4
diagonals d=1

square size e=4
diagonals d=0

Note: By construction, only these tile transitions can occur

Cache Tiles

- Tile size is based on your cache capacity
  - size
  - Associativity
- Also allows prefetching
  - Cache is loaded one “line” at a time
  - 64 bytes on x86
- Speedup?

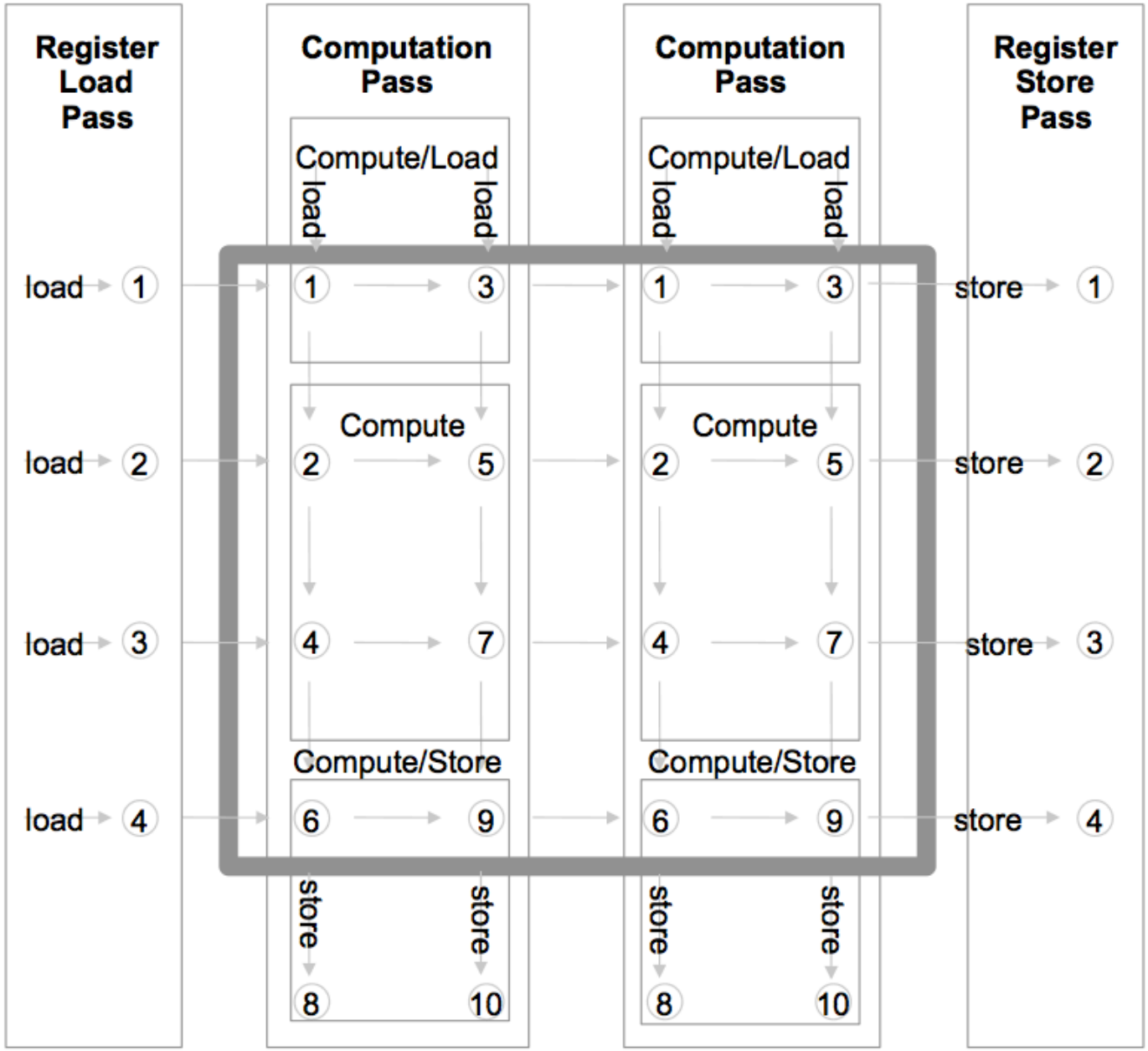
Cache Tile Speedup

- Basically zero
- Did we do something wrong?
- Did we at least reduce cache operations
  - Hardware performance counters
    - PAPI
    - CPC
- We got something right...
- Do we give up? Any other ideas?
  - What to measure next?

Register Tiles

- Maybe there is not enough compute per word for cache tiles to help
  - Even with less cache access, we stall the CPU
- What about register tiles?
  - Another chance to not stall the CPU
- Note: we are trying to get what GMP had
  - Peak IPC
  - For the problem we are optimizing
- Speedup?

Tile Computation



Code Generation

1,000 lines of Perl produces

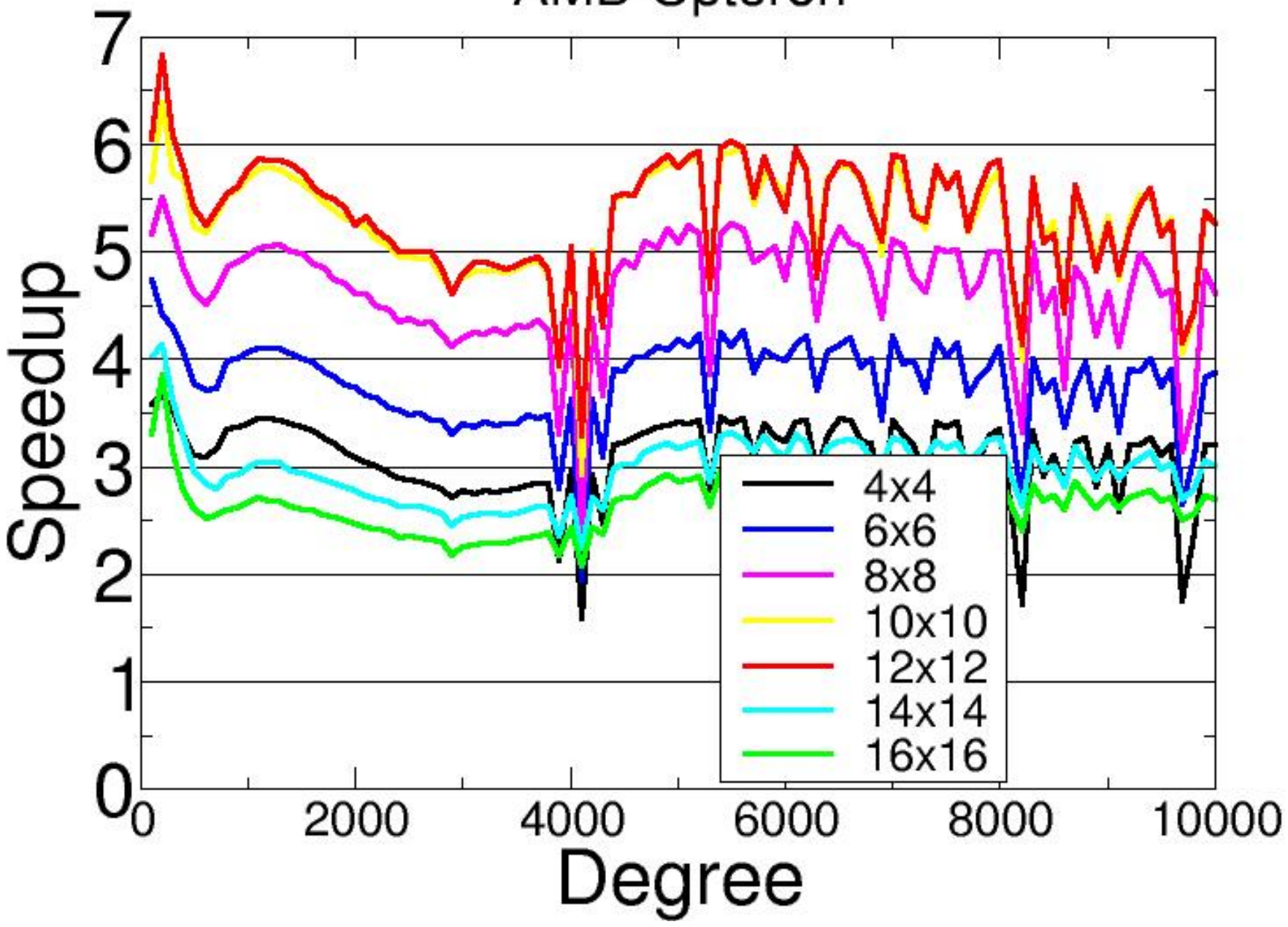
e (edge size)	Lines of generated C++
4	1,124
6	1,876
8	3,044
10	4,724
12	7,012
14	10,004
16	13,796
Total	**41,580**

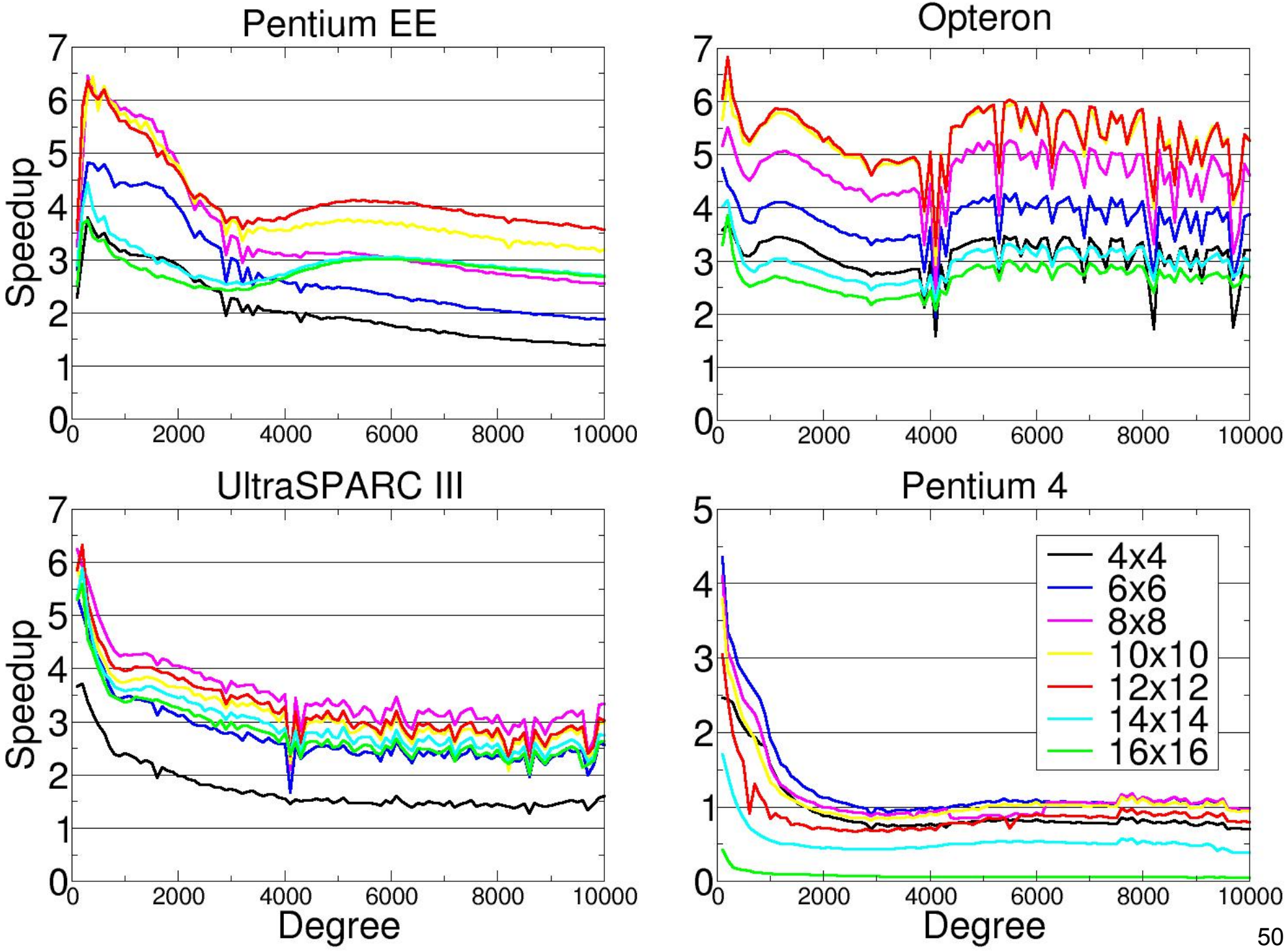
Processor architectures

processor	word-length	registers	IEUs	cache assoc.	optimal tile- size
Pentium4	32	8	4	8- way	6x6
UltraSPARC III	64	32	2	4- way	8x8
Pentium EE	64	16	4	8- way	12x12
Opteron	64	16	3	4- way	12x12

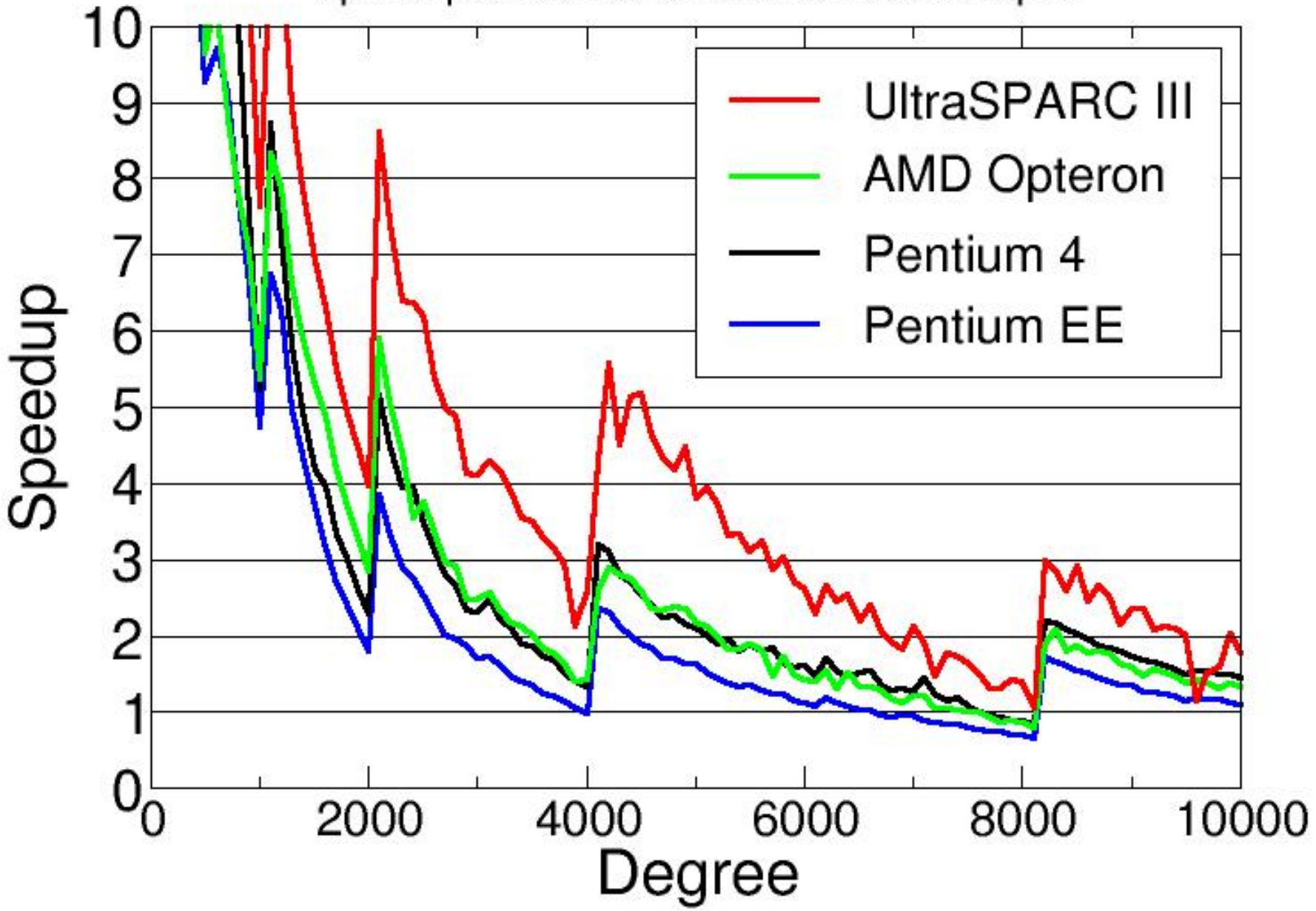
=> optimal tile size

Impact of register tile size on performance



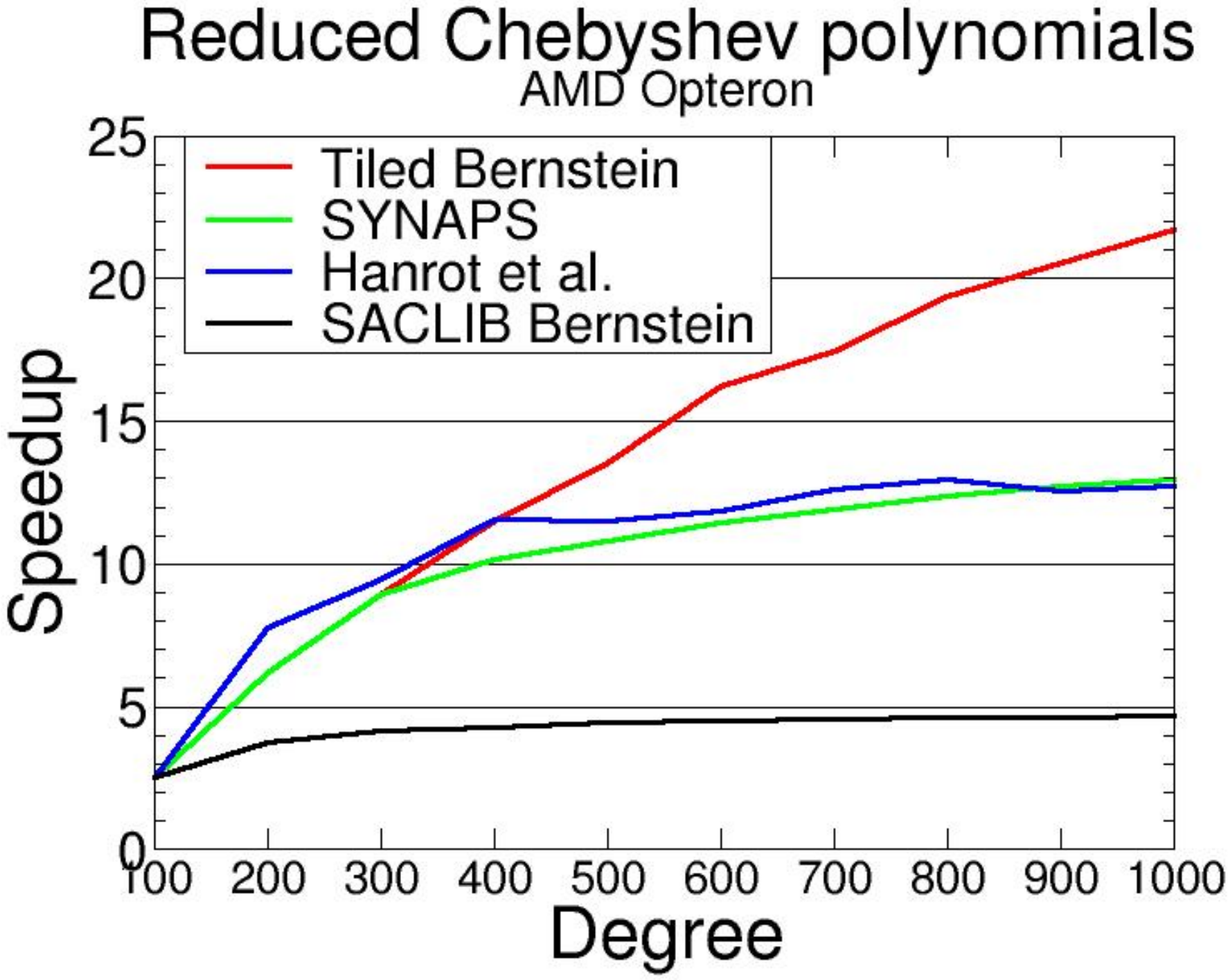

Pentium EE
Opteron
UltraSPARC III
Pentium 4
Speedup
Degree
0
1
2
3
4
5
6
7
0
2000
4000
6000
8000
10000
4x4
6x6
8x8
10x10
12x12
14x14
16x16

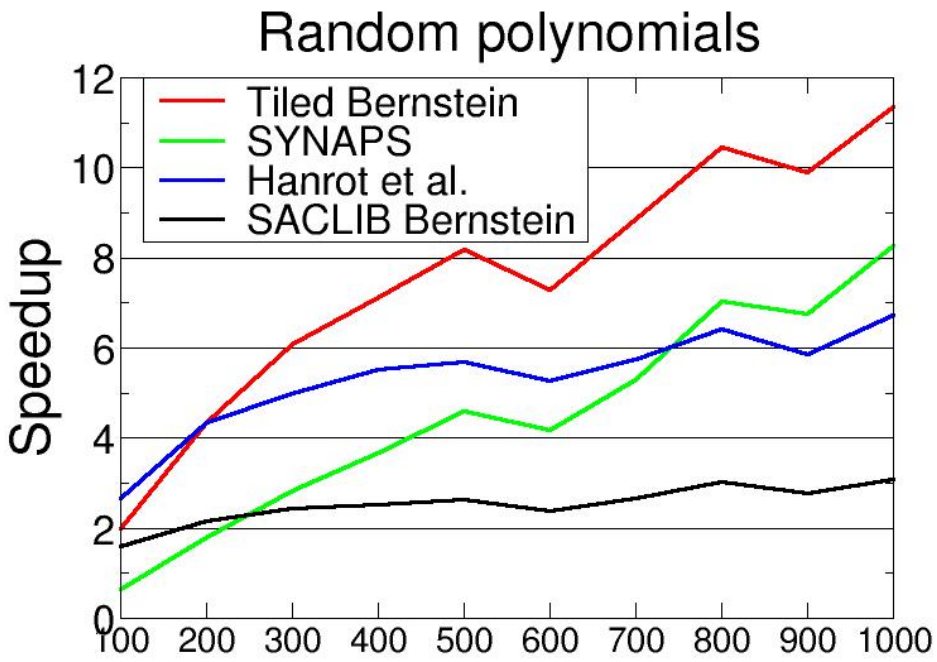
Classical vs. asymptotically fast Taylor shift

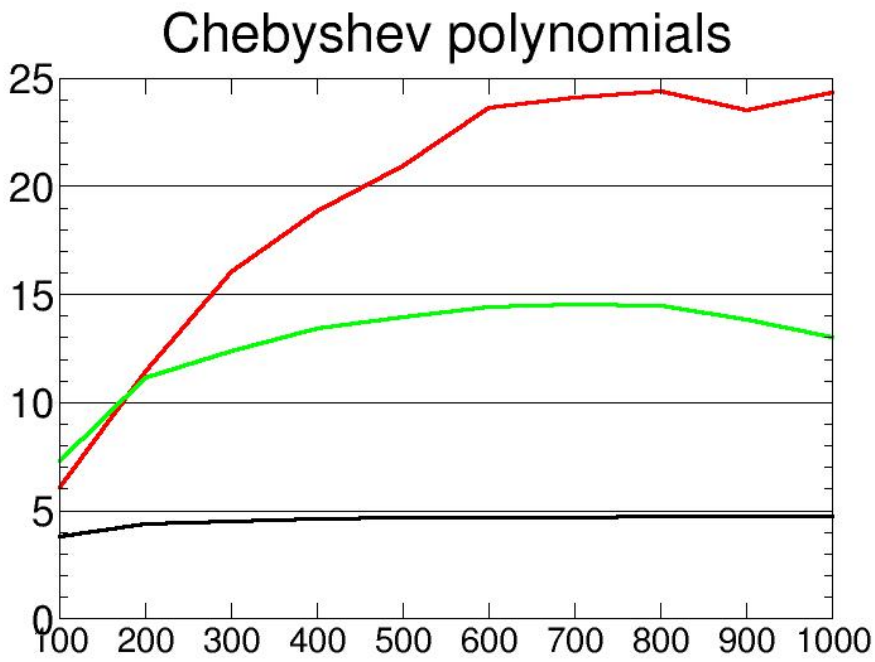


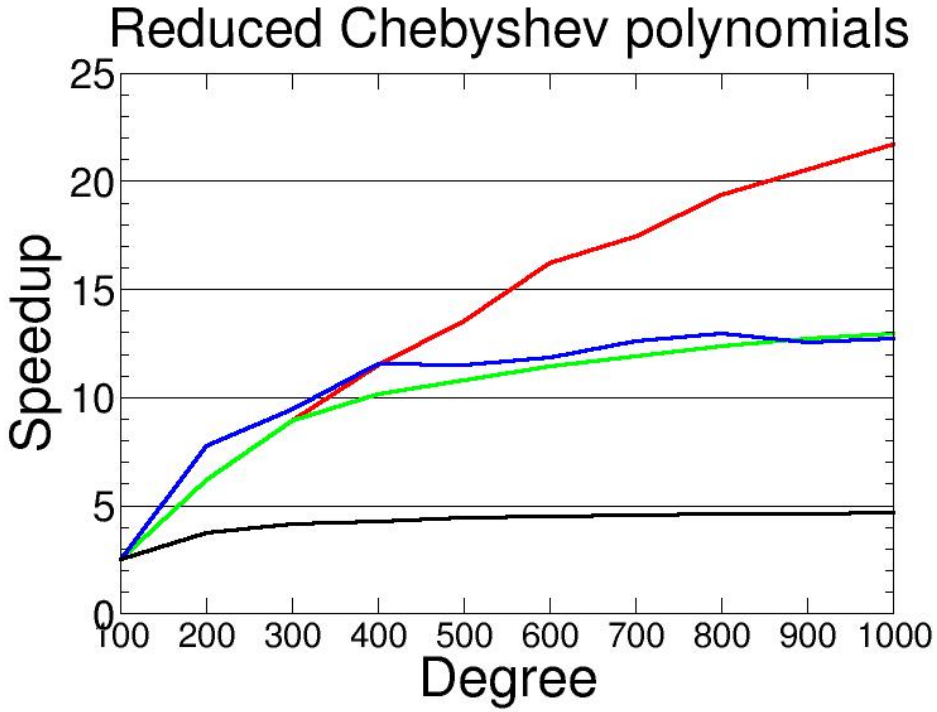
Input polynomials

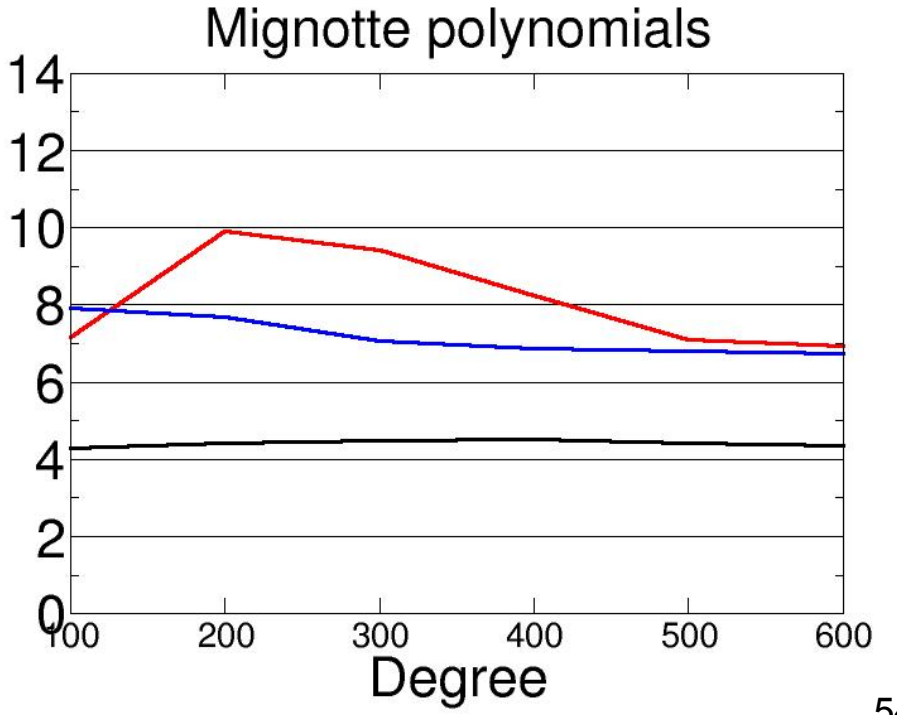
- Two major costs of root isolation
  - Tile evaluations
  - Recursion depth
- Random polynomials: random coefficients (same for all implementations) few nodes, degrees 100, 200,...,1000.
- Chebyshev polynomials: all roots real (and known), many nodes, degrees 100,200,...,1000. Hanrot: degree reduction by pre-processing x -> sqrt(x) when possible.
- Reduced (Chebyshev polynomial of degree n) has degree n/2.
- Mignotte polynomials: deep recursion trees, worst case for Descartes method, degrees 100,200,...,600. SYNAPS works up to degree 80.


Reduced Chebyshev polynomials
AMD Opteron
Tiled Bernstein
SYNAPS
Hanrot et al.
SACLIB Bernstein
Speedup
0
5
10
15
20
25
100
200
300
400
500
600
700
800
900
1000
Degree


Random polynomials
Tiled Bernstein
SYNAPS
Hanrot et al.
SACLIB Bernstein
Speedup
0
2
4
6
8
10
12
100 200 300 400 500 600 700 800 900 1000


Chebyshev polynomials
0
5
10
15
20
25
100 200 300 400 500 600 700 800 900 1000


Reduced Chebyshev polynomials
Speedup
0
5
10
15
20
25
100 200 300 400 500 600 700 800 900 1000
Degree


Mignotte polynomials
0
2
4
6
8
10
12
14
100 200 300 400 500 600
Degree

Publications

Jeremy R. Johnson, Werner Krandick, Anatole D. Ruslanov. **Architecture-aware classical Taylor shift by 1**, *Proceedings of the 2005 international symposium on Symbolic and algebraic computation*

Jeremy R. Johnson, Werner Krandick, Kevin M. Lynch, David G. Richardson, and Anatole D. Ruslanov. **High-Performance Implementations of the Descartes Method**, *Proceedings of the 2006 International Symposium on Symbolic and Algebraic Computation.*