A Comparison of Search Heuristics for Empirical Code Optimization

Keith Seymour, Haihang You, Jack Dongarra

International Workshop on Automatic Performance Tuning
Oct 1, 2008

ICL UT
INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY of TENNESSEE
Department of Electrical Engineering and Computer Science

Motivation

- Can ATLAS-like techniques be applied to arbitrary code?

- What do we mean by ATLAS-like techniques?
  - Blocking
  - Loop unrolling
  - Data prefetch
  - etc.

- Referred to as *empirical optimization* or *autotuning*
  - Generate many variations
  - Pick the best implementation by measuring the performance

Motivation (cont)

- Why use empirical optimization?
  - Traditional compiler optimization can't extract full machine performance
  - Hand tuning is difficult and non-portable
  - Allows optimizing less common routines which have not been hand tuned

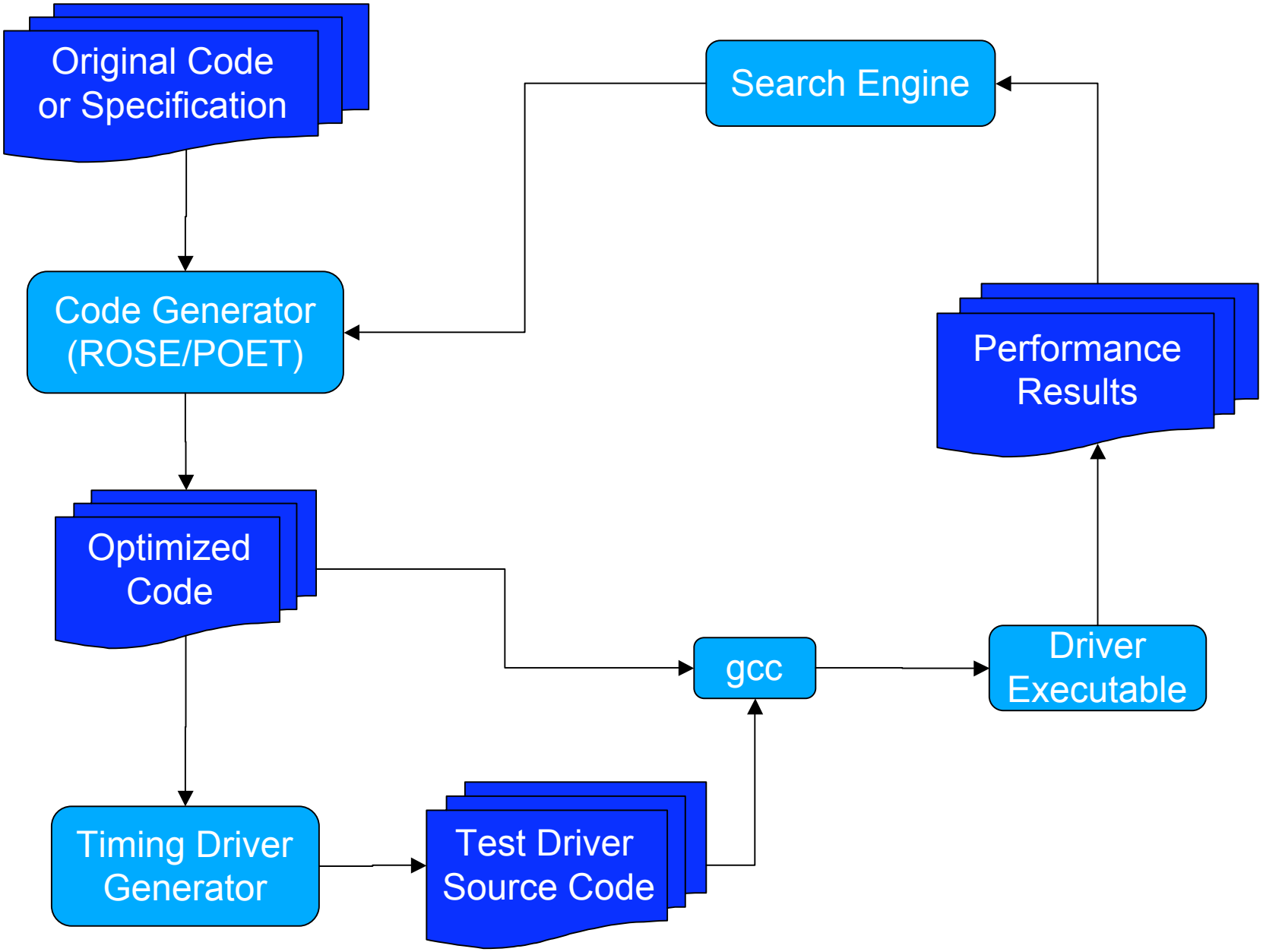
- Related Work
  - ATLAS
  - FFTW
  - PHiPAC
  - OSKI

Empirical Optimization Components

- Compiler infrastructure
  - Generates the various implementations
  - ROSE LoopProcessor from Lawrence Livermore National Lab (Yi, Q.; Quinlan, D.; Vuduc, R.)
  - POET - Parameterized Optimizations for Empirical Tuning (Yi, Q.; Seymour, K.; You, H.; Vuduc, R.; Quinlan, D.)

- Timing driver generator
  - Generates code to evaluate the performance

- Search engine
  - Can't exhaustively try implementations
  - Simplex
  - Genetic Algorithm
  - Random
  - Etc.


Original Code or Specification
Search Engine
Code Generator (ROSE/POET)
Performance Results
Optimized Code
gcc
Driver Executable
Timing Driver Generator
Test Driver Source Code

What is the user responsible for?

- Specifying the argument sizes:

```
/*$ATLAS ROUTINE DGEMM */
/*$ATLAS SIZE 400:400:50 */
/*$ATLAS ARG M           IN     int     $size */
/*$ATLAS ARG N           IN     int     $size */
/*$ATLAS ARG L           IN     int     $size */
/*$ATLAS ARG ALPHA       IN     double  1.0 */
/*$ATLAS ARG A[M][L]     IN     double  $rand */
/*$ATLAS ARG B[L][N]     IN     double  $rand */
/*$ATLAS ARG C[M][N]     INOUT  double  $rand */
```

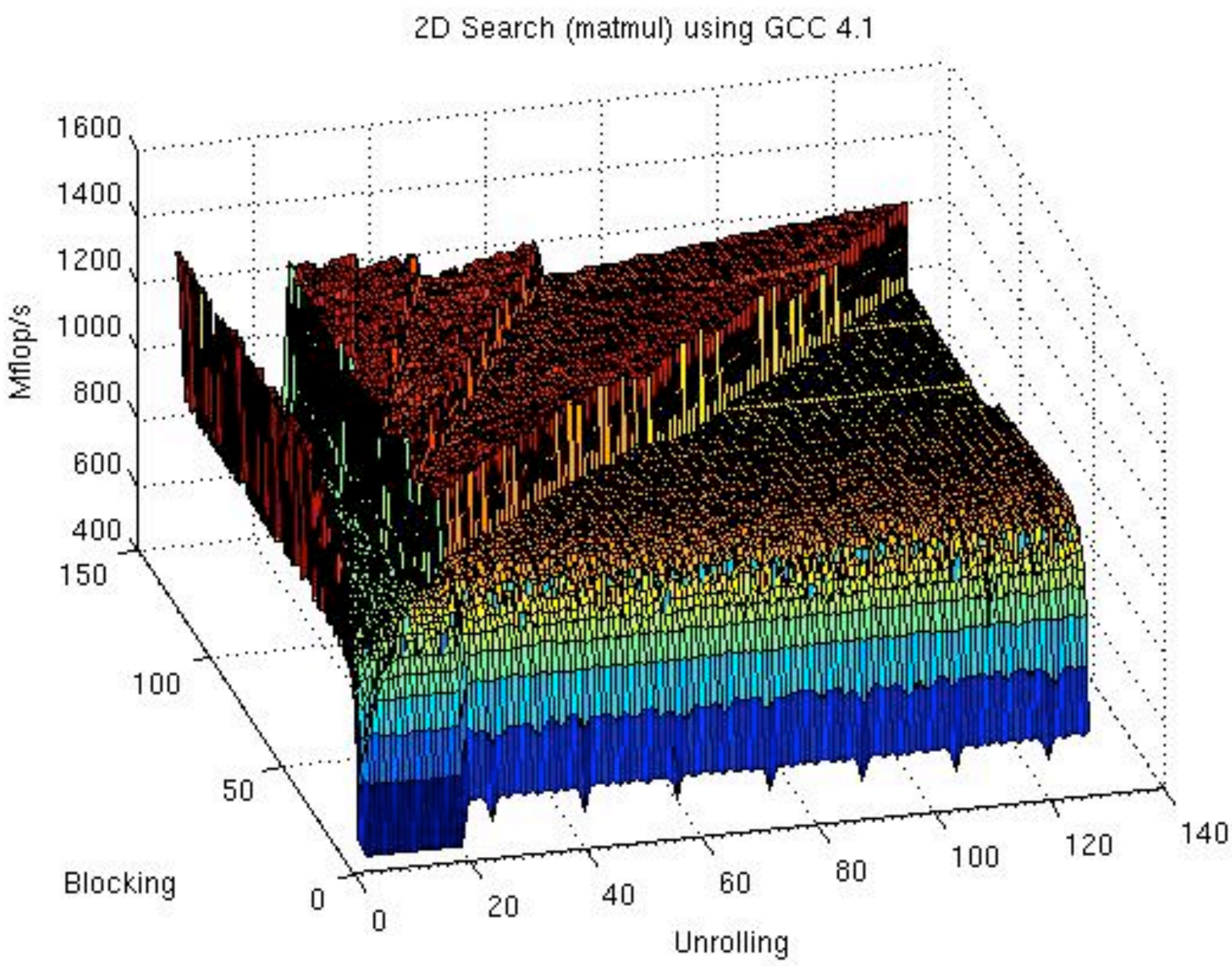
- Script that specifies the search bounds and evaluates a candidate:

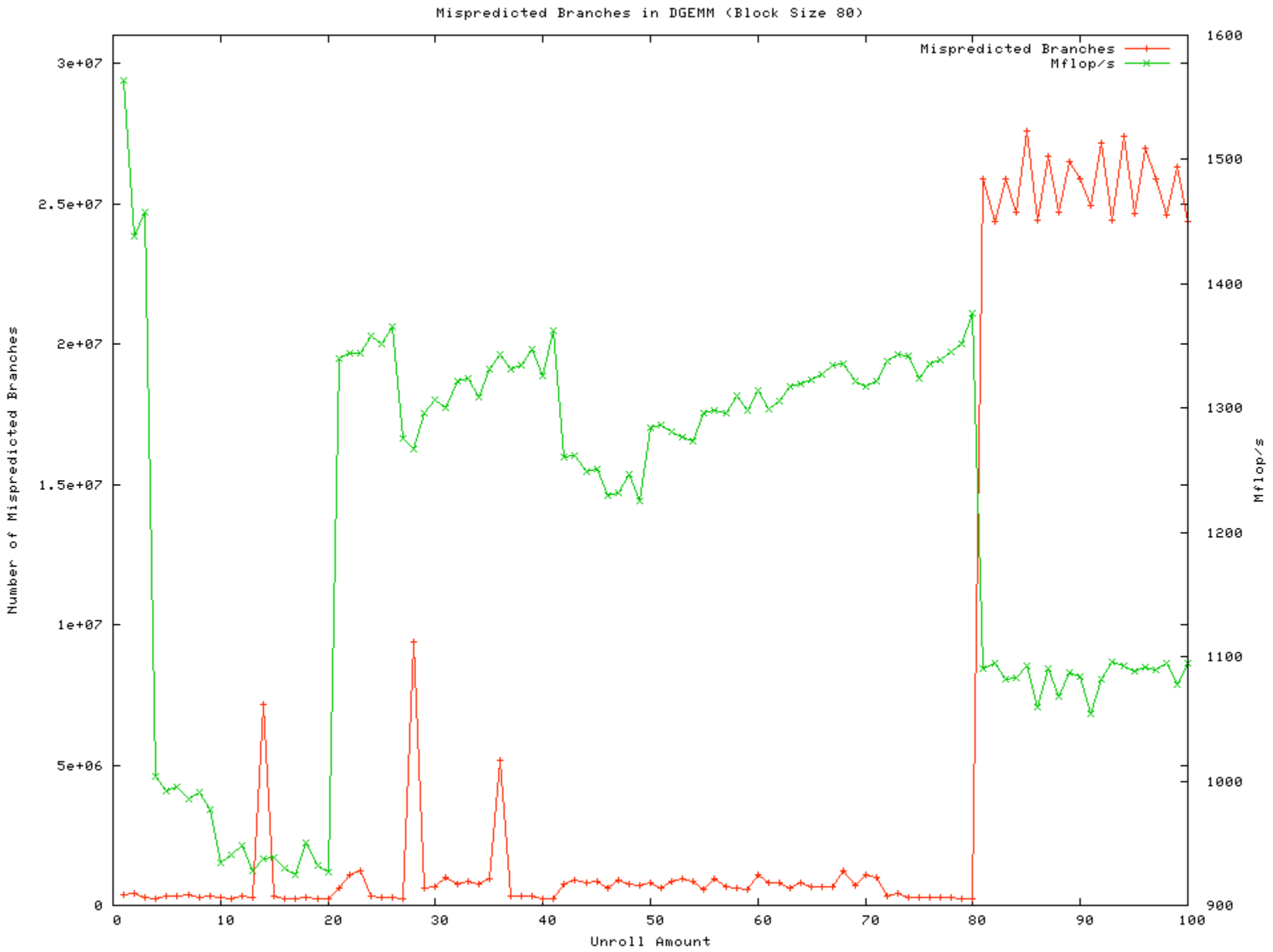
```
#DIM    5
#LB     2     2     2   2 1
#UB   128   128   128 128 6
#CONSTRAINT p1 < p2
#CONSTRAINT (p1 % 2) == 0
```

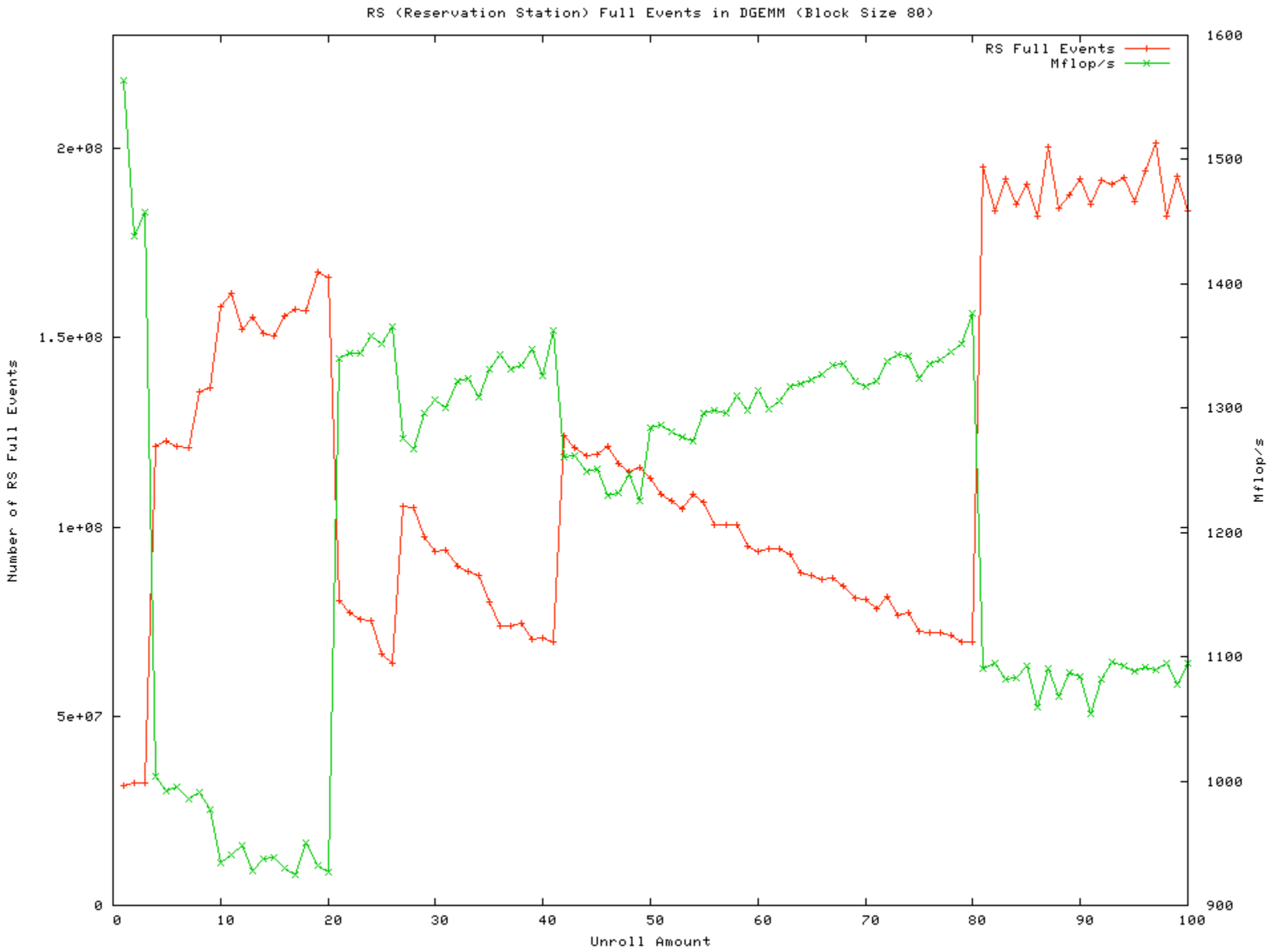
Test Environment

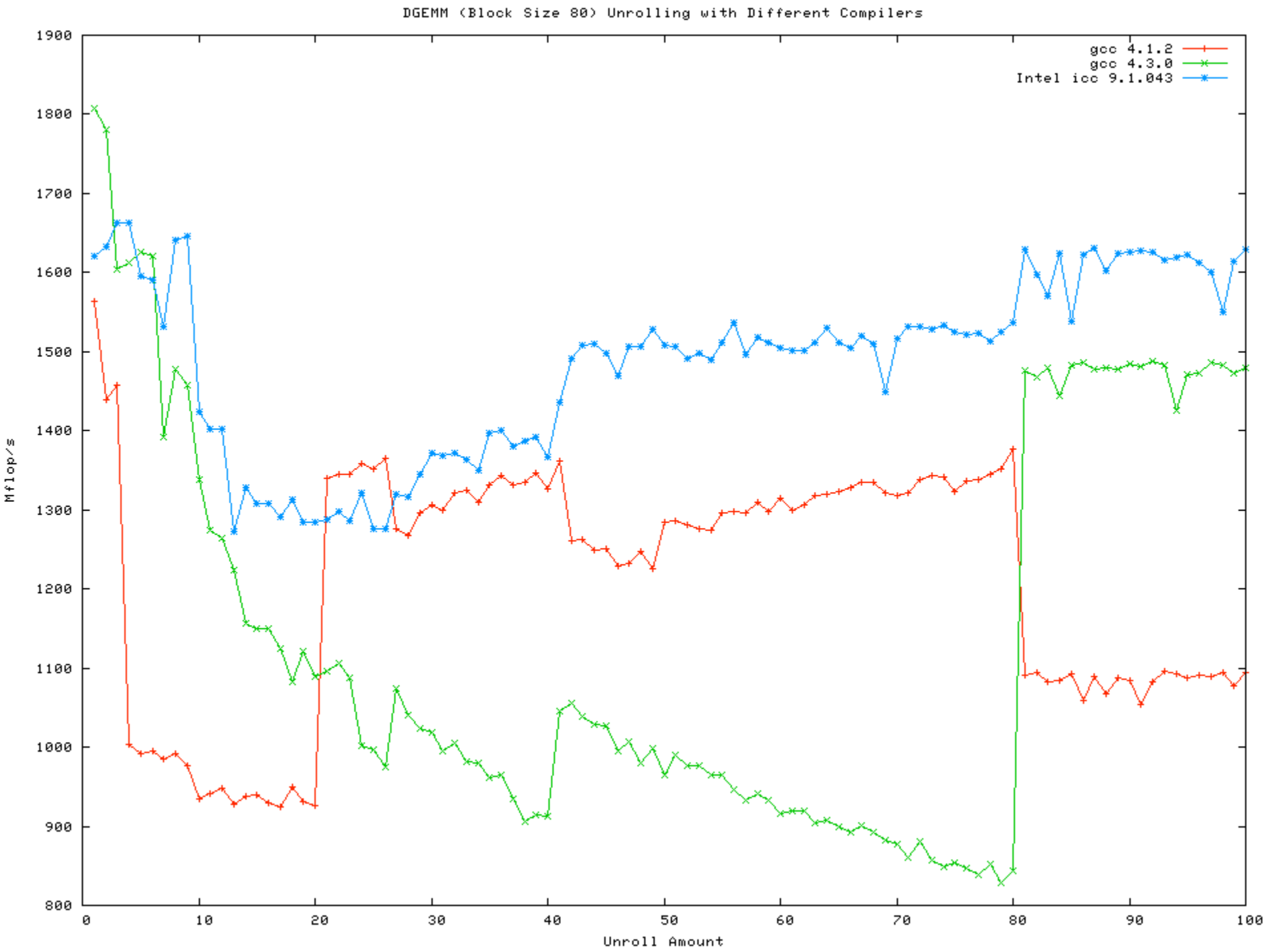
- 2.66GHz Intel Xeon X5355
- Fedora core 6 (kernel 2.6.22-10-perfctr)
- PAPI 3.6.0
- gcc 4.1.2

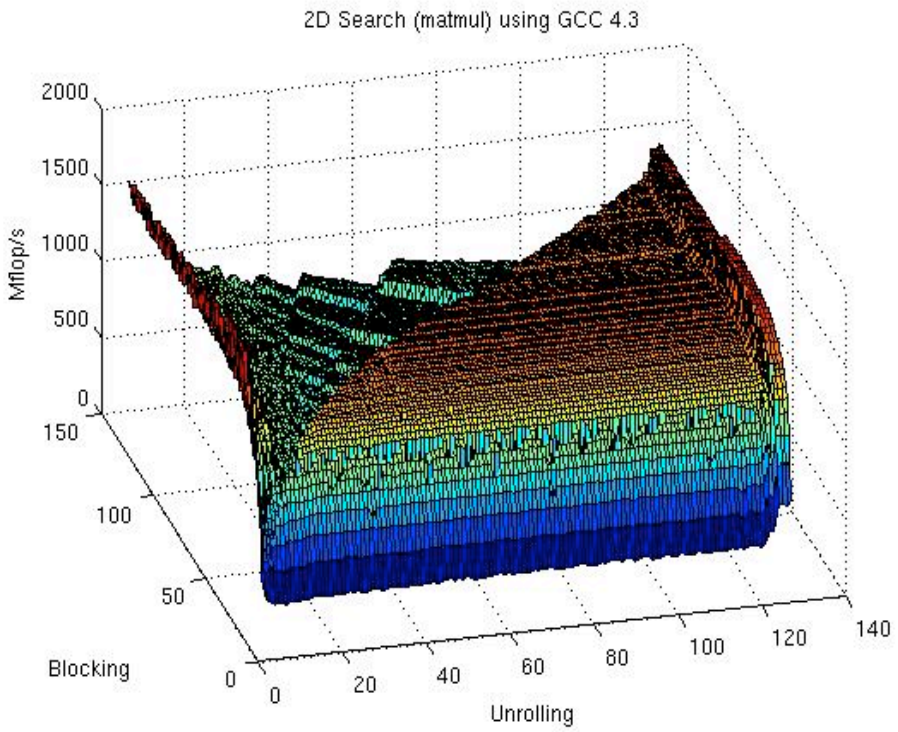
- Matrix multiply
  - 2d (blocking/unrolling)
  - 5d (multi-blocking/unrolling/loop order)
  - 21d (5d plus 16 dimensions of compiler flags)
- Matrix-vector multiply
  - 4d (multi-blocking/unrolling/loop order)


2D Search (matmul) using GCC 4.1
Mflop/s
1600
1400
1200
1000
800
600
400
Blocking
150
100
50
0
Unrolling
0
20
40
60
80
100
120
140

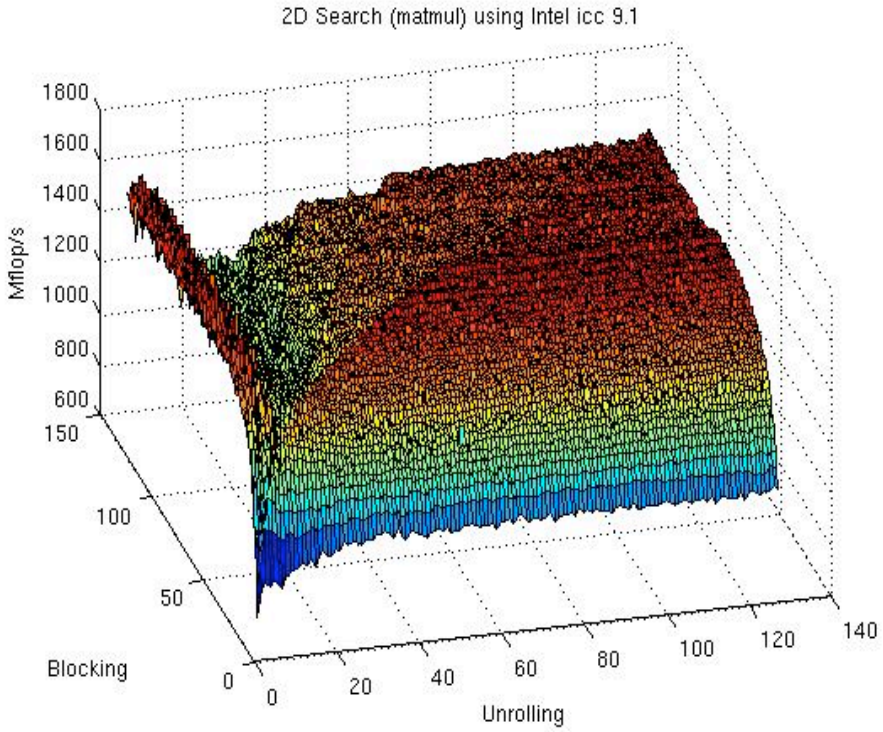

Mispredicted Branches in DGEMM (Block Size 80)
Mispredicted Branches
Mflop/s
Number of Mispredicted Branches
Mflop/s
Unroll Amount
0
5e+06
1e+07
1.5e+07
2e+07
2.5e+07
3e+07
900
1000
1100
1200
1300
1400
1500
1600
0
10
20
30
40
50
60
70
80
90
100


RS (Reservation Station) Full Events in DGEMM (Block Size 80)
RS Full Events
Mflop/s
Number of RS Full Events
Mflop/s
Unroll Amount
0
5e+07
1e+08
1.5e+08
2e+08
900
1000
1100
1200
1300
1400
1500
1600
0
10
20
30
40
50
60
70
80
90
100


DGEMM (Block Size 80) Unrolling with Different Compilers
gcc 4.1.2
gcc 4.3.0
Intel icc 9.1.043
Mflop/s
Unroll Amount
1900
1800
1700
1600
1500
1400
1300
1200
1100
1000
900
800
0
10
20
30
40
50
60
70
80
90
100



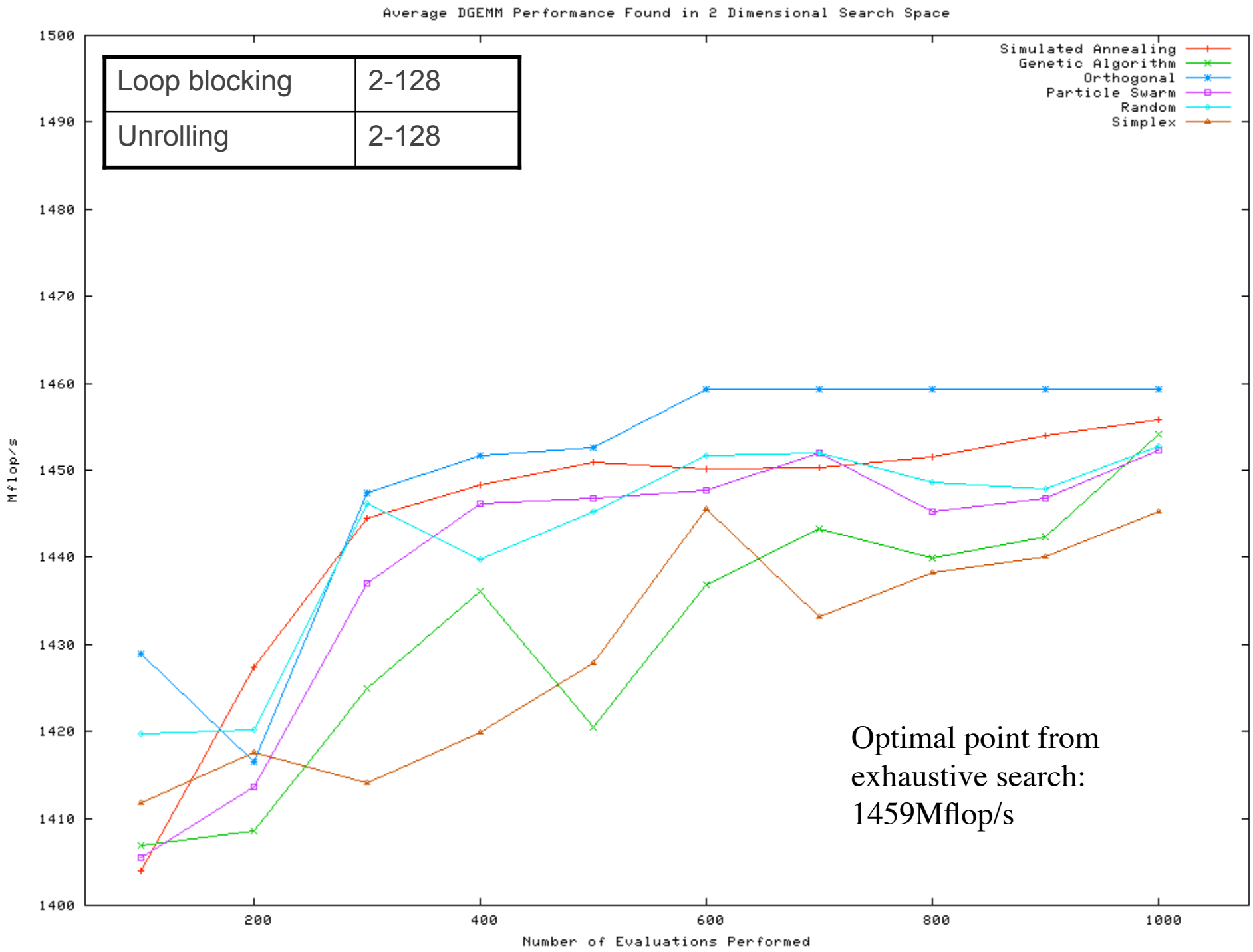
GCC 4.3

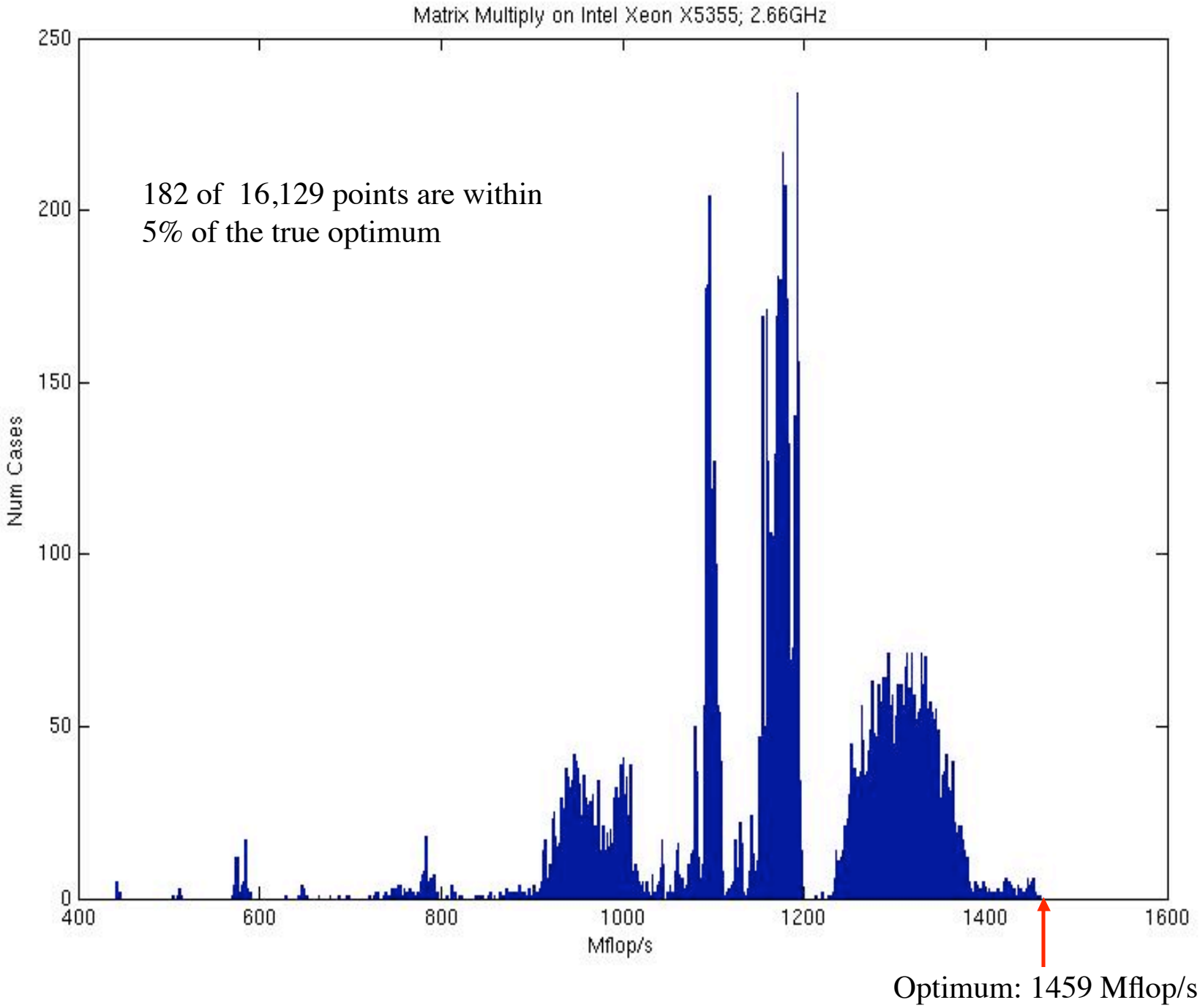


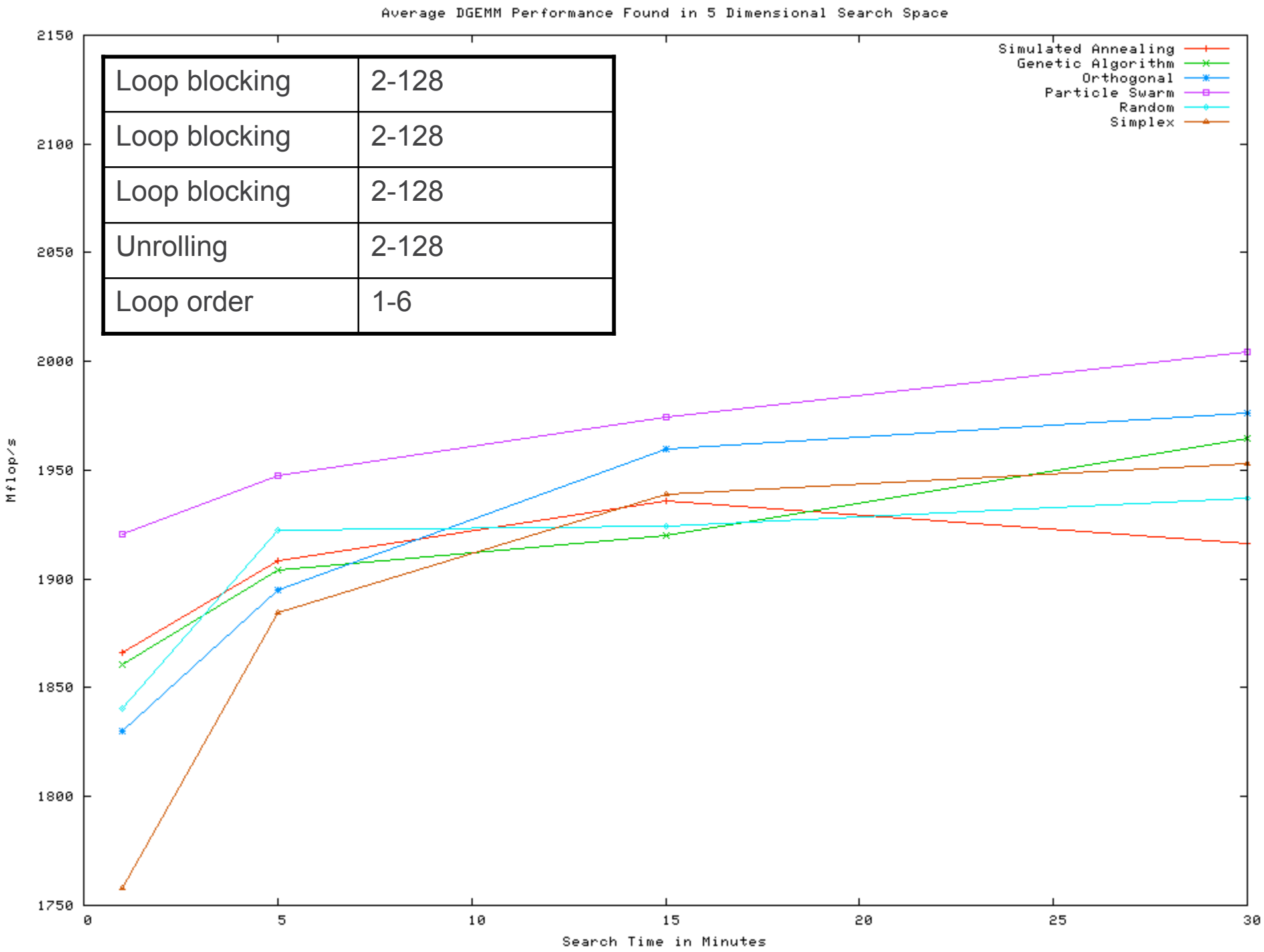
Intel icc 9.1

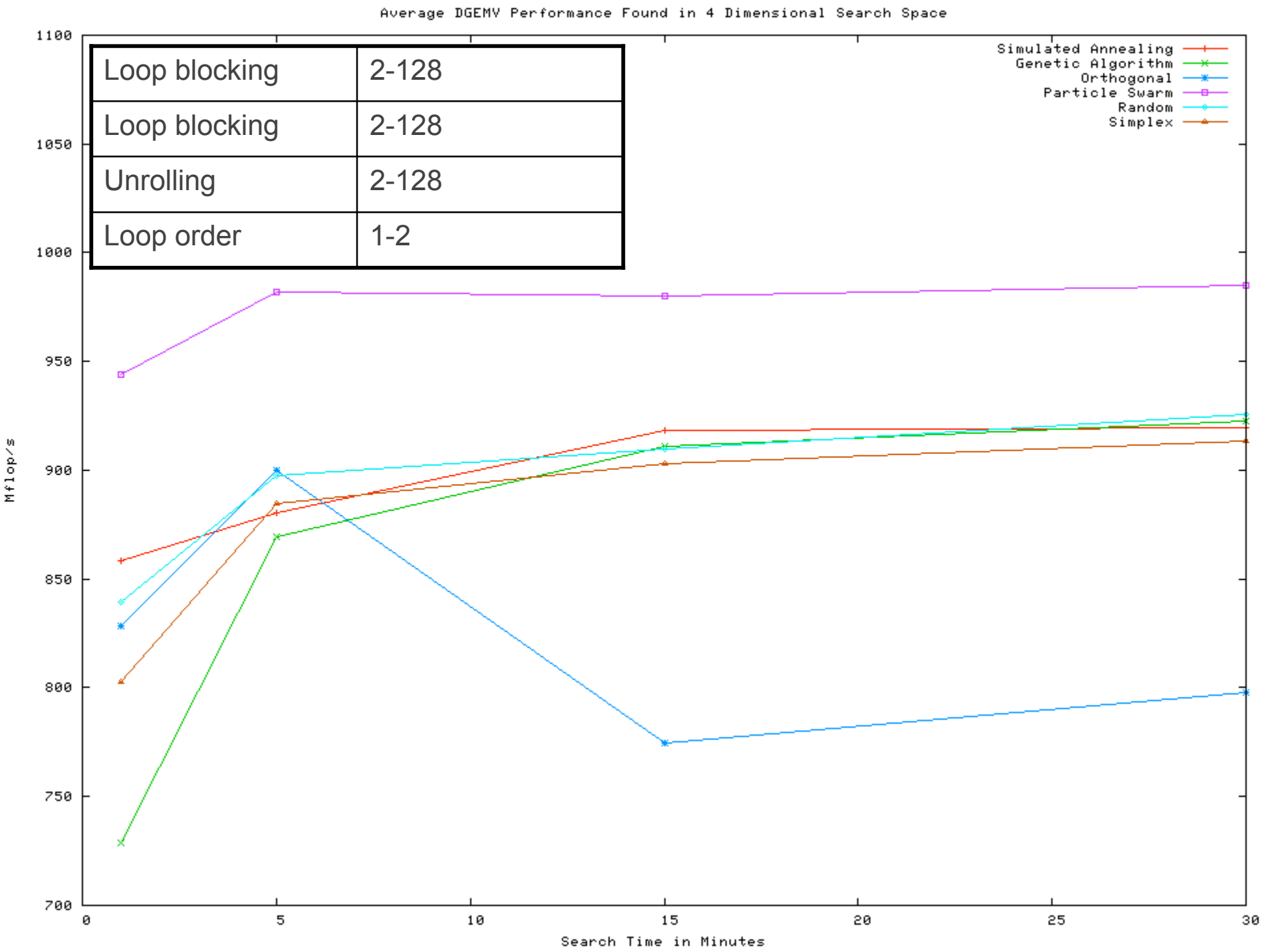
Search Techniques

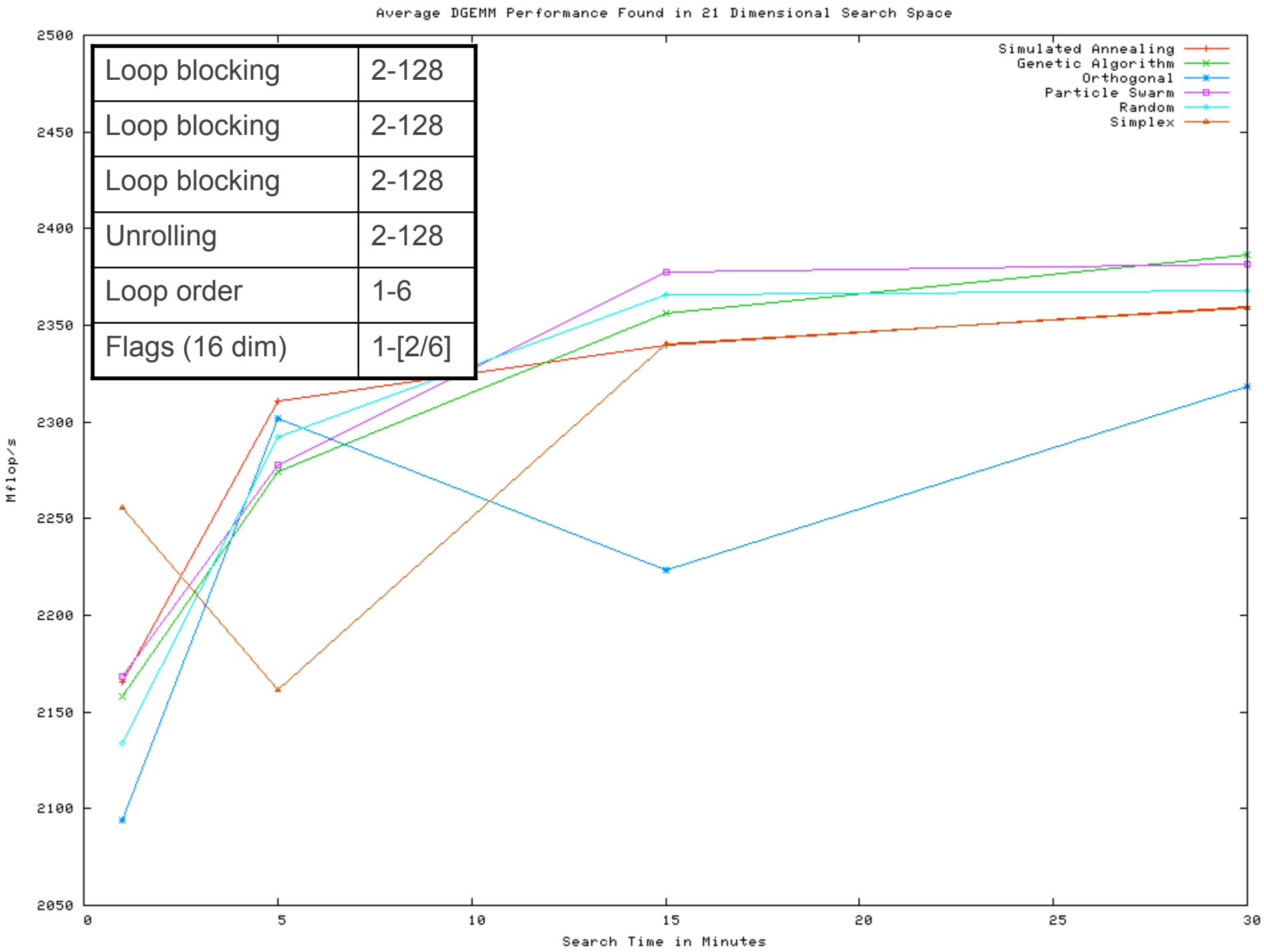
- Simplex
  - Set of vertices that contracts to an optimum by *reflection*, *expansion*, *contraction*, and *shrink*
- Orthogonal
  - Fully search one dimension at a time
- Genetic Algorithm
  - Adapt a population of candidates via natural selection (selection, mutation, crossover)
- Simulated Annealing
  - Based on annealing of substances such as metals – go through heating/cooling cycles until the characteristics change
- Particle Swarm Optimization
  - Based on flocking of birds – allow a population of particles to fly through the space, attracted to better performing areas
- Random


Average DGEMM Performance Found in 2 Dimensional Search Space
Loop blocking 2-128
Unrolling 2-128
Simulated Annealing
Genetic Algorithm
Orthogonal
Particle Swarm
Random
Simplex
Mflop/s
Number of Evaluations Performed
1500
1490
1480
1470
1460
1450
1440
1430
1420
1410
1400
200
400
600
800
1000
Optimal point from exhaustive search: 1459Mflop/s


Matrix Multiply on Intel Xeon X5355; 2.66GHz
182 of 16,129 points are within
5% of the true optimum
Num Cases
Mflop/s
0
50
100
150
200
250
400
600
800
1000
1200
1400
1600
Optimum: 1459 Mflop/s


Average DGEMM Performance Found in 5 Dimensional Search Space
Loop blocking 2-128
Loop blocking 2-128
Loop blocking 2-128
Unrolling 2-128
Loop order 1-6
Simulated Annealing
Genetic Algorithm
Orthogonal
Particle Swarm
Random
Simplex
Mflop/s
2150
2100
2050
2000
1950
1900
1850
1800
1750
0
5
10
15
20
25
30
Search Time in Minutes


Average DGEMV Performance Found in 4 Dimensional Search Space
Loop blocking 2-128
Loop blocking 2-128
Unrolling 2-128
Loop order 1-2
Simulated Annealing
Genetic Algorithm
Orthogonal
Particle Swarm
Random
Simplex
Mflop/s
Search Time in Minutes
1100
1050
1000
950
900
850
800
750
700
0
5
10
15
20
25
30


Average DGEMM Performance Found in 21 Dimensional Search Space
Loop blocking 2-128
Loop blocking 2-128
Loop blocking 2-128
Unrolling 2-128
Loop order 1-6
Flags (16 dim) 1-[2/6]
Simulated Annealing
Genetic Algorithm
Orthogonal
Particle Swarm
Random
Simplex
Mflop/s
2500
2450
2400
2350
2300
2250
2200
2150
2100
2050
0
5
10
15
20
25
30
Search Time in Minutes

Recent Work

- Uniform distributions for population based methods (GA, PSO)
  - Halton and Faure sequences
- Hybrid Particle Swarm / Simulated Annealing
- Orthogonal search modifications
  - Hybrid random/orthogonal search
  - Sort dimensions based on size
- Application to different kinds of problems

Conclusions

- Performance can be improved with a modest amount of search time
- Random search is pretty good!
  - Is the search space too easy?
- Otherwise PSO seems to have the slight edge
- Non-architectural features (e.g. compiler behavior) can have a huge effect