Statistical Models for Automatic Performance Tuning

Richard Vuduc, James Demmel (U.C. Berkeley, EECS)
`{richie,demmel}@cs.berkeley.edu`

Jeff Bilmes (Univ. of Washington, EE)
`bilmes@ee.washington.edu`

May 29, 2001

International Conference on Computational Science

Special Session on Performance Tuning

Context: High Performance Libraries

- Libraries can isolate performance issues
  - BLAS/LAPACK/ScaLAPACK (linear algebra)
  - VSIPL (signal and image processing)
  - MPI (distributed parallel communications)
- Can we implement libraries …
  - automatically and portably?
  - incorporating machine-dependent features?
  - that match our performance requirements?
  - leveraging compiler technology?
  - using domain-specific knowledge?
  - with relevant run-time information?

Generate and Search:
An Automatic Tuning Methodology

- Given a library routine
- Write parameterized code generators
  - input: parameters
    - machine (e.g., registers, cache, pipeline, special instructions)
    - optimization strategies (e.g., unrolling, data structures)
    - run-time data (e.g., problem size)
    - problem-specific transformations
  - output: implementation in "high-level" source (e.g., C)
- **Search parameter spaces**
  - generate an implementation
  - compile using native compiler
  - measure performance (time, accuracy, power, storage, …)

Recent Tuning System Examples

- Linear algebra
  - PHiPAC (Bilmes, Demmel, et al., 1997)
  - ATLAS (Whaley and Dongarra, 1998)
  - Sparsity (Im and Yelick, 1999)
  - FLAME (Gunnels, et al., 2000)
- Signal Processing
  - FFTW (Frigo and Johnson, 1998)
  - SPIRAL (Moura, et al., 2000)
  - UHFFT (Mirković, et al., 2000)
- Parallel Communication
  - Automatically tuned MPI collective operations (Vadhiyar, et al. 2000)

Tuning System Examples (cont'd)

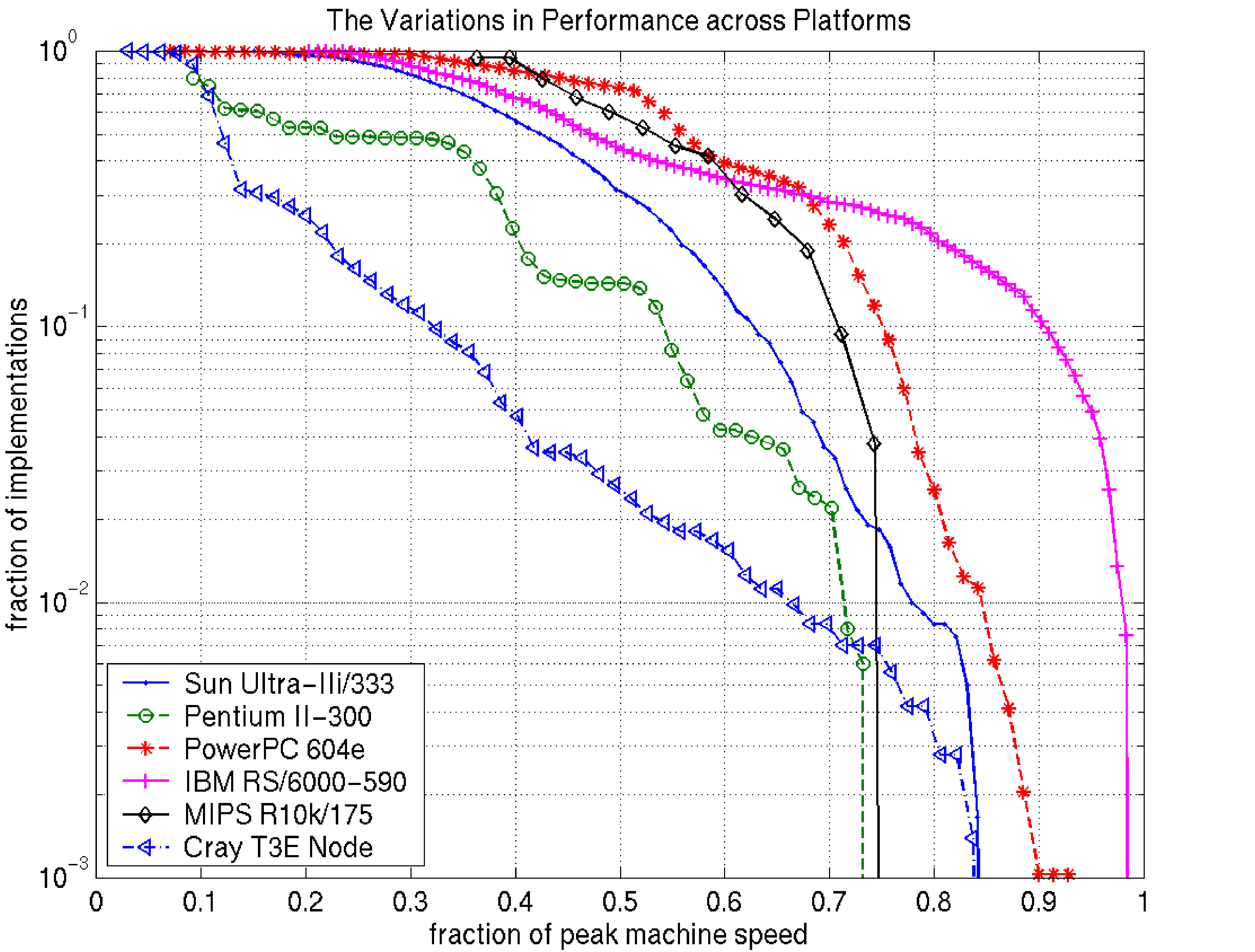
- Image Manipulation (Elliot, 2000)
- Data Mining and Analysis (Fischer, 2000)
- Compilers and Tools
  - Hierarchical Tiling/CROPS (Carter, Ferrante, et al.)
  - TUNE (Chatterjee, et al., 1998)
  - Iterative compilation (Bodin, et al., 1998)
  - ADAPT (Voss, 2000)

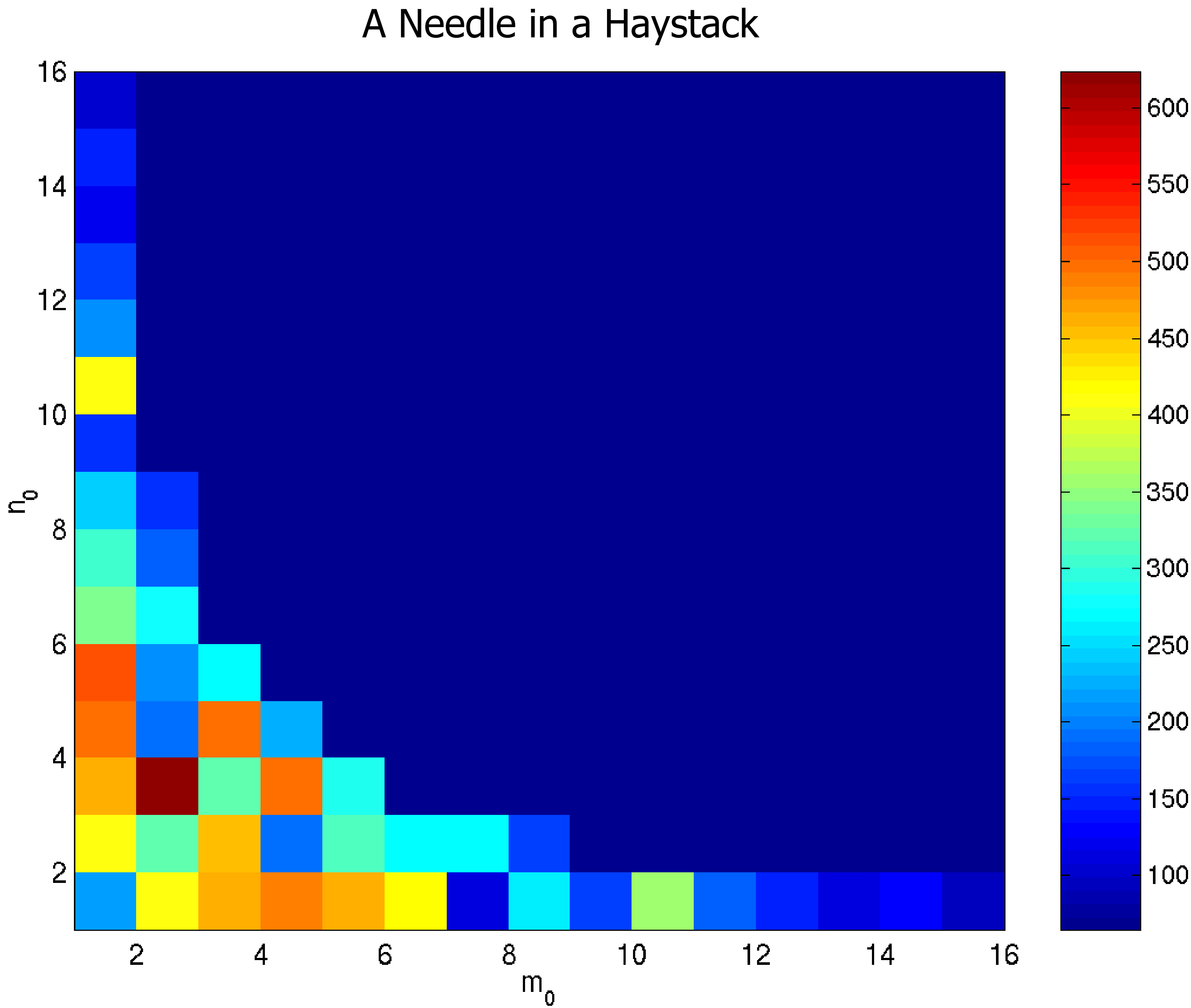
Road Map

- Context
- **Why search?**
- Stopping searches early
- High-level run-time selection
- Summary

The Search Problem in PHiPAC

- PHiPAC (Bilmes, et al., 1997)
  - produces dense matrix multiply (matmul) implementations
  - generator parameters include
    - size and depth of fully unrolled "core" matmul
    - rectangular, multi-level cache tile sizes
    - 6 flavors of software pipelining
    - scaling constants, transpose options, precisions, etc.
- An experiment
  - fix scheduling options
  - vary register tile sizes
  - 500 to 2500 "reasonable" implementations on 6 platforms


The Variations in Performance across Platforms
fraction of implementations
fraction of peak machine speed
10^{0}
10^{-1}
10^{-2}
10^{-3}
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Sun Ultra-IIi/333
Pentium II-300
PowerPC 604e
IBM RS/6000-590
MIPS R10k/175
Cray T3E Node


A Needle in a Haystack
16
14
12
10
8
6
4
2
n_0
2
4
6
8
10
12
14
16
m_0
600
550
500
450
400
350
300
250
200
150
100

Road Map

- Context
- Why search?
- **Stopping searches early**
- High-level run-time selection
- Summary

Stopping Searches Early

- Assume
  - dedicated resources limited
    - end-users perform searches
    - run-time searches
  - near-optimal implementation okay
- Can we stop the search early?
  - how early is "early?"
  - guarantees on quality?
- PHiPAC search procedure
  - generate implementations uniformly at random *without* replacement
  - measure performance

An Early Stopping Criterion

- Performance scaled from 0 (worst) to 1 (best)
- Goal: Stop after t implementations when

$$\text{Prob}[\ M_t \leq 1\text{-}\varepsilon\ ] < \alpha$$

  - M_t max observed performance at t
  - ε proximity to best
  - α degree of uncertainty
  - example: "find within top 5% with 10% uncertainty"
    - $\varepsilon = .05$, $\alpha = .1$

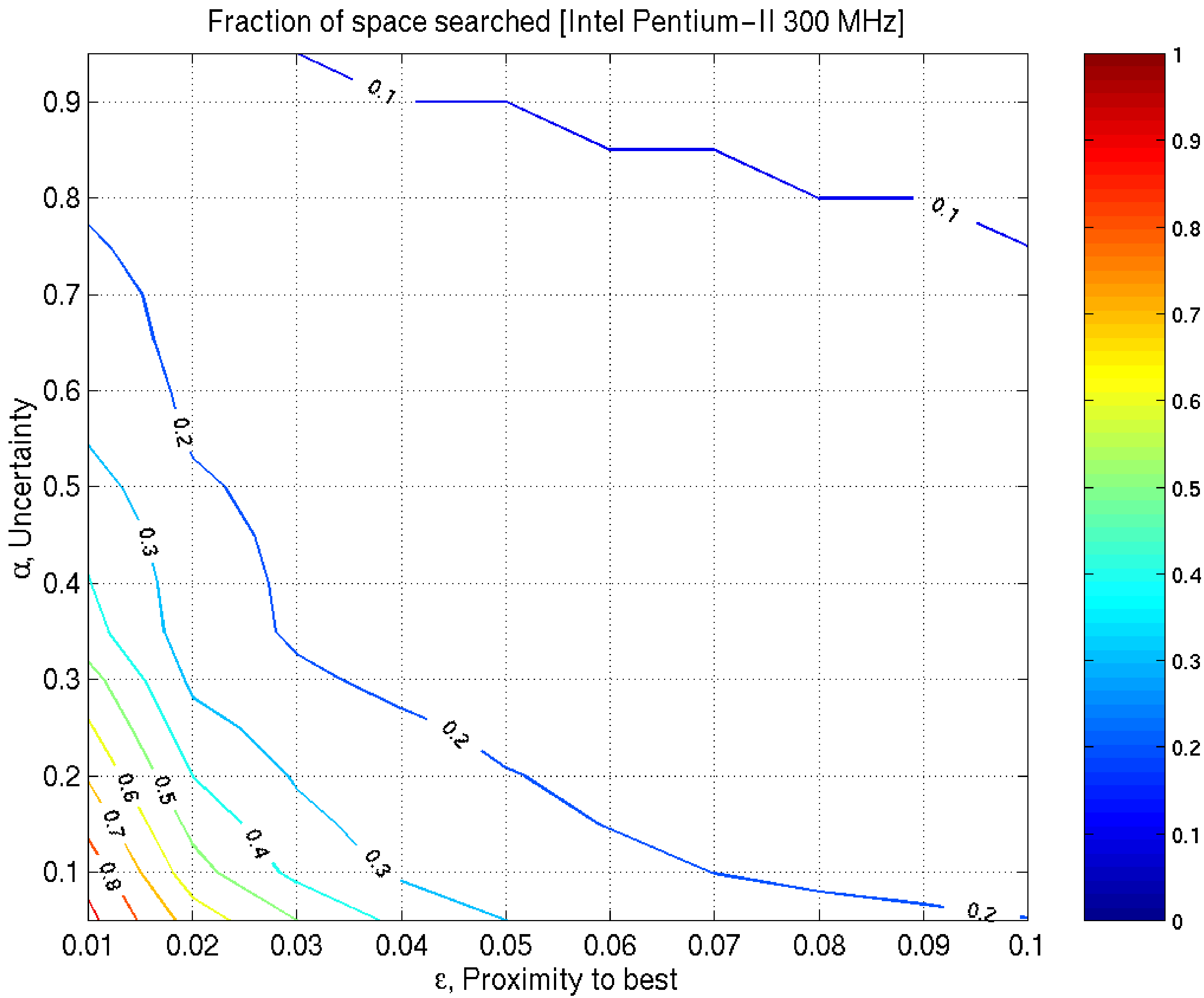
- Can show probability depends only on

$$F(x) = \text{Prob}[\ \mathit{performance} <= x\ ]$$

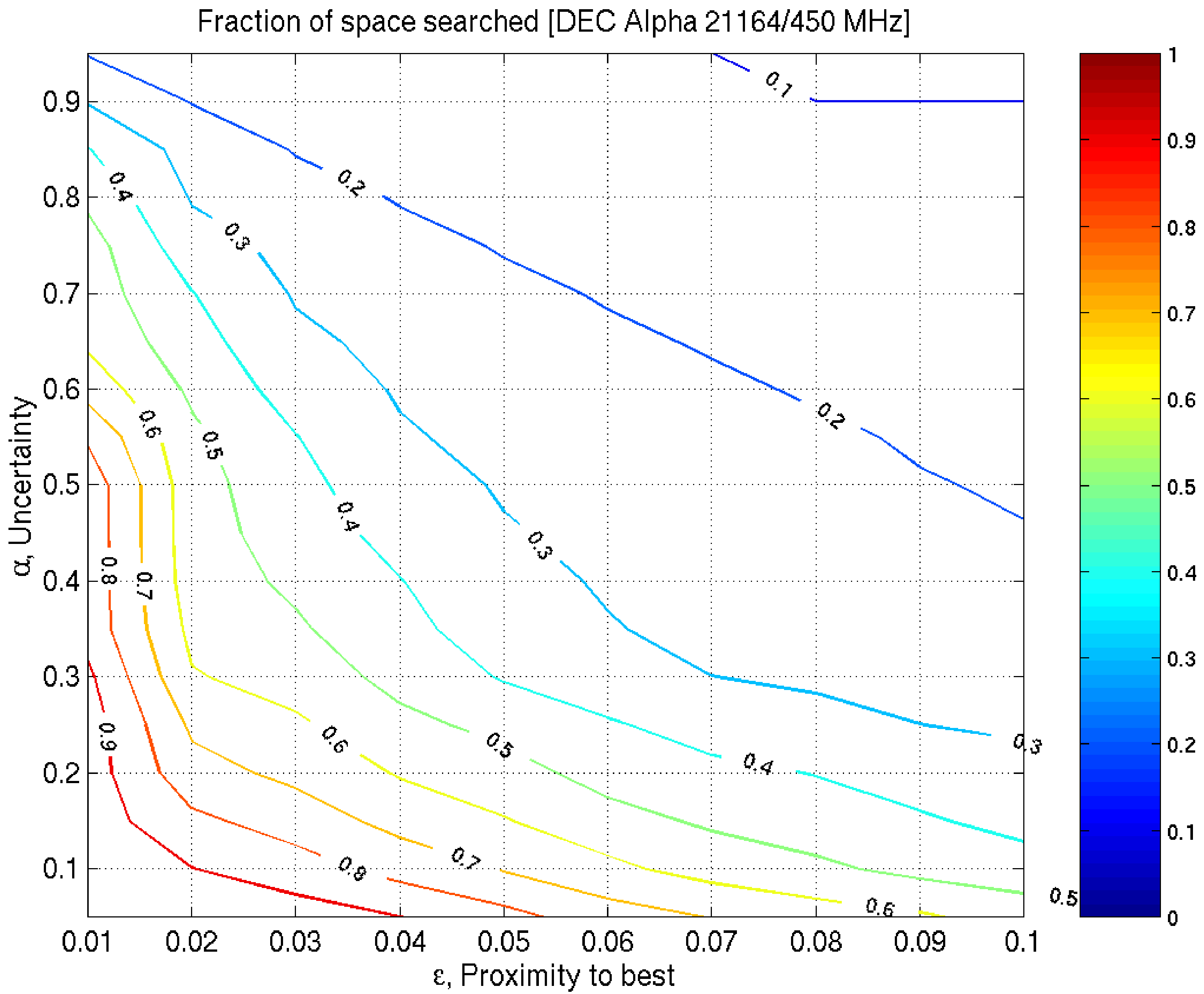
- Idea: Estimate $F(x)$ using observed samples

Stopping Algorithm

- User or library-builder chooses ε, α
- For each implementation t
  - Generate and benchmark
  - Estimate $F(x)$ using all observed samples
  - Calculate p := $\mathrm{Prob}[\, M_t <= 1-\varepsilon \,]$
  - Stop if $p < \alpha$
- Or, if you must stop at $t=T$, can output ε, α



Fraction of space searched [Intel Pentium−II 300 MHz]
α, Uncertainty
ε, Proximity to best
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
1
0



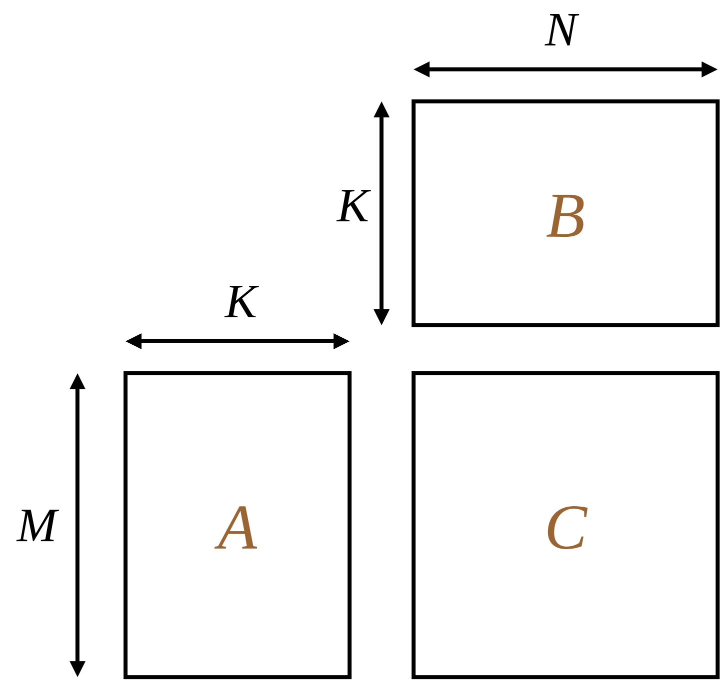
Fraction of space searched [DEC Alpha 21164/450 MHz]
α, Uncertainty
ε, Proximity to best
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
1
0

Road Map

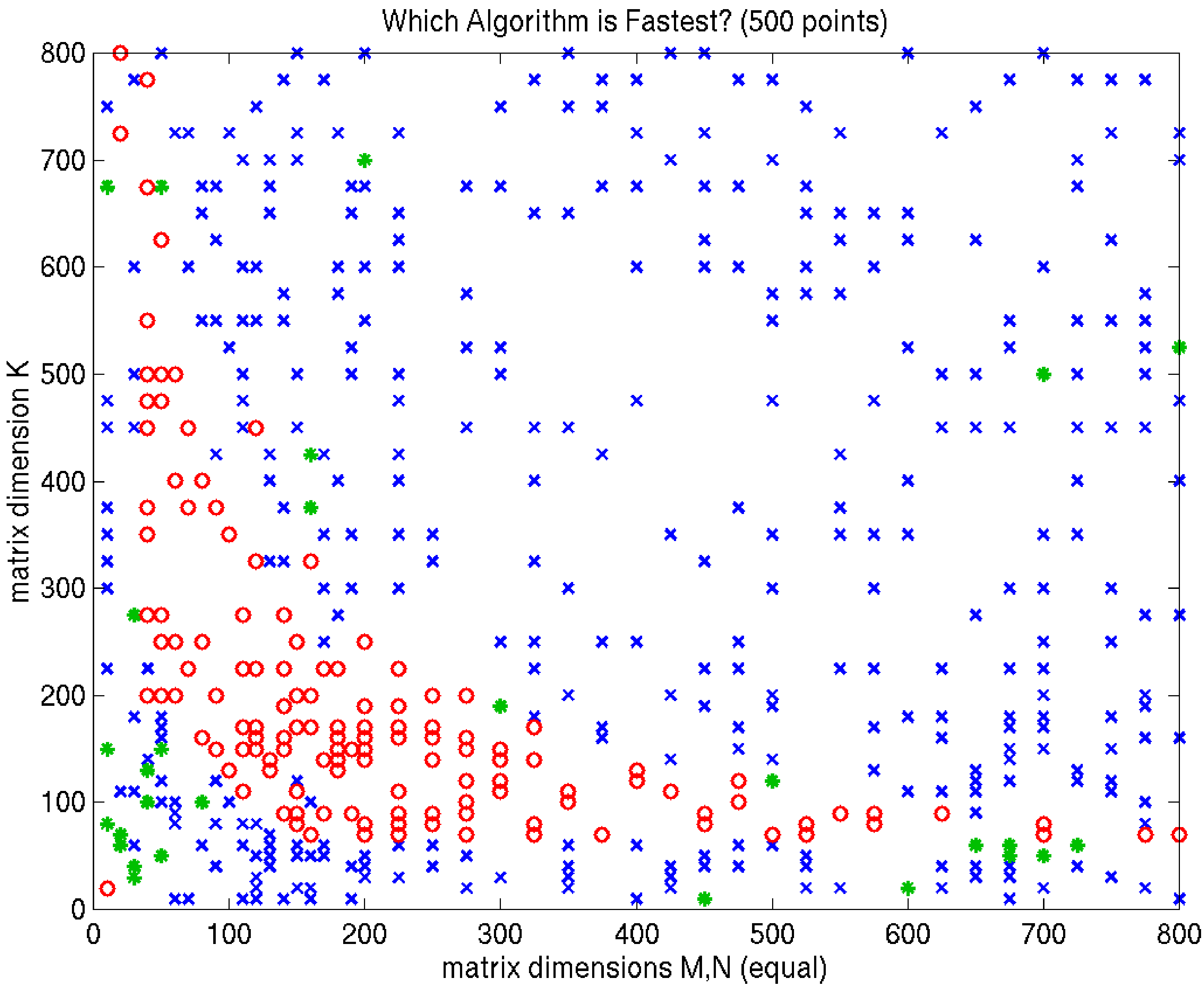
- Context
- Why search?
- Stopping searches early
- **High-level run-time selection**
- Summary

Run-Time Selection

- Assume
  - one implementation is *not* best for all inputs
  - a few, good implementations known
  - can benchmark
- How do we choose the "best" implementation at run-time?
- Example: matrix multiply, tuned for small (L1), medium (L2), and large workloads



$C = C + A*B$

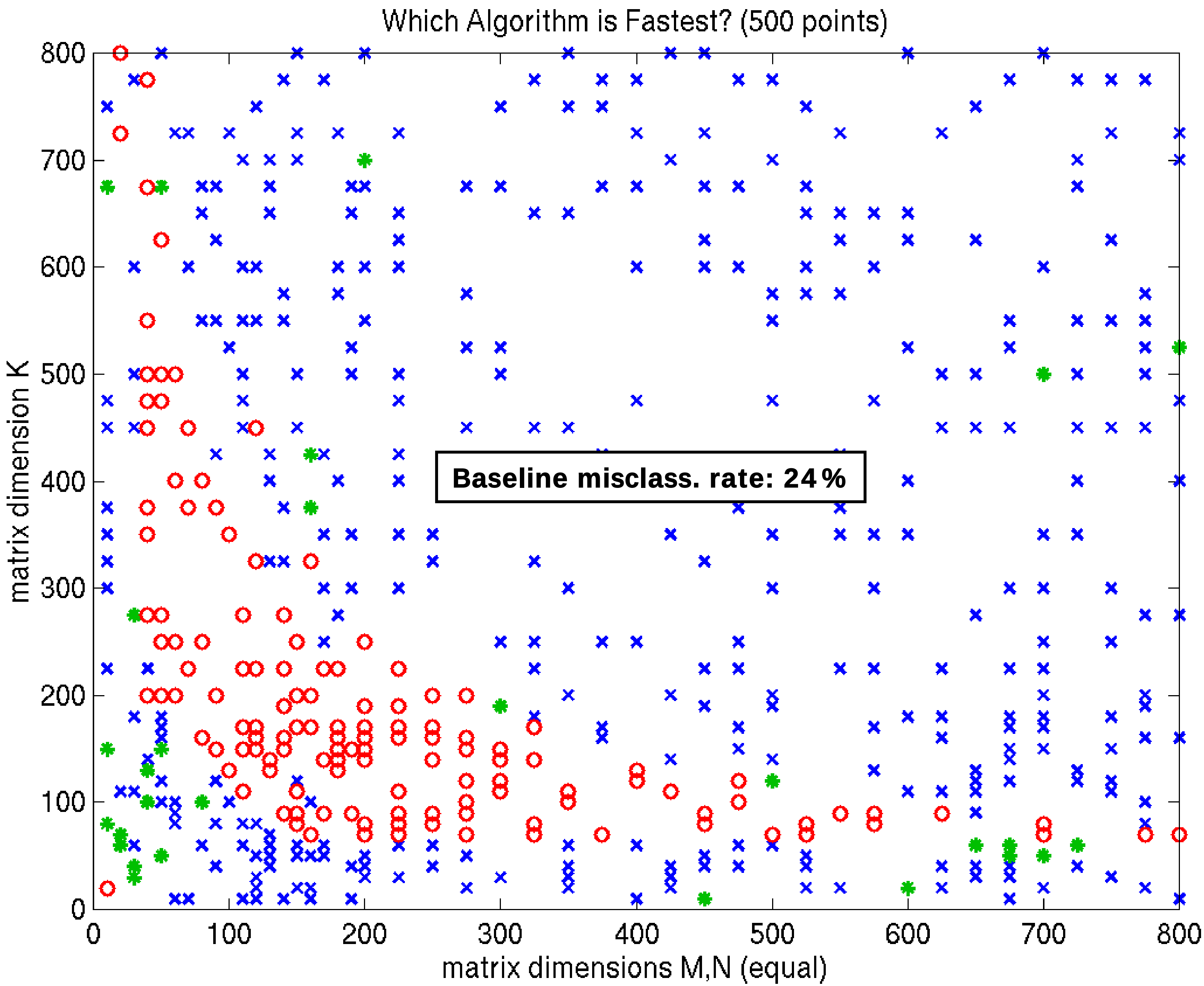

Which Algorithm is Fastest? (500 points)
matrix dimension K
matrix dimensions M,N (equal)
0
100
200
300
400
500
600
700
800

A Formal Framework

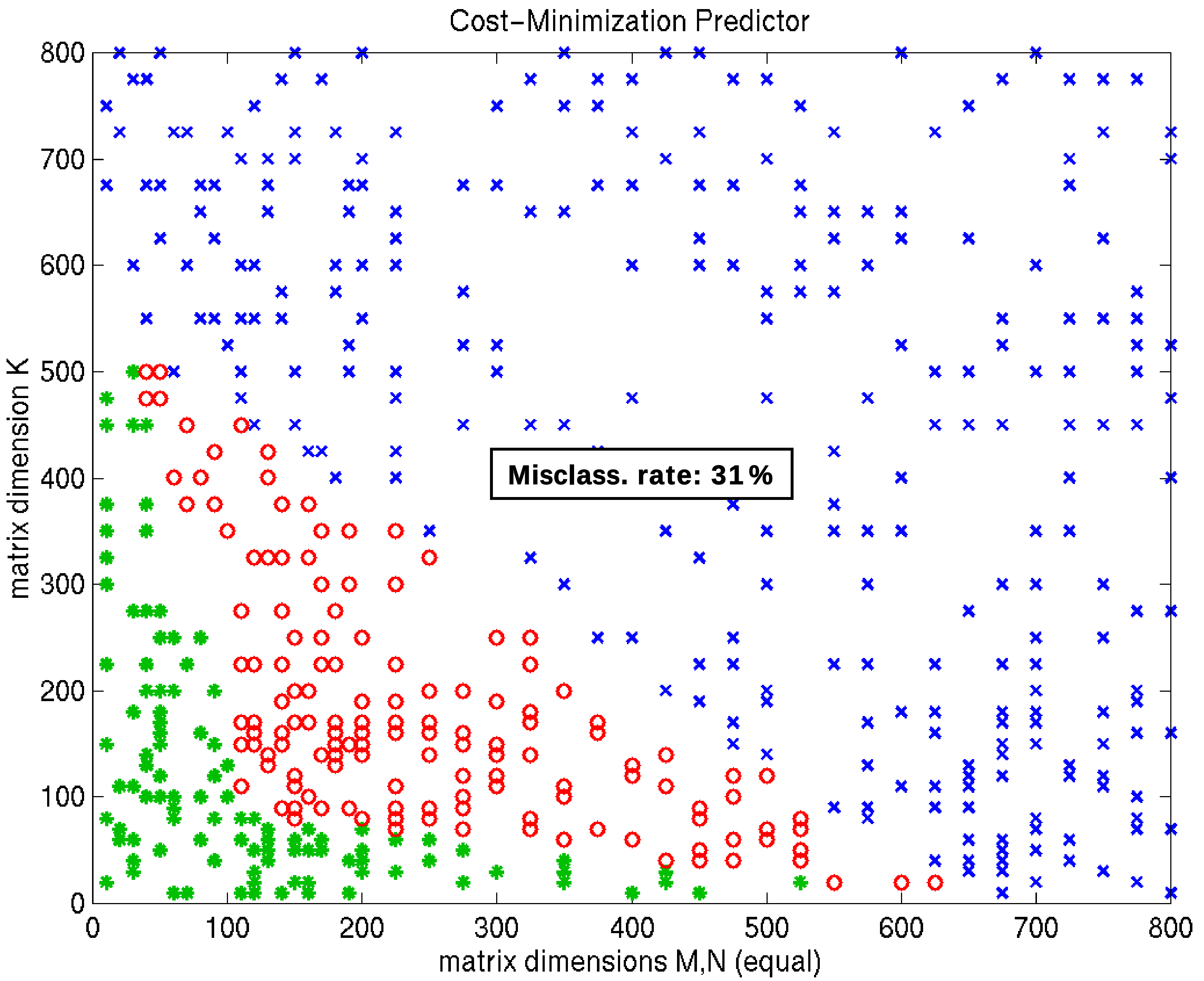
- Given
  - m implementations $A = \{a_1, a_2, \ldots, a_m\}$
  - n sample inputs (training set) $S_0 = \{s_1, s_2, \ldots, s_n\} \subseteq S$
  - execution time $T(a, s) : a \in A, s \in S$
- Find
  - decision function *f(s)* $f : S \to A$
  - returns "best" implementation on input s
  - *f(s)* cheap to evaluate

Solution Techniques (Overview)

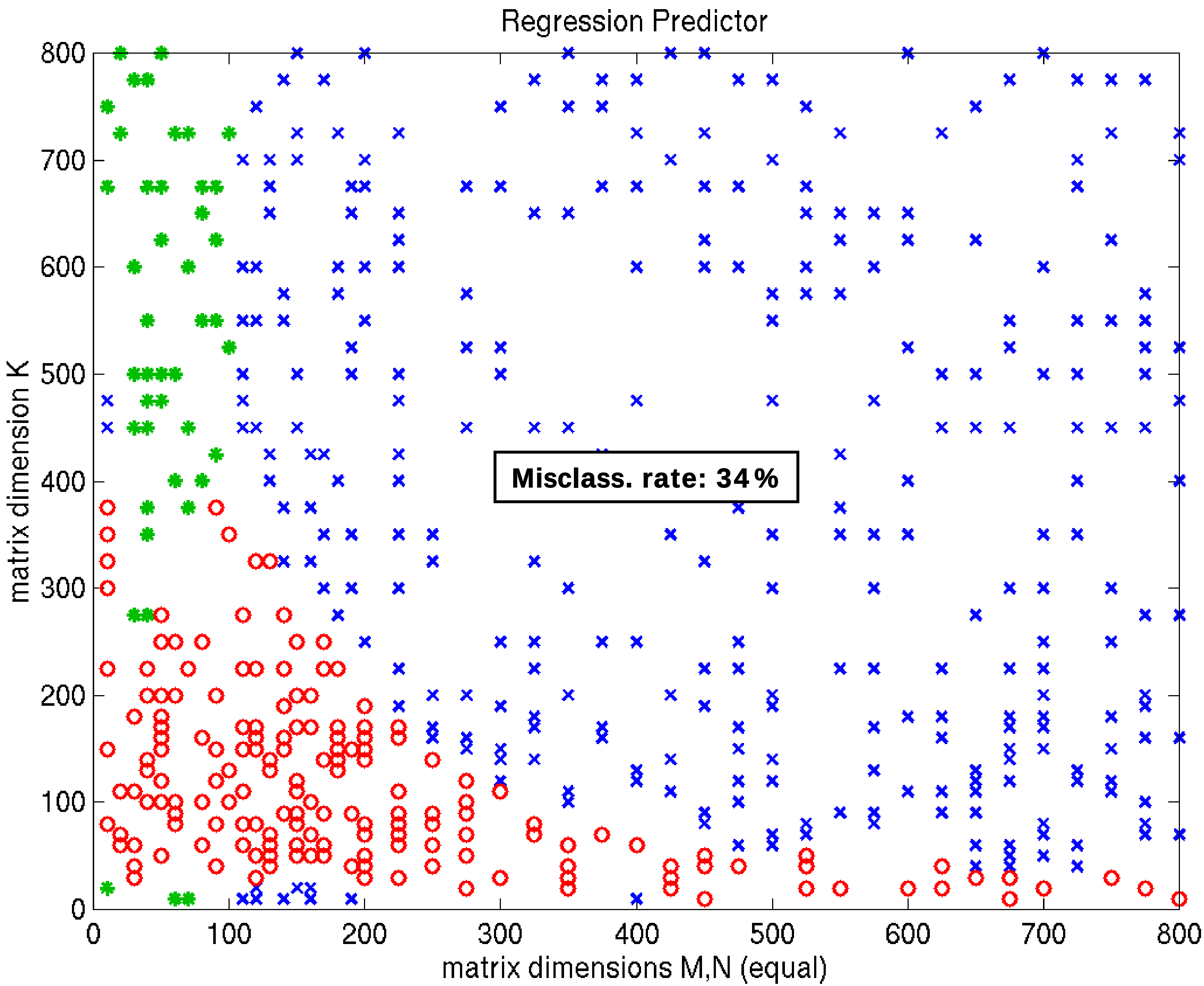
- **Method 1**: Cost Minimization
  - select geometric boundaries that minimize overall execution time on samples
    - pro: intuitive, $f(s)$ cheap
    - con: ad hoc, geometric assumptions
- **Method 2**: Regression (Brewer, 1995)
  - model run-time of each implementation
    e.g., $T_a(N) = b_3 N^3 + b_2 N^2 + b_1 N + b_0$
    - pro: simple, standard
    - con: user must define model
- **Method 3**: Support Vector Machines
  - statistical classification
    - pro: solid theory, many successful applications
    - con: heavy training and prediction machinery



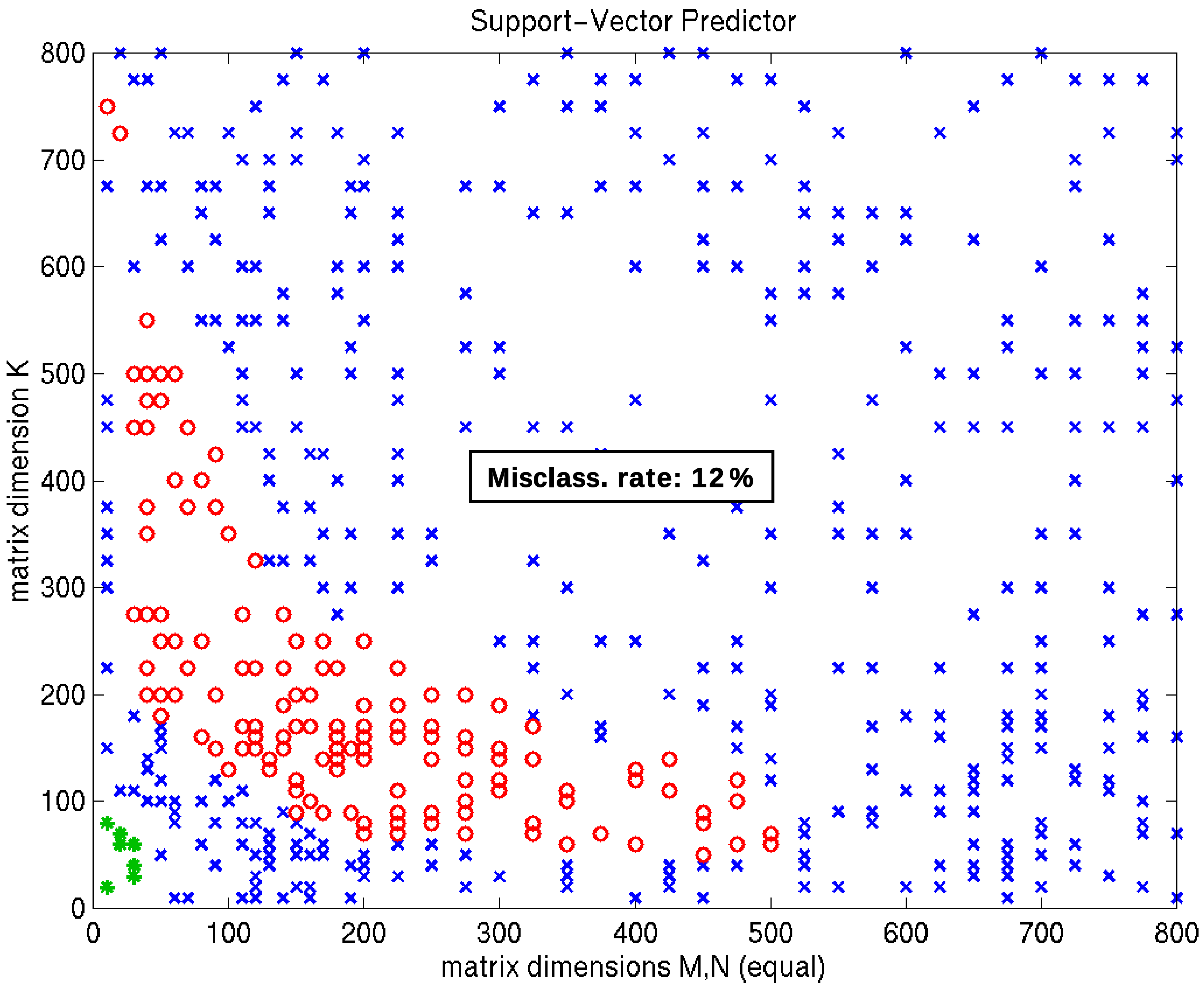
Which Algorithm is Fastest? (500 points)
Baseline misclass. rate: 24%
matrix dimension K
matrix dimensions M,N (equal)
0
100
200
300
400
500
600
700
800

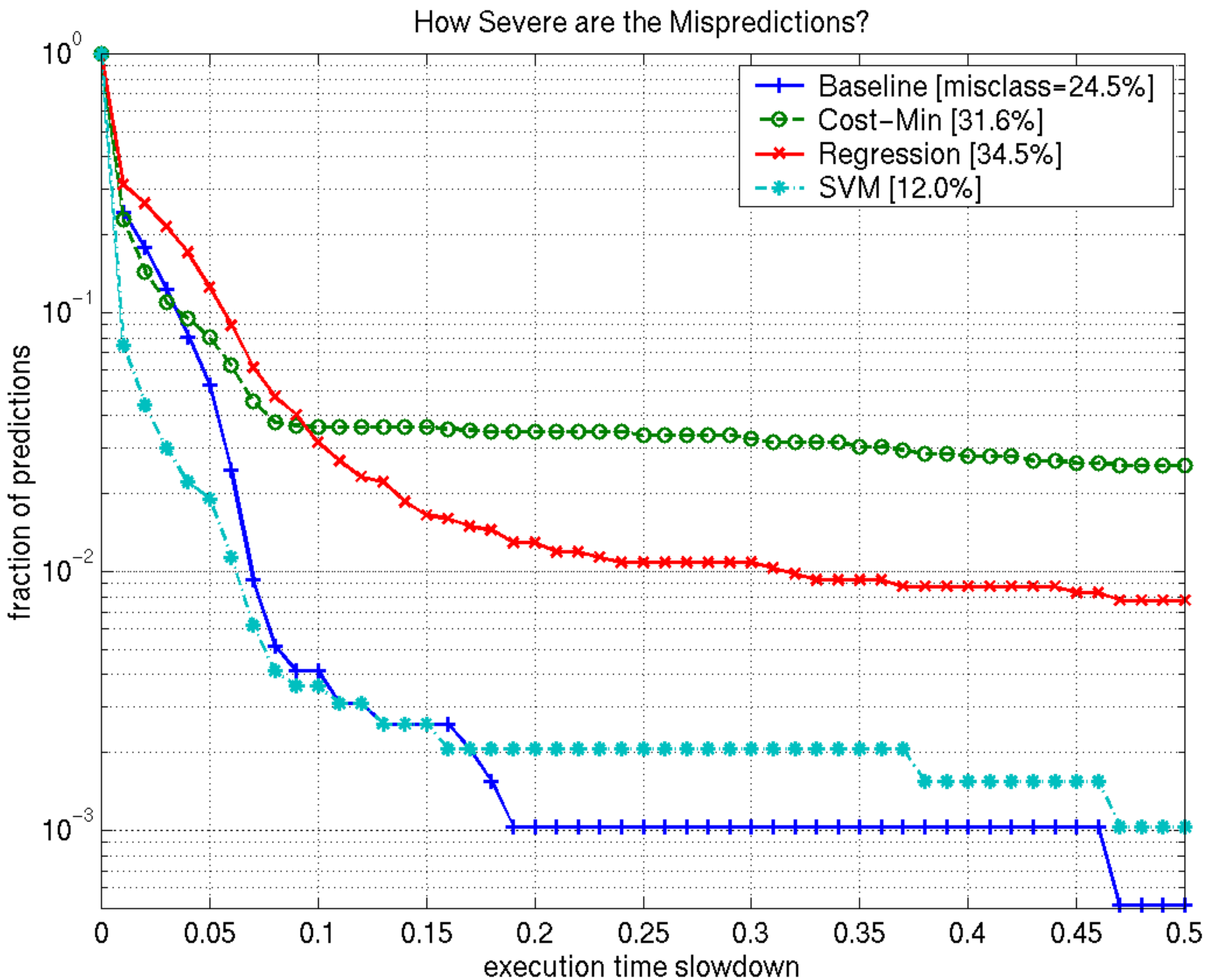


Cost−Minimization Predictor
Misclass. rate: 31 %
matrix dimension K
matrix dimensions M,N (equal)
0
100
200
300
400
500
600
700
800



Regression Predictor
Misclass. rate: 34 %
matrix dimension K
matrix dimensions M,N (equal)
0
100
200
300
400
500
600
700
800


Support−Vector Predictor
Misclass. rate: 12 %
matrix dimension K
matrix dimensions M,N (equal)
0
100
200
300
400
500
600
700
800



Notes:

- "Baseline" predictor always chooses the implementation that was best on the majority of sample inputs.
- Cost of cost-min and regression predictions: ~O(3x3) matmul.
- Cost of SVM prediction: ~O(64x64) matmul.

Road Map

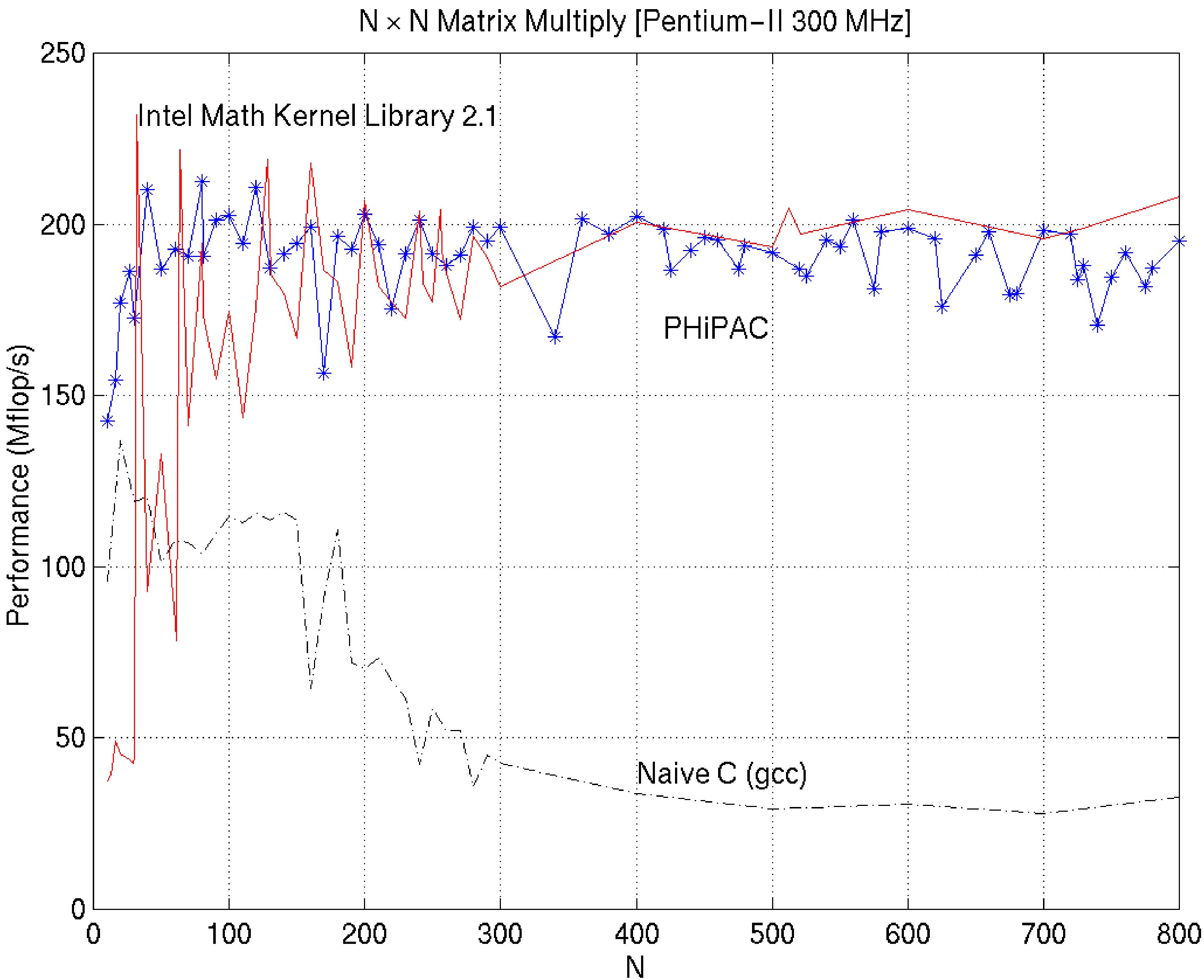
- Context
- Why search?
- Stopping searches early
- High-level run-time selection
- **Summary**

Summary

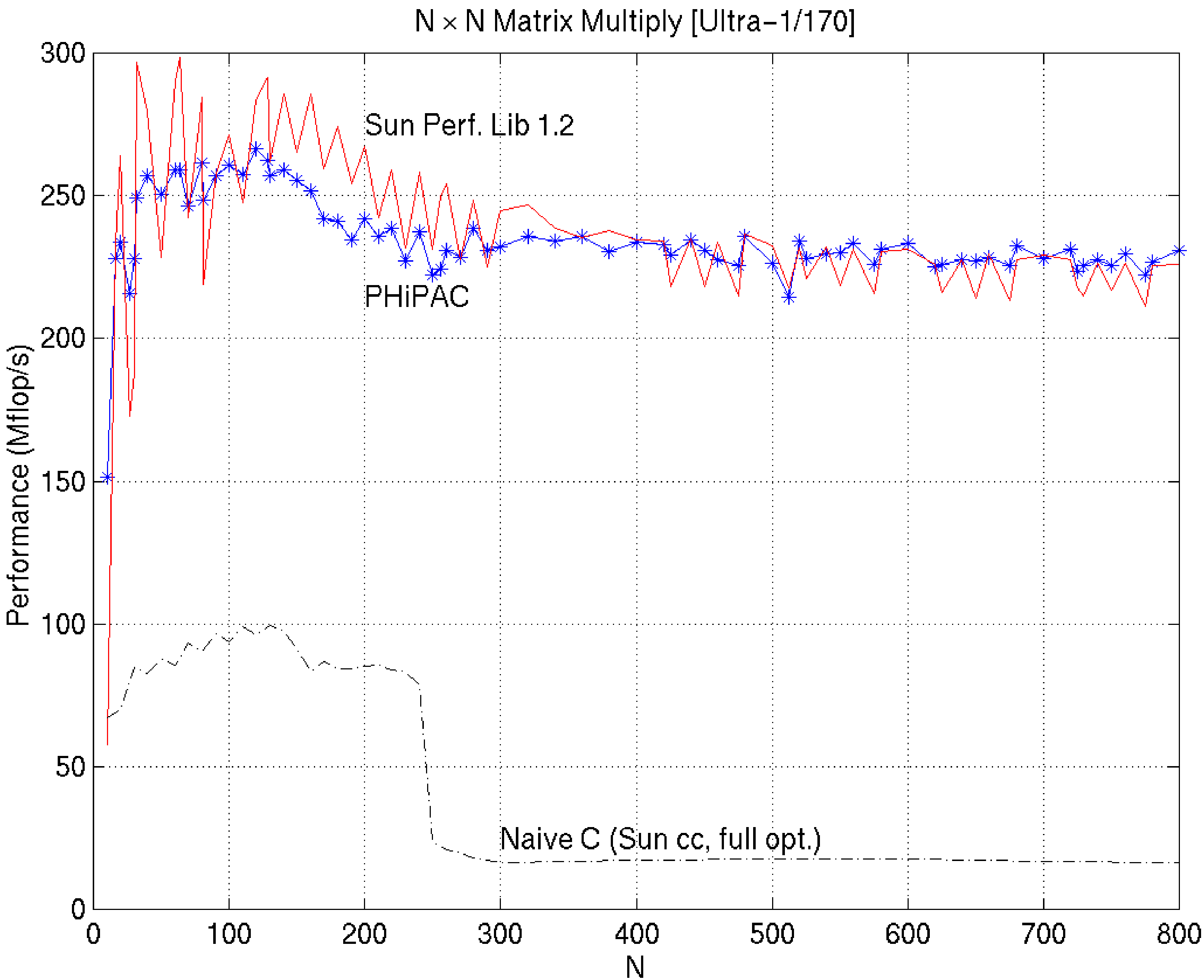
- Finding the best implementation can be like searching for a needle in a haystack
- Early stopping
  - simple and automated
  - informative criteria
- High-level run-time selection
  - formal framework
  - error metrics
- More ideas
  - search directed by statistical correlation
  - other stopping models (cost-based) for run-time search
    - E.g., run-time sparse matrix reorganization
  - large design space for run-time selection

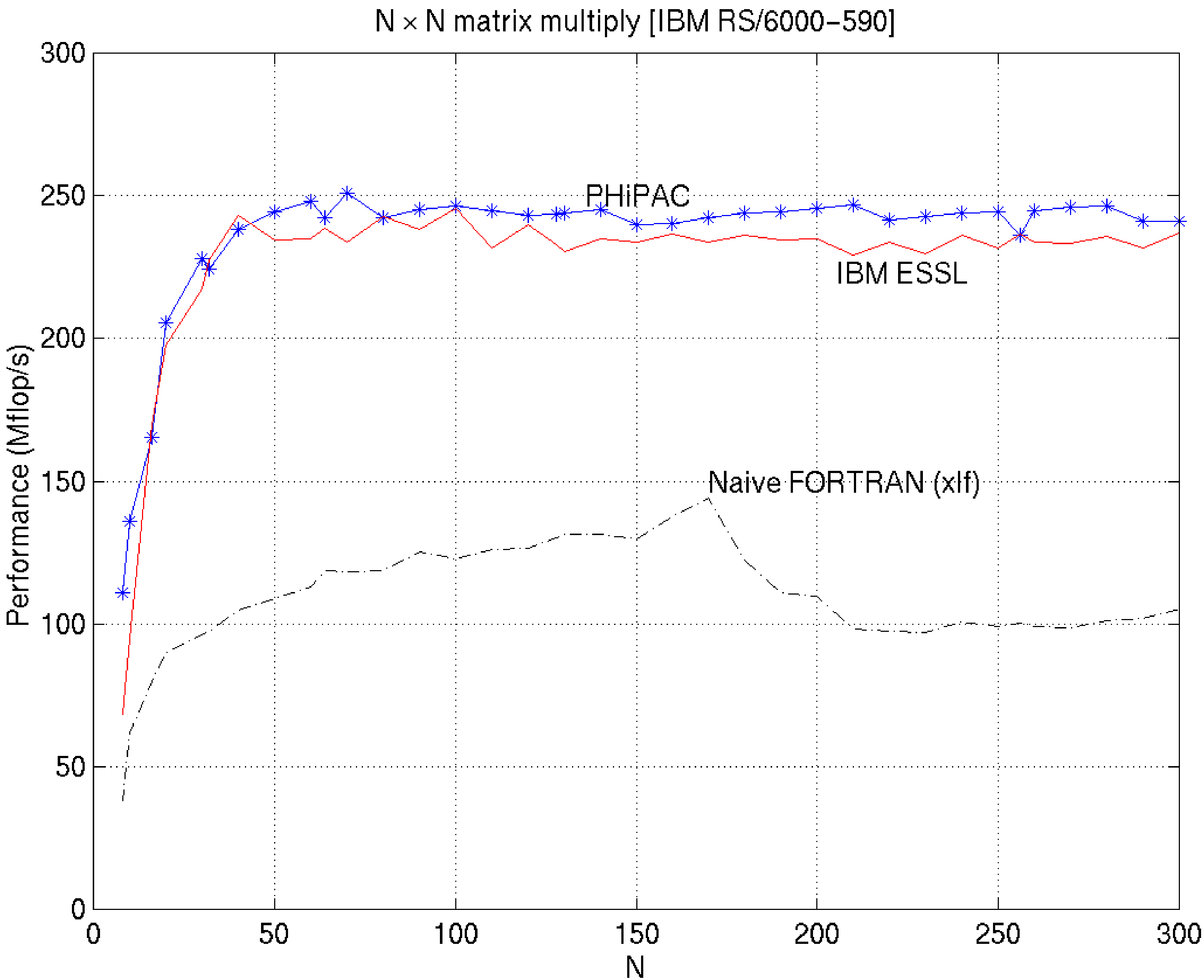
Extra Slides

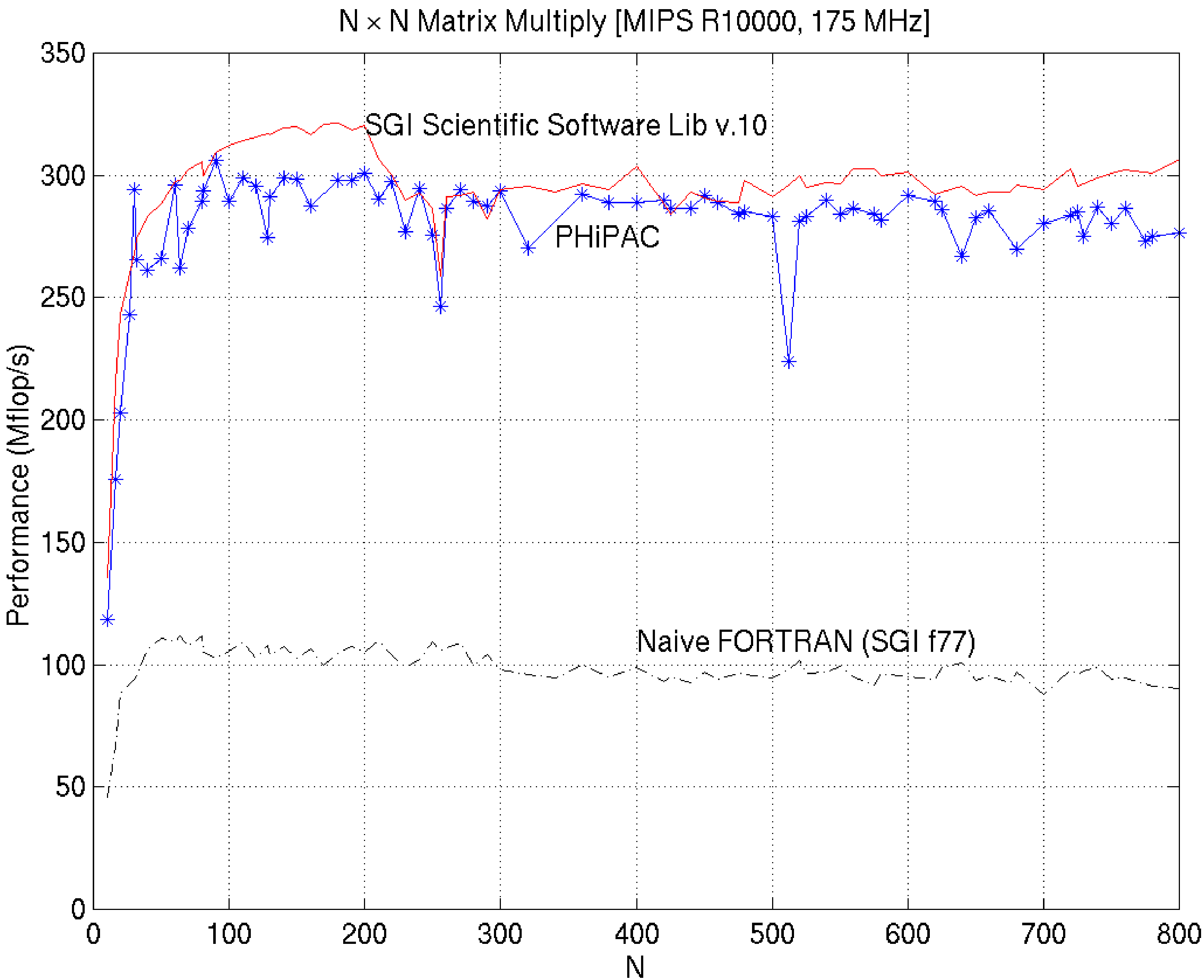
More detail (time and/or questions permitting)

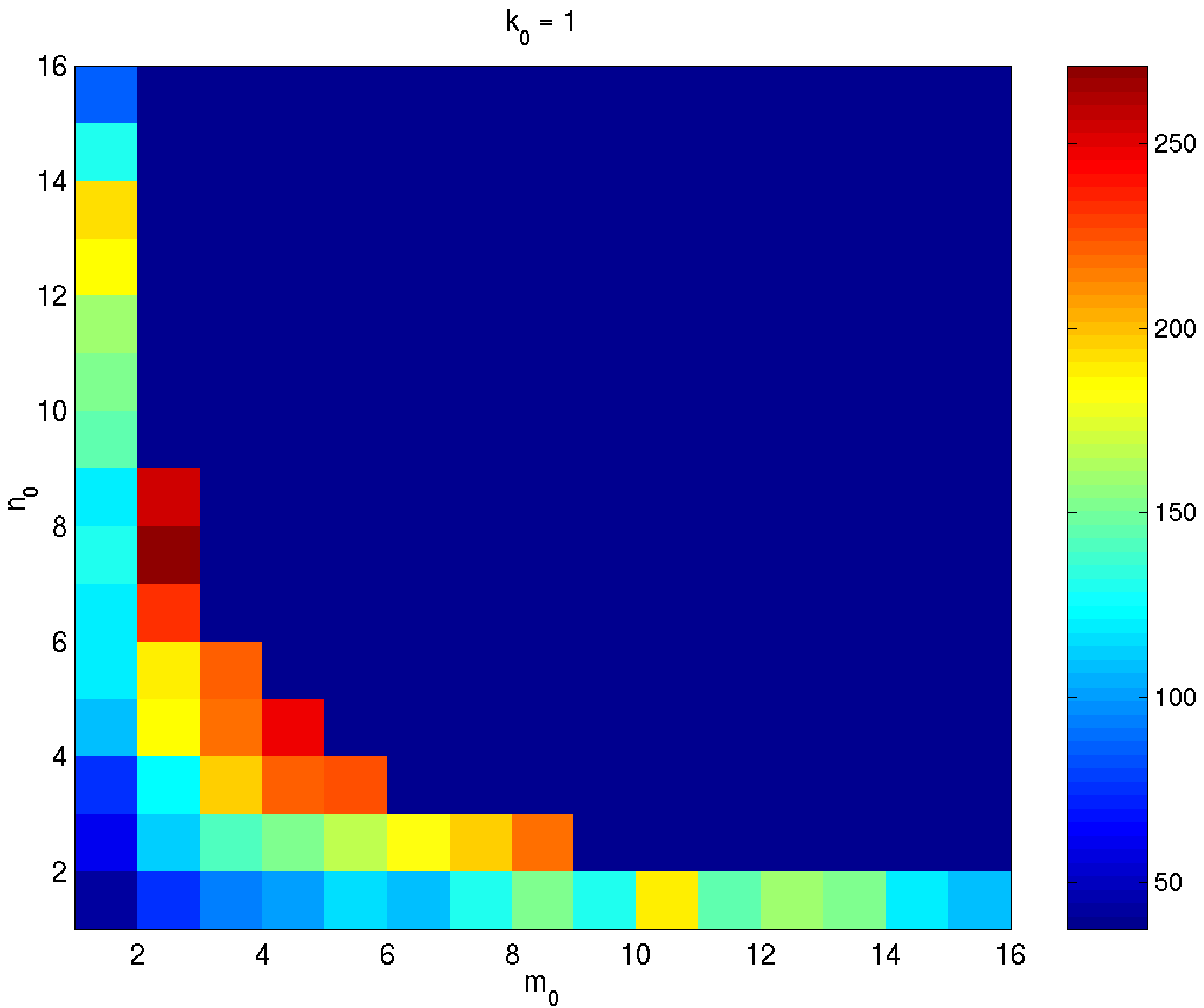


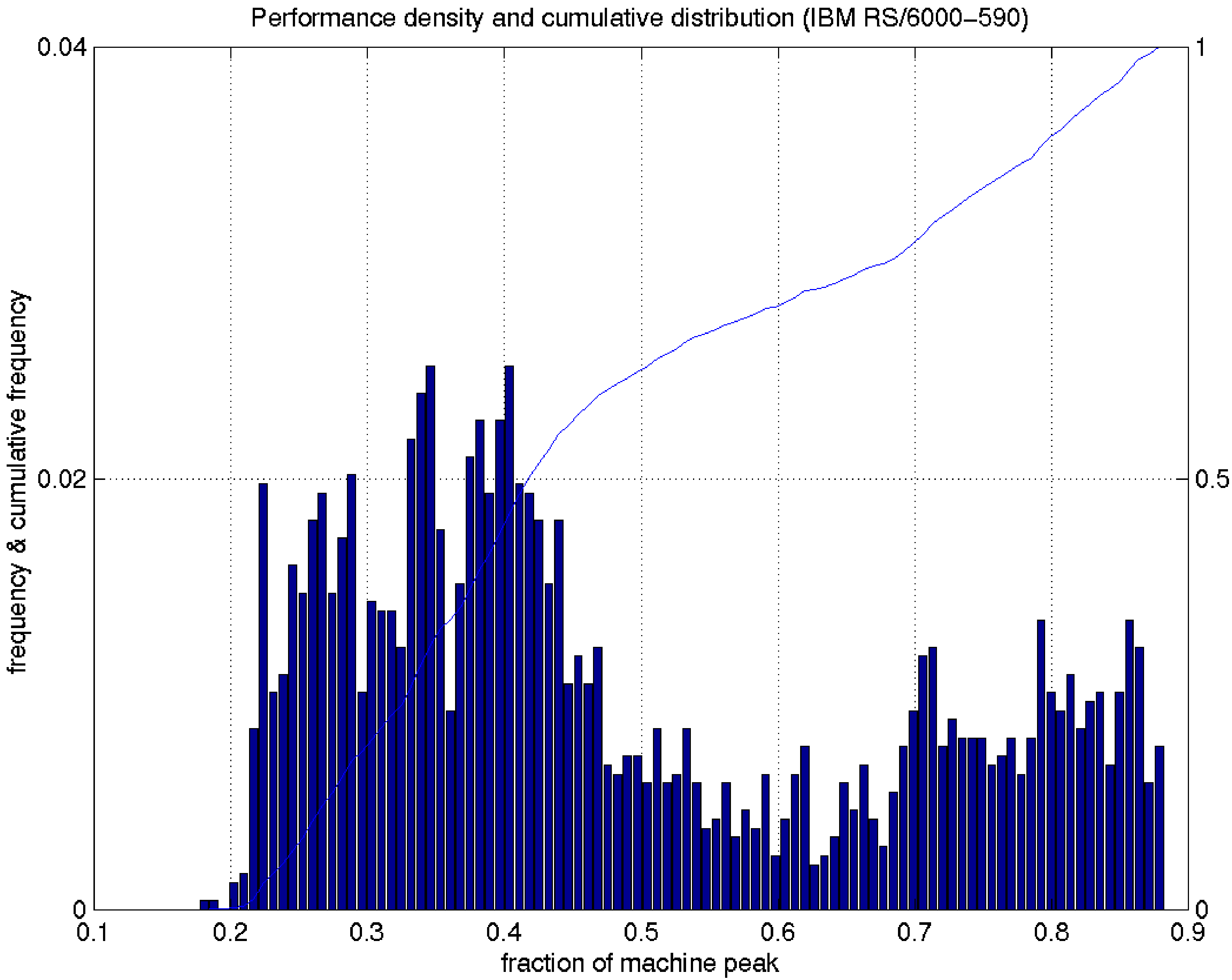
N × N Matrix Multiply [Pentium−II 300 MHz]
Intel Math Kernel Library 2.1
PHiPAC
Naive C (gcc)
Performance (Mflop/s)
N
0
50
100
150
200
250
0
100
200
300
400
500
600
700
800


N × N Matrix Multiply [Ultra−1/170]
Sun Perf. Lib 1.2
PHiPAC
Naive C (Sun cc, full opt.)
Performance (Mflop/s)
N
0
50
100
150
200
250
300
0
100
200
300
400
500
600
700
800

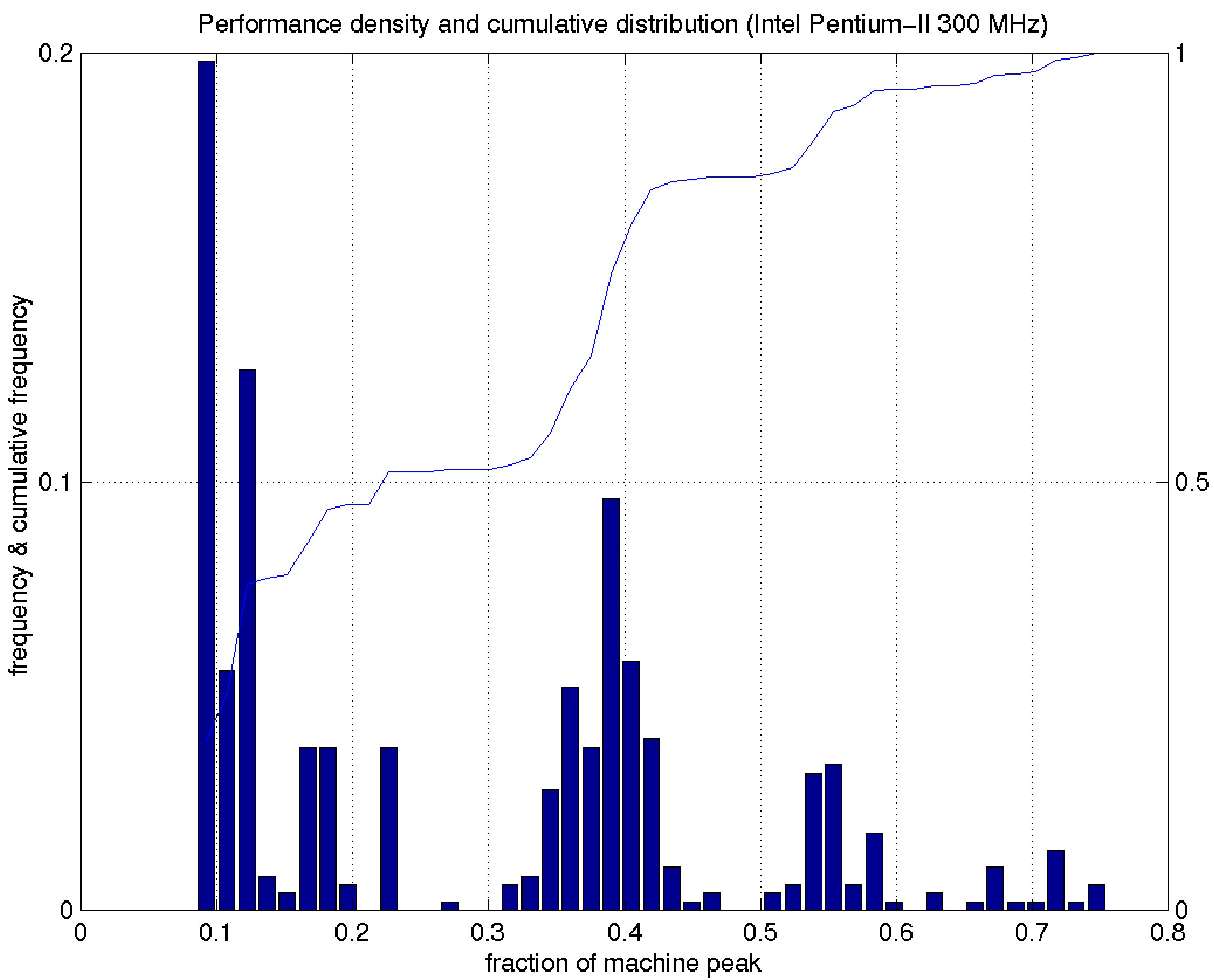

N × N matrix multiply [IBM RS/6000-590]
PHiPAC
IBM ESSL
Naive FORTRAN (xlf)
Performance (Mflop/s)
N
0
50
100
150
200
250
300

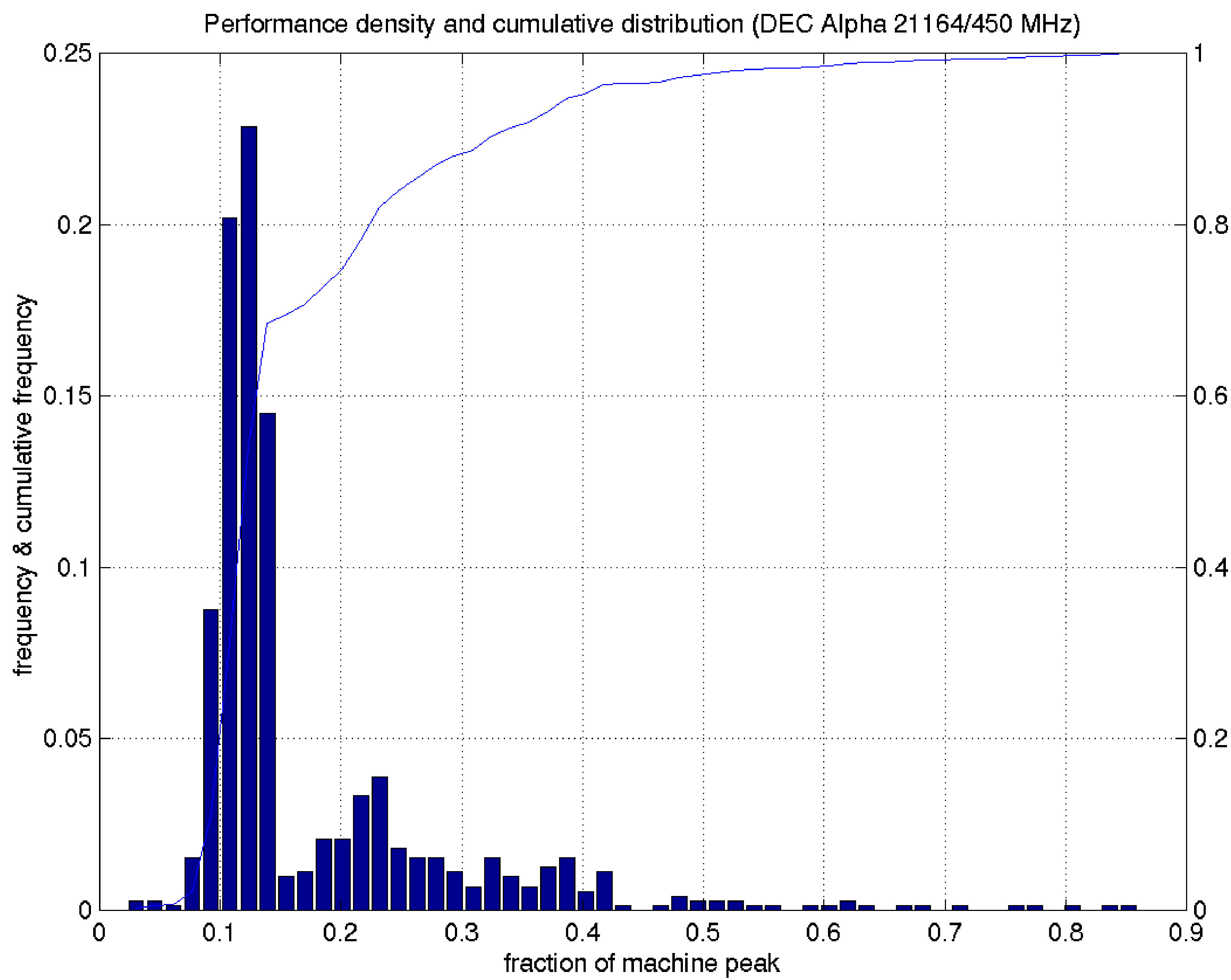

N × N Matrix Multiply [MIPS R10000, 175 MHz]
SGI Scientific Software Lib v.10
PHiPAC
Naive FORTRAN (SGI f77)
Performance (Mflop/s)
N
0
50
100
150
200
250
300
350
0
100
200
300
400
500
600
700
800


k_0 = 1
n_0
m_0
16
14
12
10
8
6
4
2
2
4
6
8
10
12
14
16
250
200
150
100
50

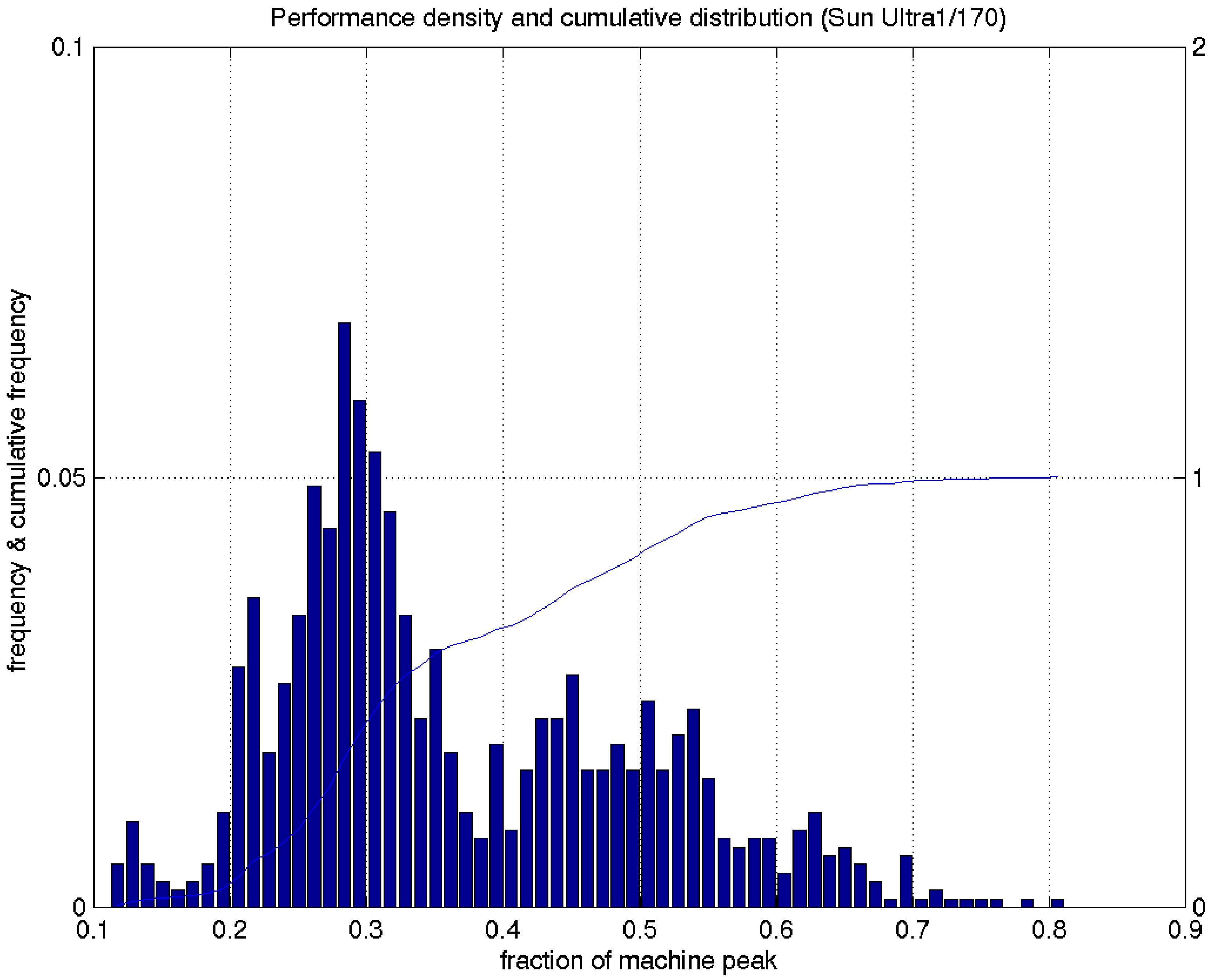


Performance density and cumulative distribution (IBM RS/6000−590)
0.04
0.02
0
1
0.5
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
fraction of machine peak
frequency & cumulative frequency

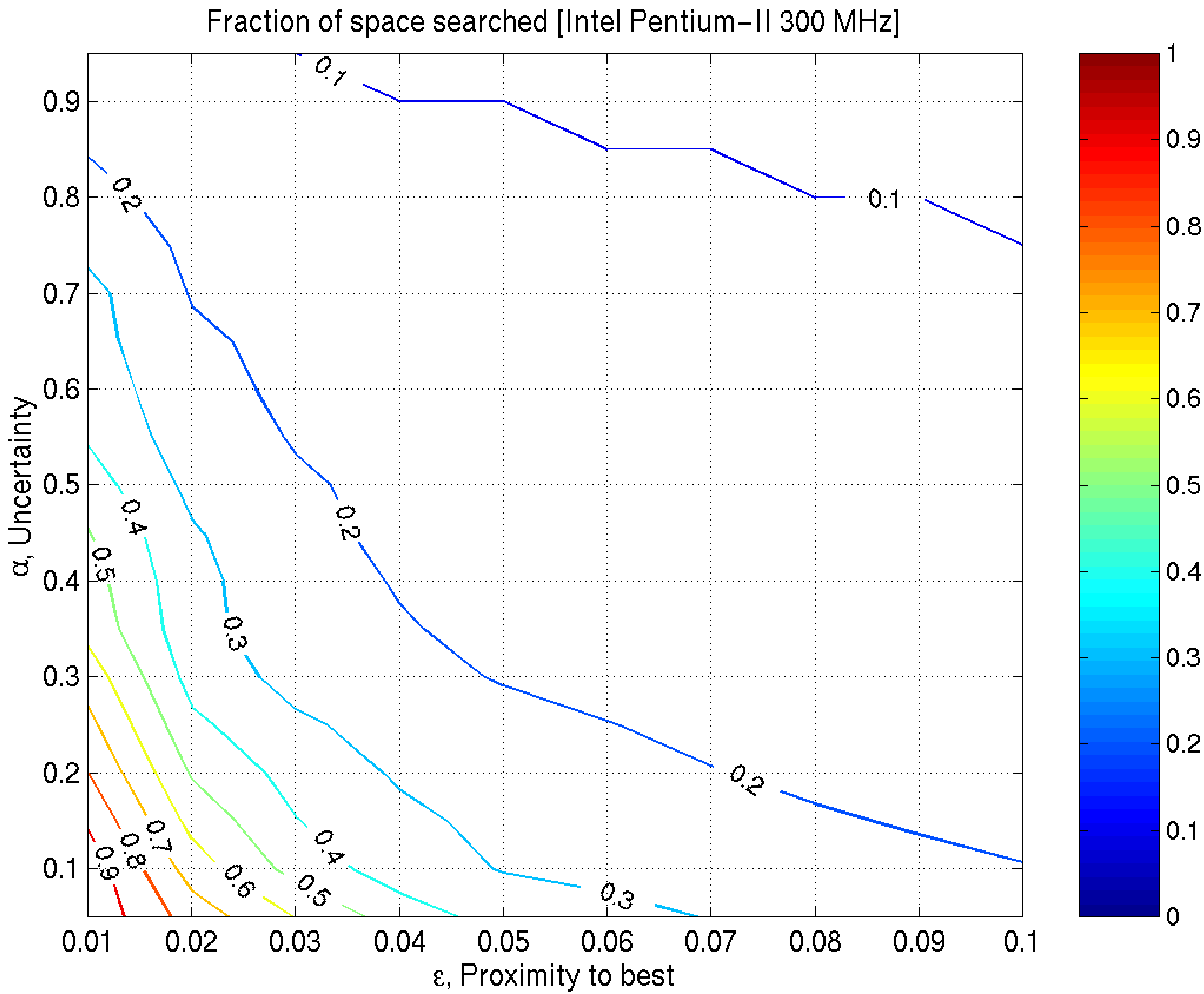

Performance density and cumulative distribution (Intel Pentium−II 300 MHz)
frequency & cumulative frequency
fraction of machine peak
0.2
0.1
0
1
0.5
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8

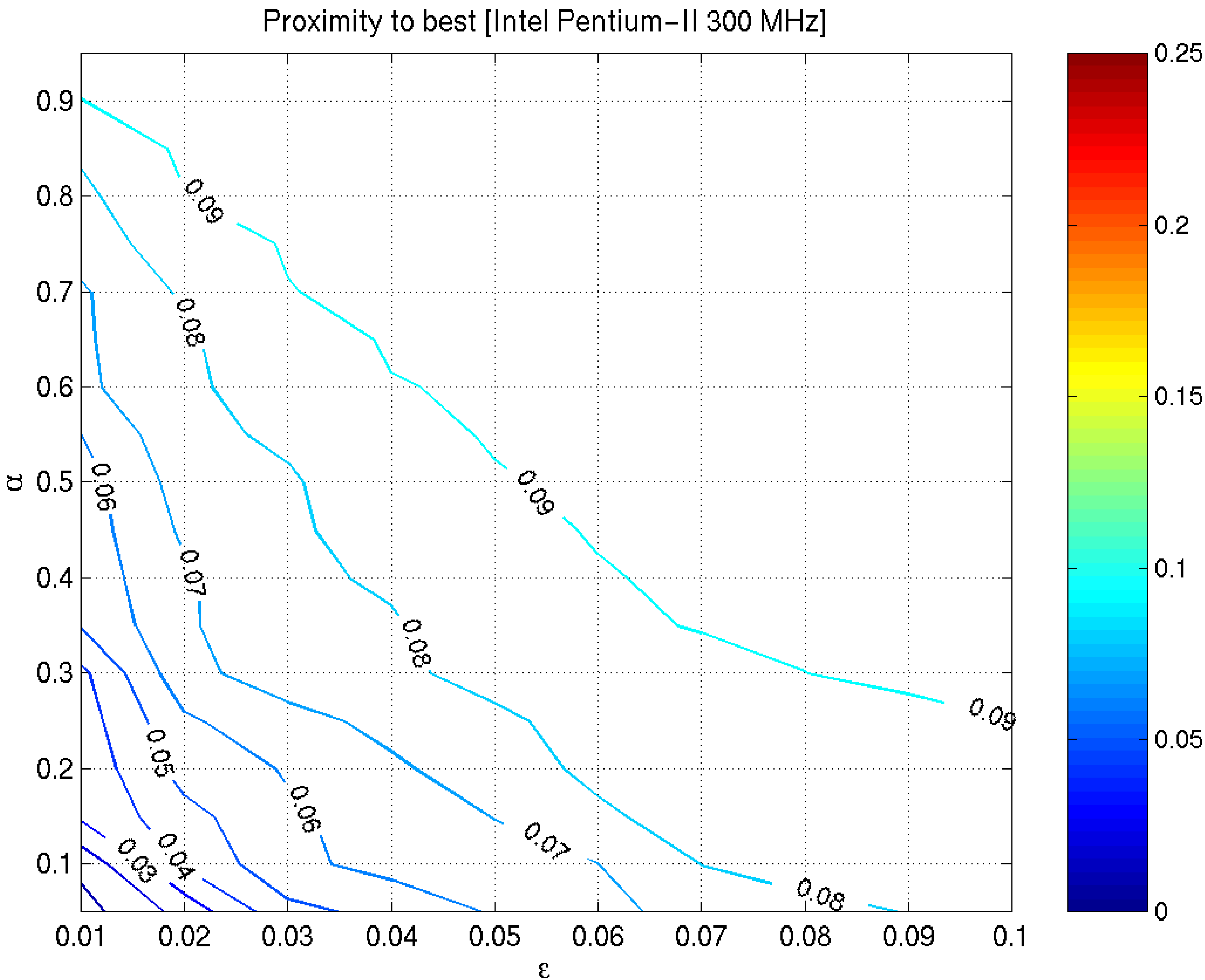


Performance density and cumulative distribution (DEC Alpha 21164/450 MHz)
frequency & cumulative frequency
fraction of machine peak
0.25
0.2
0.15
0.1
0.05
0
1
0.8
0.6
0.4
0.2
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

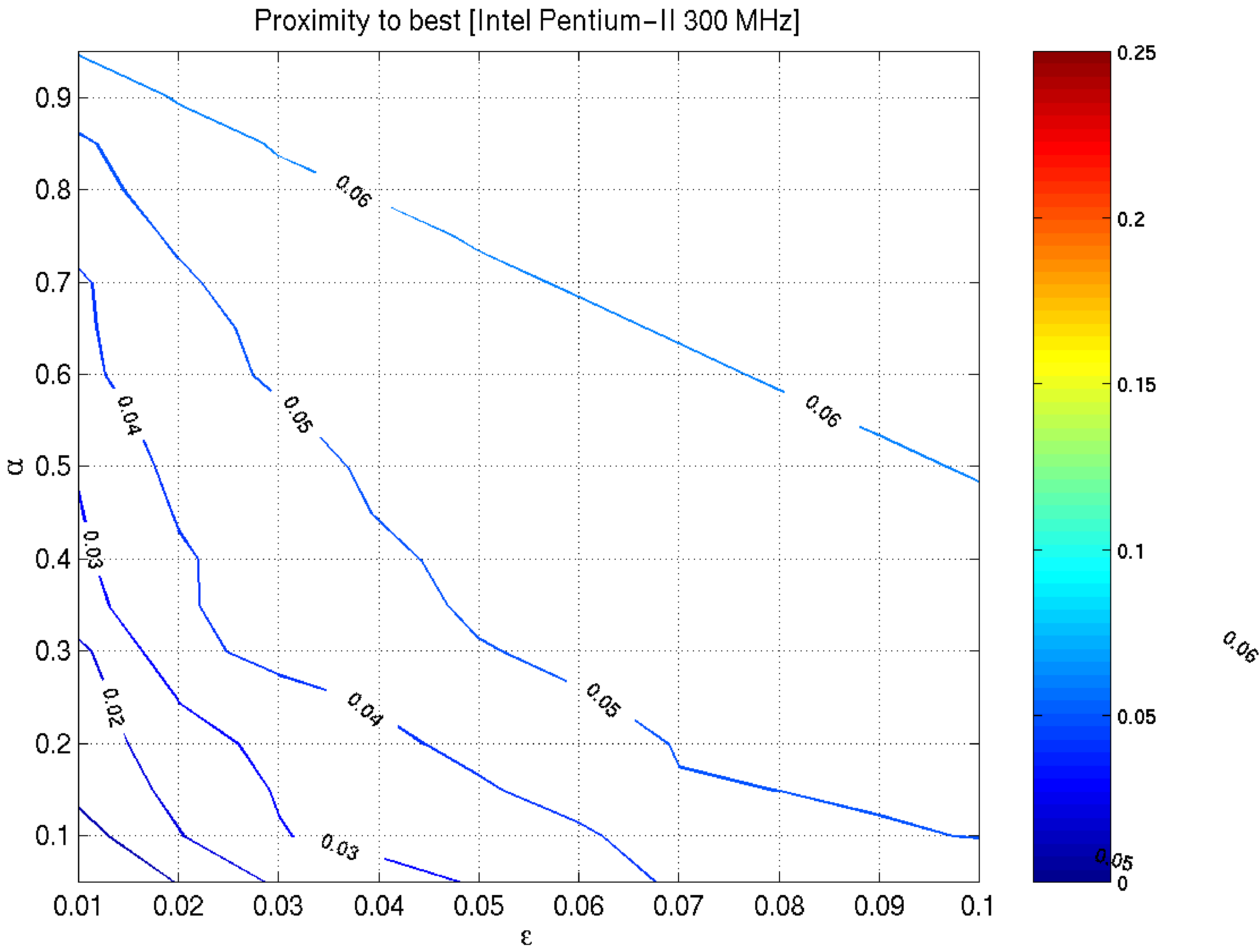


Performance density and cumulative distribution (Sun Ultra1/170)
frequency & cumulative frequency
fraction of machine peak
0.1
0.05
0
2
1
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

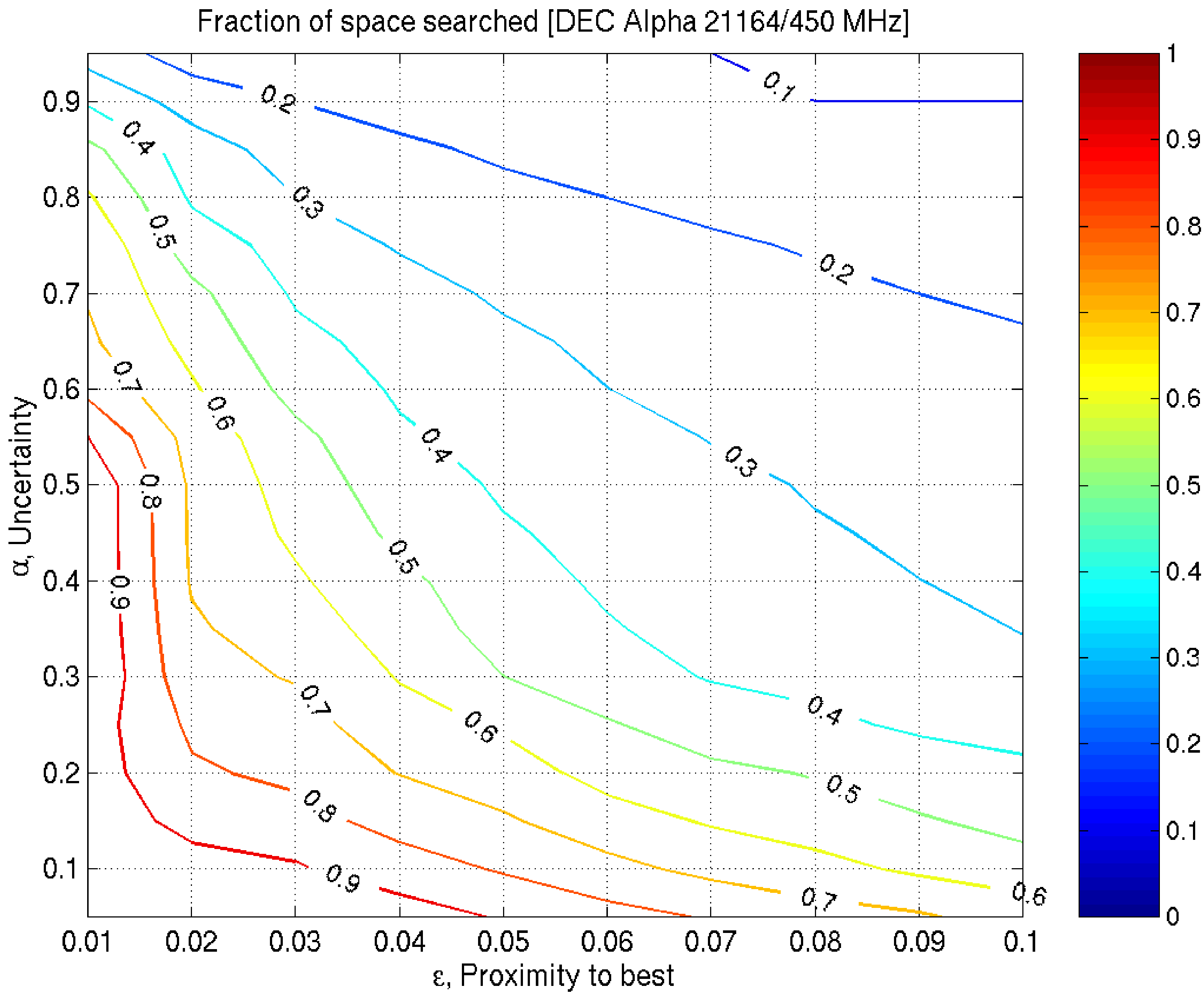

Fraction of space searched [Intel Pentium−II 300 MHz]
α, Uncertainty
ε, Proximity to best
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
0
1

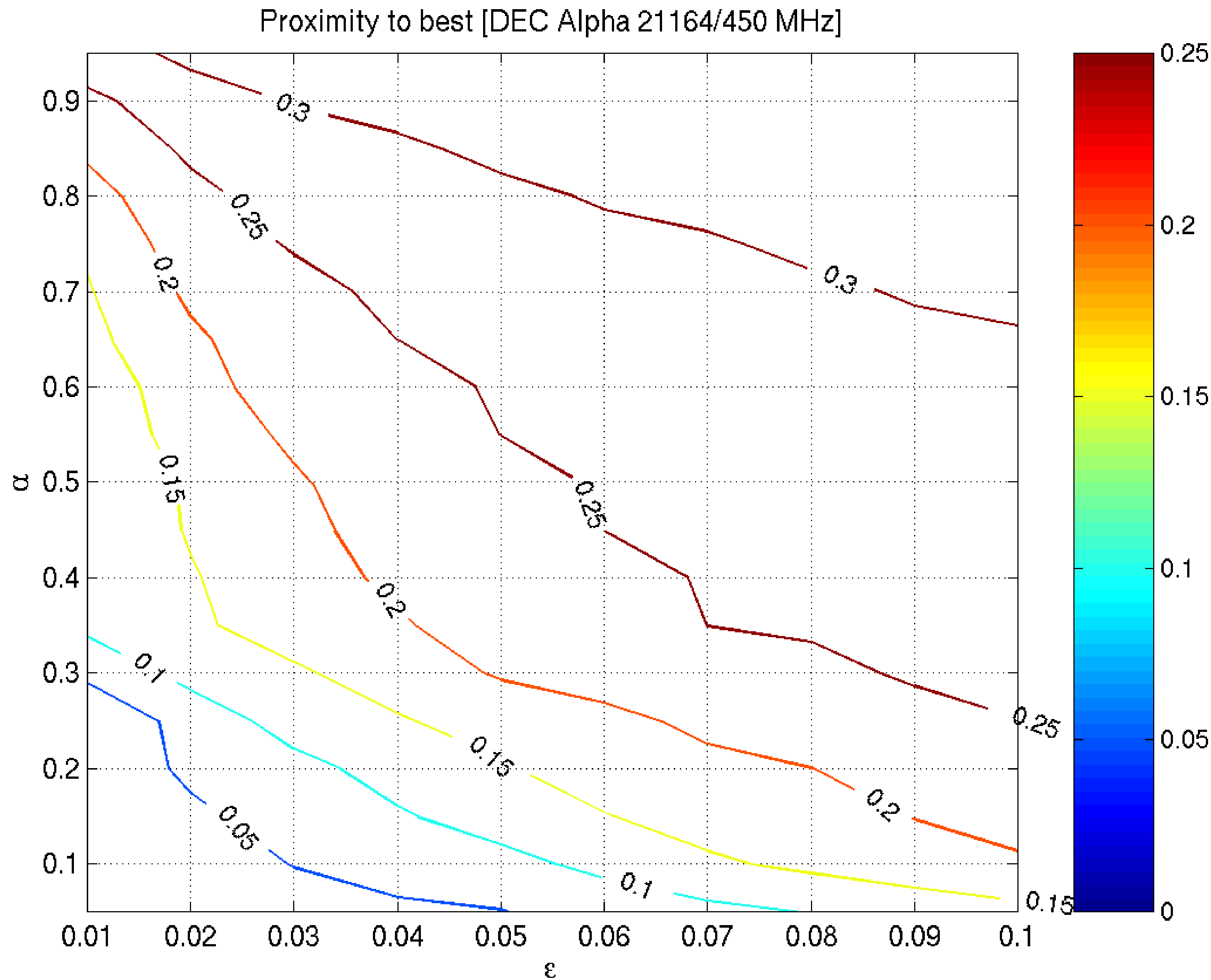


Proximity to best [Intel Pentium−II 300 MHz]
α
ε
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
0.25
0.2
0.15
0.1
0.05
0
0.03
0.04
0.05
0.06
0.07
0.08
0.09

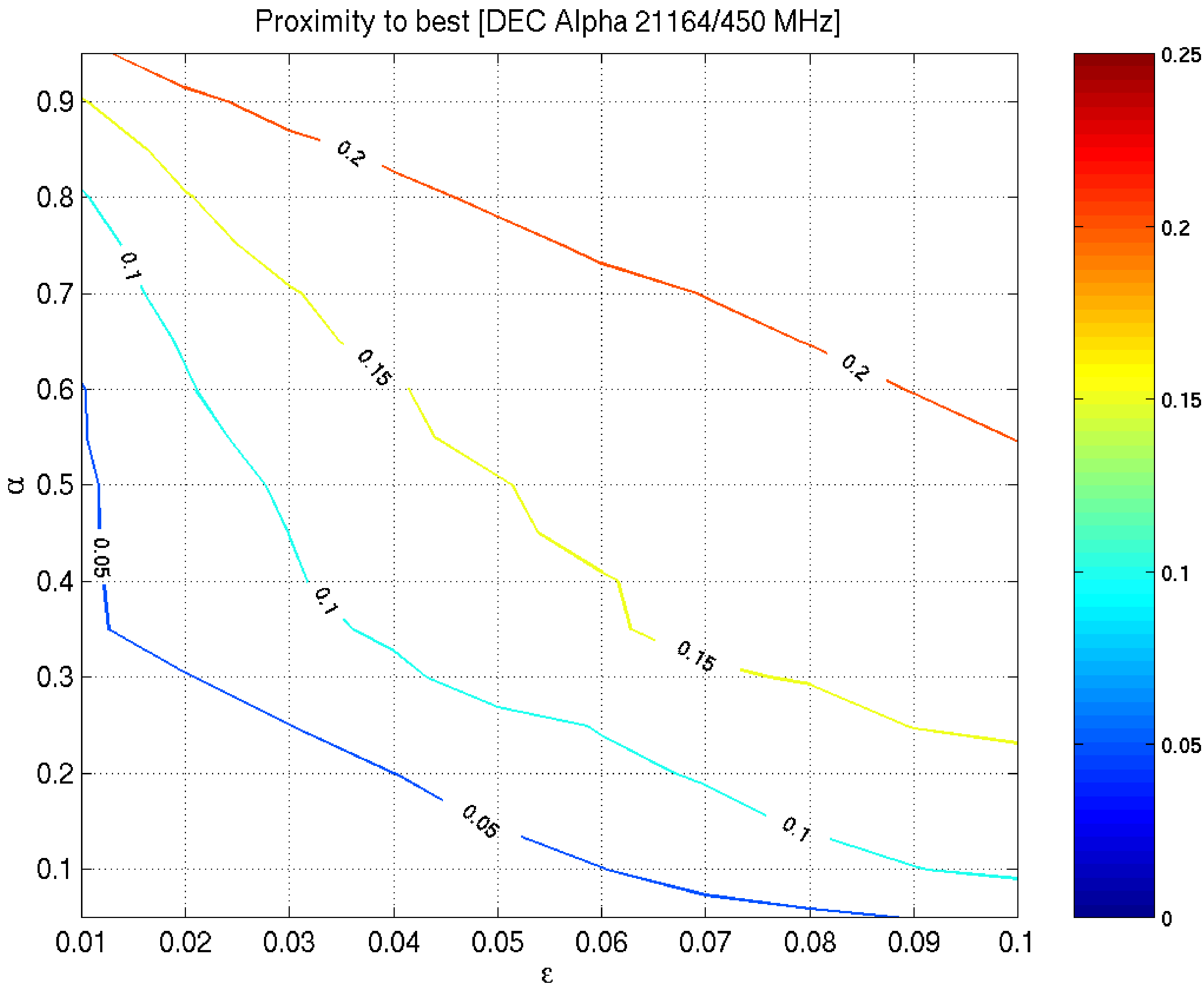


Proximity to best [Intel Pentium−II 300 MHz]
α
ε
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
0.02
0.03
0.04
0.05
0.06
0
0.05
0.1
0.15
0.2
0.25


Fraction of space searched [DEC Alpha 21164/450 MHz]
α, Uncertainty
ε, Proximity to best
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
0
1



Proximity to best [DEC Alpha 21164/450 MHz]
0.3
0.3
0.25
0.25
0.25
0.2
0.2
0.2
0.15
0.15
0.15
0.1
0.1
0.05
α
ε
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.25
0.2
0.15
0.1
0.05
0



Proximity to best [DEC Alpha 21164/450 MHz]
0.2
0.2
0.15
0.15
0.1
0.1
0.1
0.05
0.05
α
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
ε
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
0.1
0.25
0.2
0.15
0.1
0.05
0

Cost Minimization

- Decision function

$$f(s) = \underset{a \in A}{\operatorname{argmax}} \left\{ w_{\theta_a}(s) \right\}$$

- Minimize overall execution time on samples

$$C(\theta_{a_1}, \ldots, \theta_{a_m}) = \sum_{a \in A} \sum_{s \in S_0} w_{\theta_a}(s) \cdot T(a, s)$$

- *Softmax* weight (boundary) functions

$$w_{\theta_a}(s) = \frac{e^{\theta_a{}^T s + \theta_{a,0}}}{Z}$$

Regression

- Decision function

$$f(s) = \operatorname*{argmin}_{a \in A} \{ T_a(s) \}$$

- Model implementation running time (e.g., square matmul of dimension N)

$$T_a(s) = \beta_3 N^3 + \beta_2 N^2 + \beta_1 N + \beta_0$$

- For general matmul with operand sizes (M, K, N), we generalize the above to include all product terms
  - MKN, MK, KN, MN, M, K, N

Support Vector Machines

- Decision function

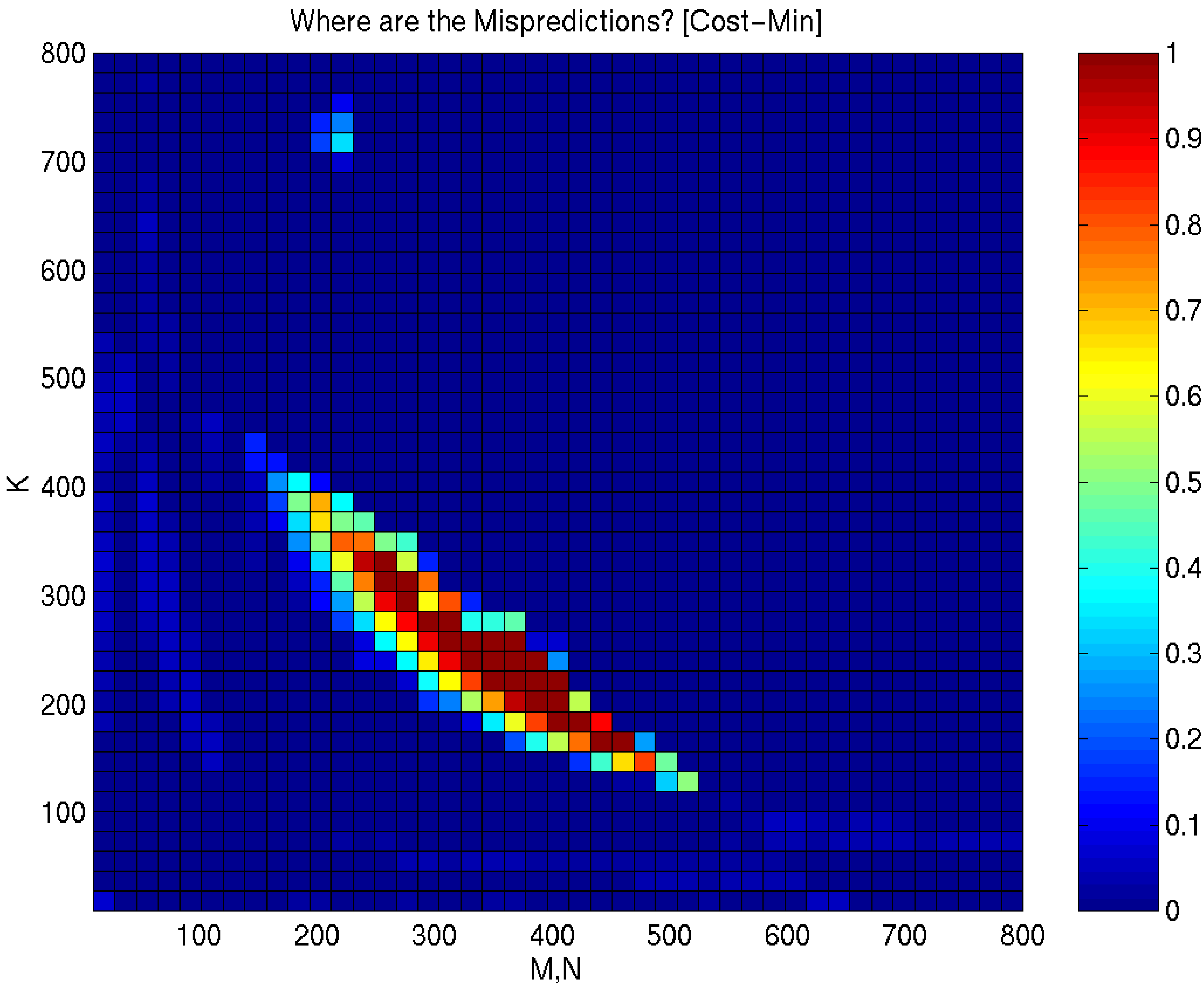
$$f(s) = \operatorname*{argmax}_{a \in A} \{ L_a(s) \}$$

- Binary classifier

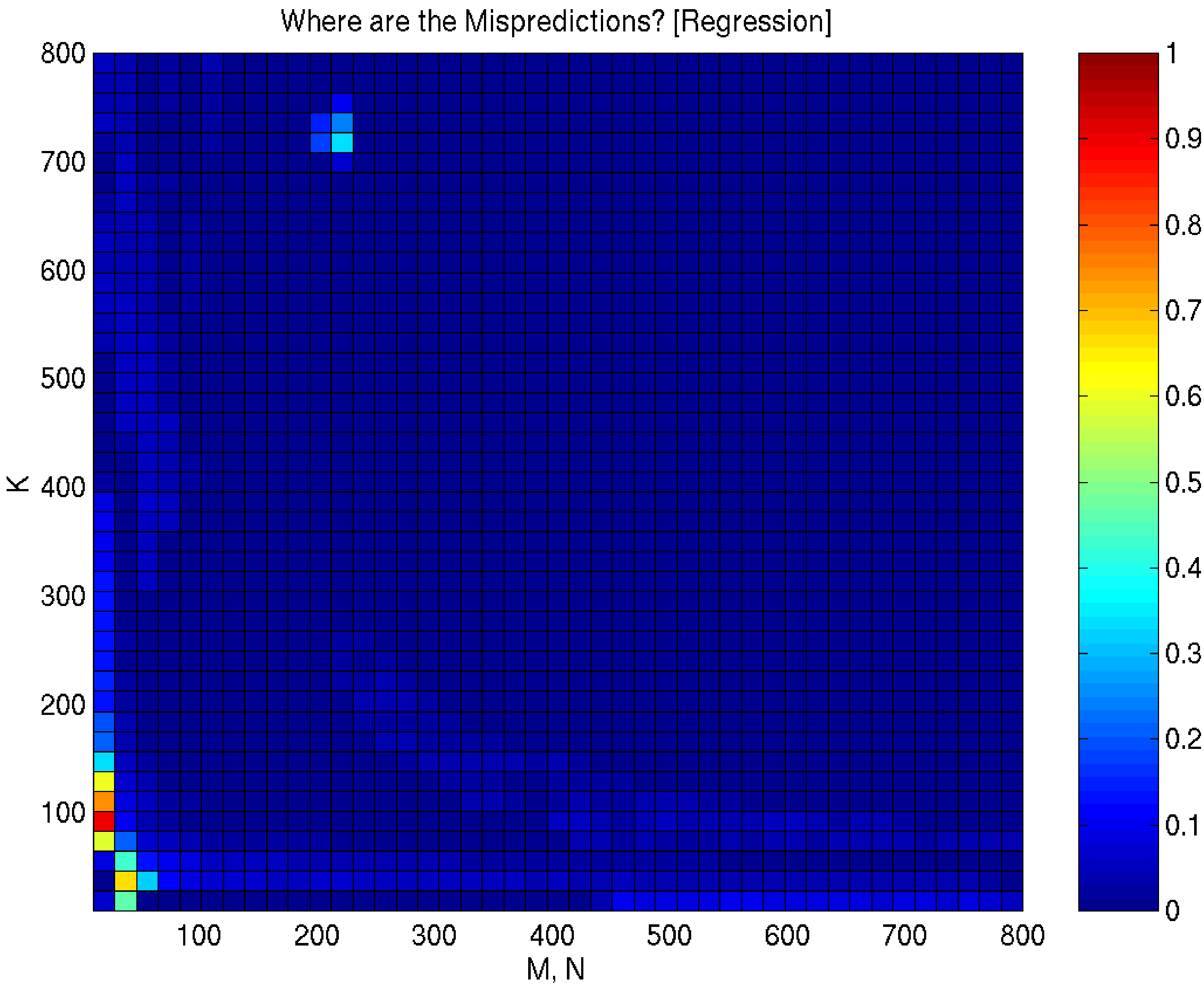
$$L(s) = -b + \sum_i \beta_i y_i K(s_i, s)$$

$$y_i \in \{-1,1\}$$

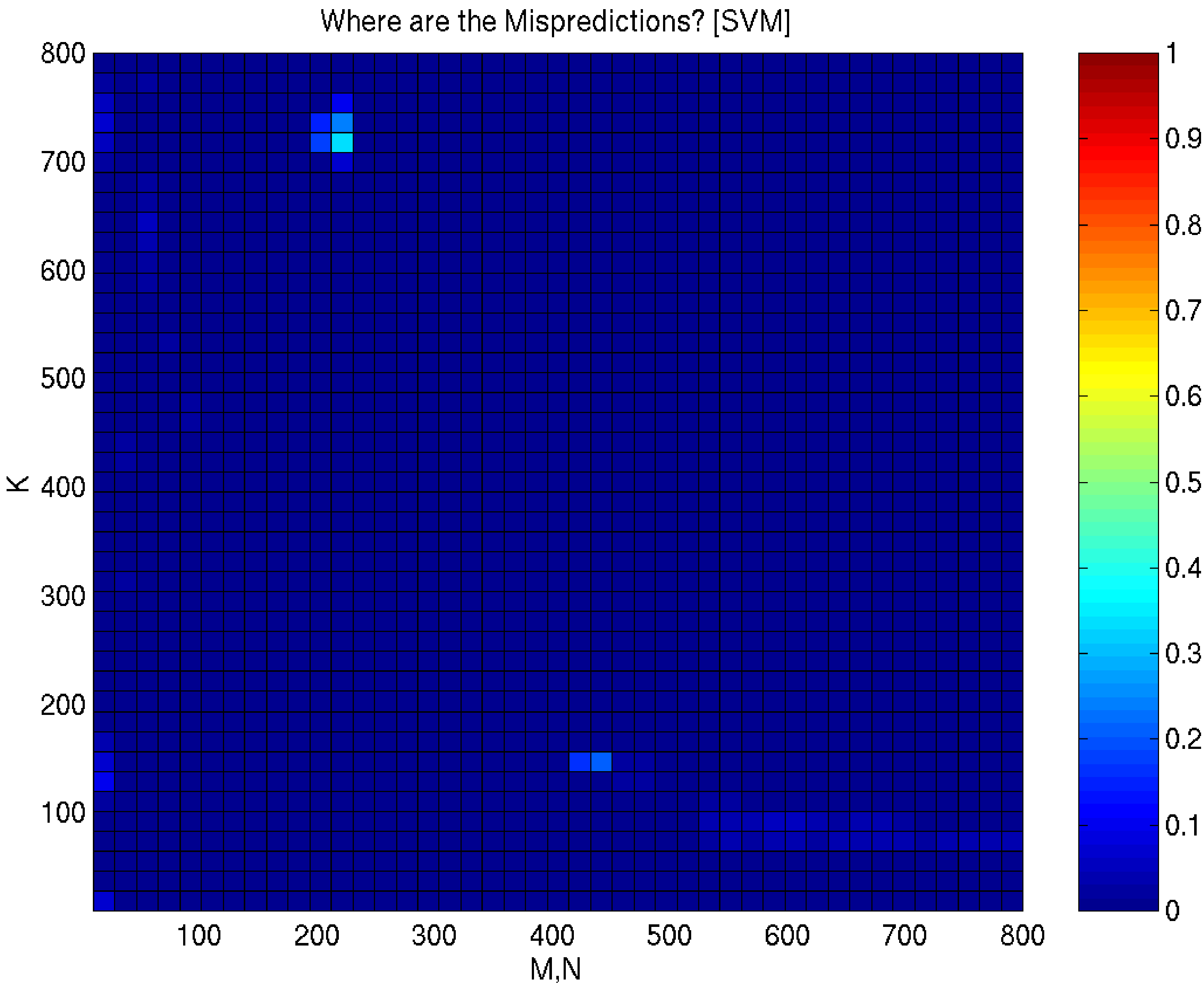
$$s_i \in S_0$$

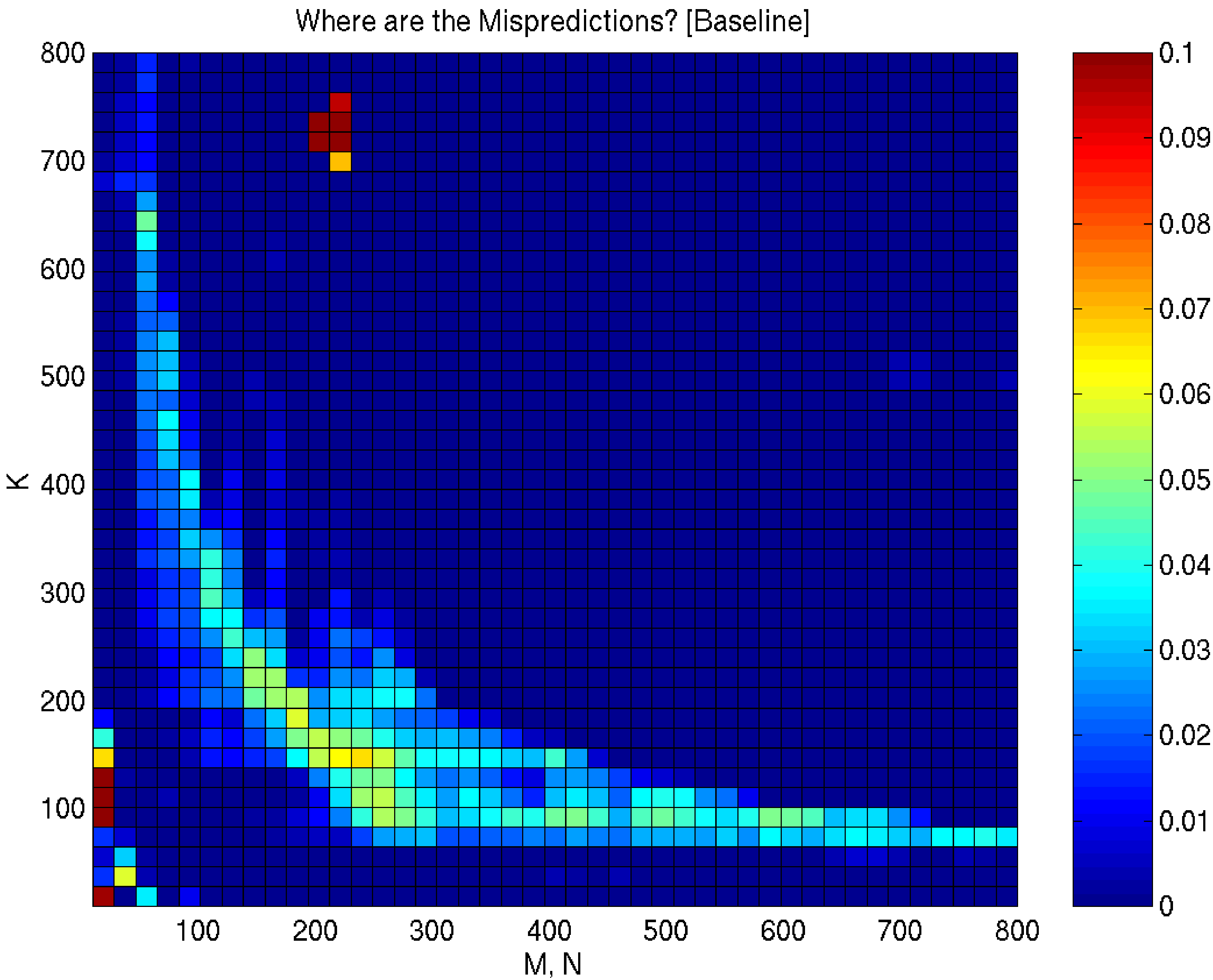


Where are the Mispredictions? [Cost-Min]
K
M,N
800
700
600
500
400
300
200
100
100
200
300
400
500
600
700
800
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0



Where are the Mispredictions? [Regression]
800
700
600
500
400
300
200
100
K
100
200
300
400
500
600
700
800
M, N
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0


Where are the Mispredictions? [SVM]
800
700
600
500
400
300
200
100
K
100
200
300
400
500
600
700
800
M,N
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0



Where are the Mispredictions? [Baseline]
K
800
700
600
500
400
300
200
100
M, N
100
200
300
400
500
600
700
800
0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0

Quantitative Comparison

Method	*Misclass.*	*Average error*	*Best 5%*	*Worst 20%*	*Worst 50%*
Regression	34.5%	2.6%	90.7%	1.2%	0.4%
Cost-Min	31.6%	2.2%	94.5%	2.8%	1.2%
SVM	12.0%	1.5%	99.0%	0.4%	~0.0%

Note:
Cost of regression and cost-min prediction ~O(3x3 matmul)
Cost of SVM prediction ~O(64x64 matmul)