Automating the Modeling and Optimization of the Performance of Signal Processing Algorithms

Bryan Singer

December 13, 2001

Thesis Committee:
Manuela Veloso (Chair)
Scott Fahlman
John Lafferty
Jeremy Johnson (Drexel University)

Overview

- Background and Motivation
- Optimizing Performance by Searching
- Modeling Performance
- Generating Fast Formulas
- Conclusions

Signal Processing

Many signal processing algorithms:

- take as input a signal X as a vector
- produce transformation of signal $Y = A\,X$

Issue:

- Naïve implementation of matrix multiplication is slow

Example signal processing applications:

- Real time audio, image, speech processing
- Analysis of large data sets

Factoring Signal Transforms

- Transformation matrices are highly structured
- Can factor transformation matrices
- Factorizations allow for faster implementations

Discrete Fourier Transform (DFT)

Highly structured, for example:

$$DFT(2^2) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

Cooley-Tukey factorization or break down rule:

$$DFT(rs) = (DFT(r) \otimes I_s)\, T_s^{rs}\, (I_r \otimes DFT(s))\, L_r^{rs}$$

Can recursively apply break down rule

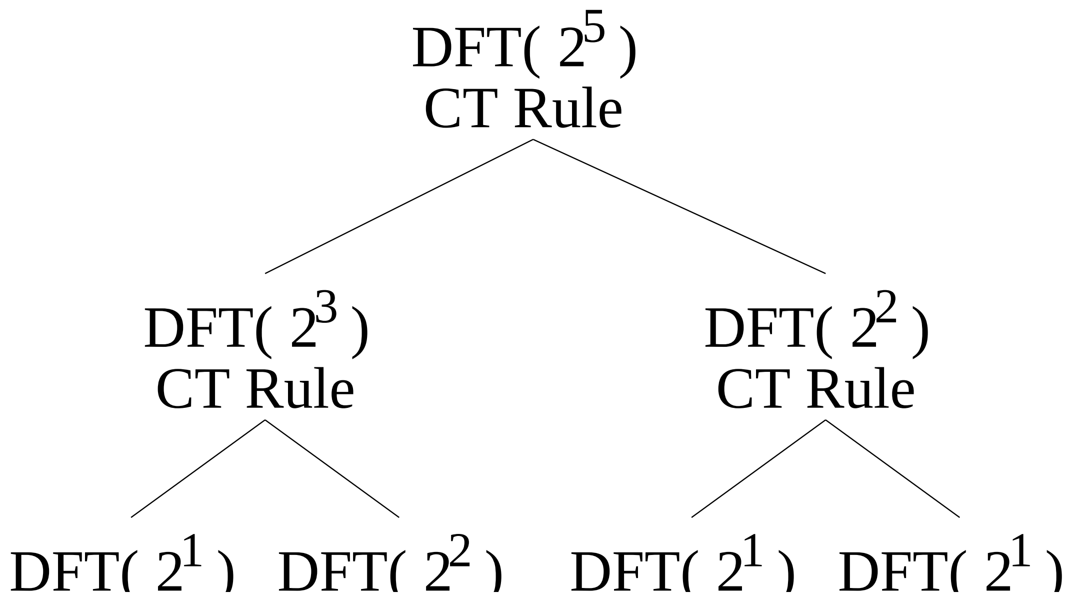
Yielding $\theta(n \log n)$ algorithm (FFT)

DFT Example

$$DFT(2^5)$$
$$= (DFT(2^3) \otimes I_4)\, T_4^{32}\, (I_8 \otimes DFT(2^2))\, L_8^{32}$$
$$= ([(DFT(2^1) \otimes I_4)\, T_4^8\, (I_2 \otimes DFT(2^2))\, L_2^8] \otimes I_4)\, T_4^{32}$$
$$(I_8 \otimes [(DFT(2^1) \otimes I_2)\, T_2^4\, (I_2 \otimes DFT(2^1))\, L_2^4])\, L_8^{32}$$

We can visualize this

as a split tree:



Walsh-Hadamard Transform (WHT)

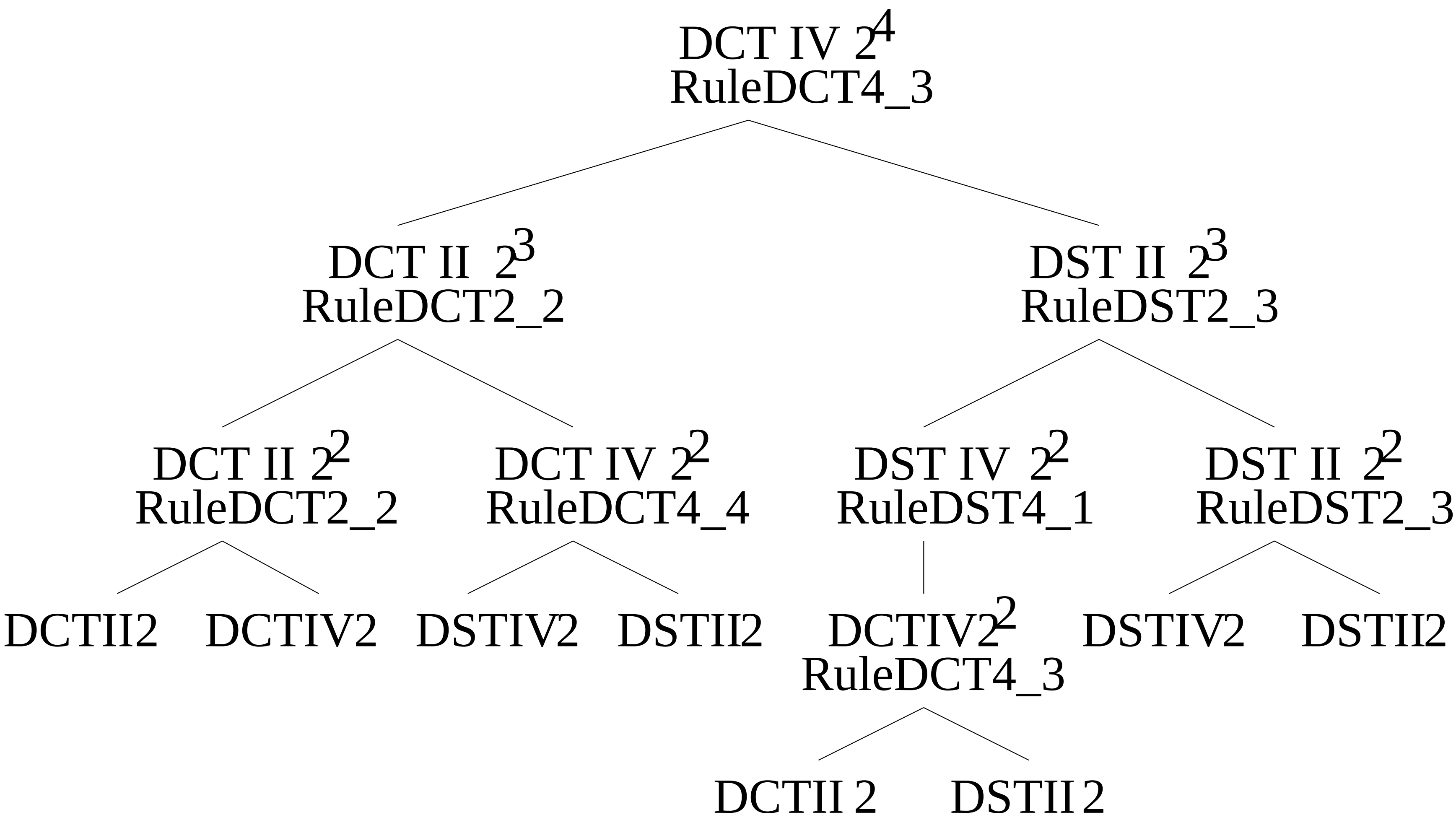
$$WHT(2^2) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

Break down rule:

$$WHT(2^n) = \prod_{i=1}^{t} (I_{2^{n_1+\cdots+n_{i-1}}} \otimes WHT(2^{n_i}) \otimes I_{2^{n_{i+1}+\cdots+n_t}})$$

for positive integers n_i such that $n = n_1 + \cdots + n_t$

Discrete Cosine Transform (DCT) Example



Search Space

Large number of factorizations:

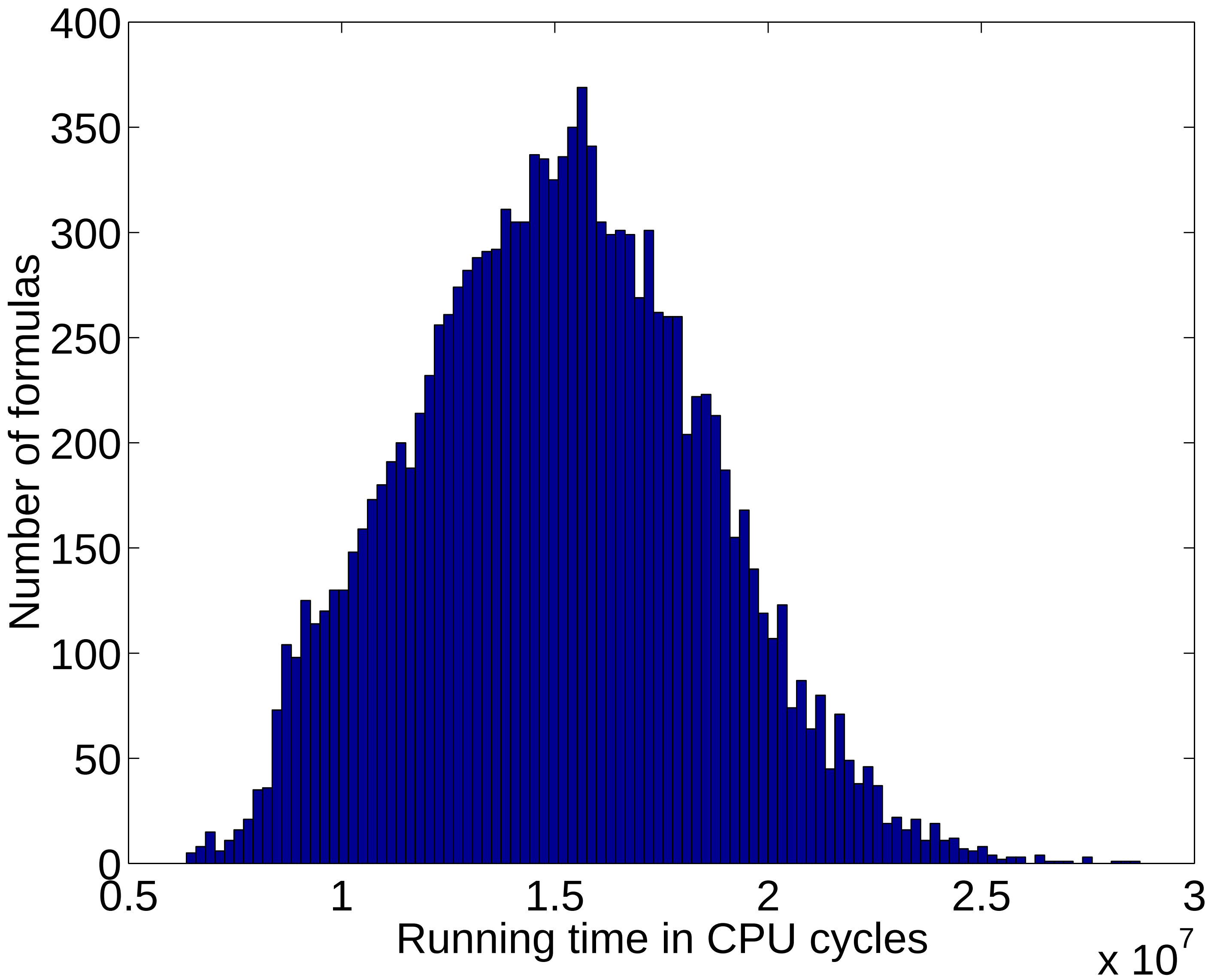
Size	DFT	WHT	DCT IV
2^1	1	1	1
2^2	6	2	10
2^3	40	6	126
2^4	360	24	31,242
2^5	258,400	112	1.9×10^9
2^6	1.8×10^{13}	568	7.3×10^{18}
2^7	7.2×10^{13}	3,032	1.1×10^{38}
2^8	7.2×10^{14}	16,768	2.3×10^{76}
2^9	1.5×10^{16}	95,199	1.1×10^{153}
2^{10}	2.3×10^{17}	551,613	2.2×10^{306}

Varying Performance

Varying performance of factorizations:

- Formulas have *very different* running times
- Same number of arithmetic operations, but different:
  - Cache performance
  - Execution unit performance
  - Register file performance
- Small changes in the split tree can lead to significantly different running times
- Optimal formulas across machines are different

Histogram of $WHT(2^{16})$ Running Times



Thesis Problem

Find the best implementation for a given:

- Transform
- Size
- Computing platform

Huge search space of implementations

Constrained by a given:

- Set of break down rules
- Code implementation strategy for formulas (possibly tunable)
- Method of obtaining runtime performance

Contributions

Search methods for optimizing performance

- Intelligently search space
- Avoid timing all formulas

Automated methods for modeling performance

- Learn models to predict performance of formulas

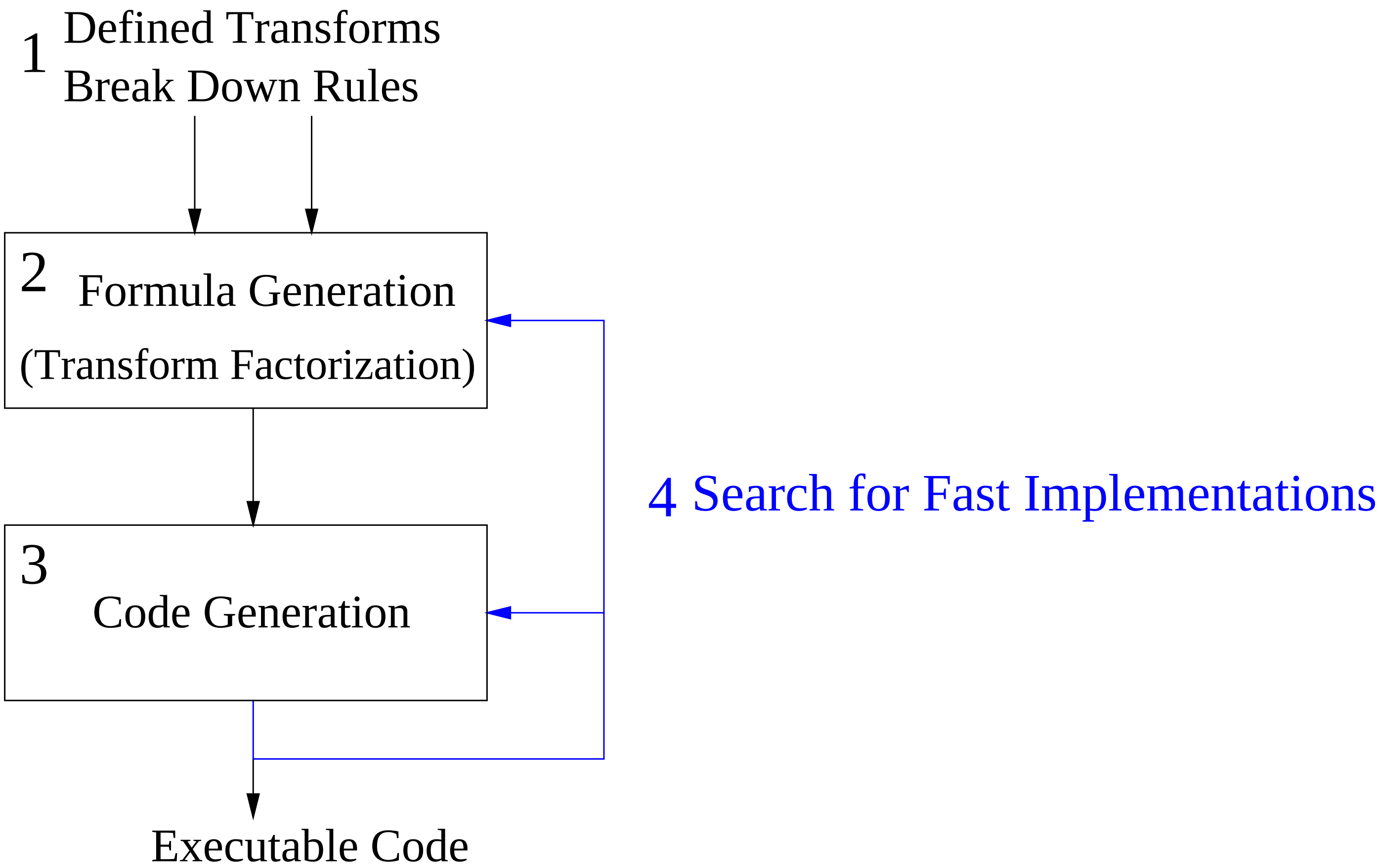
Method for generating fast implementations

- Use learned models to optimize performance
- Control the construction of formulas
- Given model, no need to time any formulas

Overview

- Background and Motivation
- Optimizing Performance by Searching
- Modeling Performance
- Generating Fast Formulas
- Conclusions

Infrastructure



SPIRAL: Signal Processing algorithms Implementation Research for Adaptable Libraries

Download system at: `http://www.ece.cmu.edu/~spiral`

Search Methods Implemented in SPIRAL

- Exhaustive Search
- Dynamic Programming (DP)
- Random Search
- Hill Climbing
- STEER (evolutionary algorithm)
- Timed Search (a meta-search algorithm)

- Search over new user-defined transforms and break down rules
- Search over formulas and options to code generator

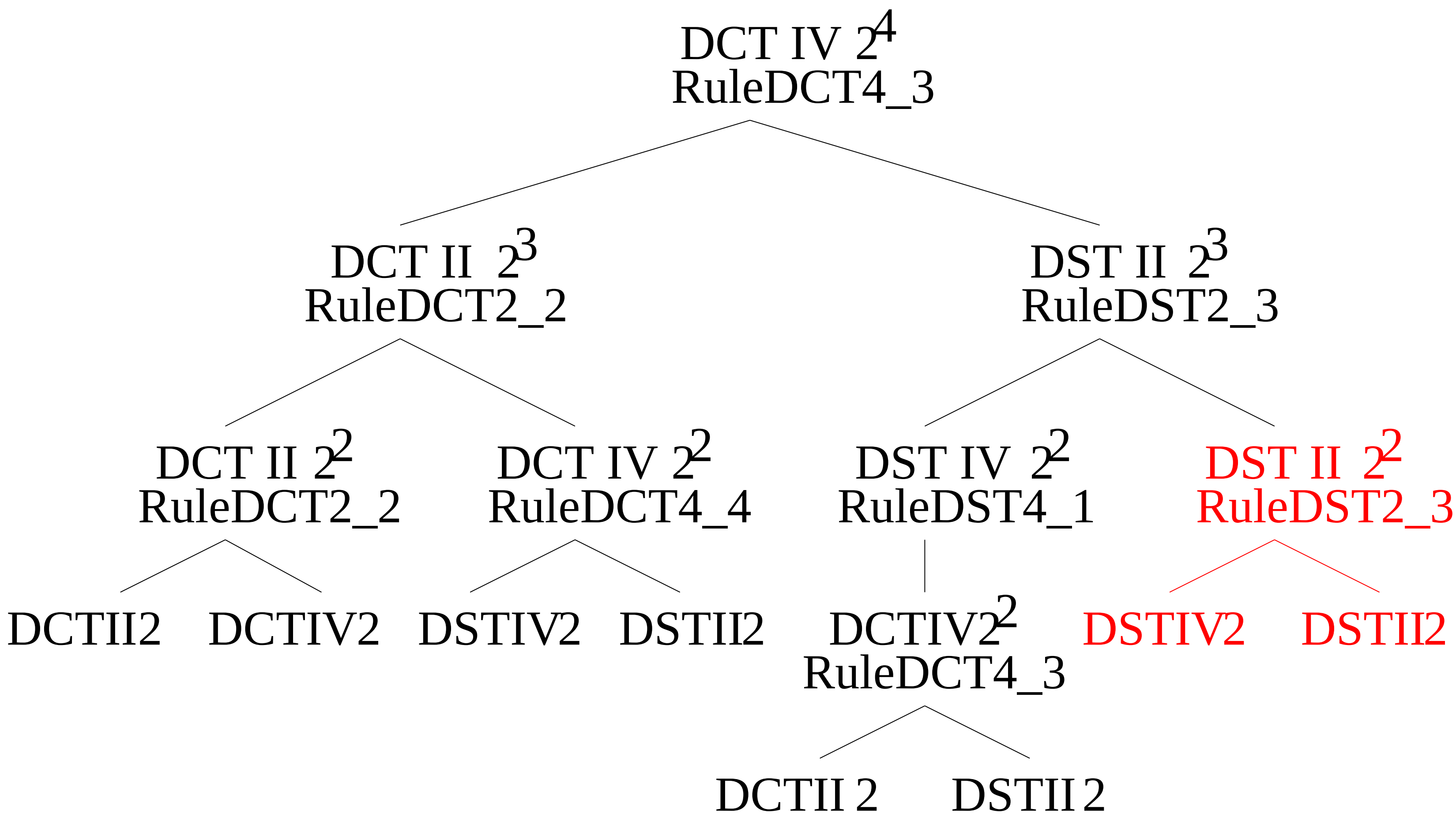
STEER: Split Tree Evolution for Efficient Runtimes

Generate a population of random legal split trees

Repeatedly "evolve" the population:

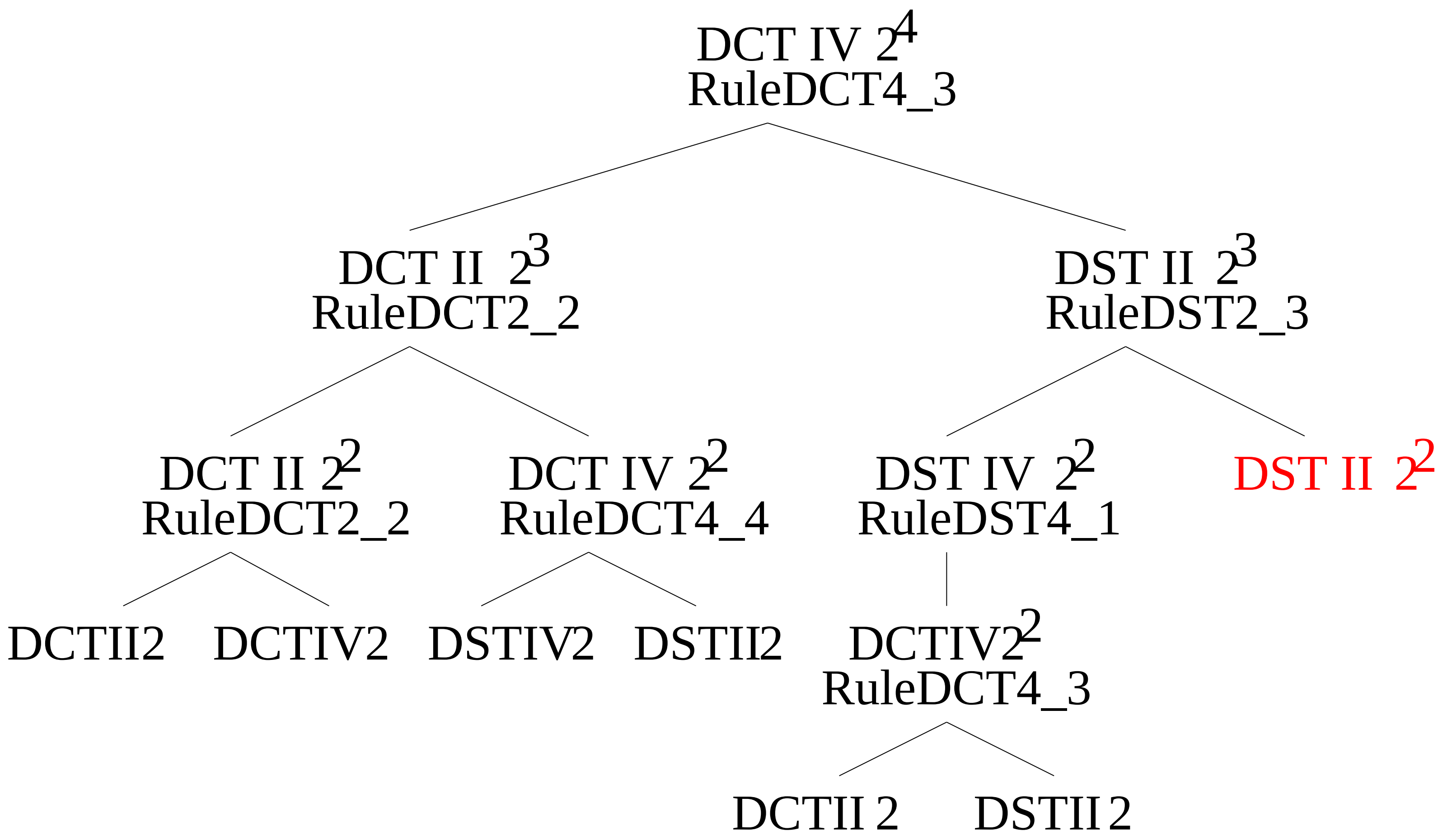
- Time trees in current set
- Generate new population with fitness proportional reproduction while:
  - Maintaining the current best trees
  - Randomly applying mutation to individual trees
  - Randomly applying crossover to pairs of trees

Mutation: Regrow



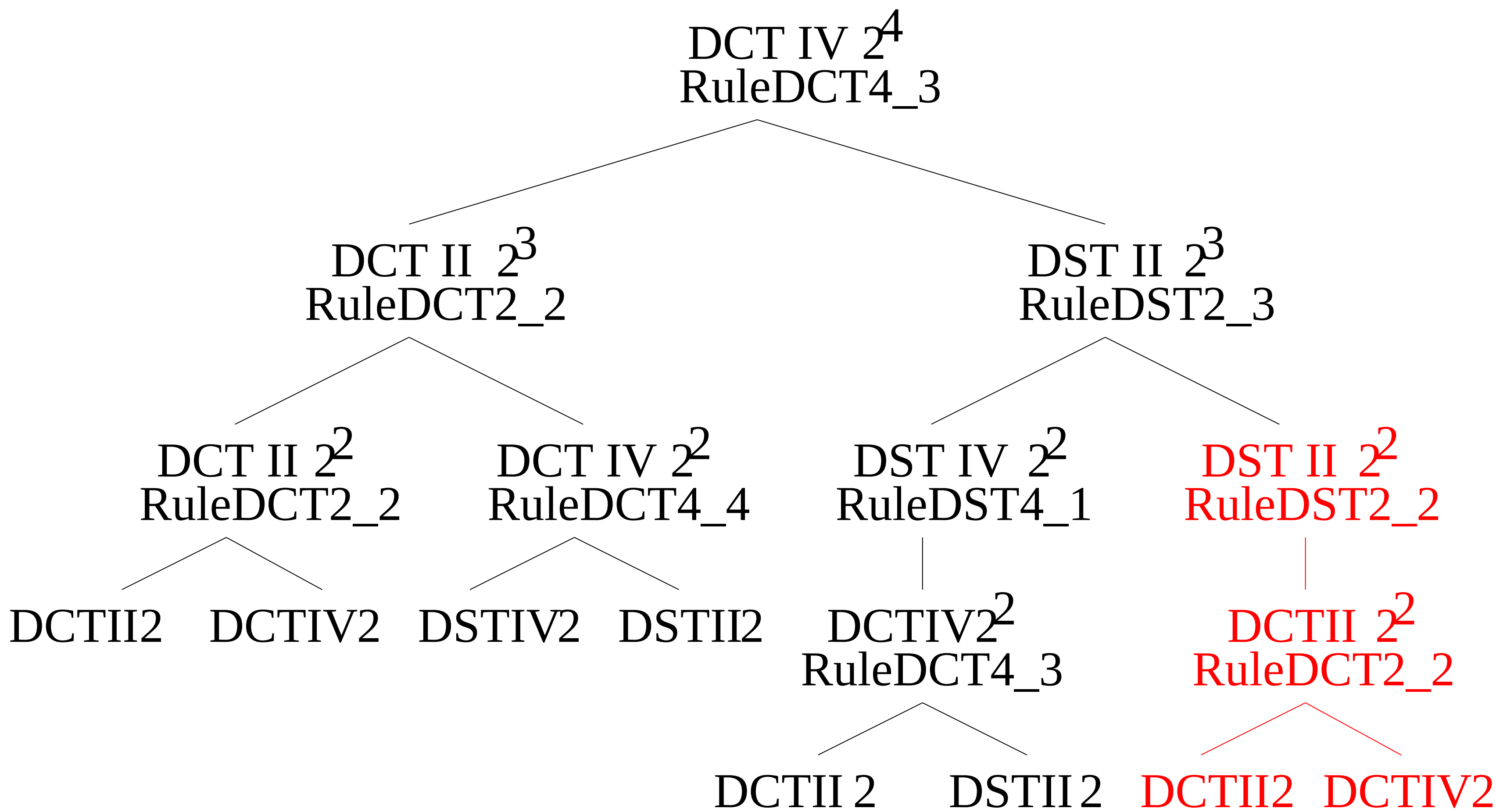
Original

Mutation: Regrow



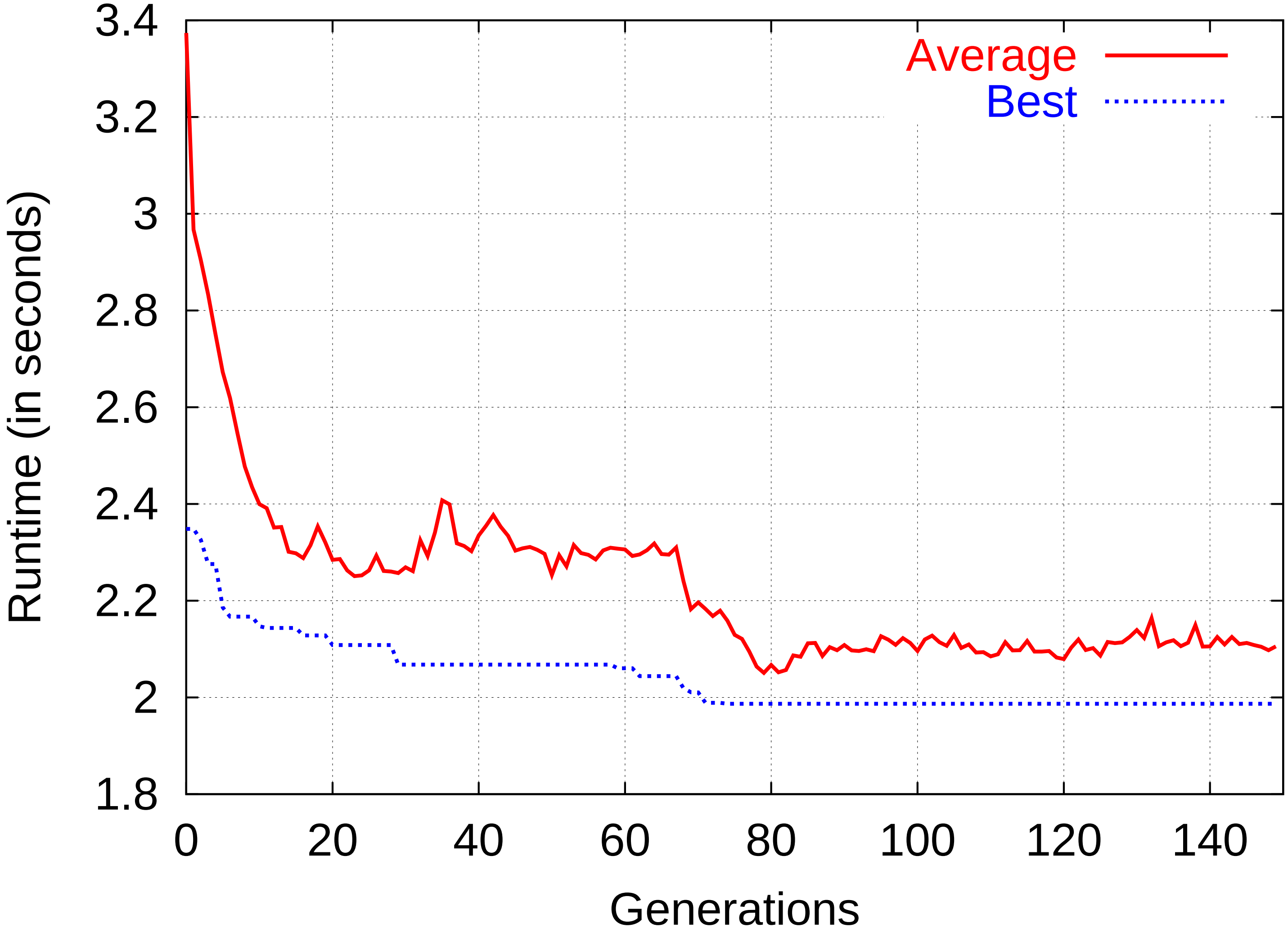
Original $\Rightarrow$ Truncate

Mutation: Regrow

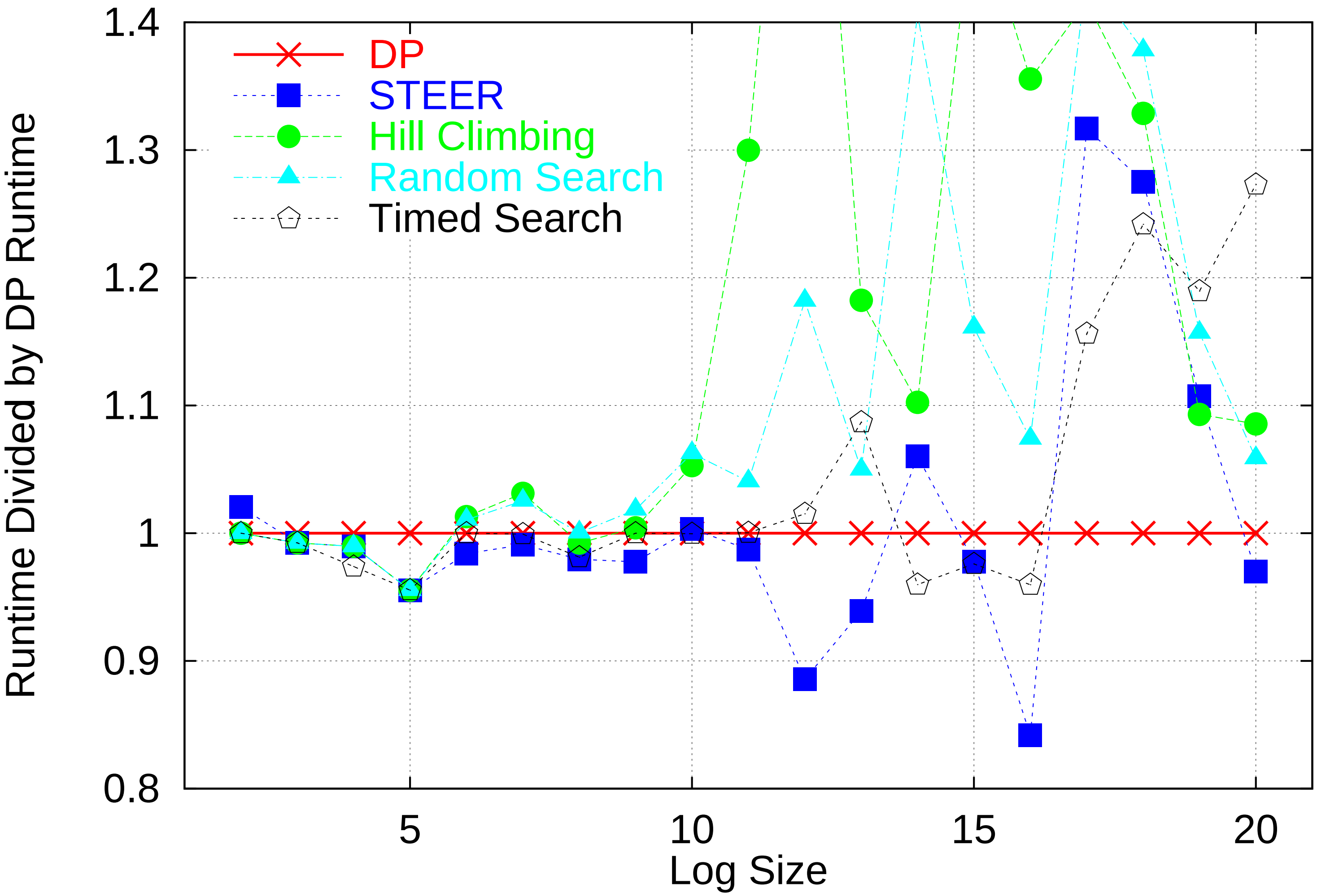


Original ⇒ Truncate ⇒ Regrow

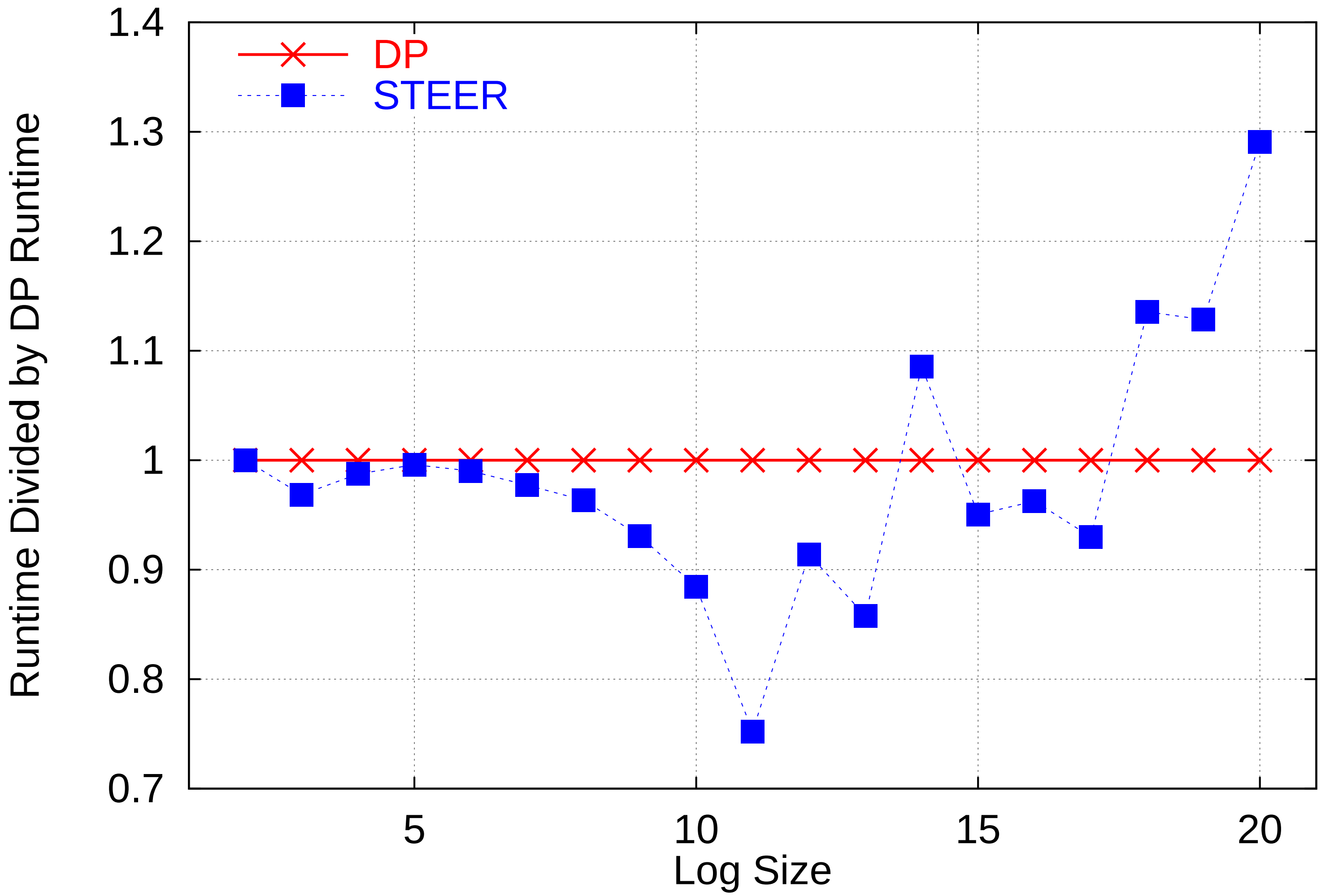
Running STEER



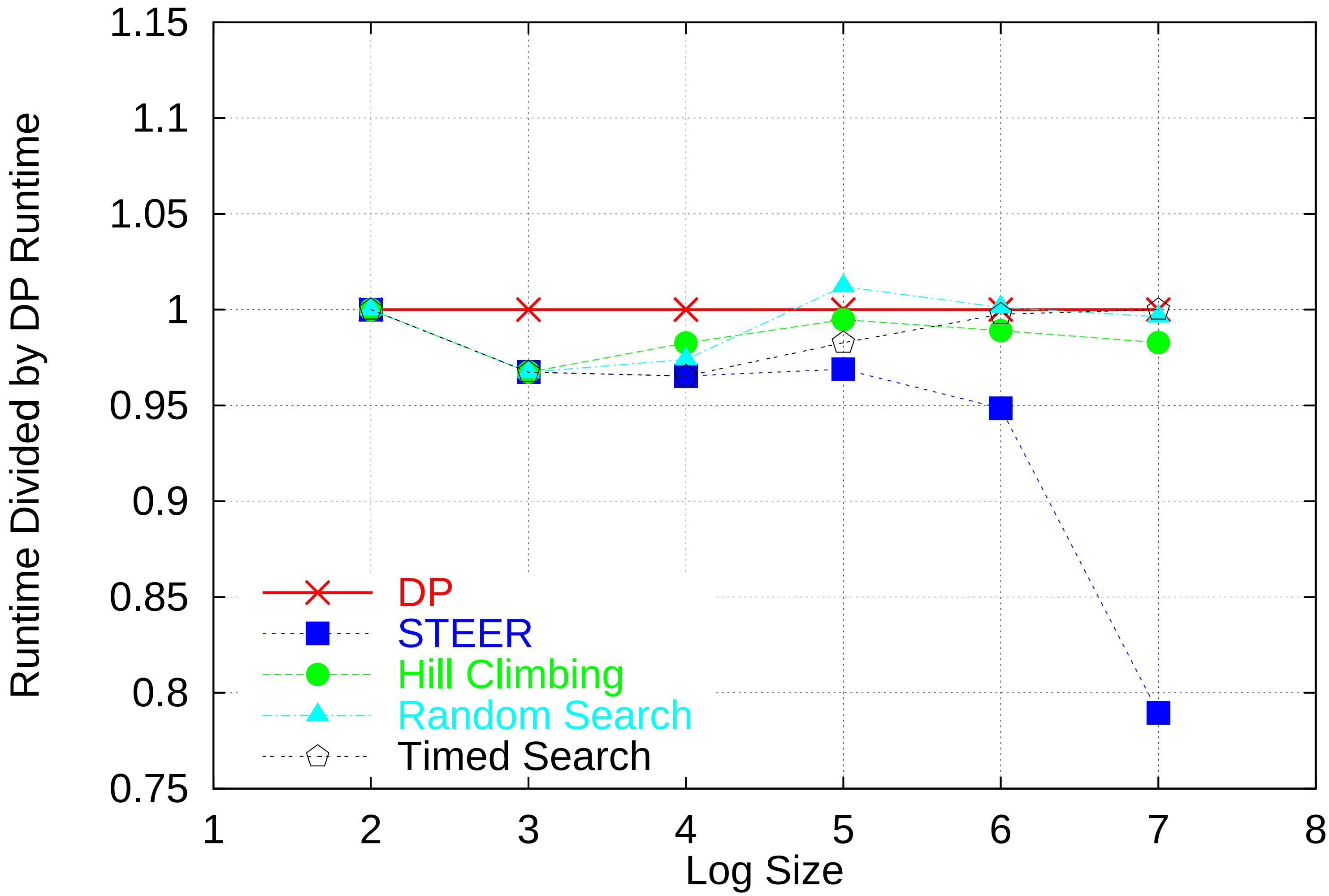
FFT on a Pentium III



FFT on a Sun UltraSparc IIi



DCT Type II on a Pentium III



Summary: Optimization by Intelligent Search

- Many search methods implemented
- No one search method dominates for all transforms and sizes
- Requires timing many formulas, but not all

Overview

- Background and Motivation
- Optimizing Performance by Searching
- Modeling Performance
- Generating Fast Formulas
- Conclusions

Learning to Predict Performance

Can we learn to predict performance of formulas?

- Can gather empirical data by running formulas
- Use automated machine learning techniques

Machine learning task:

- Predict performance for entire formulas
- Predict performance for individual nodes in split tree
  - Sum predictions for nodes to predict for formula
  - For WHT, computation occurs in leaves only
  - For FFT, computation occurs in all nodes
  - Limit FFT to Cooley-Tukey factorization

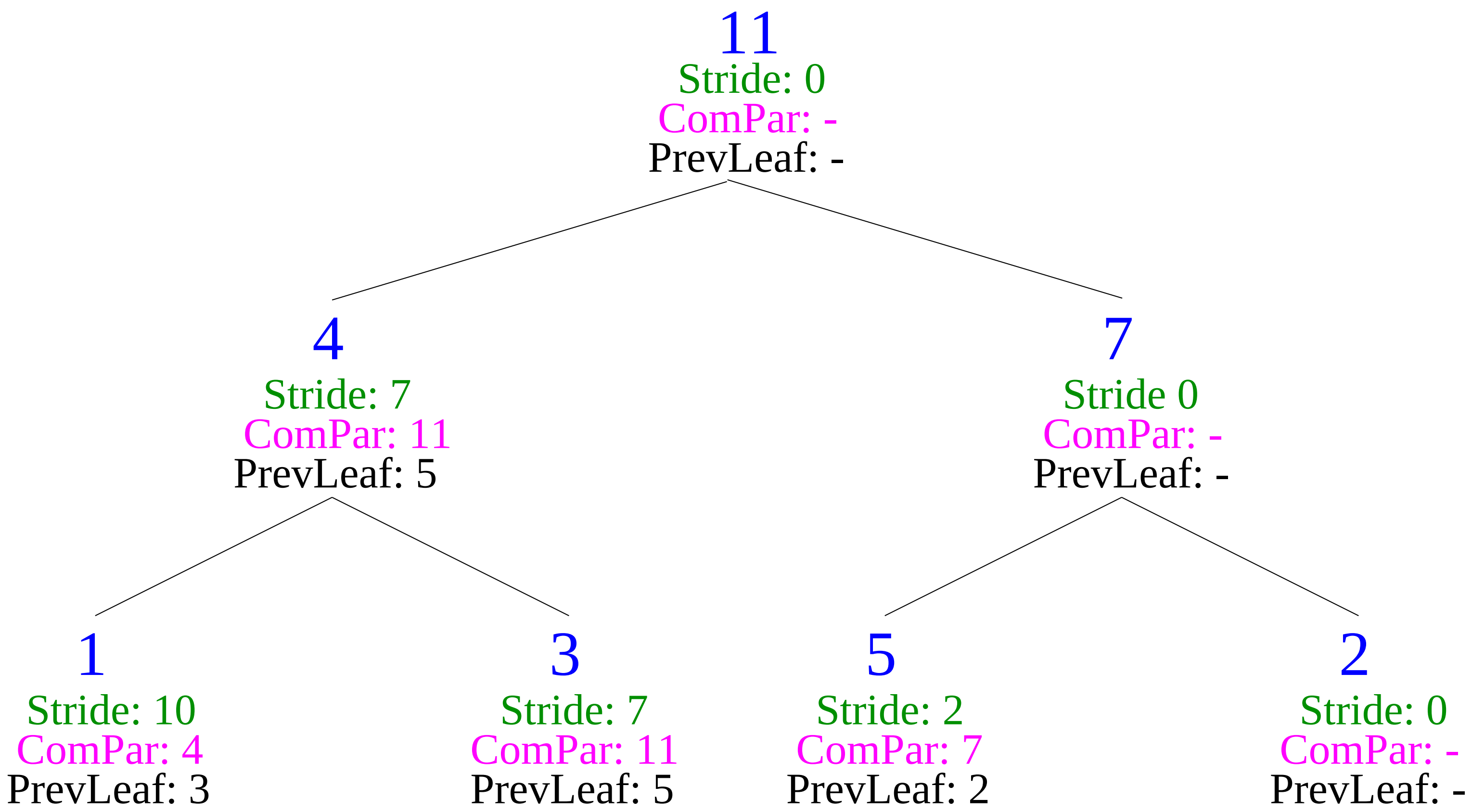
Learning Algorithm

1. Collect runtimes for nodes in split trees
2. Divide runtimes by size of overall transform
3. Train a function approximator to predict runtimes for split tree nodes

Need to describe split tree nodes with features

Features for Split Tree Nodes

- Size and stride of the given node
- Size and stride of the parent of the given node
- Size and stride of the common parent
- Size and stride of each of the children and grandchildren



Learning Algorithm

1. Collect runtimes for nodes in split trees
2. Divide runtimes by size of overall transform
3. Describe nodes with features
4. Train a function approximator to predict a node's runtime given the node's features

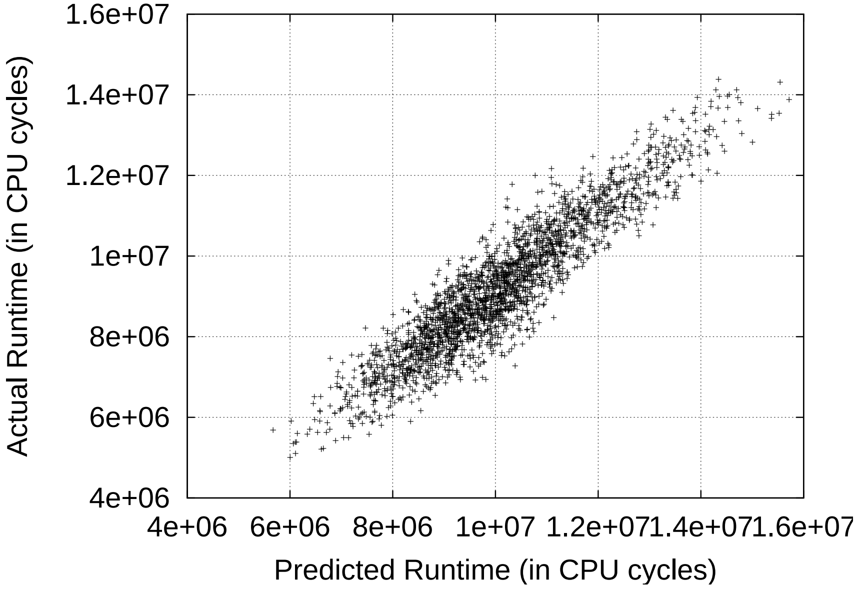
Training

- Trained regression trees using RT4.0
- Data from subsets of FFT and WHT formulas of size 2^{16}
- Trained different regression trees for:
  - WHT leaves
  - FFT leaves
  - FFT internal nodes
- Predicted for entire formulas by summing predictions for all nodes

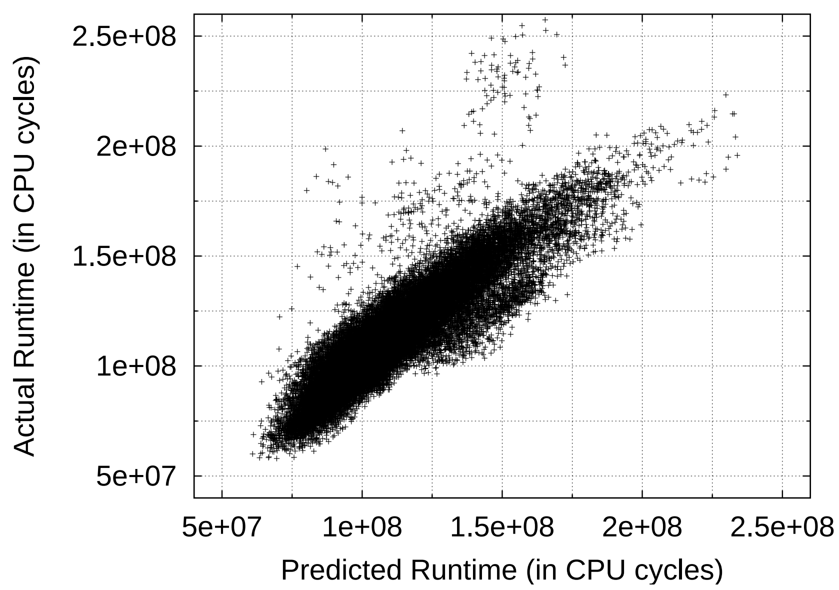
Predicted Runtime Versus Actual Runtime

FFT on a Pentium III

$FFT(2^{14})$



$FFT(2^{17})$

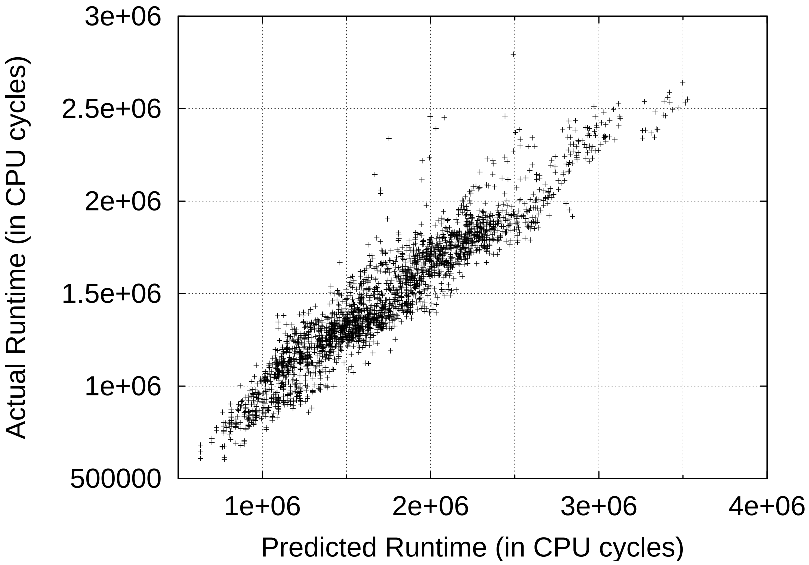


- Trained only on nodes from $FFT(2^{16})$ split trees
- Predicts well across different sizes, even larger sizes!

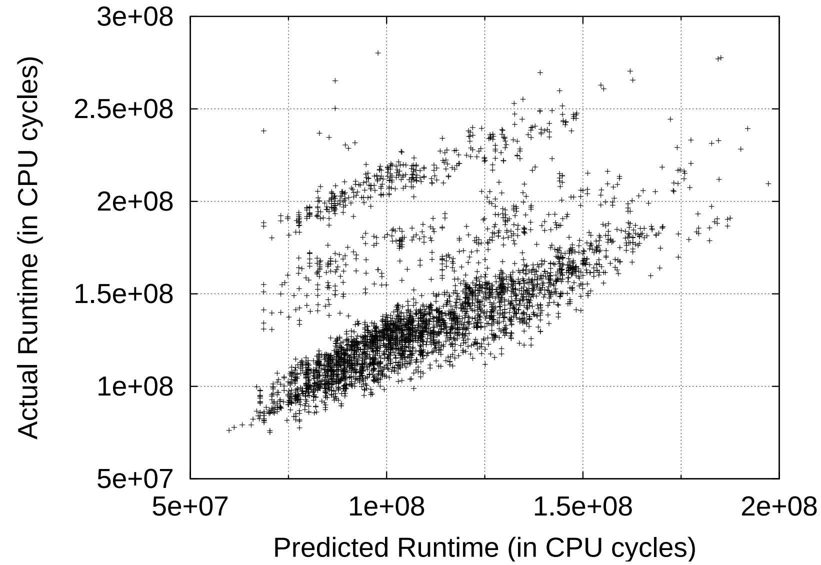
Predicted Runtime Versus Actual Runtime

WHT on a Sun UltraSparc IIi

Binary No-2^1-Leaf
$WHT(2^{14})$



Binary No-2^1-Leaf
Rightmost $WHT(2^{20})$



- Trained only on leaves from $WHT(2^{16})$ split trees
- Predicts well across different sizes, even larger sizes!

Summary: Predicting Runtimes

Train a function approximator:

- Predict runtimes for nodes
- Train using runtime data collected for nodes
- Describe nodes with numeric features

By learning to predict runtimes for nodes:

- Accurately predict runtimes for entire formulas
- Accurately predict across many transform sizes while trained on one size

Overview

- Background and Motivation
- Optimizing Performance by Searching
- Modeling Performance
- Generating Fast Formulas
- Conclusions

Generating Fast Formulas

- Can now predict runtimes for formulas
- But still MANY formulas to search through

Can we learn to generate fast formulas?

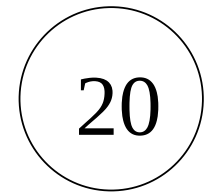
Control Learning Problem:

- Learn to control the generation of formulas to produce fast ones

Generating Fast Formulas: Approach

Want to grow the fastest split tree:

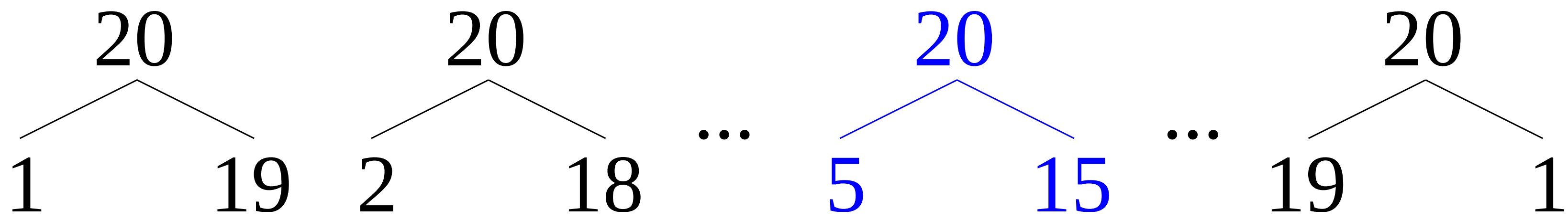
- Begin with a root node of the desired size:



Generating Fast Formulas: Approach

Want to grow the fastest split tree:

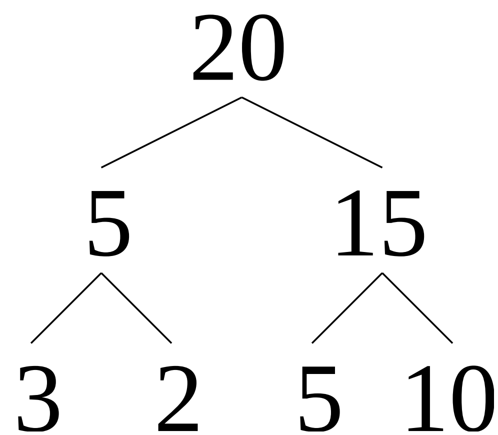
- Begin with a root node of the desired size
- Choose best set of children out of all possible:



Generating Fast Formulas: Approach

Want to grow the fastest split tree:

- Begin with a root node of the desired size
- Choose best set of children
- Recurse on each of the children:



Choosing the Best Children

How do we choose the best children?

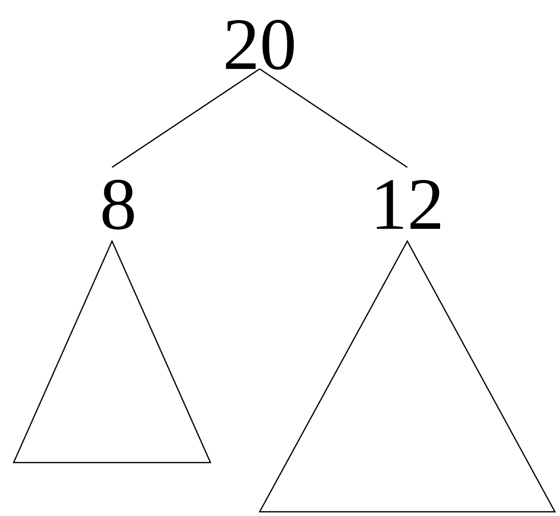
- Define a value function over nodes
- Node's value = runtime of best subtree
- Choose children with minimal sum of values

How do we calculate this value function?

Problem Structure

Overlapping Subproblems

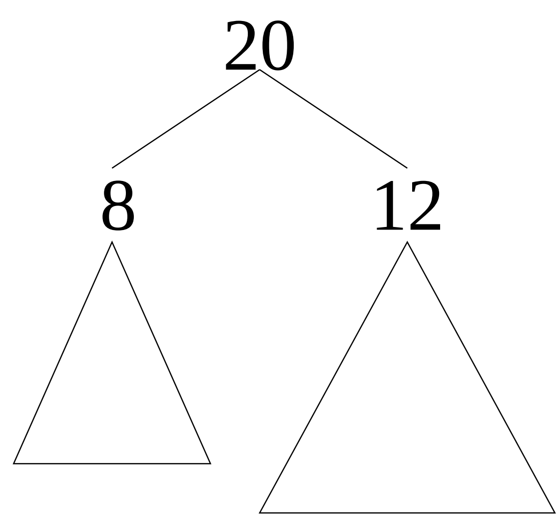
- Many duplicated subtrees in different formulas
- Consider all possible $WHT(2^{20})$ split trees
- Given subtree of node 8:
  - Appears many times in trees for size 2^{20}
  - Appears once for every different subtree of 12



Problem Structure

Optimal Substructure

- Best subtree for node 8:
  - Independent of node 12's subtree
  - But dependent on node 8's location
- Features already capture this



Dynamic Programming

Duplicated Subproblems + Optimal Substructure =

Properties needed for DP

Describe nodes with features

- State = One set of feature values, describing a node
- Features describe context not just size of node
- 2 nodes in different trees can be same state

Run DP

- Calculate values for states
- Memoize results to save duplicating work

Value Function

State = node in split tree described by features

State's value = runtime of best subtree

- Accurate runtimes are expensive to obtain
- Plus may not have a fully grown tree to run
- Use the regression trees to predict runtimes!

Mathematically: Value Function on States

State = node in split tree described by features

The value of a state is:

$$V(state) = \min_{subtrees} \sum_{node \in subtree} PredictedRuntime(node)$$

- Min over all possible subtrees of the given state

Recursive Formulation of Value Function

State = node in split tree described by features

The value of a state is:

$$
\begin{aligned}
V(state) &= \min_{splittings} \sum_{children} V(child) \\
&+ PredictedRuntime(state)
\end{aligned}
$$

DP can calculate this value function!

Computing the Value Function

Use dynamic programming to calculate value function:

- Consider all possible sets of children of the root
- Recursively call DP on each of the children states
  - Determine values of children states
  - Memoizing results
- Determine set of children with minimal sum of values
- Root's value is this minimal sum of values plus the root's predicted runtime

Generating Fast Formulas

Use value function to control generation of formulas

Generate split tree with minimal value

- Consider all possible sets of children of the root
- Look up values of children states
- Choose those that have the minimal sum of values
- Recurse on children

Generating with a Tolerance

Generates single tree with fastest predicted runtime

Two approximations made:

- Regression trees used to predict runtimes
- Assumed optimal substructure

Given a tolerance:

- Generate all trees with values within tolerance of best value
- Rank formulas according to values (predicted runtimes)

Generation Rank	1	2	3	4	...
Predicted Runtime	4.4	4.5	4.7	4.8	...
Actual Runtime	4.4	4.7	4.3	5.2	...

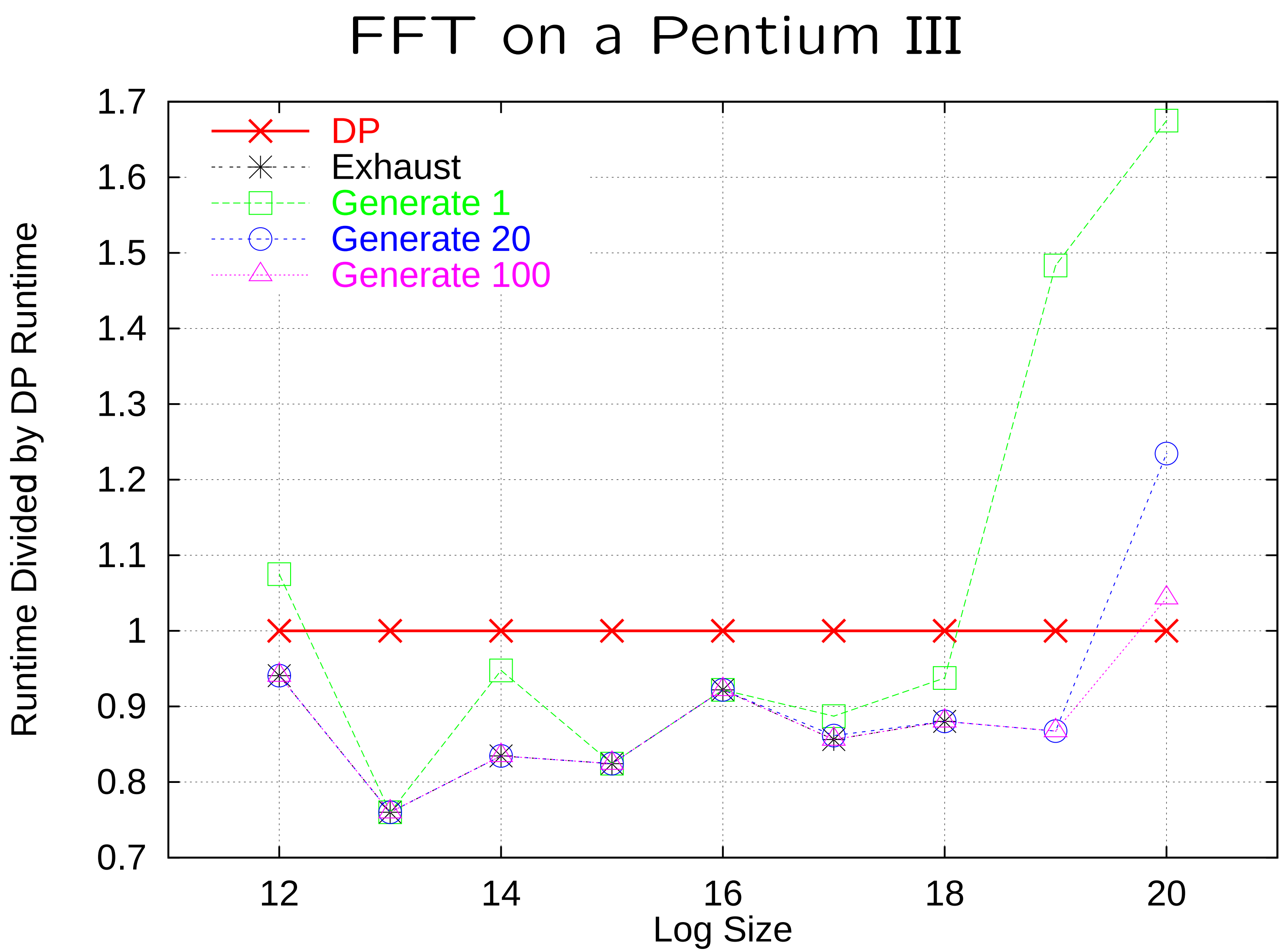
Fast Formula Generation Results

FFT on a Pentium III

Size	Generation rank of fastest formula	Rank 1 formula is X% slower than fastest formula
2^{12}	16	14.3%
2^{13}	1	0.0%
2^{14}	2	13.6%
2^{15}	1	0.0%
2^{16}	1	0.0%
2^{17}	82	3.6%
2^{18}	11	6.5%

70,376 different $FFT(2^{18})$ formulas

Fast Formula Generation Results



Fast Formula Generation Results

WHT on a Pentium III

Size	Generation rank of best known formula	Rank 1 formula is X% slower than best known formula
2^{13}	5	3.4%
2^{14}	4	3.0%
2^{15}	3	2.1%
2^{16}	4	1.7%
2^{17}	5	0.1%
2^{18}	4	2.0%
2^{19}	1	0.0%
2^{20}	4	1.7%

398,041 different $WHT(2^{20})$ formulas

Fast Formula Generation Results

WHT on a Sun UltraSparc IIi

Size	Generation rank of best known formula	Rank 1 formula is X% slower than best known formula
2^{13}	14	77.7%
2^{14}	20	12.8%
2^{15}	1	0.0%
2^{16}	2	4.3%
2^{17}	7	18.0%
2^{18}	38	5.9%
2^{19}	17	3.3%
2^{20}	47	1.4%

398,041 different $WHT(2^{20})$ formulas

Fast Formula Generation Results

- Method never sees a timing for sizes other than 2^{16}
- First formula generated is very fast
- Generates fastest known formula within first several formulas

Summary: Fast Formula Generation

Run dynamic programming:

- Determine value of different states
- Use regression trees to predict runtimes for nodes

Generate fast formulas:

- By choosing children with minimal sum of values

Excellent results:

- Generates the fastest known formulas
- Trained only on data of one transform size, and generates fast formulas of many different sizes

Overview

- Background and Motivation
- Optimizing Performance by Searching
- Modeling Performance
- Generating Fast Formulas
- Conclusions

Contributions

Search Engine

- Works with many transforms and break down rules
- Searches over formulas and compiler options
- Includes newly developed STEER

Automatic Performance Modeling

- Uses collected runtimes to train ML techniques
- Uses developed and analyzed feature sets
- Learns models that predict across sizes

Fast Formula Generation

- Generates fastest formulas
- Never sees a timing for most transform sizes

Future Work

- Extend modeling and generation to other transforms
  - For example, DTTs
  - Multiple break down rules possible
  - Children are different transforms
- Learn across different computer platforms
  - Features of the architecture
- Apply work to multiprocessors or hardware
  - New compiler options
  - Different performance metrics (e.g., power usage)
- Optimizing other signal processing algorithms beyond transforms or other numerical algorithms

Acknowledgements

Thesis Committee:

- Manuela Veloso
- Scott Fahlman
- John Lafferty
- Jeremy Johnson

SPIRAL group:

- José Moura, ECE, CMU
- Jeremy Johnson, MCS, Drexel
- Robert Johnson, MathStar
- David Padua, CS, University of Illinois
- Viktor Prasanna, CS, USC
- Markus Püschel, ECE, CMU
- Manuela Veloso, CS, CMU
- Gavin Haentjens, ECE, CMU
- David Sepiashvili, ECE, CMU
- Jianxin Xiong, CS, University of Illinois

Questions?

Extras

Cross Platform Results

timed on	fast formula for PIII	P4	Athlon	Sun
Pentium III 900 MHz	0.83	1.08	0.99	1.10
Pentium 4 1.4 GHz	0.97	0.63	0.73	1.23
Athlon 1.1 GHz	1.23	1.23	1.07	1.22
Sun UltraSparc II 450 MHz	0.95	1.67	1.42	0.82

Related Work

- Signal transform optimization
  - Minimizing arithmetic operations
  - Optimizing signal transforms for real computers FFTW (Frigo & Johnson)
    - Explicitly only considers FFTs
    - Restricted search space, chosen by hand without justification
- Automatic performance tuning and platform adaptation
  - PHiPAC (Bilmes et al.) and ATLAS (Whaley & Dongarra)
  - Using reinforcement learning (Lagoudakis & Littman; Vuduc et al.)
  - Using statistical modeling (Brewer)
  - Compiler Optimization (Moss et al.; Nisbet; Bodin et al.)
  - Combinatorial Optimization (Boyan; Zhang & Dietterich)

DP

Algorithm:

- Try all possible ways to split the root node
- For each child, use previously found best split tree
- Keep track of best found tree

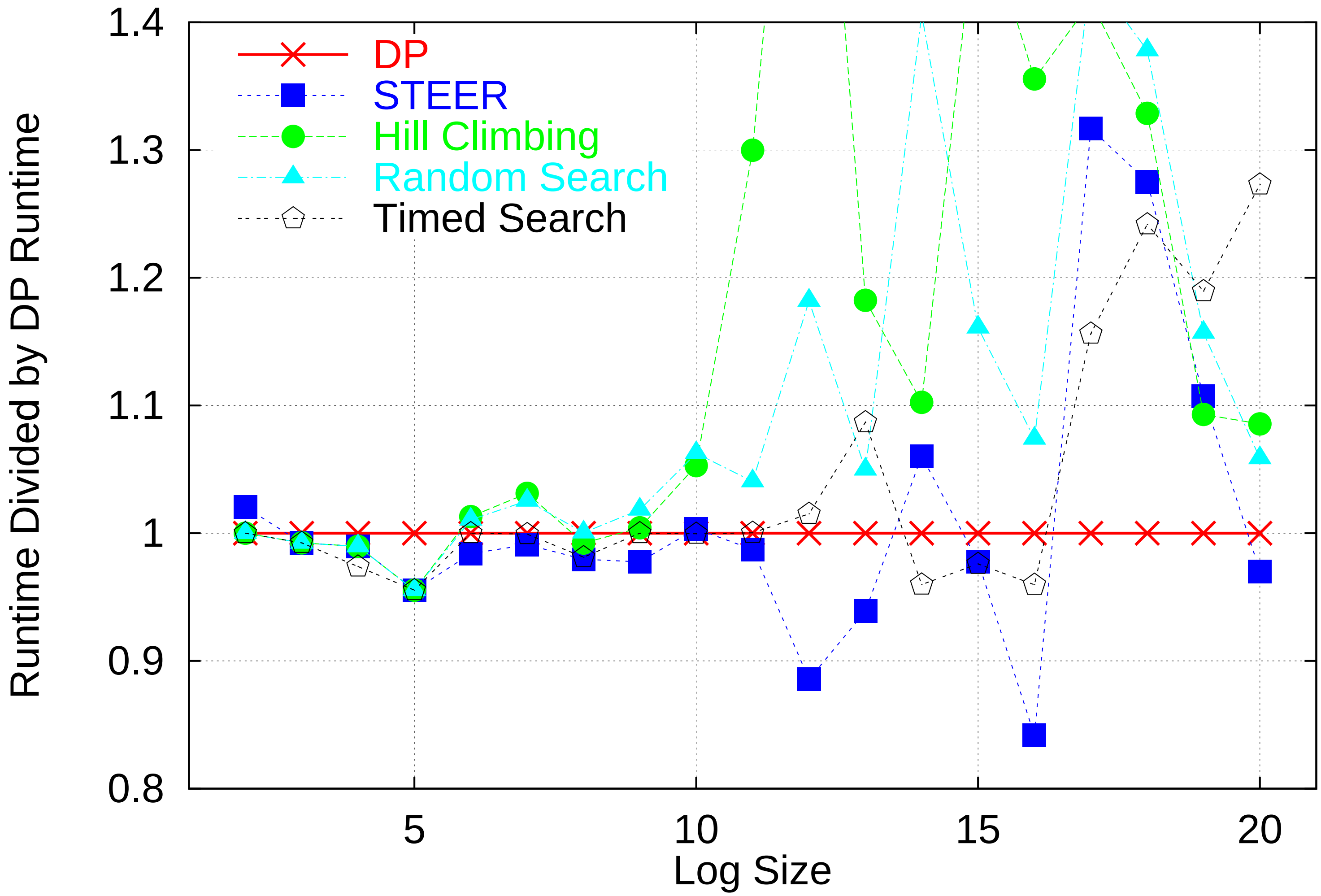
Assumes:

- Best way to split a node is independent of its location in the split tree

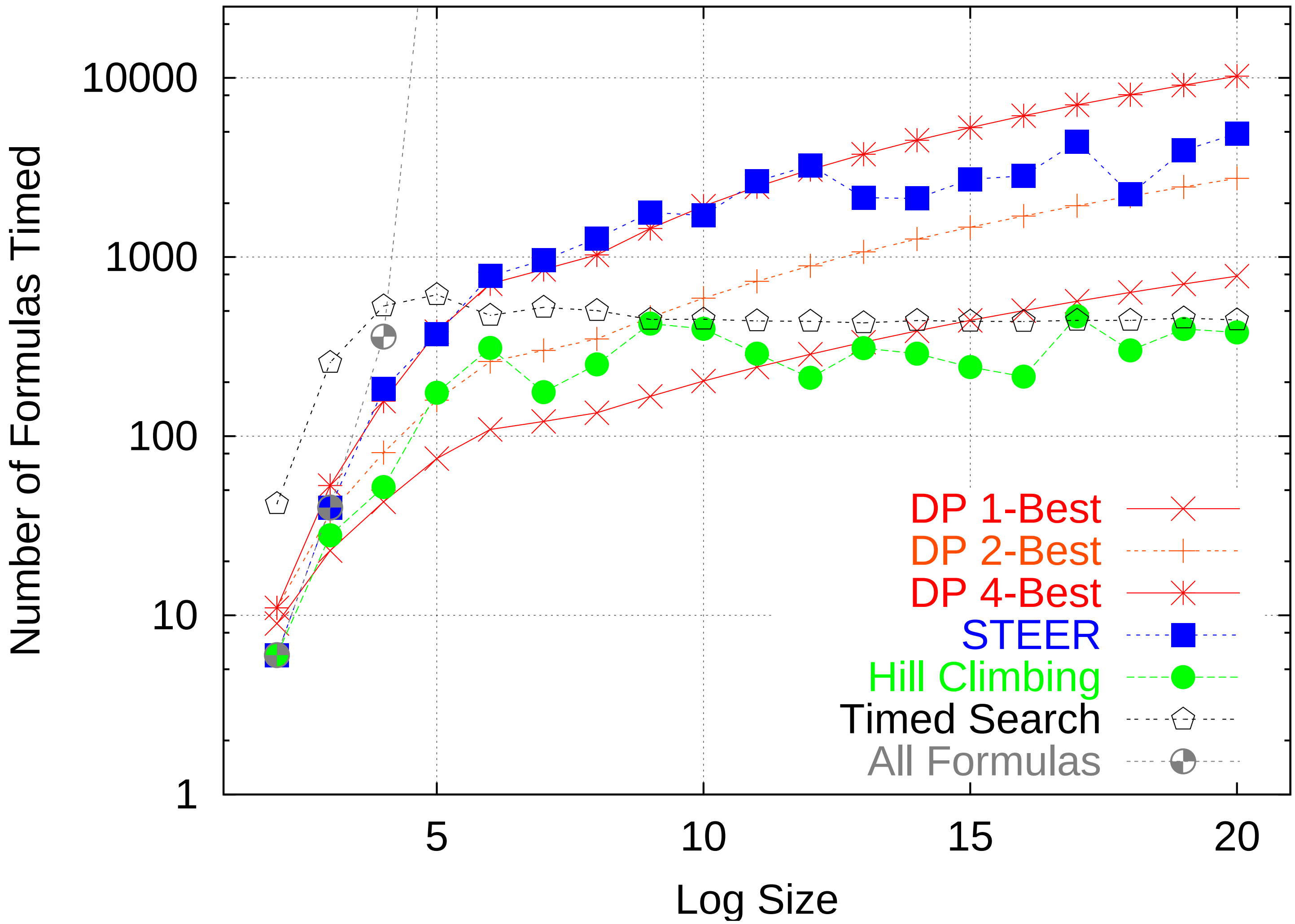
Can generalize:

- Keep track of the n-Best formulas for each transform/size

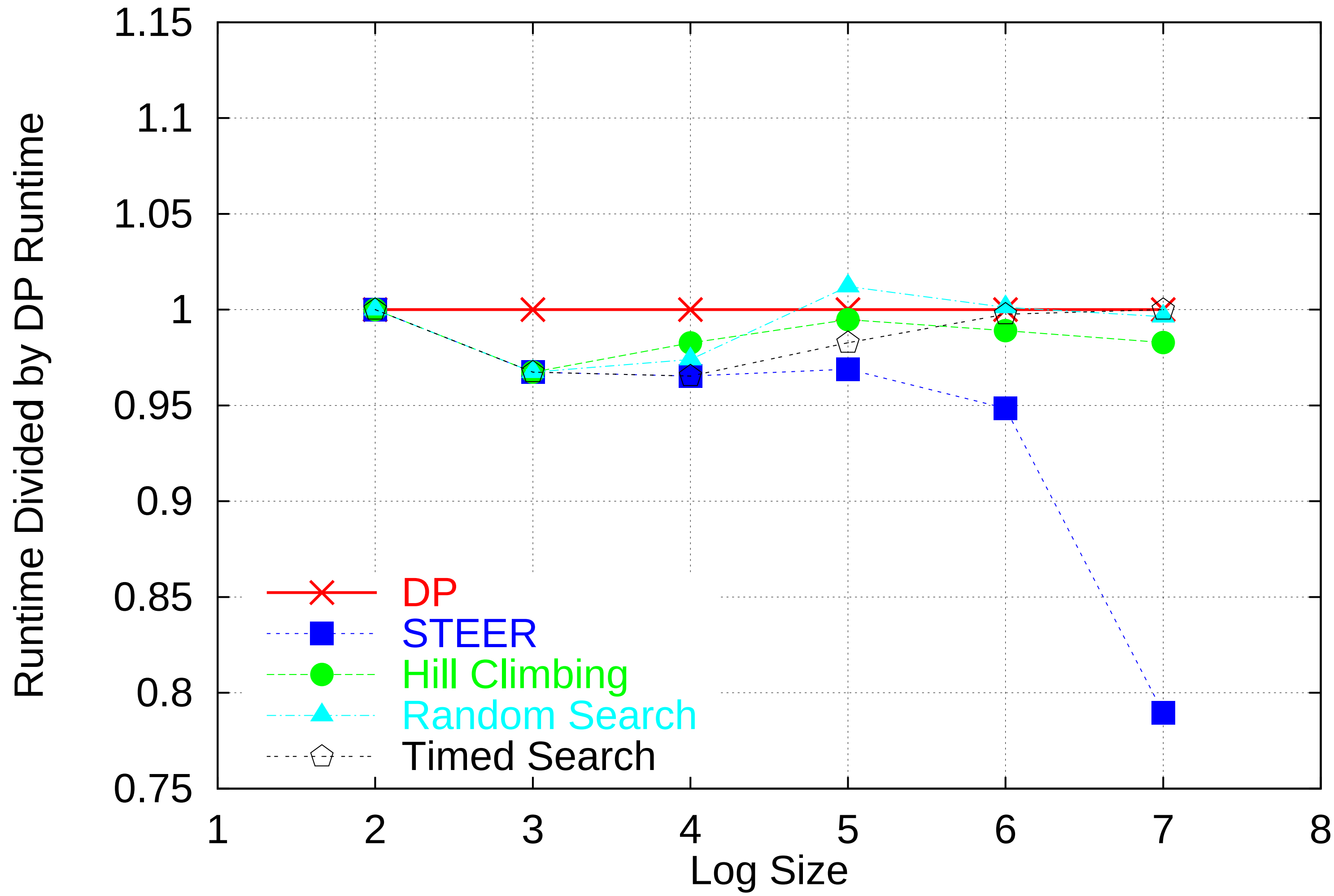
FFT on a Pentium



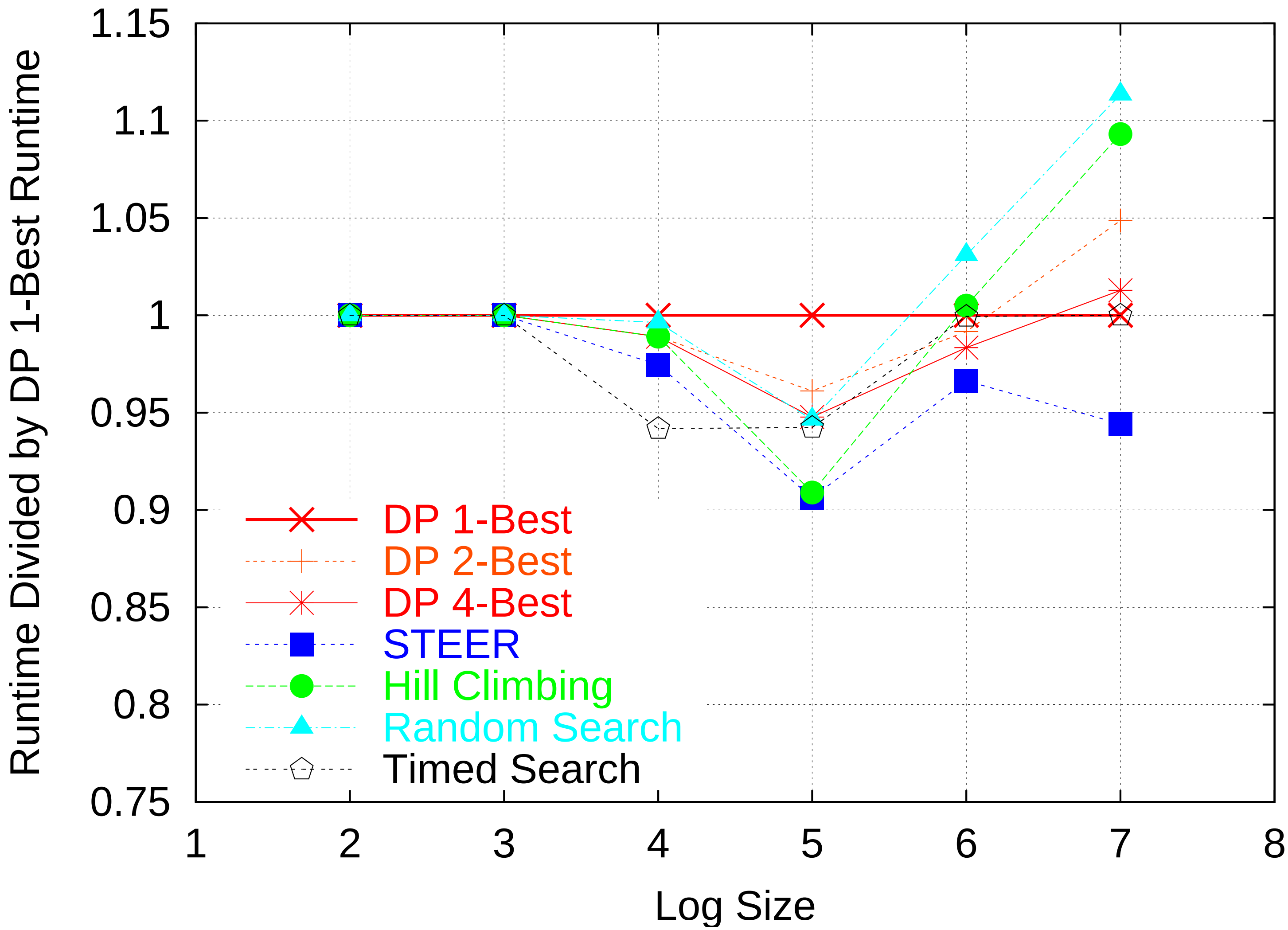
FFT on a Pentium: Number of Formulas Timed



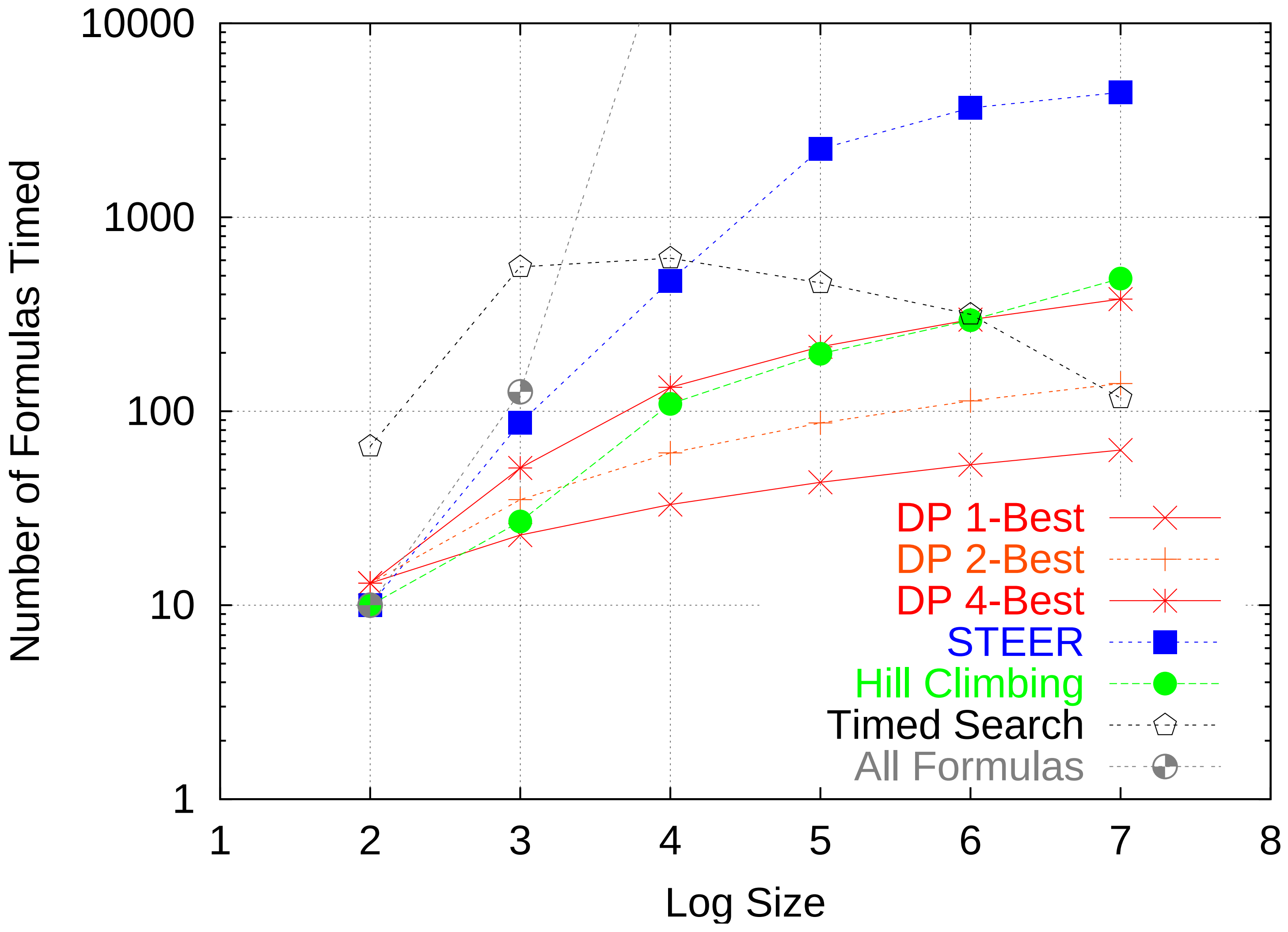
DCT Type II on a Pentium



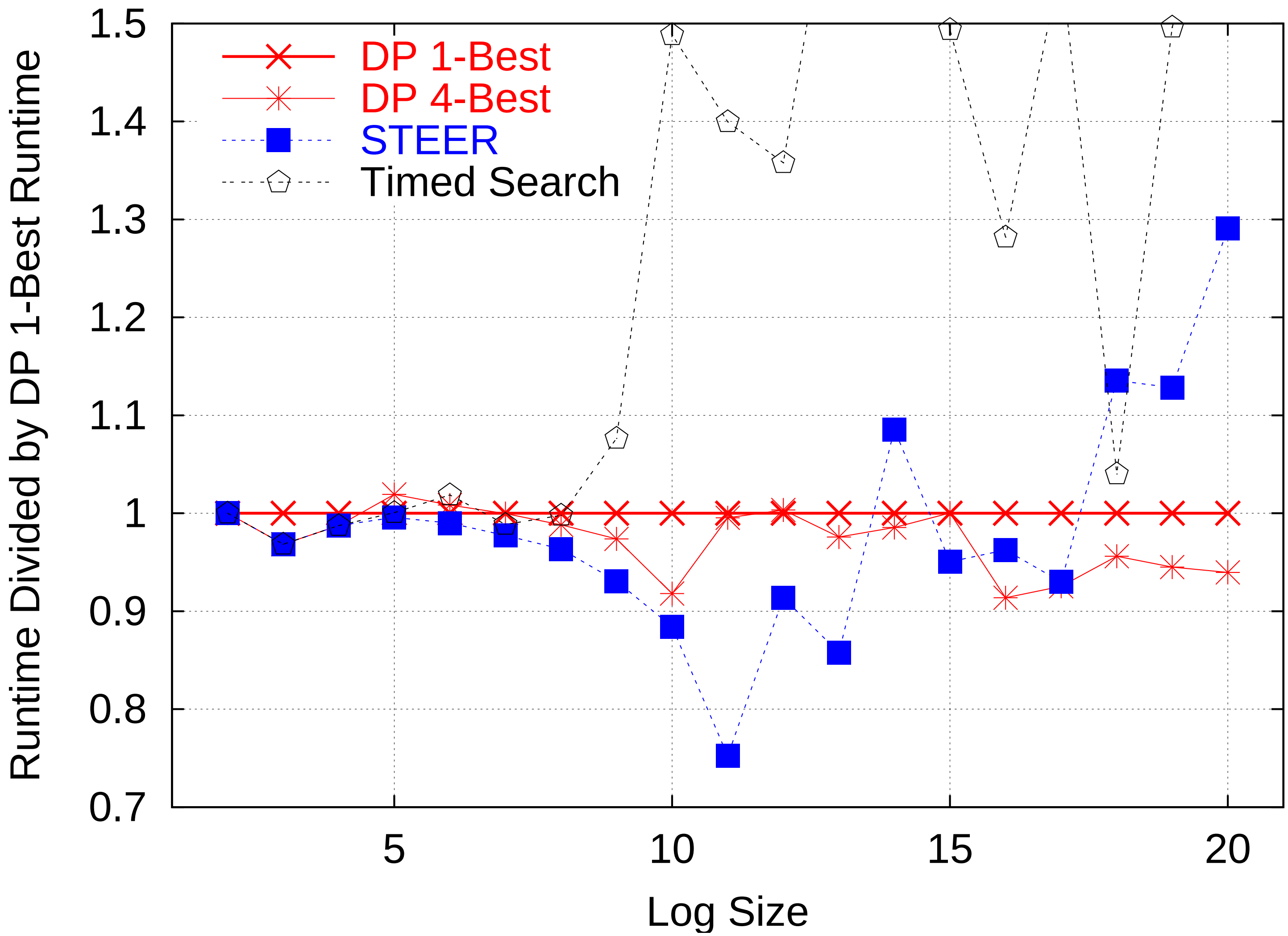
DCT Type IV on a Pentium



DCT IV on a Pentium: Number of Formulas Timed

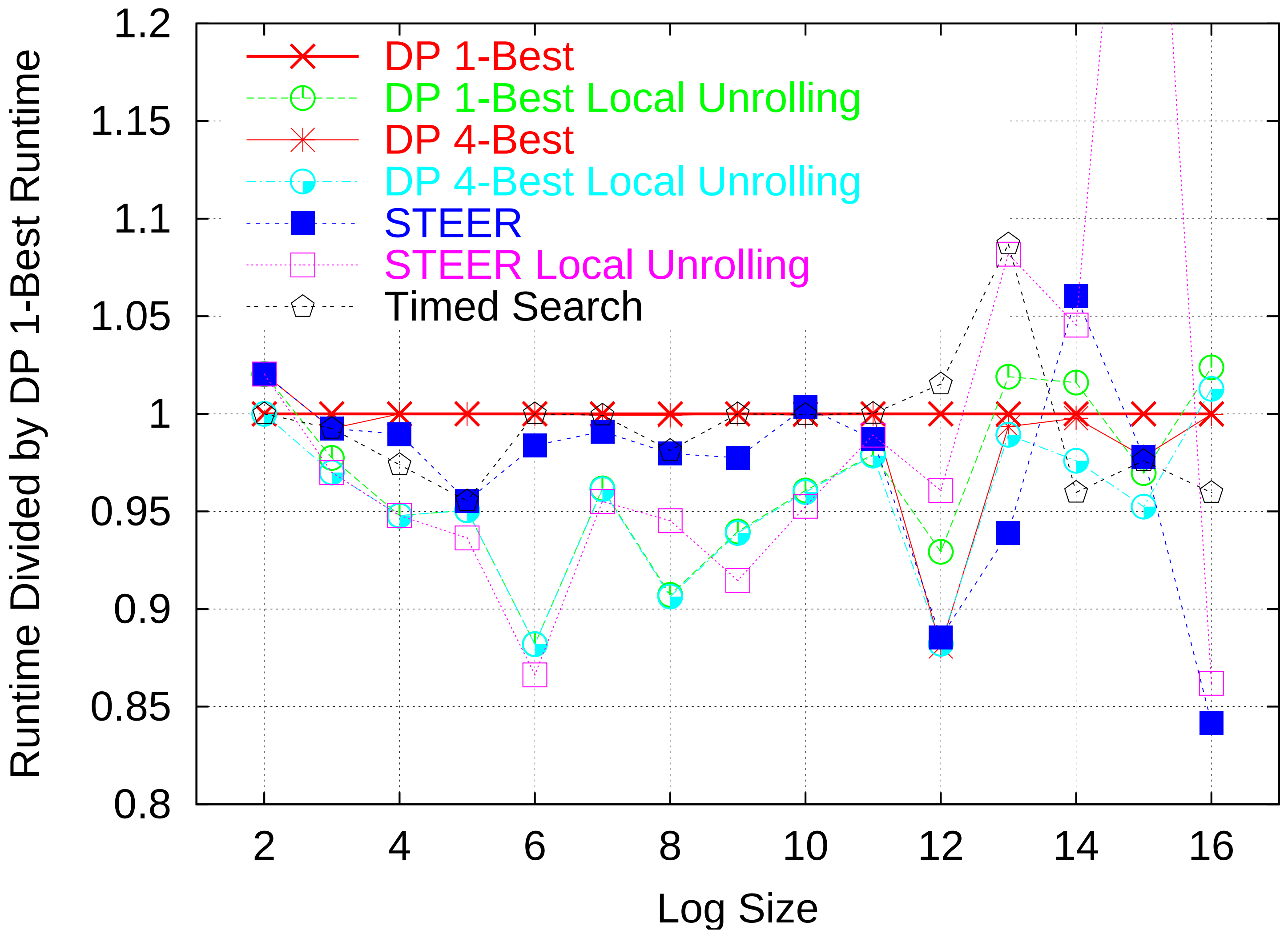


FFT on a Sun

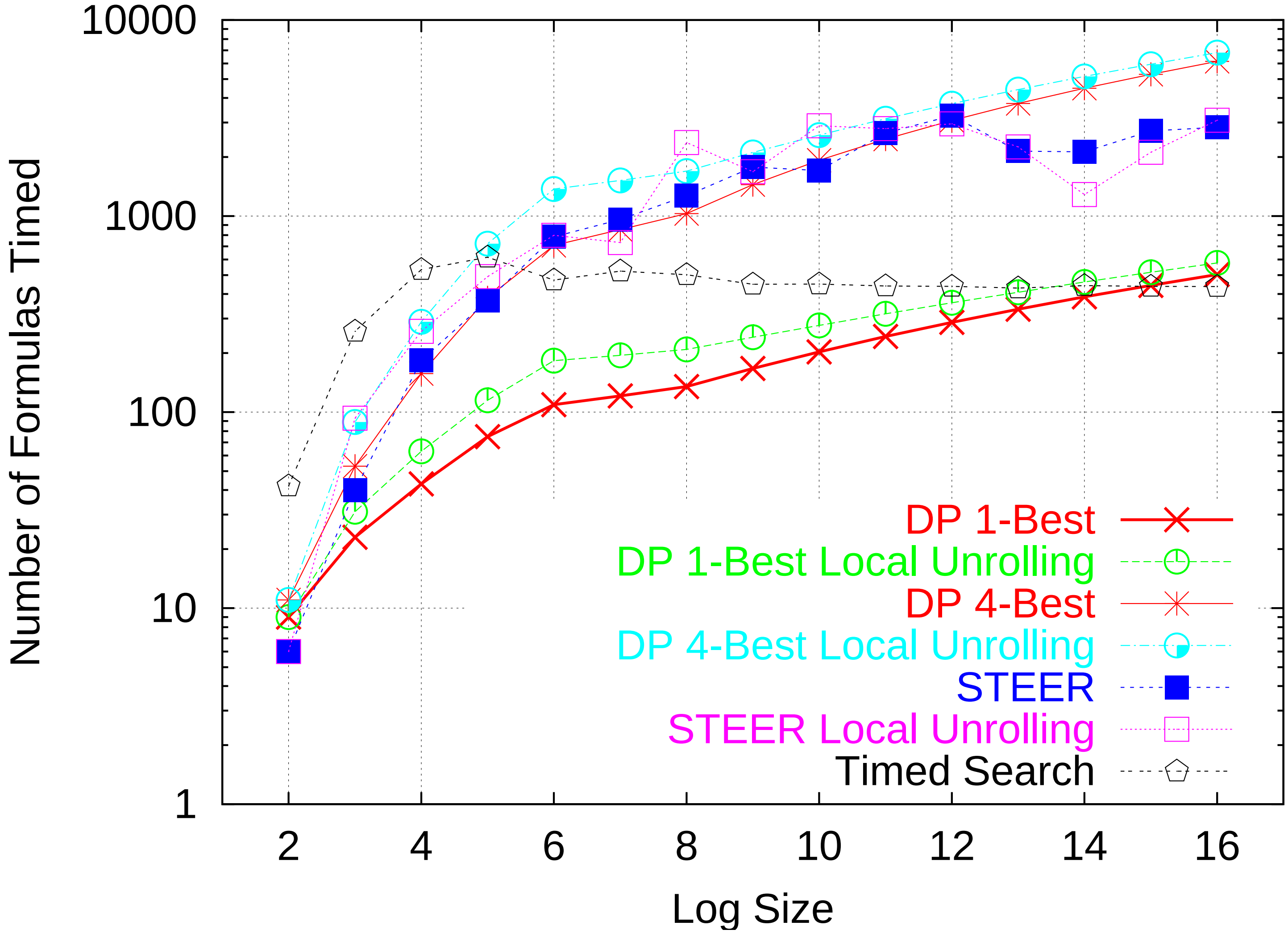


DP 1-Best
DP 4-Best
STEER
Timed Search
Runtime Divided by DP 1-Best Runtime
Log Size
1.5
1.4
1.3
1.2
1.1
1
0.9
0.8
0.7
5
10
15
20

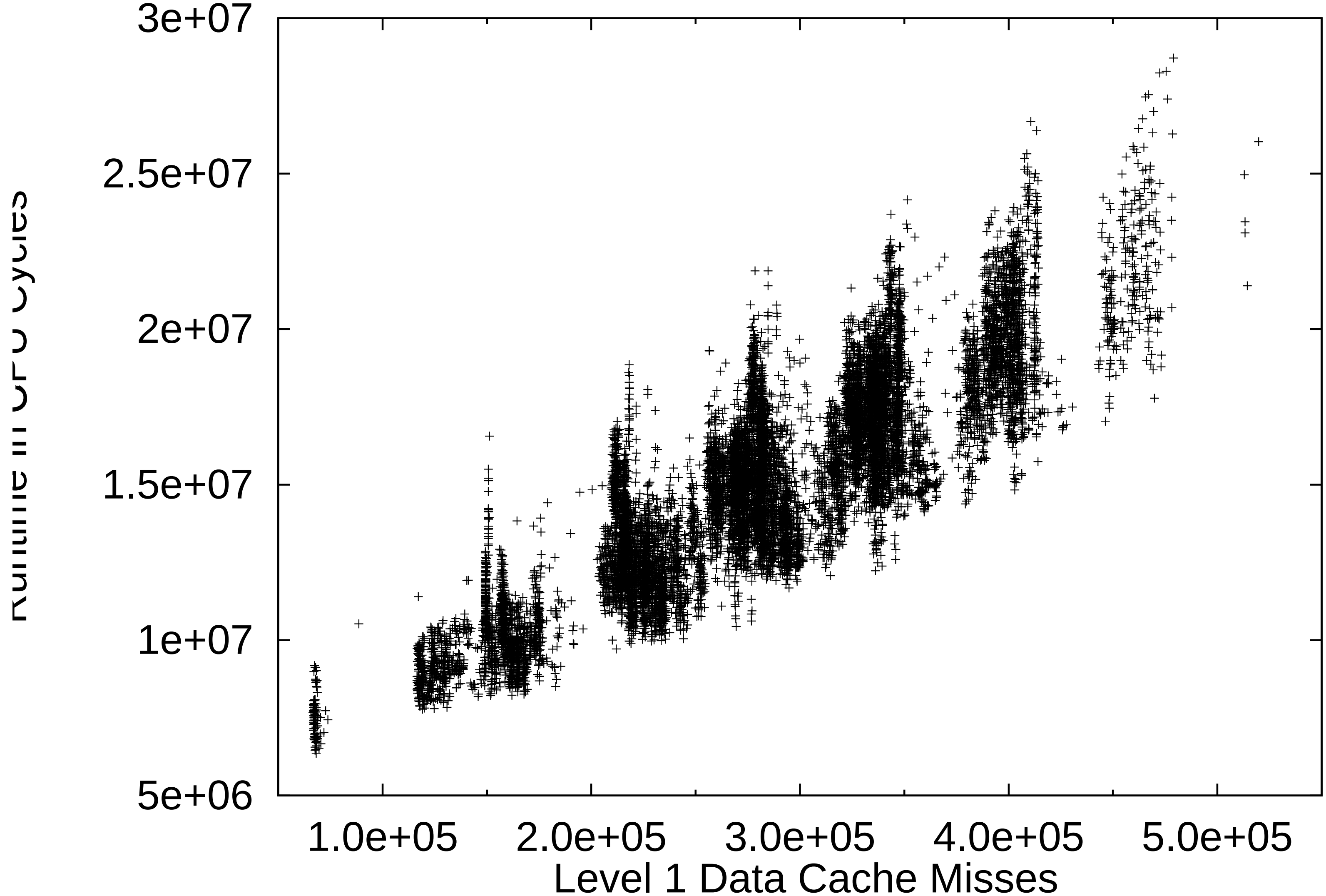
FFT on a Pentium with Local Unrolling



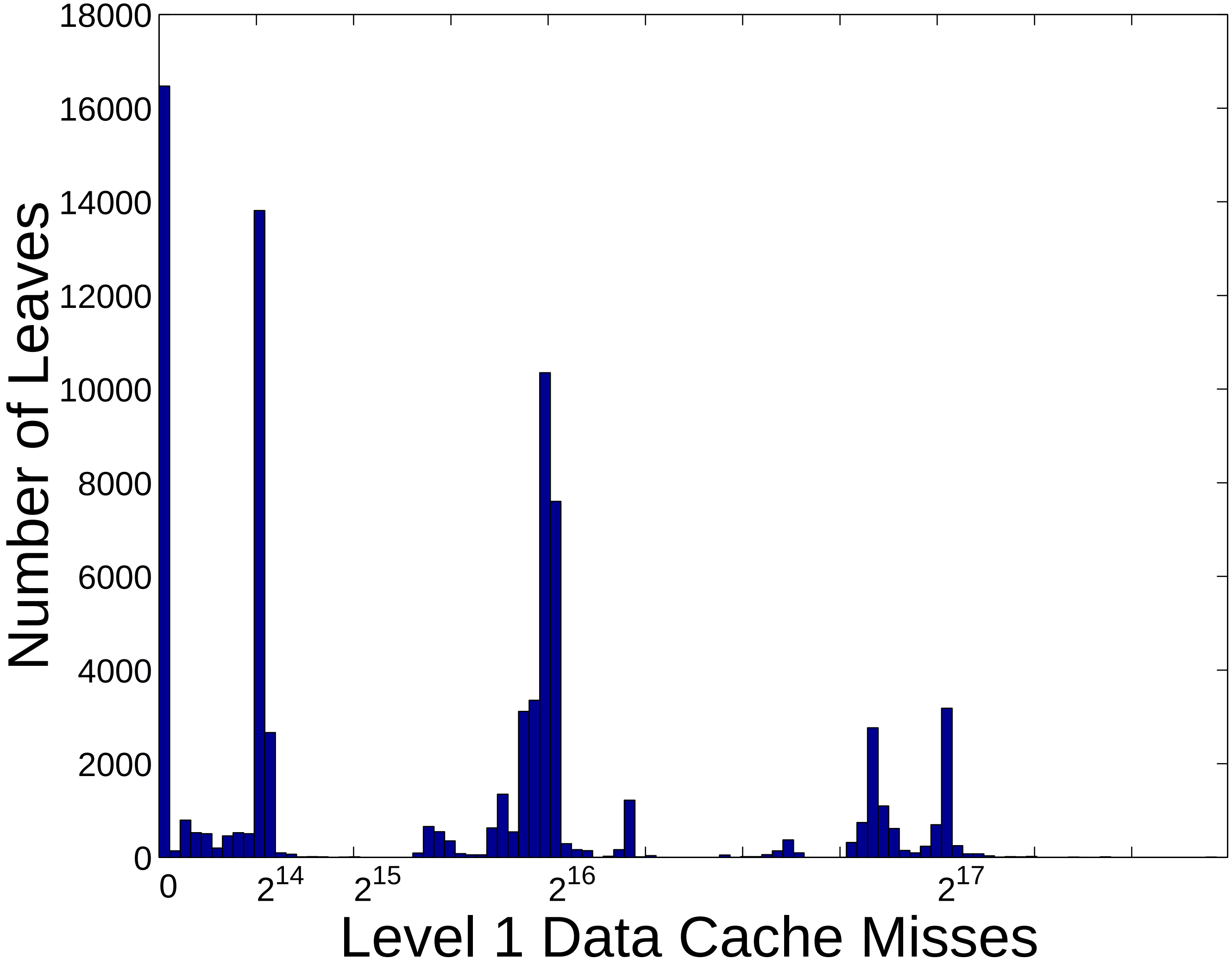
FFT with Local Unrolling: Number of Formulas Timed



WHT Runtime Vs. Cache Misses on a Pentium III



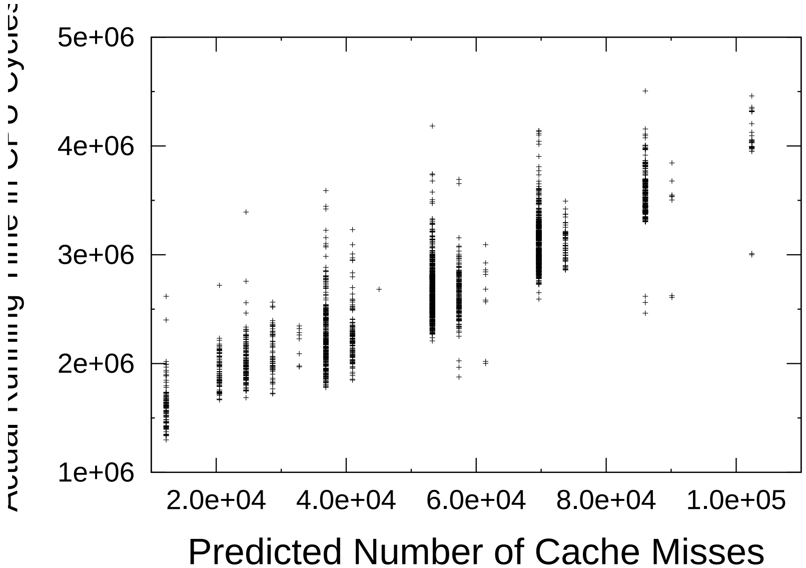
WHT Leaf Cache Misses on a Pentium III



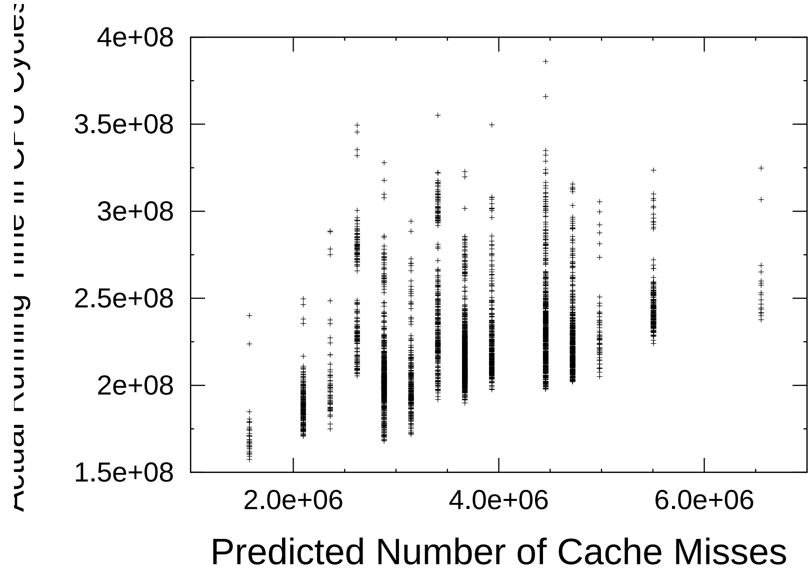
Predicted Cache Misses Versus Actual Runtime

WHT on a Pentium III

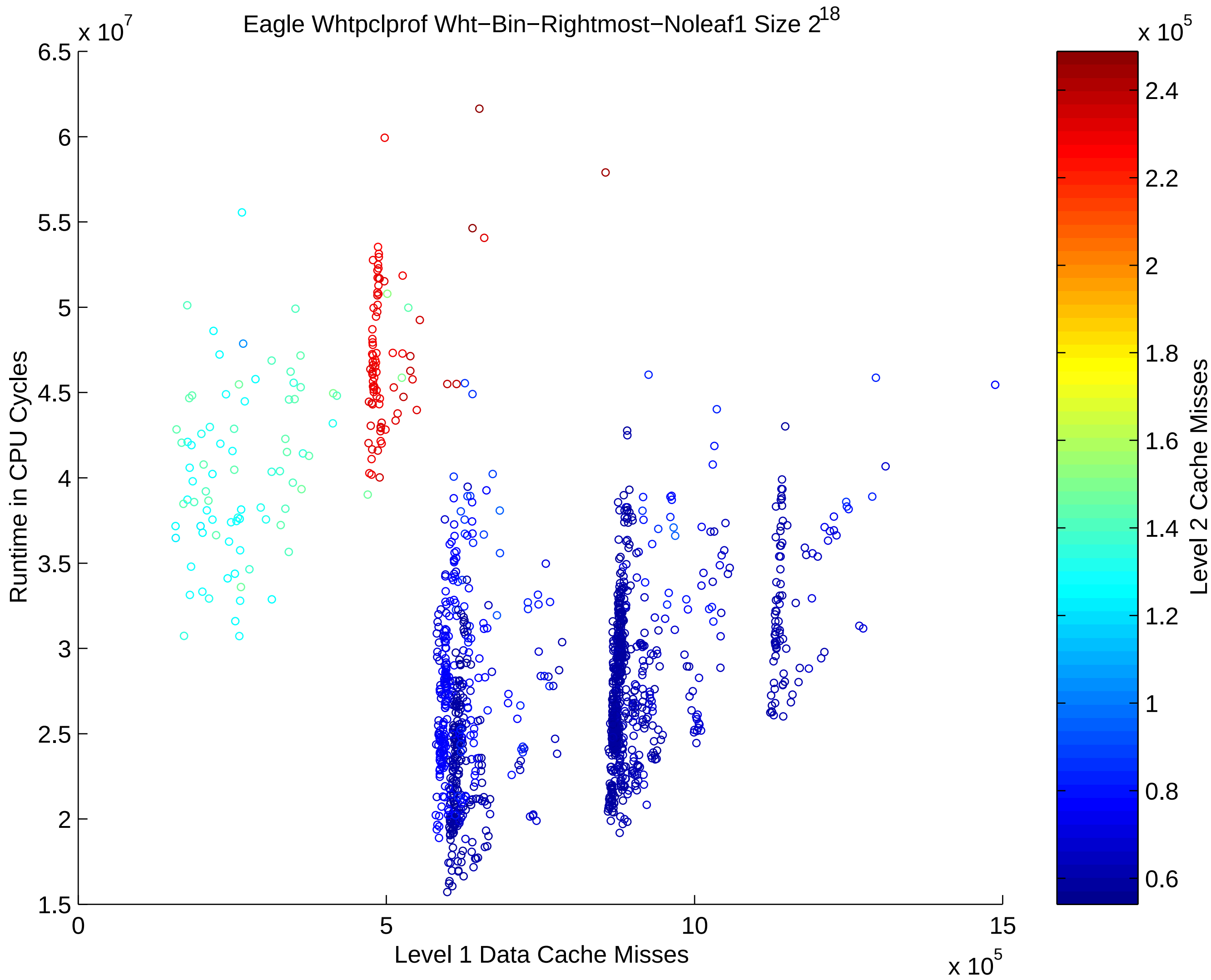
Binary No-2^1-Leaf
$WHT(2^{14})$



Binary No-2^1-Leaf
Rightmost $WHT(2^{20})$



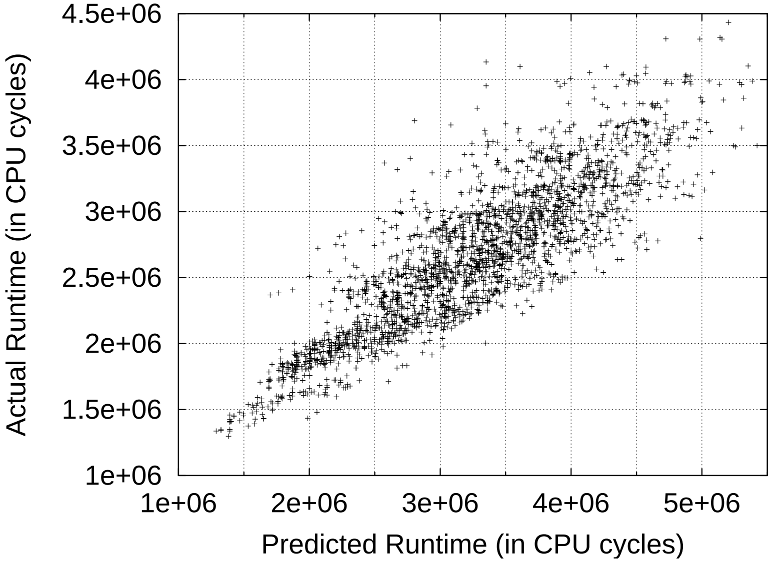
WHT on a Sun UltraSparc IIi



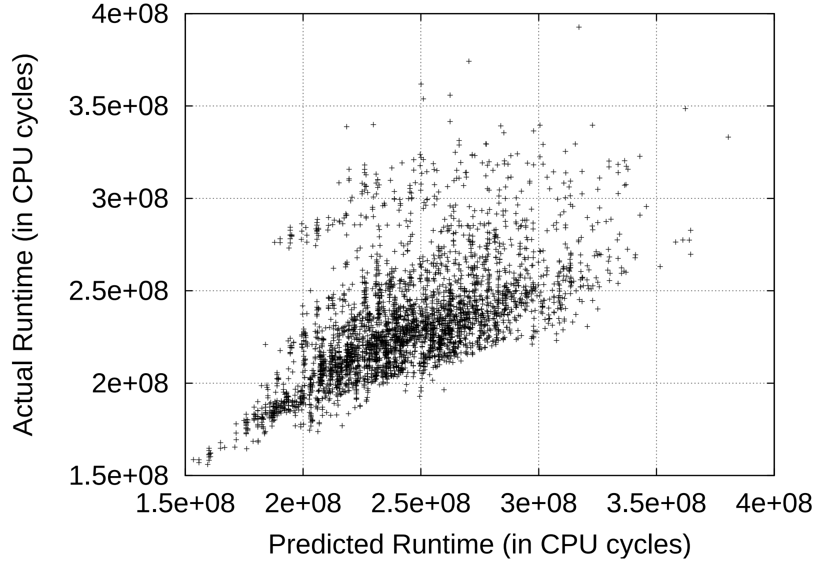
Predicted Runtime Versus Actual Runtime

WHT on a Pentium III

Binary No-2^1-Leaf
$WHT(2^{14})$



Binary No-2^1-Leaf
Rightmost $WHT(2^{20})$



Generating Fast Formulas: Approach

- Try to formulate in terms of Markov Decision Processes (MDPs) and Reinforcement Learning (RL)
- Final formulation not an MDP
- Final formulation borrows concepts from RL

MDPs

An MDP is a tuple $(\mathcal{S}, \mathcal{A}, T, C)$:

- $\mathcal{S}$ is a set of states
- $\mathcal{A}$ is a set of actions
- $T: \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ is a transition function that maps the current state and action to the next state
- $C: \mathcal{S} \times \mathcal{A} \rightarrow \Re$ is a cost function that maps the current state and action onto its real valued cost

Markov Property: T and C only depend on the current state and action

MDPs and RL

Agent:

- Observes current state
- Selects action to take
- Receives the cost for that action in that state
- Observes next state, and repeat

Reinforcement learning provides methods for finding a policy $\pi\colon \mathcal{S} \rightarrow \mathcal{A}$ that selects the best action at each state that minimizes the sum of costs incurred

Basic Formulation

Given a size, want to grow a fast split tree

Framing this problem in the MDP framework:

- States = unexpanded nodes in split tree
- Start state = root node of given size w/ no children
- Actions = ways to split a node, giving it children
  OR, make the node a leaf
- Cost Function = runtime of node
- Goal = minimize sum of costs

Detail: State Space Representation

States = unexpanded nodes in split tree

But how to represent the states???

Same features as before:

- Size and stride of the given node
- Size and stride of the parent of the given node
- Size and stride of the common parent to this node
- Size and stride of children and grandchildren if internal node

Detail: Cost Function

Ideal Cost Function =

Runtime of node represented by state

But, a node's runtime is not easily obtained

However, we can predict runtimes for nodes!

Difficulty: Transition Function

What is the transition function for this problem?

Given that 2 children of the root are grown:

- Which node is the next state?
- When will we transition back to the sibling?
- Where to transition to from a leaf node?
- And still maintain the Markov property?

We depart from the MDP framework here . . .