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 $Y = AX$

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:

\[
DFT(rs) = (DFT(r) \otimes I_s) T_{rs} (I_r \otimes DFT(s)) L_{rs}
\]

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

DCT IV $2^4$
RuleDCT4_3

DCT II $2^3$
RuleDCT2_2

DCT IV $2^2$
RuleDCT4_4

DCT II $2^2$
RuleDCT2_2

DCT II 2
RuleDCT2_2

DST II $2^3$
RuleDST2_3

DST IV $2^2$
RuleDST4_1

DST II 2
RuleDST2_3

DST IV 2
RuleDST4_3

DCT II 2
RuleDCT2_2

DCT IV 2
RuleDCT4_4

DST IV 2
RuleDST4_1

DST II 2
RuleDST2_3

DCTII2  DCTIV2  DSTIV2  DSTII2

DCTII2  DCTIV2  DSTIV2  DSTII2

DCTII2  DCTIV2  DSTIV2  DSTII2

DCTII2  DSTII2

DCTII2  DSTII2
**Search Space**

Large number of factorizations:

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

![Histogram graph showing the distribution of running times in CPU cycles for $WHT(2^{16})$. The x-axis represents running time in CPU cycles, ranging from 0.5 to $3 \times 10^7$, and the y-axis represents the number of formulas, ranging from 0 to 400. The graph displays a bell-shaped curve indicating a normal distribution of running times.](image-url)
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

1. Defined Transforms
   Break Down Rules

2. Formula Generation
   (Transform Factorization)

3. Code Generation

4. Search for Fast Implementations

Executable Code

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

DCT IV $2^4$
RuleDCT4_3

DCT II $2^3$
RuleDCT2_2

DCT IV $2^2$
RuleDCT4_4

DST II $2^3$
RuleDST2_3

DCT II $2^2$
RuleDCT2_2

DCT IV $2^2$
RuleDCT4_4

DST IV $2^2$
RuleDST4_1

DST II $2^2$
RuleDST2_3

DCT II 2
RuleDCT2_2

DCT IV 2
RuleDCT4_4

DST IV 2
RuleDST4_1

DST II 2
RuleDST2_3

DCTII2 DCTIV2 DSTIV2 DSTII2 DCTIV2 DSTIV2 DSTII2

DCTII2 DSTII2

Original
Mutation: Regrow

Original $\Rightarrow$ Truncate
Mutation: Regrow

Original $\Rightarrow$ Truncate $\Rightarrow$ Regrow
Running STEER
FFT on a Pentium III

![Graph of runtime divided by DP runtime vs log size, showing different algorithms such as DP, STEER, Hill Climbing, Random Search, and Timed Search.](image)
FFT on a Sun UltraSparc IIIi
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

```
          11
          |      Stride: 0
          |      ComPar: -
          |      PrevLeaf: -

          /|
         / |
        4 7
        |   |
        |   Stride: 7
        |   ComPar: 11
        |   PrevLeaf: 5

          |
          1
          |
          |
          |
          |
          Stride: 10
          ComPar: 4
          PrevLeaf: 3

          |
          3
          |
          |
          |
          |
          Stride: 7
          ComPar: 11
          PrevLeaf: 5

          |
          5
          |
          |
          |
          |
          Stride: 2
          ComPar: 7
          PrevLeaf: 2

          |
          2
          |
          |
          |
          |
          Stride: 0
          ComPar: -
          PrevLeaf: -
```
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

- 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: 20
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:

```
        20
       /   \
      19    20
     /     /   \   \
    1     2     18   ...  20
   /     /     /     /   \   \
  19    2     18    5     15   ...  19
```

...
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:

```
     20
    /  \
   5   15
  /    /  \
 3    2   5 10
```
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:

\[ V(state) = \min_{\text{splittings}} \sum_{\text{children}} V(child) + \text{PredictedRuntime}(state) \]

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)

<table>
<thead>
<tr>
<th>Generation Rank</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicted Runtime</td>
<td>4.4</td>
<td>4.5</td>
<td>4.7</td>
<td>4.8</td>
<td>...</td>
</tr>
<tr>
<td>Actual Runtime</td>
<td>4.4</td>
<td>4.7</td>
<td>4.3</td>
<td>5.2</td>
<td>...</td>
</tr>
</tbody>
</table>
## Fast Formula Generation Results

### FFT on a Pentium III

<table>
<thead>
<tr>
<th>Size</th>
<th>Generation rank of fastest formula</th>
<th>Rank 1 formula is $X%$ slower than fastest formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{12}$</td>
<td>16</td>
<td>14.3%</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>1</td>
<td>0.0%</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>2</td>
<td>13.6%</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>1</td>
<td>0.0%</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>1</td>
<td>0.0%</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>82</td>
<td>3.6%</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>11</td>
<td>6.5%</td>
</tr>
</tbody>
</table>

70,376 different $FFT(2^{18})$ formulas
Fast Formula Generation Results

FFT on a Pentium III

Runtime Divided by DP Runtime vs. Log Size

- DP
- Exhaust
- Generate 1
- Generate 20
- Generate 100

Log Size

0.7
0.8
0.9
1
1.1
1.2
1.3
1.4
1.5
1.6
1.7
12 14 16 18 20

Runtime Divided by DP Runtime
## Fast Formula Generation Results

**WHT on a Pentium III**

<table>
<thead>
<tr>
<th>Size</th>
<th>Generation rank of best known formula</th>
<th>Rank 1 formula is X% slower than best known formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{13}$</td>
<td>5</td>
<td>3.4%</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>4</td>
<td>3.0%</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>3</td>
<td>2.1%</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>4</td>
<td>1.7%</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>5</td>
<td>0.1%</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>4</td>
<td>2.0%</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>1</td>
<td>0.0%</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>4</td>
<td>1.7%</td>
</tr>
</tbody>
</table>

398,041 different $WHT(2^{20})$ formulas
## Fast Formula Generation Results

<table>
<thead>
<tr>
<th>Size</th>
<th>Generation rank of best known formula</th>
<th>Rank 1 formula is ( X )% slower than best known formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2^{13})</td>
<td>14</td>
<td>77.7%</td>
</tr>
<tr>
<td>(2^{14})</td>
<td>20</td>
<td>12.8%</td>
</tr>
<tr>
<td>(2^{15})</td>
<td>1</td>
<td>0.0%</td>
</tr>
<tr>
<td>(2^{16})</td>
<td>2</td>
<td>4.3%</td>
</tr>
<tr>
<td>(2^{17})</td>
<td>7</td>
<td>18.0%</td>
</tr>
<tr>
<td>(2^{18})</td>
<td>38</td>
<td>5.9%</td>
</tr>
<tr>
<td>(2^{19})</td>
<td>17</td>
<td>3.3%</td>
</tr>
<tr>
<td>(2^{20})</td>
<td>47</td>
<td>1.4%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Timing Platform</th>
<th>fast formula for</th>
<th>PIII</th>
<th>P4</th>
<th>Athlon</th>
<th>Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium III 900 MHz</td>
<td></td>
<td>0.83</td>
<td>1.08</td>
<td>0.99</td>
<td>1.10</td>
</tr>
<tr>
<td>Pentium 4 1.4 GHz</td>
<td></td>
<td>0.97</td>
<td>0.63</td>
<td>0.73</td>
<td>1.23</td>
</tr>
<tr>
<td>Athlon 1.1 GHz</td>
<td></td>
<td>1.23</td>
<td>1.23</td>
<td>1.07</td>
<td>1.22</td>
</tr>
<tr>
<td>Sun UltraSparc II 450 MHz</td>
<td></td>
<td>0.95</td>
<td>1.67</td>
<td>1.42</td>
<td>0.82</td>
</tr>
</tbody>
</table>
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)
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

![Graph showing runtime divided by DP runtime vs log size for different search methods: DP, STEER, Hill Climbing, Random Search, Timed Search.](image)
FFT on a Pentium: Number of Formulas Timed

![Graph showing the number of formulas timed against log size for different methods: DP 1-Best, DP 2-Best, DP 4-Best, STEER, Hill Climbing, Timed Search, and All Formulas.]
DCT Type II on a Pentium

![Graph showing runtime divided by DP runtime for different log sizes. The graph compares DP, STEER, Hill Climbing, Random Search, and Timed Search algorithms.]
DCT Type IV on a Pentium

Runtime Divided by DP 1-Best Runtime

Log Size
DP 1-Best
DP 2-Best
DP 4-Best
STEER
Hill Climbing
Random Search
Timed Search
FFT on a Sun

![Graph showing runtime divided by DP 1-Best runtime against log size. The graph compares DP 1-Best, DP 4-Best, STEER, and Timed Search methods.](image-url)
**FFT on a Pentium with Local Unrolling**

<table>
<thead>
<tr>
<th>Log Size</th>
<th>DP 1-Best</th>
<th>DP 1-Best Local Unrolling</th>
<th>DP 4-Best</th>
<th>DP 4-Best Local Unrolling</th>
<th>STEER</th>
<th>STEER Local Unrolling</th>
<th>Timed Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graph showing runtime divided by DP 1-Best runtime vs log size](image)
FFT with Local Unrolling: Number of Formulas Timed

Number of Formulas Timed vs Log Size for different algorithms:
- DP 1-Best
- DP 1-Best Local Unrolling
- DP 4-Best
- DP 4-Best Local Unrolling
- STEER
- STEER Local Unrolling
- Timed Search
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 IIIi
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 $(S, A, T, C)$:

- $S$ is a set of states
- $A$ is a set of actions
- $T: S \times A \rightarrow S$ is a transition function that maps the current state and action to the next state
- $C: S \times A \rightarrow \mathbb{R}$ 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 $\pi: S \rightarrow A$ 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!
Difficult: 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 . . .