Learning to Generate
Fast Signal Processing Implementations

Bryan Singer

Joint work with Manuela Veloso

Shorter version to be presented at ICML-2001
Overview

- Background and Motivation
- Key Signal Processing Observations
- Predicting Leaf Cache Misses
- 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
Walsh-Hadamard Transform (WHT)

Highly structured, for example:

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

Factorization or break down rule:

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

for positive integers \( n_i \) such that \( n = n_1 + \cdots + n_t \)
WHT Example

\[ WHT(2^5) = [WHT(2^3) \otimes I_{2^2}] [I_{2^3} \otimes WHT(2^2)] \]
\[ = \left\{ (WHT(2^1) \otimes I_{2^2}) (I_{2^1} \otimes WHT(2^2)) \right\} \otimes I_{2^2} \]
\[ [I_{2^3} \otimes \left\{ (WHT(2^1) \otimes I_{2^1}) (I_{2^1} \otimes WHT(2^1)) \right\} ] \]

We can visualize this as a **split tree**:

```
  5
 / \  /
3   2 /   /
|   |1 2 1
1   1
```

1-1 correspondence between split trees and formulas
Search Space

Large number of factorizations:

- $WHT(2^n)$ has $\Theta((4 + \sqrt{8})^n/n^{3/2})$ different split trees
- $WHT(2^n)$ has $\Theta(5^n/n^{3/2})$ different binary split trees
- $WHT(2^{10})$ has 51,819 binary split trees
Varying Performance

Varying performance of factorizations:

- Formulas have very different running times
- Small changes in the split tree can lead to significantly different running times
- Optimal formulas across machines are different

Reasons:

- Cache performance
- Utilization of execution units
- Number of registers
Histogram of $WHT(2^{16})$ Running Times
Problem

Huge search space of formulas

Want to find the fastest formula
  • For a given transform
  • For a given size
  • For a given machine
  • But for any input vector

Our Approach: Learn to generate fast formulas
  • Learn to predict cache misses for leaves
  • Use this as the base cases for determining values of different splittings
  • Construct fast formulas by choosing best splittings
Overview

• Background and Motivation
• Key Signal Processing Observations
• Predicting Leaf Cache Misses
• Generating Fast Formulas
• Conclusions
Run Times and Cache Misses

- Runtime in CPU Cycles
- Level 1 Data Cache Misses

![Graph showing relationship between runtime and level 1 data cache misses.](image-url)
Run Times and Cache Misses

- Fastest formula has minimal number of cache misses
- Minimizing cache misses produces small group of formulas which contains the fastest formula
WHT Leaves

- WHT leaves are implemented as unrolled code (sizes $2^1$ to $2^8$)
- Internal nodes recursively call their children
- All run time and cache misses occur in the leaves
- Total run time or cache misses of a formula is the sum of that incurred by the leaves
- If we can predict for leaves, then we can predict for entire formulas
Leaf Cache Misses: \( WHT(2^{16}) \) example
Leaf Cache Misses

- The number of cache misses incurred by leaves is only of a few possible values
- These values are fractions of the transform size
- We can predict one of a few categories instead of real valued number of cache misses
- We can learn across different sizes by learning the categories corresponding to fractions of the transform size
Review of Observations

- Fastest formula has minimal number of cache misses
- All computation performed in the leaves
- Leaf cache misses only have a few values
- Leaf cache misses are fractions of transform size
Overview

- Background and Motivation
- Key Signal Processing Observations
- Predicting Leaf Cache Misses
- Generating Fast Formulas
- Conclusions
Predicting Leaf Cache Misses

- Want to learn to accurately predict leaf cache misses
- Should then be able to predict cache misses for entire formulas
Learning Algorithm

1. Collect cache misses for leaves of WHT formulas

2. Classify \((\text{cache misses} / \text{transform size})\) as:
   - near-zero if less than \(1/8\)
   - near-quarter if less than \(1/2\)
   - near-whole if less than \(3/2\)
   - large otherwise.

3. Train a classification algorithm to predict one of the four classes given a leaf
Features for WHT Leaves

Need to describe WHT leaves with features

Could use:

- Size of the given leaf
- Stride of the given leaf

Stride:

- Determines how a node accesses its input and output data
- Greatly impacts cache performance
- Determined by location of node in split tree
More Features for WHT Leaves

- Size and stride of the given leaf
- Size and stride of the parent of the given leaf
- Size and stride of the common parent
Review: Learning Algorithm

1. Collect cache misses for leaves of WHT formulas
2. Classify (cache misses / transform size) as:
   - near-zero if less than 1/8
   - near-quarter if less than 1/2
   - near-whole if less than 3/2
   - large otherwise.
3. Describe leaves with features
4. Train a classification algorithm to predict one of the four classes given features for a leaf
Evaluation

- Trained a decision tree
- Used a random 10% of leaves of all binary $WHT(2^{16})$ split trees with no leaves of size $2^1$
- Evaluated performance using subsets of formulas known to be fastest
- Can not evaluate over all formulas because there are too many possible formulas
Leaf Cache Miss Category Performance

Error rates for predicting cache miss category incurred by leaves

<table>
<thead>
<tr>
<th>Binary No-2(^1)-Leaf</th>
<th>Size</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(2^{12})</td>
<td>0.5%</td>
</tr>
<tr>
<td></td>
<td>(2^{13})</td>
<td>1.7%</td>
</tr>
<tr>
<td></td>
<td>(2^{14})</td>
<td>0.9%</td>
</tr>
<tr>
<td></td>
<td>(2^{15})</td>
<td>0.9%</td>
</tr>
<tr>
<td></td>
<td>(2^{16})</td>
<td>0.7%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary No-2(^1)-Leaf Rightmost</th>
<th>Size</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(2^{17})</td>
<td>1.7%</td>
</tr>
<tr>
<td></td>
<td>(2^{18})</td>
<td>1.7%</td>
</tr>
<tr>
<td></td>
<td>(2^{19})</td>
<td>1.7%</td>
</tr>
<tr>
<td></td>
<td>(2^{20})</td>
<td>1.6%</td>
</tr>
<tr>
<td></td>
<td>(2^{21})</td>
<td>1.6%</td>
</tr>
</tbody>
</table>

Trained on one size, performs well across many!
Predicting Cache Misses for Entire Formulas

Average percentage error for predicting cache misses for entire formulas

<table>
<thead>
<tr>
<th>Binary No-2\textsuperscript{1}-Leaf</th>
<th>Binary No-2\textsuperscript{1}-Leaf Rightmost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Errors</td>
</tr>
<tr>
<td>(2^{12})</td>
<td>12.7%</td>
</tr>
<tr>
<td>(2^{13})</td>
<td>8.6%</td>
</tr>
<tr>
<td>(2^{14})</td>
<td>6.7%</td>
</tr>
<tr>
<td>(2^{15})</td>
<td>5.2%</td>
</tr>
<tr>
<td>(2^{16})</td>
<td>4.6%</td>
</tr>
</tbody>
</table>

Error = \(\frac{1}{|TestSet|} \sum_{i \in TestSet} \frac{|a_i - p_i|}{a_i}\), where \(a_i\) and \(p_i\) are the actual and predicted number of cache misses for formula \(i\).
Runtime Versus Predicted Cache Misses

Binary No-2\(^1\)-Leaf

\[ WHT(2^{14}) \]

Actual Running Time in CPU Cycles

Predicted Number of Cache Misses

Binary No-2\(^1\)-Leaf

Rightmost \[ WHT(2^{20}) \]

Actual Running Time in CPU Cycles

Predicted Number of Cache Misses
Review: Predicting Cache Misses

By learning to predict leaf cache misses:

- Accurately predict cache misses for entire formulas
- Fastest formulas have fewest predicted cache misses
- Predict accurately across many transform sizes while trained on one size
Overview

• Background and Motivation
• Key Signal Processing Observations
• Predicting Leaf Cache Misses
• Generating Fast Formulas
• Conclusions
Generating Fast Formulas

- Can now quickly predict cache misses for a formula
- Fastest formulas have minimal cache misses
- But still MANY formulas to search through

Can we learn to generate fast formulas?
Generating Fast Formulas: Approach

Control Learning Problem:

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

Want to grow the fastest WHT split tree:

- Begin with a root node of the desired size
Generating Fast Formulas: Approach

Control Learning Problem:

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

Want to grow the fastest WHT split tree:

• Begin with a root node of the desired size
• Grow best possible children
Generating Fast Formulas: Approach

Control Learning Problem:

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

Want to grow the fastest WHT split tree:

- Begin with a root node of the desired size
- Grow best possible children
- Recurse on each of the children
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 WHT split tree

• 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 =
  • Zero when giving children to a node
  • The leaf’s run time when making a node a leaf
• Goal = minimize sum of costs
Detail: State Space Representation

States = unexpanded nodes in split tree
But how to represent the states???

Modified leaf features for arbitrary 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 to this node
Detail: Cost Function

Ideal Cost Function =

- Zero when giving children to a node
- The leaf’s run time when making a node a leaf

But, a leaf’s runtime is not easily obtained

However, we can predict cache misses for leaves!

Used Cost Function =

- Zero when giving children to a node
- The leaf’s predicted cache misses when making a node a leaf

Now we really minimize the number of cache misses
Difficultly: 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 . . .
Our Approach

Problem advantages:

• Deterministic and known actions
• Deterministic and known cost function
  (learned decision tree)

Approach:

• Define an optimal value function on states
• Run DP to determine value function
  (basically like solving an MDP)
Value Function

Define an optimal value function on states:

- Value of a state is the cost of the best subtree
- Value of root node is the cost of the best formula
- Choose children that have minimal sum of values
Mathematically: Value Function on States

State = unexpanded node in split tree, described by 6 features

The optimal value of a state is:

$$V^*(state) = \min_{subtrees} \sum_{leaf \in subtree} CacheMisses(leaf)$$

- Min over all possible subtrees of the given state
- $CacheMisses()$ returns the predicted number of cache misses for the given leaf
Recursive Formulation of Value Function

Define:

\[ \text{LeafCM}(state) = \begin{cases} \text{CacheMisses}(state), & \text{if state can be a leaf} \\ \infty, & \text{if state cannot be a leaf} \end{cases} \]

and

\[ \text{SplitV}(state) = \min_{\text{splittings}} \sum_{\text{child} \in \text{splitting}} V^*(\text{child}) \]

Then:

\[ V^*(state) = \min\{\text{LeafCM}(state), \text{SplitV}(state)\} \]
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, memoizing results
- Determine set of children with minimal sum of values
- Root’s value is this minimal sum of values
Generating Fast Formulas

Generate split tree with minimal Value (or near minimal)
- Consider all possible sets of children of the root
- Choose those that have the minimal sum of values
- Recurse on children
**Evaluation**

**Difficulty:**
- Do not know what the optimal formula is
- Too many formulas to exhaust at larger sizes

**Possible:**
- Exhaust over limited subspaces
- Limit based on signal processing knowledge and prior experience from using different search methods
- Compare my method with best found by this limited exhaust
## Fast Formula Generation Results

<table>
<thead>
<tr>
<th>Size</th>
<th>Number of Formulas Generated</th>
<th>Generated Included the Fastest Known</th>
<th># of Fastest Formulas in Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{12}$</td>
<td>101</td>
<td>yes</td>
<td>77</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>86</td>
<td>yes</td>
<td>4</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>101</td>
<td>yes</td>
<td>70</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>86</td>
<td>yes</td>
<td>11</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>101</td>
<td>yes</td>
<td>68</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>86</td>
<td>yes</td>
<td>15</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>101</td>
<td>yes</td>
<td>25</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>86</td>
<td>yes</td>
<td>16</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>101</td>
<td>yes</td>
<td>16</td>
</tr>
</tbody>
</table>
Histograms: $WHT(2^{20})$

Limited Exhaust
Binary No-$2^1$-Leaf Rightmost

Our method
Overview

- Background and Motivation
- Key Signal Processing Observations
- Predicting Leaf Cache Misses
- Generating Fast Formulas
- Conclusions
Conclusions

• New method for constructing fast WHT formulas
• Generates fastest known formulas!
• Method can be trained on data for one size and perform well across many sizes
• Also, can learn to accurately predict cache misses of formulas

On going and future work:
• Test and extend to other architectures
• Extend to other transforms
Acknowledgements

SPIRAL group:

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