Evolving Feature Selectors to Inform Compiler Optimizations

CMPUT 680 Final Project Report

Chris Rayner
Dept. Computing Science
University of Alberta

Dec 11, 2009

Abstract Machine learning (ML) has the potential to help automate and improve the design of static compiler heuristics. The core challenge in applying ML is *representation*: how do you accurately summarize a program as a fixed-length feature vector? This experimental study is modeled after Leather *et al.*'s 2009 work [4]. A framework is developed to automatically evolve a collection of small programs ('feature selectors') that map arbitrary program code to real-valued vectors. These features ultimately inform loop unrolling decisions with the ambition of achieving faster compiled code.

The resulting prototype – a modification of GCC 4.3.1 – is slower at compiling but occasionally makes beneficial unrolling decisions. Performance may be held back by the limited dataset size and the simple classifier used.

1 Introduction

The goal of this project is to find *feature selectors* that map *program information* to *useful numbers*. A feature selector is defined as a small program based on a simple grammar. Program information is defined as a hierarchical representation of a compiler's static internal representation. *Useful* numbers well inform the loop unrolling decisions of a particular compiler.

Loop unrolling duplicates the code inside a loop body, which decreases the loop's tripcount and associated control flow. In general, this transformation can provide a compiler more freedom to perform scheduling, so can lead to runtime speedups. There is the obvious risk of increased code size, which can lead to instruction cache misses due to loop instructions spanning a wider memory space. A more subtle problem is that poor register allocation by the scheduler may increase register pressure [8]. Due to these competing factors, deciding whether to unroll—and by how much—is a nontrivial process, but remains a popular testbed for machine-learned compiler heuristics [4, 7].

Data from the MediaBench benchmark suite [5] is collected by examining *GCC* 4.3.1's intermediate representation. This suite is freely available and is a subset of those benchmarks used by Leather *et al.* [4], facilitating a limited comparison. MediaBench also contains small, computationally manageable benchmarks, allowing me to generate data in only a few days. A brief description of the MediaBench suite and any modifications I made to the individual applications is available in Appendix A.

2 Approach

This section describes the representation I use to represent hierarchical loop information, the grammar used to define individual feature selectors, and a short overview of the genetic algorithms framework used.

2.1 Representation

Natural loops are defined as an ergodic subgraph of the control flow graph in which the loop header dominates all other nodes. By looking at the dominator tree of this subgraph, a natural hierarchical representation emerges, which is represented using the following syntax,

$$\ell = \begin{cases} [L, \mathbf{depth}, \mathbf{numInsns}, \mathbf{avgNumInsns}, \mathbf{avgIterations}, \\ \quad [B, \mathbf{avgFrequency}, \mathbf{oncePerIteration}, [I, \mathbf{insnType}], \ldots \\ \quad\quad [I, \mathbf{insnType}], [B, \ldots]] \ldots]]. \end{cases}$$

Here a marker L marks the root node which contains static loop information. B marks a node representing a basic block. And I indicates a node representing an instruction. The variables are defined as follows:

- **depth** – the nesting depth of ℓ, where 1 means no nest

- **numInsns** – a count of the instructions contained in the loop
- **avgNumInsns** – a static compiler-generated estimate of the average number of instructions executed per loop iteration
- **avgFrequency** – a static compiler-generated estimate of the frequency with which the corresponding basic block is executed
- **oncePerIteration** – a indicator as to whether the corresponding basic block is executed once per iteration
- **insnType** – the instruction *type* in terms of the register transfer language, which discerns between jumps, barriers, unary instructions, *etc.*

For example, the following loop has been extracted from the *jpeg* benchmark, which is described in Section A.

$$\ell = \begin{cases} [L, 2, 3, 6, 9, [B, 89, 1, [I, 9], [I, 10], [I, 5], [I, 5], [I, 6], \\ \quad [B, 84, 1, [I, 9], [I, 10], [I, 5], [I, 6], \\ \qquad [B, 80, 1, [I, 10], [I, 5]]]]]. \end{cases}$$

2.2 Feature Selectors

A *feature selector* is a small program S that that maps program information to a real number, $S : \mathcal{P} \to \mathbb{R}$. Feature selectors are defined using a small language with a simple grammar (Figure 1). Definitions of the language's built-in functions are in Appendix B. The feature selector language is modeled after that used by Leather *et al.* [4][1] and is specialized for analyzing hierarchical data of the sort that represents natural loops.

Note that in Figure 1, ℓ is a hierarchical representation of program information (in this case a loop) as extracted from the compiler's intermediate representation, and $\mathbf{N} \in [0, 1)$, $\mathbf{M} \in \mathbb{Z}^+$. I motivate $\mathbf{N}$'s unconventional choice of domain by the need to have randomly generated feature selectors more often evaluable on loops than not. For example, when a feature selector

[1]No specification of the original grammar used by Leather *et al.* was available, and the implementations are slightly different. Those functions analogous to *filters* in Leather *et al.*'s study support disjunction, and the instruction type categories have finer granularity. Meanwhile, my grammar supports unions over sequences, and products over features.

feature	: count (*sequence*)	A *feature* expresses a value in $\mathbb{R}$.
	| attribute (*node*, $\mathbf{N}$)	
	| sum (*sequence*, *expr*)	
	| average (*feature, feature*)	
	| product (*feature, feature*)	
	| max (feature, feature)	
node	: child (*node*, $\mathbf{N}$)	*Nodes* are tree nodes in ℓ.
	| ℓ	
sequence	: select (*sequence*, *filter*)	A *sequence* is a set of nodes.
	| union (*sequence*, *sequence*)	
	| descendants (*node*)	
	| children (*node*)	
filter	: nodeType $(\cdot) = \mathbf{M}$	A *filter* maps nodes to $\{0, 1\}$.
	| attribute $(\cdot, \mathbf{N}) = \mathbf{M}$	
	| attribute $(\cdot, \mathbf{N}) > \mathbf{M}$	
	| attribute $(\cdot, \mathbf{N}) < \mathbf{M}$	
expr	: nodeType$(\cdot)$	An *expr* maps nodes to $\mathbb{R}$.
	| attribute $(\cdot, \mathbf{N})$	
	| 1 iff attribute $(\cdot, \mathbf{N}) = \mathbf{M}$	

Figure 1: An outline of the feature selection grammar. Nonterminals *emphasized*, numeric values in **bold**. ℓ is a hierarchical representation of program information, $\mathbf{N} \in [0, 1)$, and $\mathbf{M} \in \mathbb{Z}^+$. Function definitions in Appendix B.

refers to child(*node*,0.5), it is referring to the median child in the ordered list of *node*'s children. But if a feature selector were to make reference to the fifth node of a child, there would be many loops that this feature selector would not be evaluable on. Some simple example feature selectors[2] include:

$$S_1(\ell) = \begin{cases} \text{count} \\ \quad (\text{descendants}(\ell)) \end{cases} \qquad S_2(\ell) = \begin{cases} \text{count(select} \\ \quad (\text{descendants}(\ell), \\ \quad\quad \text{type} = B)). \end{cases}$$

Given the following example loop ℓ, which was extracted from the compiler during compilation of the *jpeg* MediaBench benchmark, we have:

[2]See Appendix D for examples of feature selectors in raw code.

$$\ell = \begin{cases} [L, 2, 3, 6, 9, [B, 89, 1, [I, 9], [I, 10], [I, 5], [I, 5], [I, 6], \\ \quad [B, 84, 1, [I, 9], [I, 10], [I, 5], [I, 6], \\ \qquad [B, 80, 1, [I, 10], [I, 5]]]]] \end{cases}$$
$$\Rightarrow S_1(\ell) = 15, \;\; S_2(\ell) = 3.$$

A feature selector is *good* if it generates numbers that help a machine learning module (in this case, k-NN) to better discern the right optimization decision. Unfortunately, the space of feature selectors is huge, and most feature selectors are useless. The question of how we can automatically create good feature selectors is examined in Section 2.3

2.3 Genetic Algorithms

How can feature selectors be automatically generated? Searching such a space is generally challenging. It is like finding the highest peaks in a landscape of spikes with a narrow view of the terrain. Fortunately, it is for this type of problem that stochastic optimization techniques are well suited. I follow Leather *et al.* in the use of genetic algorithms [4], but simulated annealing, particle swarm optimization, and other nonconventional stochastic search algorithms [6] may be potential alternatives.

Within the genetic algorithm (GA) framework, fit candidates (feature selectors) produce *offspring* (similar feature selectors) while unfit candidates are removed from consideration. An algorithmic overview of such an approach is as follows:

- *Initialize*: Begin with a random pool of individuals
- *Repeat* until termination condition:
    1. Measure each individual's *fitness*
    2. Delete unfit individuals
    3. Refill the pool with fit individuals' *offspring*

As a general summary, the GA procedure takes a random set (or pool) of feature selectors and repeatedly *(i)* filters the set for its best candidates, and *(ii)* introduces the best candidates' offspring into the pool.

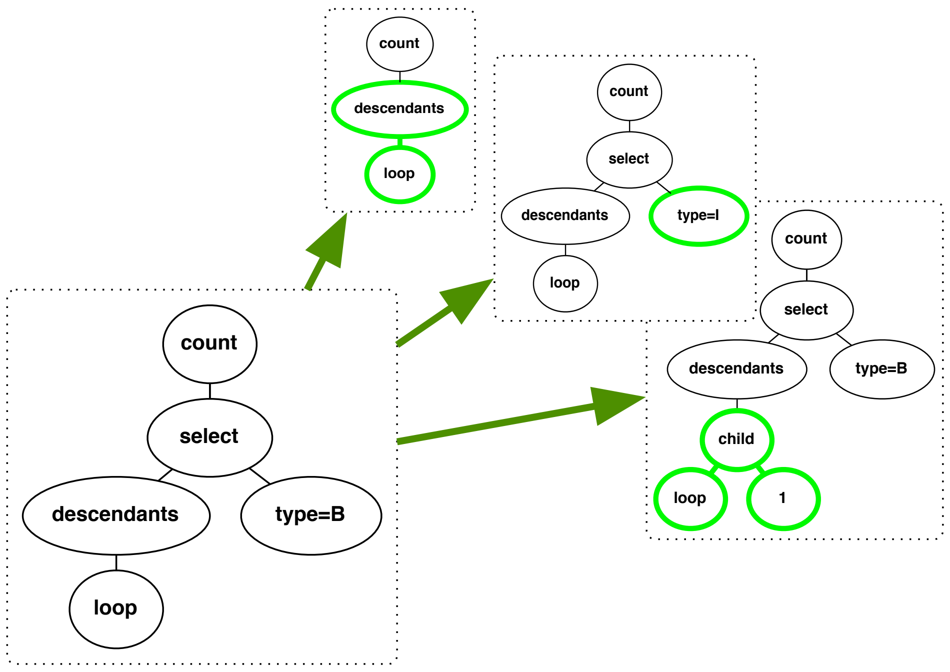

Figure 2: Examples of the *mutation* operator. The parent feature selector on the left has a random nonterminal node selected. This nonterminal is randomly re-expanded, which may lead to some of the results on the right.

Offspring: Offspring are created through *mutation* and *crossover* operations. Mutation re-expands a random nonterminal of an existing feature selector. Crossover takes a random subtree from one feature selector and replaces it with a random subtree of another feature selector. These two operations are depicted in Figure 2 and Figure 3 respectively. The question of which features to create offspring for is determined by *fitness*.

Fitness: To determine a feature selector's fitness, I provide the numbers it produces as an additional dimension of information to a machine learning module. The subsequent effect in the model's accuracy, when given a validation loop and told to classify its best unroll factor, is proportional to the feature selector's fitness. In my evaluation of feature fitness, I use a simple machine learning module. Given a validation loop whose best unroll factor is to be determined, k nearest neighbors (k-NN) [2] (with k=1) looks within its training set for the nearest matching loop in the Euclidean distance between feature vectors, and uses the optimal unroll factor of that loop as its answer.

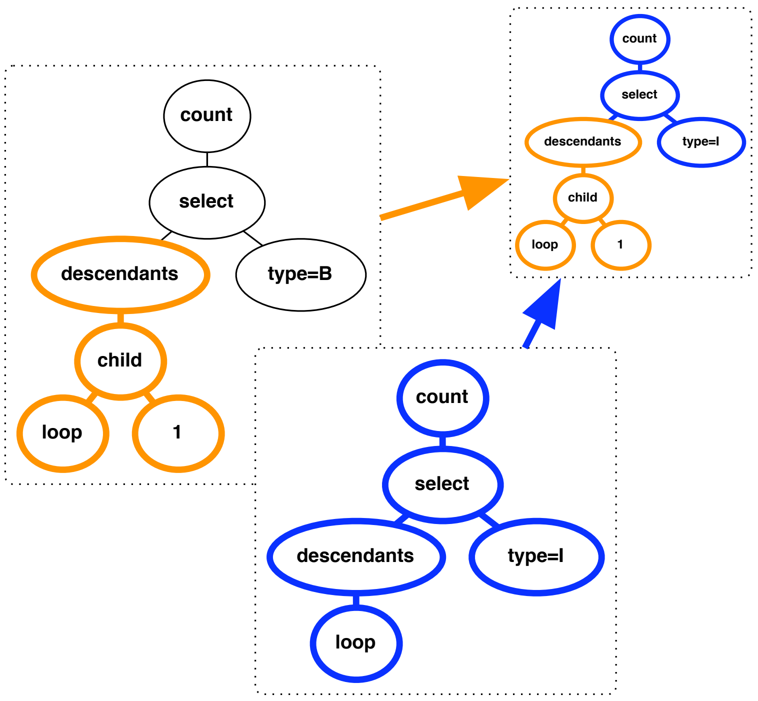

Figure 3: An example of the *crossover* operator. A subtree is randomly selected from the first parent (left). This subtree replaces a randomly selected subtree of the second parent (bottom). The only constraint is that the subtrees have as roots the same type of nonterminal.

The use of k-NN is in contrast to Leather *et al.*'s use of C-4.5 decision tree. I find k-NN to be an appropriate candidate for this study because it has no training cost, making it ideally suited to the rapid evaluation of new feature values demanded by the GA. I considered but discarded standard support vector machines (SVMs) for use in this study. SVMs can have excellent classification performance. However, they take *significant* time when used to rapidly evaluate feature values, define only binary decision boundaries (so must be used in conjunction to distinguish between multiple unroll factors), and are frequently infeasible in the quadratic programming sense when the feature selectors provide poor numbers (which is often).

3 Data Collection

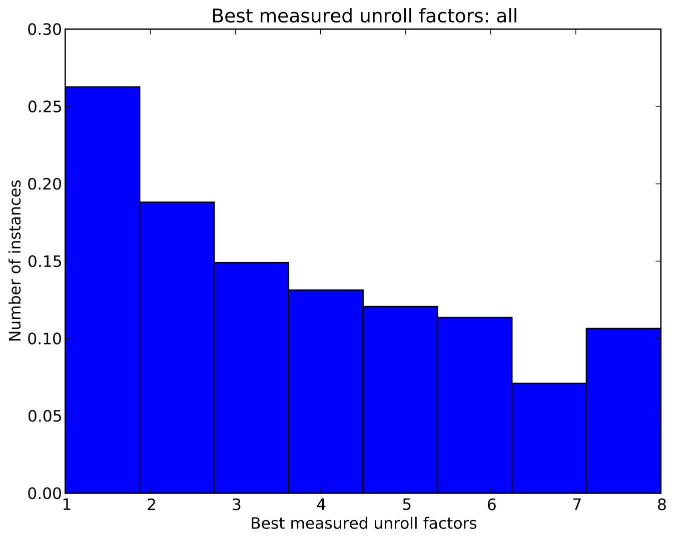

Figure 4: Proportion of best unroll factors across all benchmarks.

To learn good feature selectors, the system must know what good decisions look like. But the space of all loop unroll factors on all loops, even on *small* benchmarks, is immense. Following a sampling method from previous work [4, 7] I build an *oracle* which consists of a set of *trainable samples*. These are loops whose unroll factor, considered independent of other transformations, makes a significant difference to the benchmark's performance.

Specifically, in the *oracle* approach, individual loops are unrolled one at a time. The best associated unroll factor is then stored in the oracle, which is the eventual corpus of training data that is used to develop feature selectors.[3] The distribution of best unroll factors for the MediaBench suite is shown in Figure 4.[4] This figure reveals an expected trend: as the unroll factor increases, fewer loops benefit. Informally, it appears as though it is

[3]Using the oracle as a training target may very well lead to machine-learned heuristics that do not match the true goal of program speedup. That is, two transformations that work well independently can actually be bad in tandem. But the space must be sampled, and this approach may provide a good first pass.

[4]Distributions for individual benchmarks are shown in Appendix D.

benchmark	best	worst
adpcm	0.009	0.010
epic	0.099	0.101
ghostscript	0.274	0.282
gsm	0.054	0.058
jpeg	1.65	1.67
mesa	0.0600	0.0635
mpeg2	0.357	0.360

samples	benchmark
6	*adpcm*
33	*epic*
66	*ghostscript*
28	*gsm*
96	*jpeg*
72	*mesa*
21	*mpeg2*
322	total

Figure 5: Left, best and worst average runtimes (in seconds of user time). Right, final number of samples for each benchmark.

a good idea to unroll—but if one does not know how much to unroll, it is safest to not unroll at all. Note the 'lip' on the distribution in the right of Figure 4. I believe this may be because the rightmost bucket has captured loops that would have benefited from an even greater unroll factor than 8.

Thresholding: An unroll factor is deemed better than all others if its selection makes a significant difference (in terms of benchmark runtime) over the other unroll factors for the same loop. Data is collected as follows:

1. For each of 10 innermost loops ℓ and each unroll factor $n \in [1, 8]$
    (a) Recompile benchmark with ℓ unrolled n times
    (b) $t \leftarrow$ average (5 runs) benchmark runtime
    (c) Add $\langle n, t \rangle_\ell$ to temporary database
2. For each loop ℓ, save the entry $\langle n_{\text{best}}, t_{\text{best}} \rangle_\ell$ if t_{best} is B% better than t_{worst}, i.e.,

$$\frac{t_{\text{worst}} - t_{\text{best}}}{t_{\text{worst}}} \geq \frac{B}{100}$$

Note the parameter B used by this algorithm limits the trainable data to only those loops whose unroll factor makes a percentage runtime difference of greater than B%. Unfortunately, the choice of B creates a dilemma. Filtering trainable instances for larger (more significant) runtime differences naturally results in fewer trainable instances.[5] To address this, I hand-selected the largest value of B that I believed provided a reasonable number of trainable samples to the oracle (close to 100 per benchmark if possible). The number of trainable samples resulting from my selection of B, as well as the best runtimes and worst runtimes for each benchmark, are tallied in Figure 5.

4 Experiments

This section describes experimental parameters, results on classification accuracy and runtime performance changes, and ends with comment on the novelty of the generated features.

4.1 Parameters

My results were developed on a dual-core Intel Pentium 4 with CPUs clocked at 3.40GHz, running Linux kernel version 2.6.18-53.1.14.

Compilation was performed using GCC 4.3.1. This is the same compiler used in Leather *et al.*'s study [4]. Each benchmark[6] was compiled only with the flag -O1, GCC's first level of optimization. This optimization level performs some relatively safe optimizations, some of which may have tangible effects on loop unrolling. For instance, it takes constant expressions out of loops, simplifies the branching conditions, and strength reduces some costly operations in expressions. When known to be supported by the architecture, some attempts are made to reschedule instructions after branches. Heuristics based on static aspects of the control flow graph are used to predict branching probabilities. Attempts are made to transform conditional jumps into equivalent code without control (using predicate registers when supported by the architecture, e.g., IA-64). No instruction scheduling is done, which may reduce potential gains from loop unrolling.

For each benchmark, one is designated as the *target* benchmark (i.e., its loops become our testing examples) and the rest are used to generate *training* data for the genetic algorithm (i.e., their loops become our training examples). The genetic algorithm itself evolves relatively small pools of size 25, and runs until at most four features are generated. A maximum of

[5]The interested reader can see plots of how the number of trainable instances tapers off across the various benchmarks in Appendix D.

[6]Also see Appendix A for descriptions of and modifications to benchmarks.

	accuracy	**runtimes**		
		learned	*no unroll*	*GCC*
adpcm	2/6 (**33.3**%)	0.0108	0.0100	0.0098
epic	7/33 (21.2%)	**0.1024**	0.1016	**0.1093**
ghostscript	11/66 (16.7%)	**0.2816**	**0.2818**	0.2798
gsm	8/28 (28.6%)	**0.0578**	**0.0596**	0.0552
jpeg	15/96 (15.6%)	1.699	1.668	1.609
mesa	8/72 (**11.1**%)	0.0659	0.0631	0.0655
mpeg2	3/21 (14.2%)	0.3807	0.3707	0.3480

Figure 6: Left, classifier accuracy on unroll factor selection, with best and worst accuracies **bold**. Right, runtimes after compiling with the learned heuristic, no unrolling, and GCC's hardcoded heuristic. Runtime entries **bold** when learned heuristic beats one of the two competing columns.

eight generations per epoch is allowed, or until five generations yield no improvement.[7] On each new epoch, 100 training examples and 100 validation examples, taken randomly from all benchmarks except the target benchmark, are supplied to the GA to determine feature fitness via k-NN, per Section 2.3.

4.2 Results

The left of Figure 6 depicts the classifier's accuracy in terms of its ability to predict the same unroll factor as contained in the oracle. In general these results are only slightly better than choosing unroll factors uniformly at random (which would be correct approximately 12.5% of the time). However, even though the system is not able to determine the single best unroll factor every time, perhaps it is still choosing beneficial unroll factors.

To investigate, I usurp GCC's unroll heuristic and have my classifier choose the unroll factor. It is important to note that, for each target benchmark, the classifier was never exposed to any loops in that benchmark. To determine if my classifier has a beneficial effect I compare *(i)* my system, *(ii)* GCC with no unrolling, and *(iii)* GCC with unrolling performed by the expertly defined heuristics, and tally the results in Figure 6, right. These

[7]These numbers are significantly smaller than the numbers Leather *et al.* use in their study. By contrast, Leather *et al.*'s pool sizes are 100; an arbitrary number of features generated; and a maximum of 200 generations per epoch are allowed as long as 15 consecutive generations yield no improvement.

results provide evidence of the difficulty inherent in designing good loop unrolling heuristics. I have presented the results for three radically different approaches to the problem of determining unroll factors (i.e., learning from example data, not unrolling at all, and expert-designed thresholds). And while GCC's hard-coded heuristics generally seem to achieve better performance, none of the three approaches is the clear winner.

A final caveat: in introducing the new heuristic I ran into a problem. Loops with as many as 80 nodes were being unrolled 8 times, which across hundreds of loops is so extreme as to make compilation intractable. To mitigate this, I had to add an additional heuristic to restrict my classifier from being activated only on the 10 innermost loops (since these were the loops that training data was generated on), and for loops consisting of fewer than 25 nodes in the control flow graph—otherwise the unroll factor is 1.

4.3 Novelty of Generated Features

In the study on which these experiments are based Leather *et al.* state that, "the [feature selectors] are [not] obvious and are unlikely to be picked by a compiler writer, demonstrating the strength of our approach" [4]. Is this statement true of my own generated features? A principled answer to this question can make use of a statistical comparison of the *generated* feature values to *static* feature values that are available during GCC 4.3.1's unrolling decisions. These static features are all based on static analysis:

- the loop's nesting depth
- a count of the instructions in the loop
- an expected number of iterations
- an expected number of instructions to be executed in one iteration

I define a vector of generated features f to be *novel* if any linear combination of f is difficult to capture as any linear combination of these static features.

I use a technique called canonical correlation analysis (CCA) to determine novelty: suppose an arbitrary linear combination of all *generated* feature values is taken. Next an arbitrary linear combination of the aforementioned *static* features is taken. The *correlation* ρ between the scalar results of these operations measures linear dependence on one upon the other. CCA can be

benchmark	correlation factors
ghostscript	1.0, 1.0, 1.0, 0.67
mesa	1.0, 1.0, 0.18, 0.15
gsm	1.0, 0.99, 0.72, 0.10
adpcm	1.0, 0.99, 0.70, 0.11
jpeg	1.0, 0.89, 0.41, 0.06
mpeg2	1.0, 0.85, 0.22, 0.05
epic	1.0, 0.99, 0.22

Table 1: Measures of correlation between generated features and static features along orthogonal components of feature space. Maximum possible correlation 1.0 indicates a linear combination of the generated features is exactly proportional to some linear combination of the static features, demonstrating a *lack* of novelty in the generated features.

used to extract the *maximum* such correlation possible over all such linear combinations.[8] De Bie *et al.* provide a good overview of CCA [1].

The results of performing CCA on the generated features against the static figures is tallied in Table 1. Note the *epic* benchmark has only three components with nonzero correlation. This is because the feature selectors for the *epic* benchmark, despite being four-dimensional, only have three intrinsic degrees of freedom (one dimension of the generated features, appropriately scaled, is always equal to another dimension).[9]

As for the novelty, all benchmarks show at least two strong (nearly 1) components of correlation. This means that at least two degrees of freedom in the generated features are strongly correlated with two degrees of freedom in the static features, suggesting to a lack of novelty. However, some benchmarks (*mpeg2*, *epic*, *jpeg*, *mesa*) show low correlation in the remaining two degrees of freedom, which lends weight to the statement that the features are providing novel information to the classifier.

I stress that my feature selectors were created using prototypical framework. As described in Section 5, my system likely has not explored the space of feature generators as thoroughly as Leather *et al.*'s system.

[8]CCA also provides the particular coefficients corresponding to each linear operation, but for brevity I do not report those numbers here.

[9]This points to a general weakness in the design of the system. Dimensionality reduction (e.g., principal components analysis) should be employed to filter out highly redundant feature selectors.

5 Analysis and Conclusion

In the course of this project I encountered concepts and ideas that I believe any new practitioner should be mindful of.

My results are not as good as Leather *et al.*. My GAs are more tamely parametrized as described in Section 4.1, and I have significantly less training data in each instance. But it is possibly my use of only one benchmark suite that hurt me the most. The MediaBench suite appears specifically designed so that *each benchmark is different*. This occurred to me after looking at the distribution of loops across best unroll factors for different benchmarks (these are shown in Appendix D). That is, any two benchmarks in this suite were likely to have been selected so as to have little in common. My application of ‘leave-one-benchmark-out’ cross-validation then may be inherently flawed. Specifically, I may be developing feature selectors that are good on benchmarks that are purposefully unalike to the target benchmark!

Developing the system naturally involved multiple debugging runs. Although I do not report on it in detail, I noticed quite high variance in the accuracy (as depicted in Figure 6) induced by the feature selectors. That is to say, if I had taken more time to train, or re-run the system from the beginning a few more times, or used larger feature selector pools, I may could have generated significantly more accurate feature selectors.

Without explicitly being stopped after creating four features (as specified in Section 4.1) there is the potential for the system to develop an arbitrarily large number of feature selectors. But at some point, each additional feature will simply serve to bias the model towards the current training set. Such feature selectors lack generality and will generally be useless. How can we detect or mitigate this? One answer lies in the many dimensionality reduction techniques that I left unexplored, which I briefly make mention of in my analysis of feature novelty in Section 4.3.

Whether the *oracle* approach to providing trainable samples helps or hurts is conflated by my classifier’s relatively weak accuracy (Section 4.2). A simple direction to explore unrolling each *all* loops on a given benchmark to the amount specified by the oracle for that benchmark and to measure the effects.

The benchmarks take significantly longer to compile with augmented GCC because of the communication overhead to interpreted Python code. Python made prototyping a GA framework to evolve small programs easy, but it was slow because it had to be invoked outside the compiler. It would be much faster to integrate the learner directly into the compiler source.

6 Acknowledgments

Thanks: Ricardo Sanchez, even while under NDA and unable to look at GCC 4.3.1 source code, provided me valuable advice on getting this beast compiled. Paul Berube answered general questions and advised me on benchmark timing issues. Thanks also to the instructor, Nelson Amaral, and the students of CMPUT 680, for the valuable classtime and conversations.

References

[1] Tijl De Bie, Nello Cristianini, and Roman Rosipal. Eigenproblems in Pattern Recognition. *Handbook of Geometric Computing: Applications in Pattern Recognition, Computer Vision, Neuralcomputing, and Robotics*, August 2005.

[2] Belur V. Dasarathy (ed). *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques*. 1991.

[3] GNU. *The GNU Compiler Collection (GCC)*. http://gcc.gnu.org, 2009.

[4] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. Automatic Feature Generation for Machine Learning Based Optimizing Compilation. *Code Generation and Optimization (GCO)*, pages 81–91, 2009.

[5] Chunho Lee, M. Potkonjak, and W.H. Mangione-Smith. "mediabench: a tool for evaluating and synthesizing multimedia and communications systems". *Microarchitecture, IEEE/ACM International Symposium on*, 0:330, 1997.

[6] Shaul Markovitch and Dan Rosenstein. Feature Generation Using General Constructor Functions. *Journal of Machine Learning*, 49:1:59–98, October 2002.

[7] Mark Stephenson and Saman Amarasinghe. Predicting Unroll Factors Using Supervised Classification. *Code Generation and Optimization (CGO)*, pages 123–134, March 2005.

[8] Peng Zhao and Jose Nelson Amaral. To Inline or Not to Inline? Enhanced Inlining Decisions. *Workshop on Languages and Compilers and Parallel Computing (LCPC)*, October 2003.

A Benchmarks

This section describes the individual MediaBench benchmarks, and any modifications I made to code, input files, or timing procedures. Note that, since these benchmarks are single-threaded, involve no device management, and computationally bound (especially between the reading and writing of input and output files), and since I anticipate loop unrolling affects only the computational effort required of a given benchmark, I measured user time.

- *adpcm* – compresses/decompresses speech audio files. I average over ten runs of compression and decompression.
- *epic* – compresses/decompresses grayscale images. I substituted the small benchmark image (roughly 65kb) with a larger image (roughly 200k) and averaged over ten runs.
- *ghostscript* – converts between vector graphic files and uncompressed images. I had to modify three trivial functions involving date and time. Timings averaged over five runs of vector-to-bitmap conversion.
- *gsm* – compresses/decompresses audio files. Timings averaged over encoding/decoding five times over.
- *jpeg* – performs jpeg encoding/decoding. I replaced the input file for encoding with a 36 MB image file and the input file for decoding with a 900k jpeg file. Timings were averaged over three runs of encoding (raw ppm to jpeg) and decoding (jpeg to raw ppm).
- *mesa* – is a 3D graphics library packaged with three graphical demos. Timings averaged over eight executions of each demo.
- *mpeg2* – performs encoding/decoding of video files. Timings averaged over three runs each of encoding and decoding.

These MediaBench applications were omitted for the following reasons:

- *g721* – compresses/decompresses audio. Omitted for having only *two* unrollable loops as determined by GCC 4.3.1.
- *pegwit* – encryption. Unable to compile.

- *pgp* – encryption. Compiled application does not read signature files, possibly due to hardcoded endianness assumptions.
- *sphere* – unavailable in the downloadable MediaBench package.
- *rasta* – filters noise/distortion from audio samples. Unable to compile.

B Feature Selector Function Definitions

This section describes some of the predefined functions that are a part of the feature selector language described in Section 2.2.

- count (s) returns a count of the size of sequence s.
- attribute (n,a) returns the attribute a of node n.
- sum (s,e) evaluates e on nodes in s and returns a sum.
- average (f,g) returns the average of values f and g.
- product (f,g) returns the product of values f and g.
- max (f,g) returns the max of values f and g.
- child (n,f) if node n has a child f, return that child, else 0.
- select (s,f) returns nodes s for which the filter f returns 1.
- union (s,r) returns the union of sequences s and r.
- descendants (n) returns *all* descendants of node n.
- children (n) returns node n's *immediate* descendants.
- nodeType (n) returns a value indicating node n's type. This can be one of *loop*, *basic block*, or *instruction*.

C Function Inlining

My original project proposal aimed to explore function inlining decisions, rather than loop unrolling decisions. However, I encountered insurmountable technical problems in overriding the GCC 4.3.1 inlining code.

In brief, neither dynamic memory allocation nor reading from a file appears to be safe, whether via stdio's *getline* or *fgets* (these were necessary in my framework to interface with Python). The problem persists even after clean, bootstrapped compilations of GCC, and can be traced down to a single line of code. Beyond this, all the code for data generation, feature selector evolution, *etc.* are in place, since they are nearly identical to those used for loop unrolling, so naturally I am disappointed.

Contrary to loops, the caller/callee relationship in the call graph is not inherently hierarchical. However, my plan was to modify GCC to dump a subtree of the callgraph of the following form,

$$\mathcal{S} = \begin{cases} [S, \mathbf{times}, \mathbf{numInsns}, \mathbf{growthEstimate}, \mathbf{inlinedStack}, \\ \quad [\mathrm{T}, \mathbf{numInsns}], \ldots, [\mathrm{U}, \mathbf{numInsns}], \ldots] \end{cases}$$

where a marker S indicates the root node which contains static information about the call site, T marks a node representing a callee that is being considered for inlining, and U marks a node representing any callee of the calling function that is *not* currently being considered for inlining.

Such a hierarchical representation is motivated by the fact that it enables inlining information to be considered in the same way that Leather *et al* [4] consider loops. Information about callees may provide some static indication about how integral a particular function is to the execution of a particular application. The following variables are defined:

- **times** – the number of calls from the caller to the callee that are being considered for inlining
- **numInsns** – the number of instructions contained in a caller or callee
- **growthEstimate** – a static compiler-generated estimate of the growth of the caller if inlining is performed
- **inlinedStack** – a static compiler-generated estimate of the amount of the size of the inlined callee's stack

Even without results I can speculate as to why this approach might fail for function inlining. First, this approach is based on static analysis. Some of the most helpful heuristics for function inlining make significant use of dynamic profiling [8]. Second, the assumption of a hierarchical structure is natural for loops, but strikes me as less so for call sites. The language upon which feature selectors are based is specifically designed to handle hierarchical structures. An alternative language not so much geared towards hierarchical structure may provide a better basis for evolving feature selectors that help inform inlining decisions. And, finally, whether to inline one call-site is a very different decision than whether to inline two call sites—I would expect this to be a significant consideration even moreso than it is for loops.

D Supplementary Data

Some raw code for feature selectors generated by the system include:

```
prod([nodeAttr([child([loopDump(), 0.4416]), 0.4986]),
     nodeAttr([loopDump(), 0.3441])])

 avg([nodeAttr([loopDump(), 0.6549]),
     count([descendants([loopDump()])])])

nodeAttr([child([
            child([loopDump(), 0.5829]), 0.1531]),
          0.5990])
```

The remainder of this section provides plots on *(i)* the distribution of best loop unroll factors across all benchmarks (depicted in Figure 7), and *(ii)* how the number of *trainable samples* (loops in the oracle) drops off across all benchmarks as I threshold at greater levels of discrepancy between worst and best run times (depicted in Figure 8).

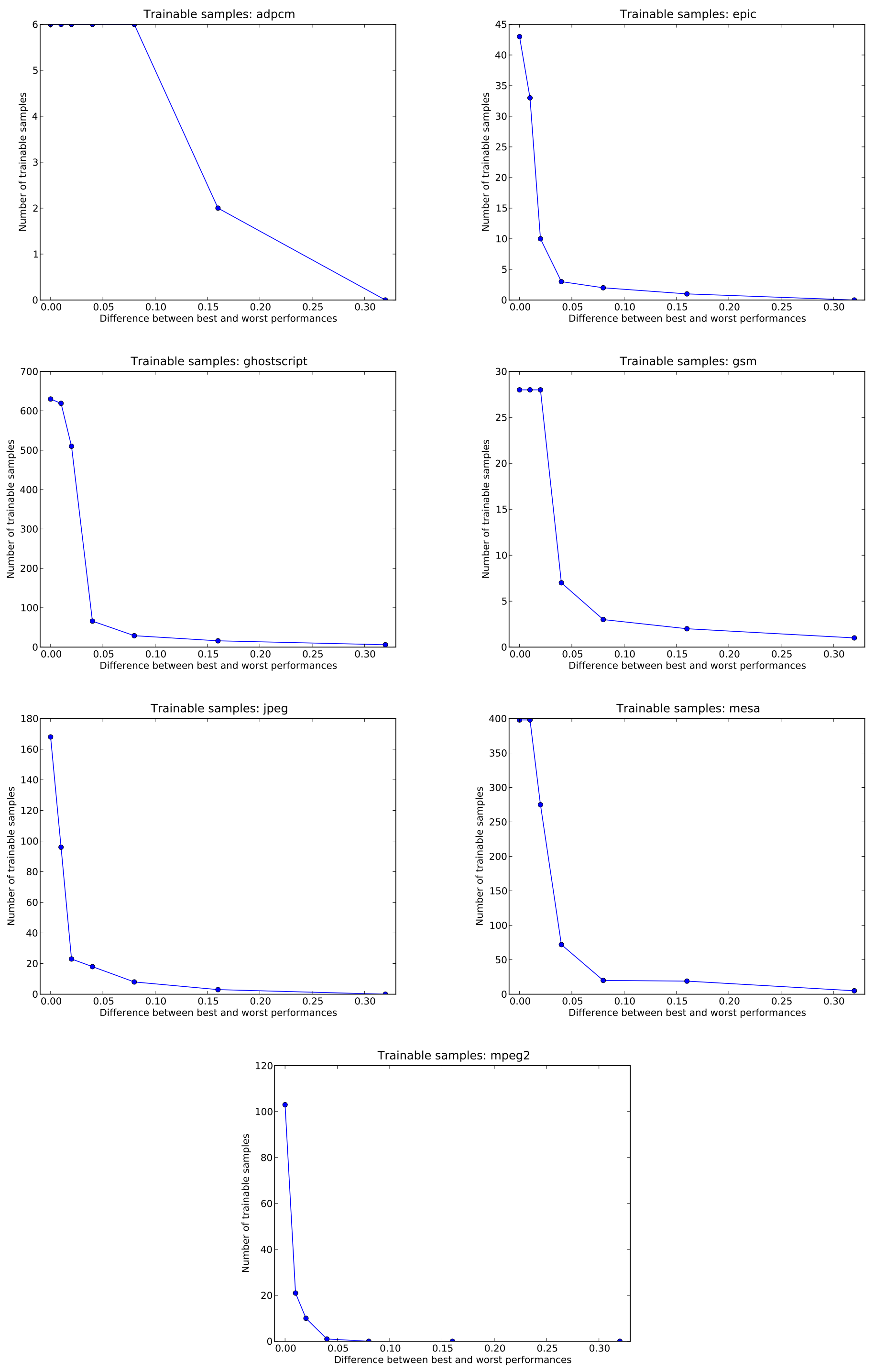

Figure 7: Dropoff in trainable samples with an increasing threshold.

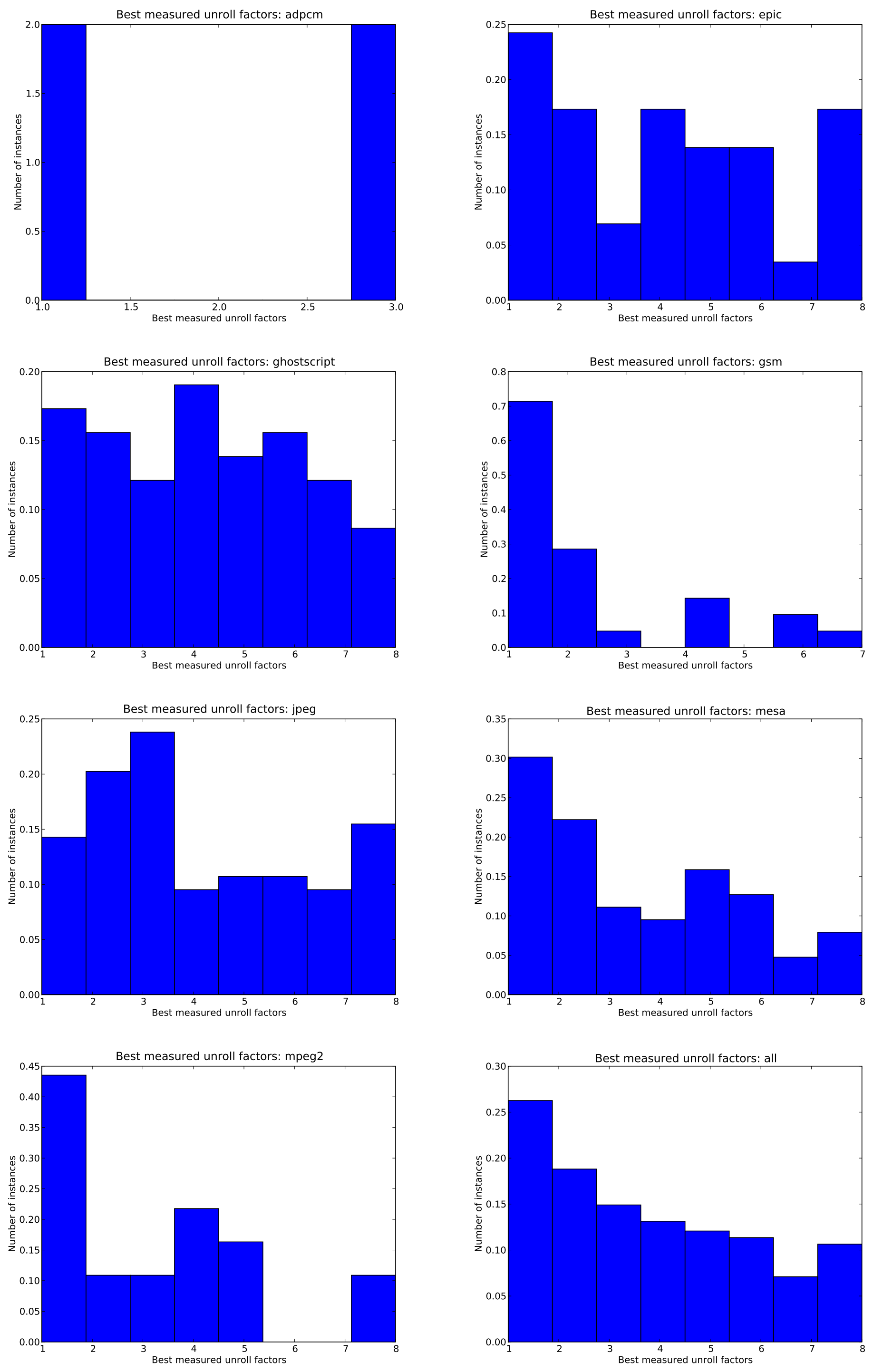

Figure 8: Distribution across all benchmarks' best unroll factors.