Machine Learning based Compilation

Michael O'Boyle

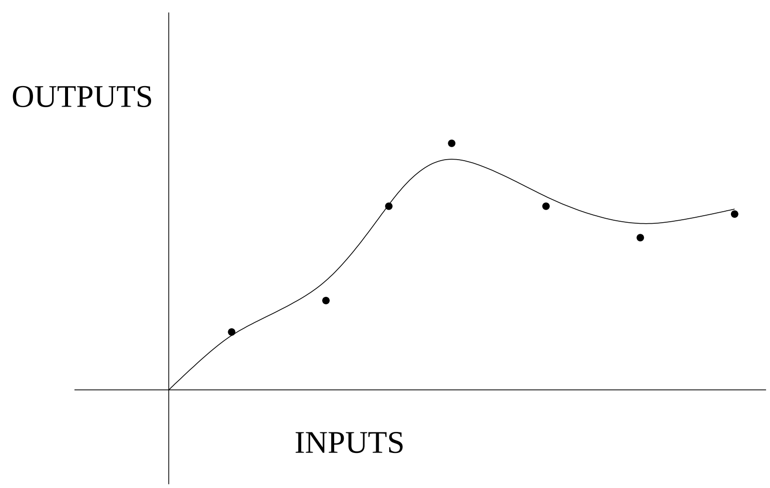
March, 2014

Overview

- Machine learning - what is it and why is it useful?
- Predictive modelling
- OSE
- Scheduling and low level optimisation
- Loop unrolling
- Limits and other uses of machine learning
- Future work and summary

Machine Learning as a solution

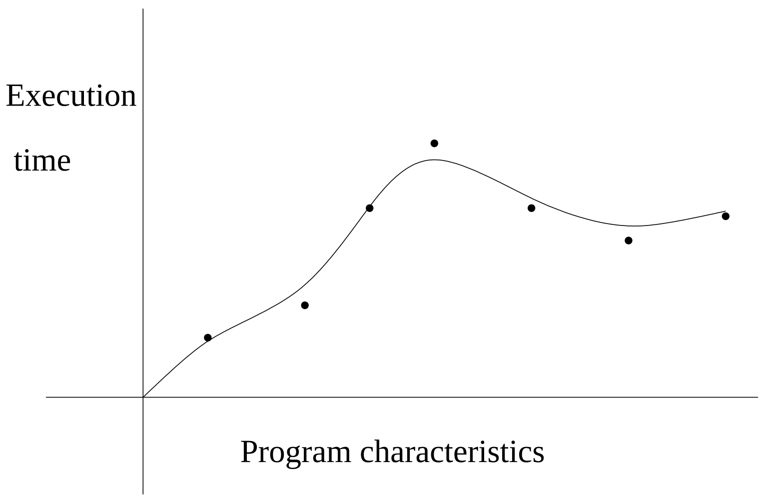
- Well established area of AI, neural networks, genetic algorithms etc. but what has AI got to do with compilation?
- In a very simplistic sense machine learning can be considered as sophisticated form of curve fitting.

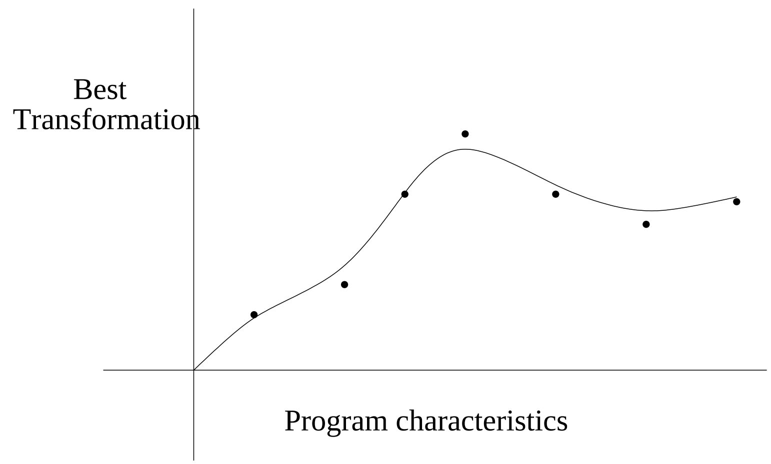


Machine Learning

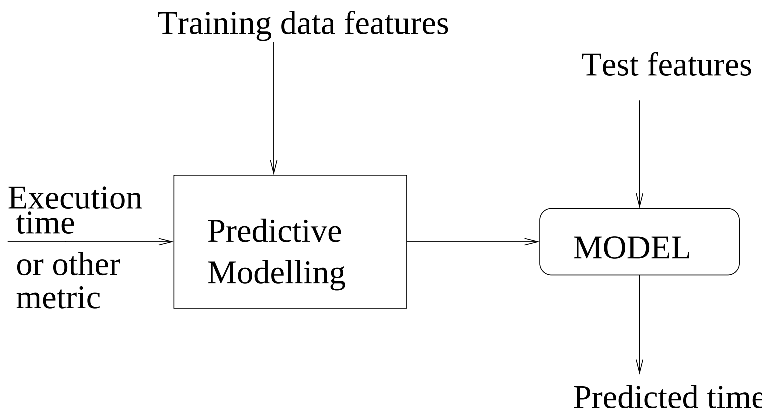
- The inputs are characteristics of the program and processor. Outputs, the optimisation function we are interested in, execution time power or code size

- Theoretically predict future behaviour and find the best optimisation





Predictive Modelling



- Predictive modelling techniques all have the property that they try to learn a model that describes the correlation between inputs and outputs
- This can be a classification or a function or Bayesian probability distribution
- Distinct training and test data. Compiler writers don't make this distinction!

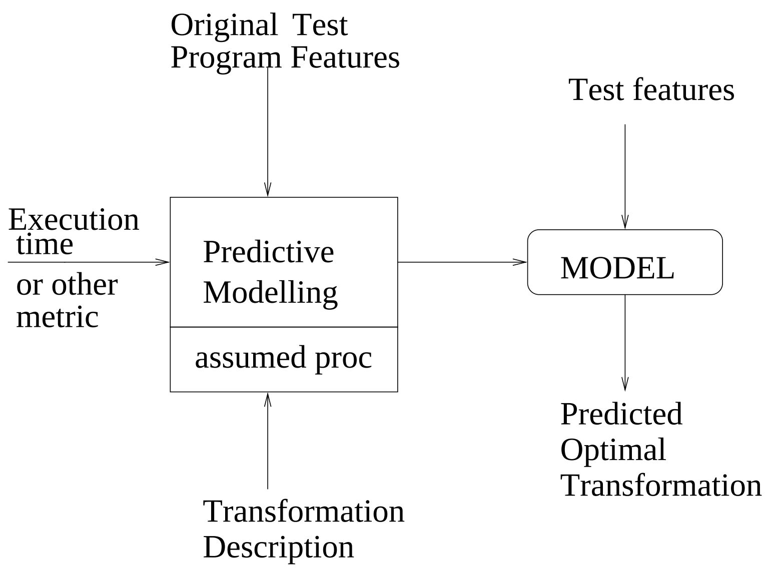
Training data

- Crucial to this working is correct selection of training data.
- The data has to be rich enough to cover the space of programs likely to be be encountered.
- If we wish to learn over different processors so that the system can port then we also need sufficient coverage here too
- In practice it is very difficult to formally state the space of possibly interesting programs
- Ideas include typical kernels and compositions of them. Hierarchical benchmark suites could help here

Feature selection of programs

- The real crux problem with machine learning is feature selection What features of a program are likely to predict it's eventual behaviour?
- In a sense, features should be a compact representation of a program that capture the essential performance related aspects and ignore the irrelevant
- Clearly, the number of vowels in the program is unlikely to be significant nor the user comments
- Compiler IRs are a good starting point as they are condensed reps.
- Loop nest depth, control-flow graph structure, recursion, pointer based accesses, data structure

Case studies



- All of the techniques have the above characterisation
- In fact it is often easier to select a good transformation rather than determine execution time. Relative vs absolute reasoning

Compiler Optimization-Space Exploration" paper by Triantafyllis et al. (CGO 2003)

- Find configurations that give good avg. performance across all programs.
- Group programs according to their performance on these configurations.
- Gradually find more specialized configurations by only considering subsets of programs.
- Idea: Pruning the search space by only considering optimisations that worked well on "similar" programs.
- Ose search tree embeds prior knowledge. Expect you to read, understand and know this paper.

Building the OSE search tree

- Arrange the best optimisation configurations C in a tree.

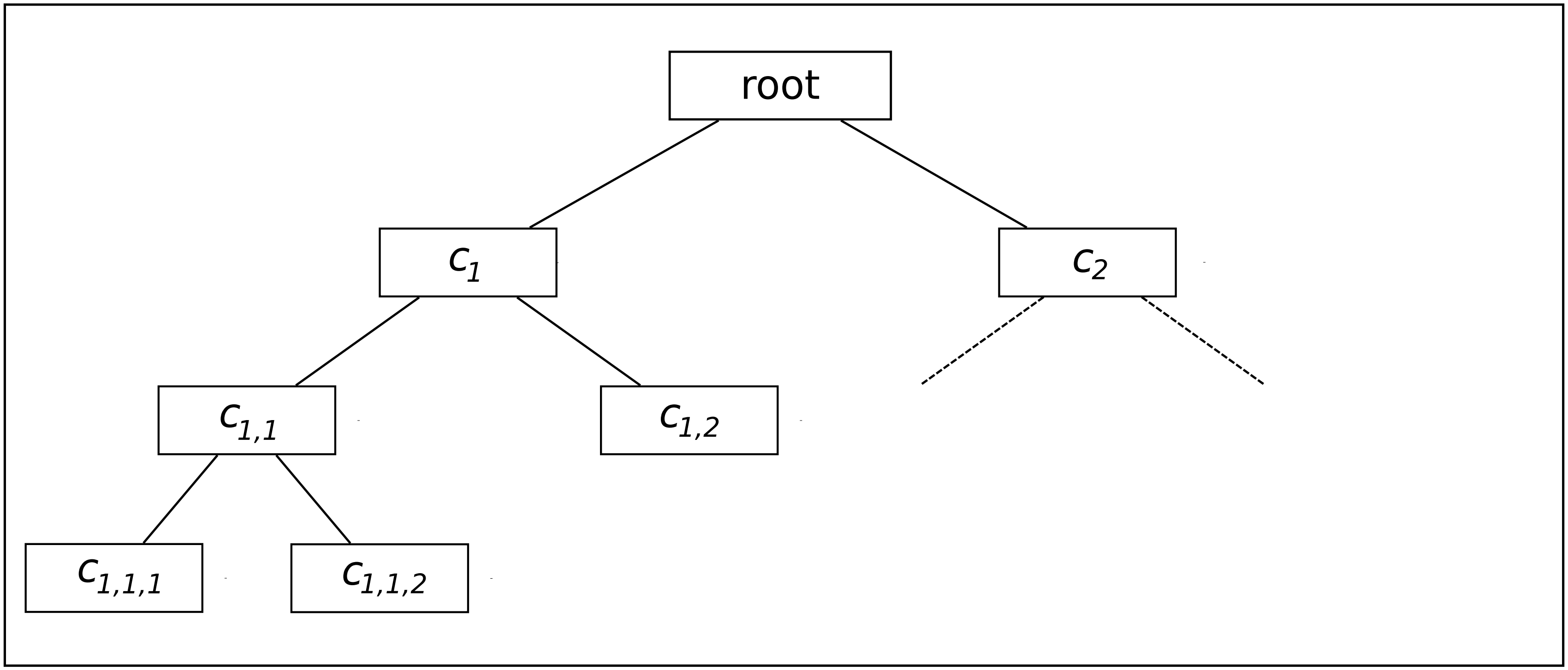
- Algorithm

Step 1: Initially, the set of programs $Q = P$
Step 2: Find configurations $c_0, c_1 \in C$ that give the best performance across Q
Step 3: Create $Q_0, Q_1 \subseteq Q$ such that $\forall p \in Q_i : perf(p, c_i) \geq perf(p, c_{1-i})$
In other words: assign each benchmark to one of two sets depending on which configuration gives the best performance.
Step 4: Start again at step 2 with $Q = Q_i$, if Q_i is not empty. Remove c_o, c_1

- Max recursion depth: 3. Note remove best avg so far

- Paper has 3 nodes per level c_o, c_1, c_2. We restrict to 2.

Constructing the Tree - An Example



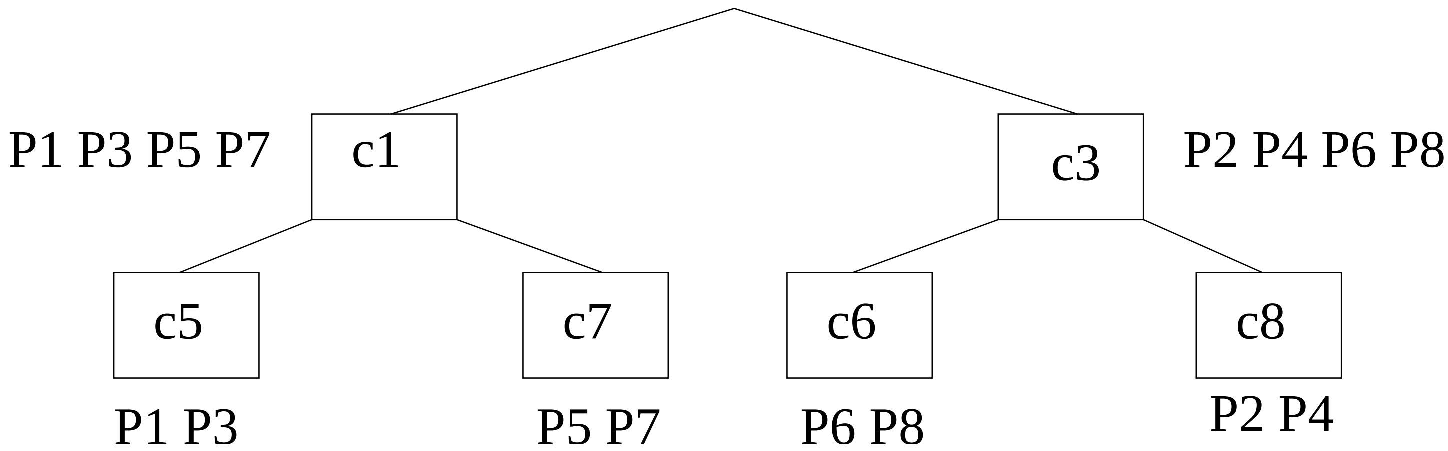
Constructing the Tree

	c1	c2	c3	c4	c5	c6	c7	c8
P1	2	5	0.9	0.1	4	1.4	3	0.25
P2	1.1	0.1	3.8	5	1.1	2	0.5	3
P3	4	0.1	1.1	0.1	2	1.4	1	0.25
P4	0.9	0.1	1.8	0.1	1.1	3	0.4	4
P5	2	0.1	0.9	5	2	1.4	4	0.25
P6	1.1	0.1	3.8	0.1	1.1	3	0.5	1
P7	4	0.1	1.1	0.1	2	1.4	3	0.25
P8	0.9	5	1.8	0.1	1.1	4	0.4	3
Avg	2.0	1.32	1.9	1.32	1.8	1.7	1.6	1.5

Configurations c1 and c3 give best avg speedup

Use them at start of tree.

Constructing the Tree - An Example



c1 and c3 are best on average.

For programs P1,3,5 and 7: configurations c5 and c7 give next best avg performance

For programs P2,4,6 and 8 : configurations c6 and c8 give next best avg performance

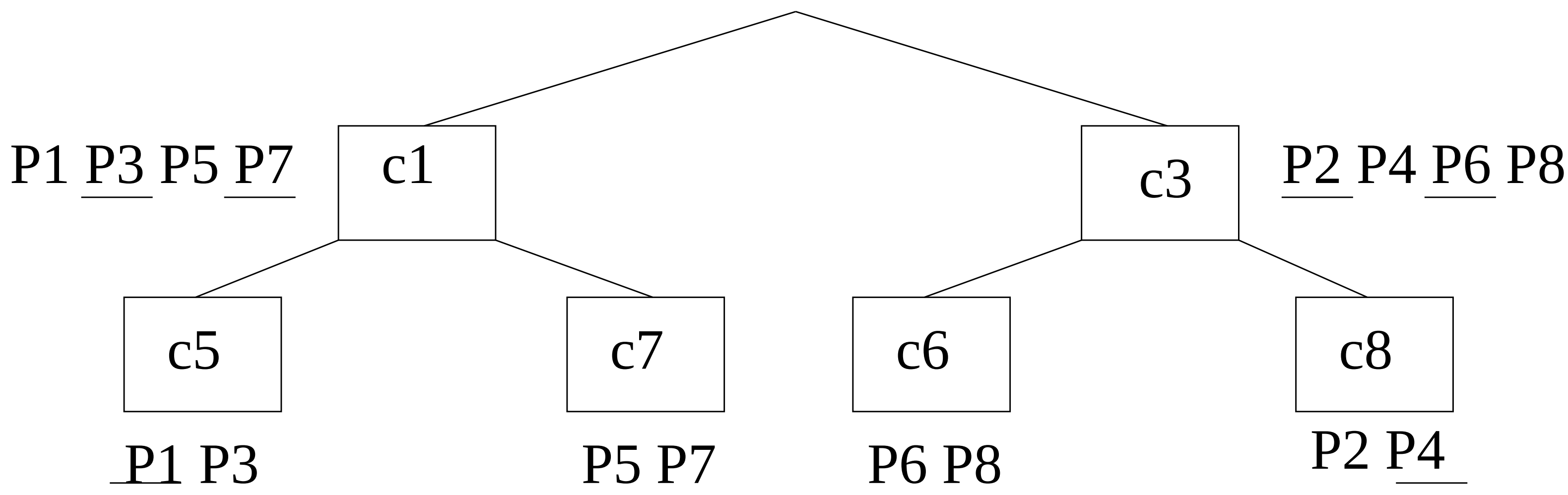
Optimizing a New Program

To quickly find a good configuration for a new program:

- Start at the root node and compare the performance of the program with the two configurations found in its child nodes.
- Move to the node with the configuration that gives a better speedup.
- Repeat these steps until you've reached a leaf node.
- Pick the configuration on the path from the root to the leaf node that gave the best performance.

Traversing the Tree

Apply to same programs for illustration



Underline denotes best

Results on applying search tree

Prog	Configs	Performance
P1	c1,c5	4
P2	c3, c8	3.8
P3	c1, c5	4
P4	c3 c8	4
P5	c1 c7	4
P6	c3 c6	3.8
P7	c1 c7	4
P8	c3 c6	4

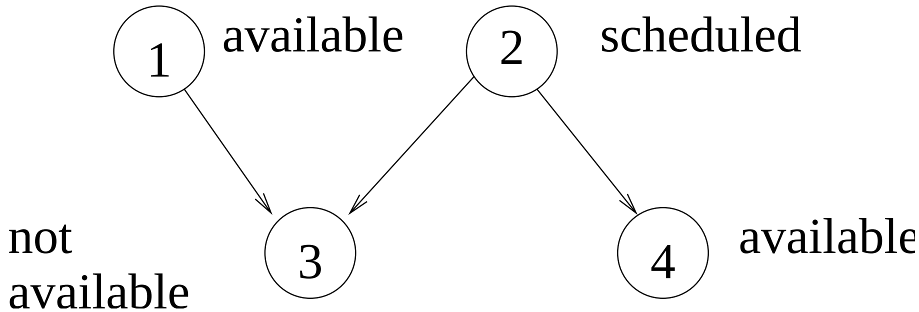
If we apply the tree to the same programs get an improvement.

Should not evaluate on training data though!

OSE uses performance models to speed up search

Learning to schedule Moss, ..,Cavazos et al

Given partial schedule 2, which instruction to schedule next 1 or 4?



- One of the first papers to investigate machine learning for compiler optimisation
- Appeared at NIPS '97 - not picked up by compiler community till later.

Learning to schedule

- The approach taken is to look at many (small to medium) basic blocks and to exhaustively determine all possible schedules.

- Next go through each block and given a (potentially empty) partial schedule and the choice of two or more instructions that may be scheduled next, select each in turn and determine which is best.

- If there is a difference, record the input tuple (P, I_i, I_j) where P is a partial schedule, I_i is the instruction that should be scheduled earlier than I_j. Record TRUE as the output. Record FALSE with (P, I_j, I_i)

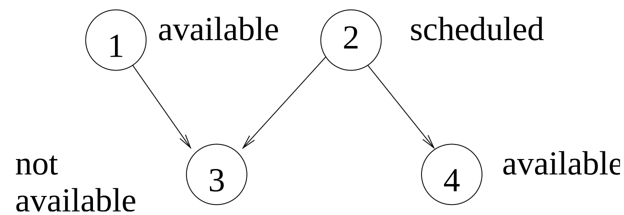
- For each variable size tuple record a fixed length vector summary based on features.

Learning to schedule

Feature selection can be a black art. Here dual issue of alpha biases choice.

- Odd Partial (odd): odd or even length schedule
- Instruction Class (ic): which class corresponds to function unit
- weighted critical path (wcp): length of dependent instructions
- Actual Dual (d): can this instruction dual issue with previous
- maxdelay (e): earliest cycle this instruction can go

Feature extraction

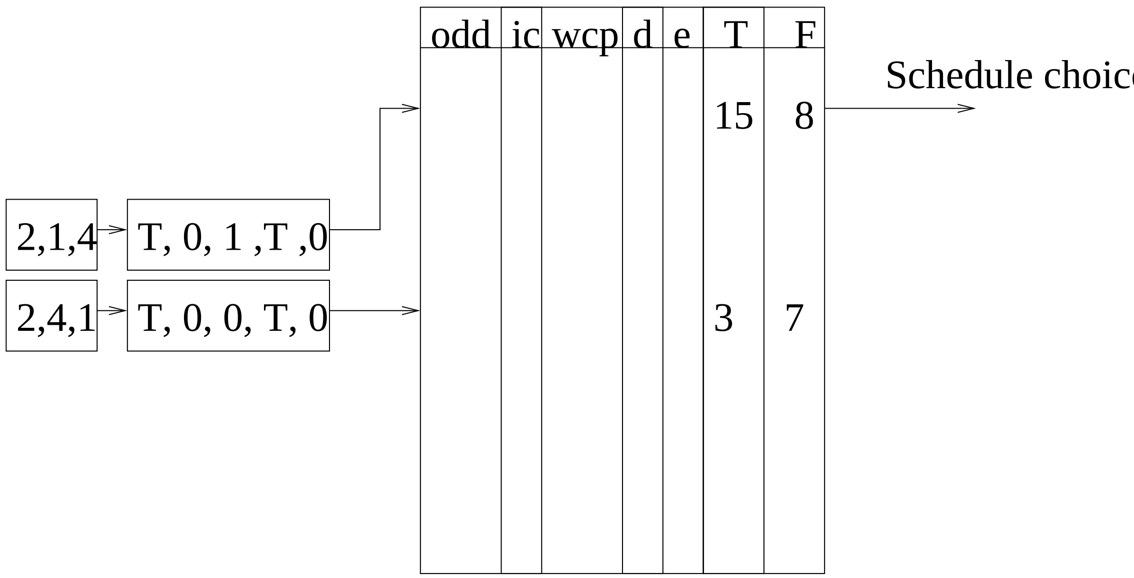


Tuple $(\{2\}, 1, 4)$: [odd:T, ic:0, wcp:1, d:T, e:0]: TRUE,

Tuple $(\{2\}, 4, 1)$: [odd:T, ic:0, wcp:0, d:T, e:0]: FALSE

- Given these tuples apply different learning techniques on data to derive a model
- Use model to select scheduling for test problems. One of the easiest is table lookup/nearest neighbour
- Others used include neural net with hidden layer, induction rule and decision tree

Example - table lookup



- The first schedule is selected as previous training has shown that it is better
- If feature vector not stored, then find nearest example. Very similar to instance-based learning

Induction heuristics

$e = second$
$e = same \wedge wcp = first$
$e = same \wedge wcp = same \wedge d = first \wedge ico = load$
$e = same \wedge wcp = same \wedge d = first \wedge ico = store$
$e = same \wedge wcp = same \wedge d = first \wedge ico = ilogical$
$e = same \wedge wcp = same \wedge d = first \wedge ico = fpop$
$e = same \wedge wcp = same \wedge d = first \wedge ico = iarith \wedge ic1 = load$ ...

- Schedule the first I_i if the max time of the second is greater

- If the same, schedule the one with the greatest number of critical dependent instruction ...

Results

- Basically all techniques were very good compared to the native scheduler Approximately 98% of the performance of the hand-tuned heuristic

- Small basic blocks were good training data for larger blocks. Relied on exhaustive search for training data - not realistic for other domains

- Technique relied on features that were machine specific so questionable portability though induction heuristic is pretty generic

- There is little head room in basic bock scheduler so hard to see benefit over standard schemes. Picked a hard problem to show improvement

- It seems leaning relative merit i vs j is easier than absolute time

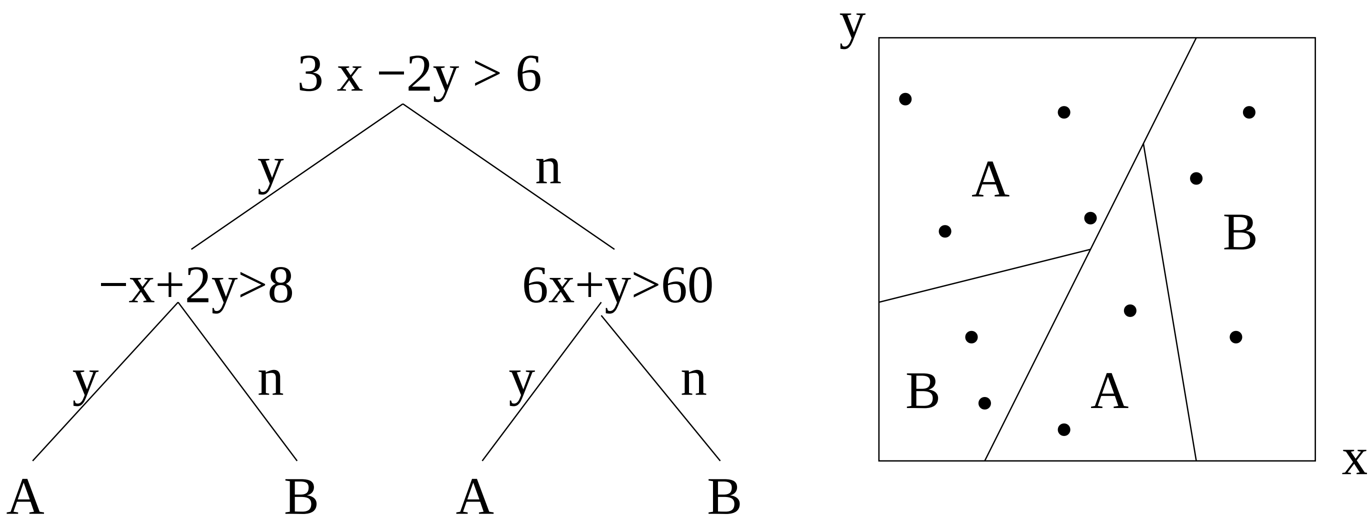
Learning to unroll Monsifort

- Monsifort uses machine learning to determine whether or not it is worthwhile unrolling a loop
- Rather than building a model to determine the performance benefit of loop unrolling, try to classify whether or not loop unrolling s worthwhile
- For each training loop, loop unrolling was performed and speedup recorded. This output was translated into good bad,or no change
- The loop features were then stored alongside the output ready for learning

Learning to unroll Monsifort

- Features used were based on inner loop characteristics.
- The model induced is a partitioning of the feature space. The space was partitioned into those sections where unrolling is good, bad or unchanged.
- This division was hyperplanes in the feature space that can easily be represented by a decision tree.
- This learnt model is the easily used at compile time. Extract the features of the loop and see which section they belong too
- Although easy to construct requires regions in space to be convex. Not true for combined transformations.

Learning to unroll Monsifort



Feature space is partitioned into regions that can be represented by decision tree.

Each constraint is linear in the features forming hyperplanes in the 6 dimensional space.

Learning to unroll Monsifort

```
do i = 2, 100

   a(i) = a(i) + a(i−1) + a(i+1)

enddo
```

statements	1
aritmetic op	2
iterations	99
array access	4
resuses	3
ifs	0

- Features try to capture structure that may affect unrolling decisions
- Again allows programs to be mapped to fixed feature vector
- Feature selection can be guided by metrics used in existing hand-written heuristics

Results

- Classified examples correctly 85% of time. Better at picking negative cases due to bias in training set
- Gave an average 4% and 6% reduction in execution time on Ultrasparc and IA64 compared to 1% and 3% from g77.Better than original heuristic.
- However g77 is an easy compiler to improve upon. Although small unrolling only beneficial on 17/22% of benchmarks
- Boosting helped classification generate a set of classifiers and select based on a weighted average of their classification
- Basic approach - unroll factor not considered.

Not a universal panacea

- Machine learning has revolutionised compiler optimisation and is becoming mainstream.
- However, it is not a panacea, solving all our problems.
- Fundamentally, it is an automatic curve fitter. We still have to choose the parameters to fit and the space to optimise over
- Runtime undecidability will not go away.
- Now being used for heterogeneous multi-cores.