Overview

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

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

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.

![Graph showing inputs and outputs](image-url)
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 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 its 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.
Learning to schedule

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

1 available  2 scheduled
not available  3
4 available

- One of the first papers to investigate machine learning for compiler optimisation
- Appeared at NIPS ’07 - 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 schedule d 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.

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

<table>
<thead>
<tr>
<th>odd</th>
<th>ic</th>
<th>wcp</th>
<th>d</th>
<th>e</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td>T</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Odd ic wcp d e T F

15837

Schedule choice

- 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.

M. O’Boyle Machine Learning based Compilation March, 2009

Induction heuristics

e = second

\[e = \text{same} \wedge \text{wcp} = \text{first}\]

\[e = \text{same} \wedge \text{wcp} = \text{same} \wedge d = \text{first} \wedge \text{ico} = \text{load}\]

\[e = \text{same} \wedge \text{wcp} = \text{same} \wedge d = \text{first} \wedge \text{ico} = \text{store}\]

\[e = \text{same} \wedge \text{wcp} = \text{same} \wedge d = \text{first} \wedge \text{ico} = \text{logical}\]

\[e = \text{same} \wedge \text{wcp} = \text{same} \wedge d = \text{first} \wedge \text{ico} = \text{fpop}\]

\[e = \text{same} \wedge \text{wcp} = \text{same} \wedge d = \text{first} \wedge \text{ico} = \text{iarith} \wedge \text{ic1} = \text{load} \ldots\]

- 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 ...

M. O’Boyle Machine Learning based Compilation March, 2009

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 block 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.

M. O’Boyle Machine Learning based Compilation March, 2009

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 is 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.

M. O’Boyle Machine Learning based Compilation March, 2009
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 easily used at compile time. Extract the features of the loop and see which section they belong to.
- Although easy to construct requires regions in space to be convex. Not true for combined transformations.

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.
- 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.
Learning to inline Cavazos

- Inlining is the number one optimisation in JIT compilers. Many papers from IBM on adaptive algorithms to get it right in Jikes
- Can we use machine learning to improve this highly tuned heuristic? Tough problem. Similar to meta-optimisation goal
- In Cavazos(2005) we looked at automatically determining inline heuristics under different scenarios.
- Opt vs Adapt - different user compiler options. Total time vs run time vs a balance - compile time is part of runtime
- x86 vs PPC - can the strategy port across platform

M. O'Boyle Machine Learning based Compilation March, 2009

Learning a heuristic

```
inliningHeuristic(calleeSize, inlineDepth, callerSize)
if (calleeSize > CALLEE_MAX_SIZE)
    return NO;
if (calleeSize < ALWAYS_INLINE_SIZE)
    return YES;
if (inlineDepth > MAX_INLINE_DEPTH)
    return NO;
if (callerSize > CALLER_MAX_SIZE)
    return NO;
// Passed all tests so we inline
return YES;
```

Focus on tuning parameters of an existing heuristic rather than generating a new one from scratch

Features are dynamic. Learn off-line and applied heuristic on-line

M. O'Boyle Machine Learning based Compilation March, 2009

Impact of inline depth on performance: Compress

- Initially tried rule induction - failed miserably. Not clear at this stage why. Difficult to determine whether optimisation has impact
- Next used a genetic algorithm to find a good heuristic.
- For each scenario asked the GA to find the best geometric mean over the training set. Using search for learning.
- Training set used - Specjvm98, test set - DaCapo including Specjbb
- Focused learning on choosing the right numeric parameters of a fixed heuristic.
- Applied this to a test set comparing against IBM heuristic.
Impact of inline depth on performance: Jess

Parameters found

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Orig</th>
<th>Adapt</th>
<th>Opt:Bal</th>
<th>Opt:Tot</th>
<th>Adapt (PPC)</th>
<th>Opt:Bal (PPC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CalleeMSize</td>
<td>23</td>
<td>49</td>
<td>10</td>
<td>10</td>
<td>47</td>
<td>35</td>
</tr>
<tr>
<td>AlwaysSize</td>
<td>11</td>
<td>15</td>
<td>16</td>
<td>6</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>MaxDepth</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>CallerMSize</td>
<td>2048</td>
<td>60</td>
<td>402</td>
<td>2419</td>
<td>1215</td>
<td>3946</td>
</tr>
<tr>
<td>HotCalleeMSize</td>
<td>135</td>
<td>138</td>
<td>NA</td>
<td>NA</td>
<td>352</td>
<td>NA</td>
</tr>
</tbody>
</table>

- Considerable variation across scenario.
- For instance on x86, Bal and Total similar except for the CallerMaxSize
- A priori these values could not be predetermined

Results

<table>
<thead>
<tr>
<th>Compilation Scenarios</th>
<th>SPECjvm98</th>
<th>DaCapo+JBB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Running</td>
<td>Total</td>
</tr>
<tr>
<td>Adapt</td>
<td>6%</td>
<td>3%</td>
</tr>
<tr>
<td>Opt:Bal</td>
<td>4%</td>
<td>16%</td>
</tr>
<tr>
<td>Opt:Tot</td>
<td>1%</td>
<td>17%</td>
</tr>
<tr>
<td>Adapt (PPC)</td>
<td>5%</td>
<td>1%</td>
</tr>
<tr>
<td>Opt:Bal (PPC)</td>
<td>1%</td>
<td>6%</td>
</tr>
</tbody>
</table>

- Does considerably better on the test data relative to inbuilt heuristic than on Spec
- Suspect Jikes writers tuned their algorithm with SPEC in mind.
- Shows that an automatic approach ports better than hand-written

Not a universal panacea

- I believe that machine learning will revolutionise compiler optimisation and will become mainstream within a decade.
- 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.
- Complexity of space makes a big difference. Tried using Gaussian process predicting on PFDC ’98 spaces - worse than random selection!