Machine Learning based Compilation

Michael O'Boyle

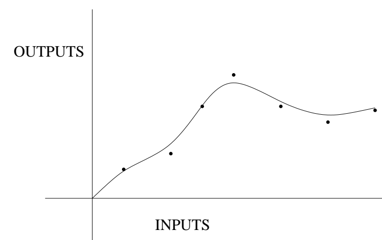
March, 2009

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

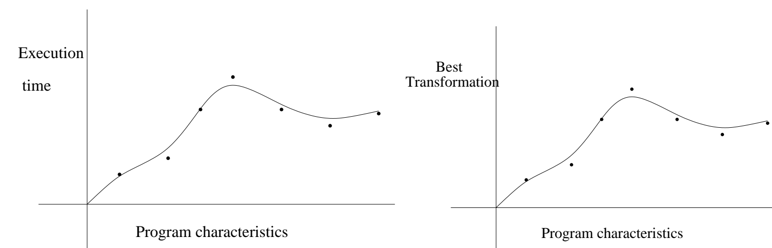
Machine Learning as a solution

- 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.

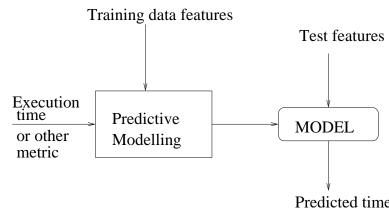


Machine Learning

- 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!

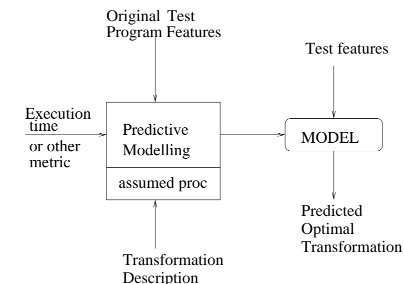
Training data

- 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

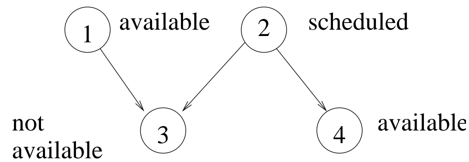
Case studies



- 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

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 '07 - not picked up by compiler community till later.

Learning to schedule

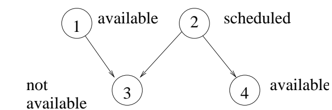
- The approach taken is to look at many (small to medium) basic blocks and to exhaustively determine all possible schedules.
- Next go through each block and given a (potentially empty) partial schedule and the choice of two or more instructions that may be schedule d next, select each in turn and determine 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.

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

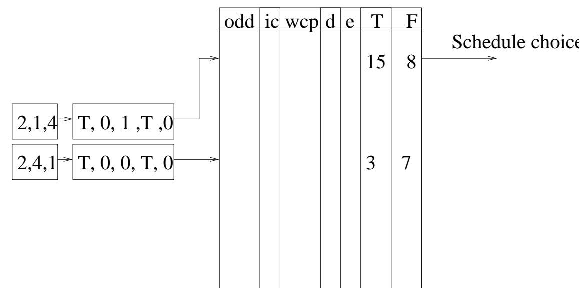


Tuple $(\{2\}, 1, 4)$: [odd:T, ic:0, wcp:1, d:T, e:0]: TRUE,

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

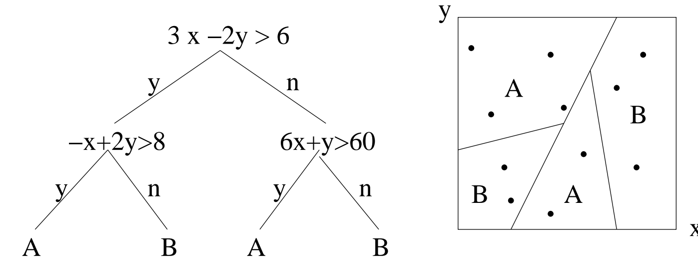
Learning to unroll Monsifort

- 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 n 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 casses 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

Learning to inline Cavazos

- 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.

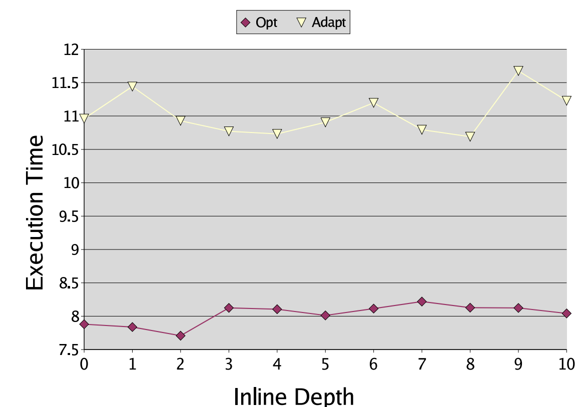
Learning a heuristic

```
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;
```

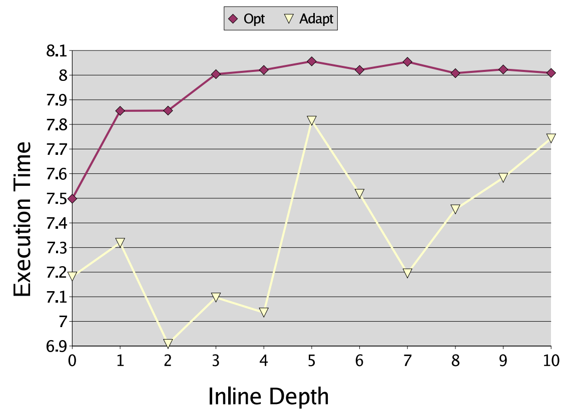
Focus on tuning parameters of an existing heuristic rather than generating a new one from scratch

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

Impact of inline depth on performance: Compress



Impact of inline depth on performance: Jess



Parameters found

Parameters	Compilation Scenarios					
	Orig	Adapt	Opt:Bal	Opt:Tot	Adapt (PPC)	Opt:Bal (PPC)
CalleeMSize	23	49	10	10	47	35
AlwaysSize	11	15	16	6	10	9
MaxDepth	5	10	8	8	2	3
CallerMSize	2048	60	402	2419	1215	3946
HotCalleeMSize	135	138	NA	NA	352	NA

- 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

Compilation Scenarios	SPECjvm98		DaCapo+JBB	
	Running	Total	Running	Total
Adapt	6%	3%	0%	29%
Opt:Bal	4%	16%	3%	26%
Opt:Tot	1%	17%	-4%	37%
Adapt (PPC)	5%	1%	-1%	6%
Opt:Bal (PPC)	1%	6%	8%	7%

- 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!