# Adaptive & Feedback Driven Compilation

Grigori Fursin

$http://homepages.inf.ed.ac.uk/gfursin/research\_desc.html$

based on the lecture course written by Michael O'Boyle
from Edinburgh University

October, 2005

## Course overview

Assume all understand basics of processor architecture and compilation process.

Focus not on individual components but the way we construct and frame compilation.

- Background and motivation
- Feedback directed approaches
- Dynamic compilation
- Machine Learning and future directions

## Background and motivation:

- Compiler construction
- Compiler optimisation - is it worthwhile?
- Classical optimisation
- Why we fail to fully optimise
- How to overcome this
- Classification of optimisation approaches

## Compilation: translation + automation

- Compilers : map user programs to hardware. Translation - must be correct
- Tackling a universal systems problems: C to x86, VHDL to netlists etc.
- Hide underlying complexity. Machines are not von Neumann
- Computer Science is the study of automation. Compilers automatically translate. Can we automate compiler construction?
- Compiler compilers exist. Retargettable systems eg gcc, CoSy
- Automatic construction of compiler optimisation is notoriously difficult. Return to this later

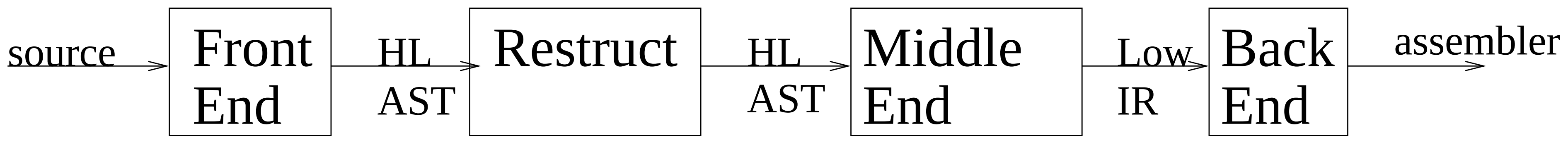
## Compiler Construction

- Little changed in structure since the 1950s
- Consists of a linear sequences of passes over an intermediate format (IR). Each pass operates in near linear time over IR
- The sequence of passes or phases makes sense but we have no hard evidence that it is the best
- Breakdown the program text into smaller items and check if the make sense. Structure similar to that of natural language processing
- If so, try to understand the *meaning* of the program and generate an equivalent version executable by the hardware

## Phase Order

- Lexical Analysis: Finds and verifies basic syntactic items lexemes,tokens using Finite state automata
- Syntax Analysis: Checks tokens follow a grammar based on a context free grammar and builds an Abstract Syntax Tree (AST)
- Semantic Analysis: Checks all names are consistently used, varies type checking schemes employed. Builds a symbol table
- Optimisation + Code generation: Most effort here, little automation

## Compiler structure



- Front end translates "strings of characters" into a structured abstract syntax tree (AST)

- Middle end attempt machine independent optimisation. Can also include "source to source" transformations - restructurer - outputs a lower level intermediate format

- Many choices for IRs. Affect form and strength of later analysis or optimisation

- Backend: code generation, instruction scheduling and register allocation

## Compilation

- Compilers map user programs to hardware. Machines are not von Neumann - designed for speed.
- Compilers always have been essential component in performance. Current focus : Optimisation go faster, smaller, cooler.
- However, we have been trying to solve this problem for more than 40+ years.
- Are we ever going to solve this? What reasons do we have for optimism?
- Gap between potential performance and actual widening
- Can compilers help or have they been shown to be poor performers?

## Importance of compilers (1)

- Increase in performance is largely due to technology
- However, technology alone provides nothing - it needs new architectures to exploit potential
- Similarly new architectures need new compilers to exploit features
- Next generation of multi-core chips will not deliver performance without improved compiler technology

## Importance of compilers (2)

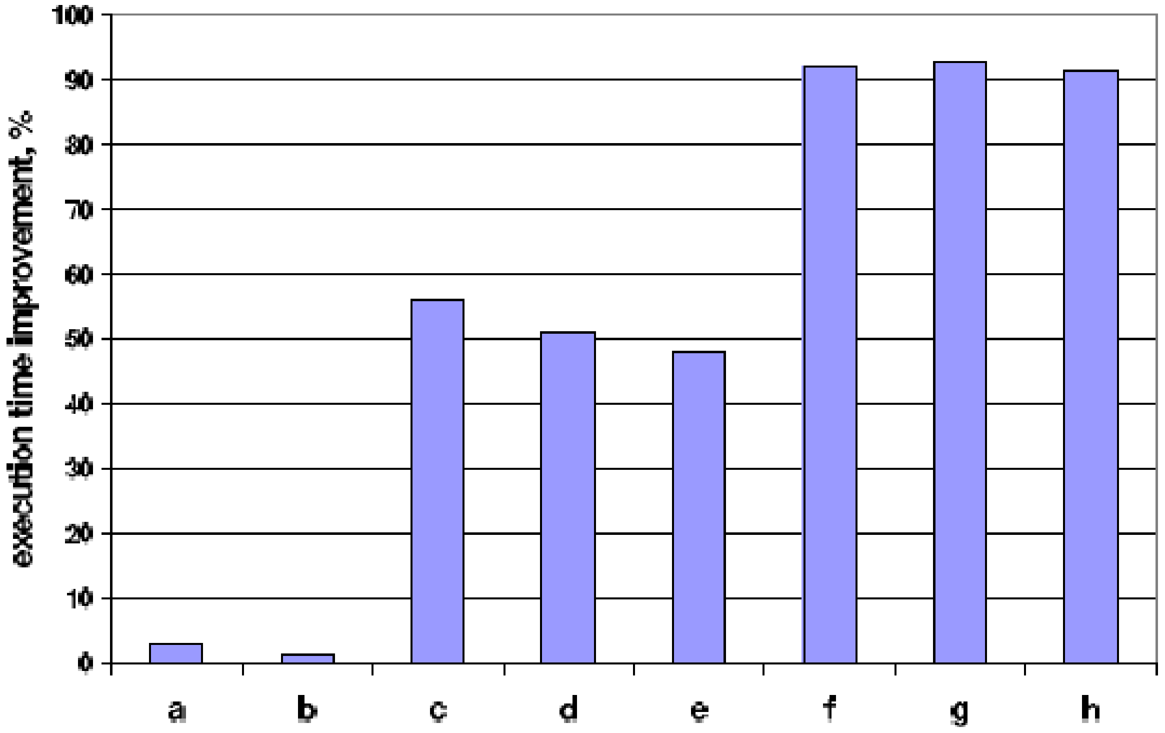
- Assembler programming and manual tuning are still wide-spread in the embedded world but hard, not portable and time consuming

- Challenge: compiler should automatically produce highly optimised code comparable to hand-tuning versions.

- Question: what percentage of optimum do we get - very hard question.

## Room for improvement? Matrix multiplication example

```
a - high level
b - feedback
c - Lam
d - Coleman
e - prediction1
f - library
g - prediction2
h - search
```



Demonstration later ...

## Compilation as translation vs optimisation

- Modern focus is on exploiting architecture features. Exploiting instruction and thread level parallelism
- Effective management of memory hierarchy registers,L1,L2,L3,Mem,Disk
- Small architectural changes have big impact. Changes in memory hierarchy do not affect ISA but have large performance impact.
- Compilers have to be architecture aware
- Optimisation at many levels source, internal formats, assembler and scopes, basic block, super/hyper blocks, loop, procedure, whole program.

## Machine dependent vs independent optimisation

- Optimisations typically split into those that are always worthwhile and machine specific.
- Example: Common sub-expression elimination
- Aim: Prevent redundant recalculation of terms

```
a = b + c + f            t = b + c
f = b + c + e            a = t + f
                         d = t + e
```

Seems always a good idea: 4 adds vs 3

BUT potentially additional variable - pressure on register allocation

## Machine dependent vs independent optimisation

- Architectural features strongly determine the best code sequence.
- Rarely are all instructions of equal cost. Even if they have the same latency, not all function units support all functions
- The "stranger" / more complex the hardware, the harder it is to determine the the best
- Code selection for embedded systems a good example - heterogeneous instruction set (CISC vs RISC)
- Example MAC (multiply accumulate a = a + x*y)

## MAC example

```
t = b * c                    MUL t,b,c
a = a + t                    ADD a,a,t
d = d + t                    ADD d,d,t

a = a + b*c                  MAC a,b,c
d = d + b*c                  MAC d,b,c
```

Simple example.

Which version to select?

Much more complex in practice especially with multimedia instructions.

## Classic optimisation: Static analysis and transformation

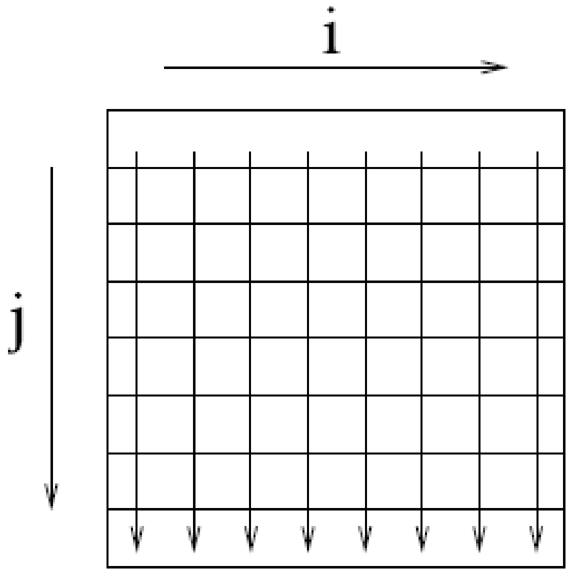
- Standard view. Statically (at compile time) analyse the program based on requirements e.g. register requirements or type checking.
- Transform program accordingly . Example stride-1 access. C has row-major layout. Makes sense to traverse data row-wise.

```
for (i = 0; i<n;i++)
 for (j = 0; j<n;j++)
   a[j][i] + b[i];
```

- This code traverses the array column-wise
- Does not exploit spatial locality. Can have excessive cache misses.

## Poor Stide

```
for (i = 0; i<n;i++)
 for (j = 0; j<n;j++)
   a[j][i] + b[i];
```



- Neighbouring fetched elements not referenced until much later
- Cache line probably evicted by then

## Classic optimisation: Static analysis and transformation

- Analysis states that the innermost iterator should be in outermost subscript - make it so!

- Transform - apply code restructuring to achieve this - loop interchange

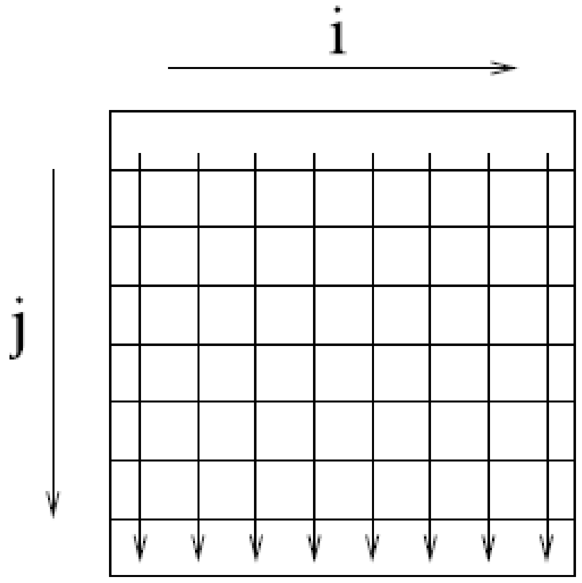
```
for (j = 0; j<n;j++)
 for (i = 0; i<n;i++)
   a[j][i] + b[i];
```

- This code now traverses the array row-wise

- Advantage linear analysis and transformation, dramatic performance improvement

## Improved Stride

```
for (i = 0; i<n;i++)
 for (j = 0; j<n;j++)
   a[j][i] + b[i];
```



- Neighbouring fetched elements referenced immediately
- Cache line unlikely to be evicted

## Classic optimisation: Static analysis and transformation

- However does not consider other costs. e.g. b[i] is no longer invariant - temporal locality lost
- Uses idealised model of machine. No account of memory hierarchy , cache replacement policy etc.
- If any of this were to change, no way of changing the compiler
- Fundamentally each analysis has a small focused scope and hardware issue to reduce complexity.
- No theory/practise to integrate views.

## Poor results

Compiler only achieve a small percentage of the maximum processor performance

Phase order means there is a long chain from source to machine code.

- Along the path there is inevitably a loss of information
- Example : Memory accesses. At assembler level virtually impossible to determine memory locations, accesses and patterns.
- Make conservative assumptions - assume accesses are to same location
- Lose parallelism/latency tolerance
- Such a problem that program languages are modified to solve it.

## Poor results

Modern architectures is technology driven.

- Not the best fit for programs written in high level languages
- *How would you rewrite your Java program if the processor changed from in-order to out-of order execution ?*
- Architectures designed around binaries of previous generation. *SPEC Alpha binaries are set in stone!*
- Compiler and architectures out of sync - compilers target yesterday's hardware.
- Application-specific systems have greater integration and performance e.g. DSP chips.

## Why fail

- Fundamental reason for failure is complexity and undecidability
- At compile time we do not know the data to be read in, so impossible to know the best code sequence
- The processor architecture behaviour is so complex that it is almost impossible to determine what the best code sequence should be even if we knew the data to be processed.
- Although individual components are simple, together impossible to derive realistic model
- O-O execution and cache have non-deterministic behaviour!

## Case Study

**Problem** Matrix multiplication

**Problem size** $N = 400$ and $N = 512$

**Processors** UltraSparc, R10000, Pentium Pro, Alpha, TriMedia

**Transformation space** Loop unrolling 1 – 20, Tiling 1 – 100, Padding 1 – 10

Well studied by static techniques (PFDC '98 with PACT'98)

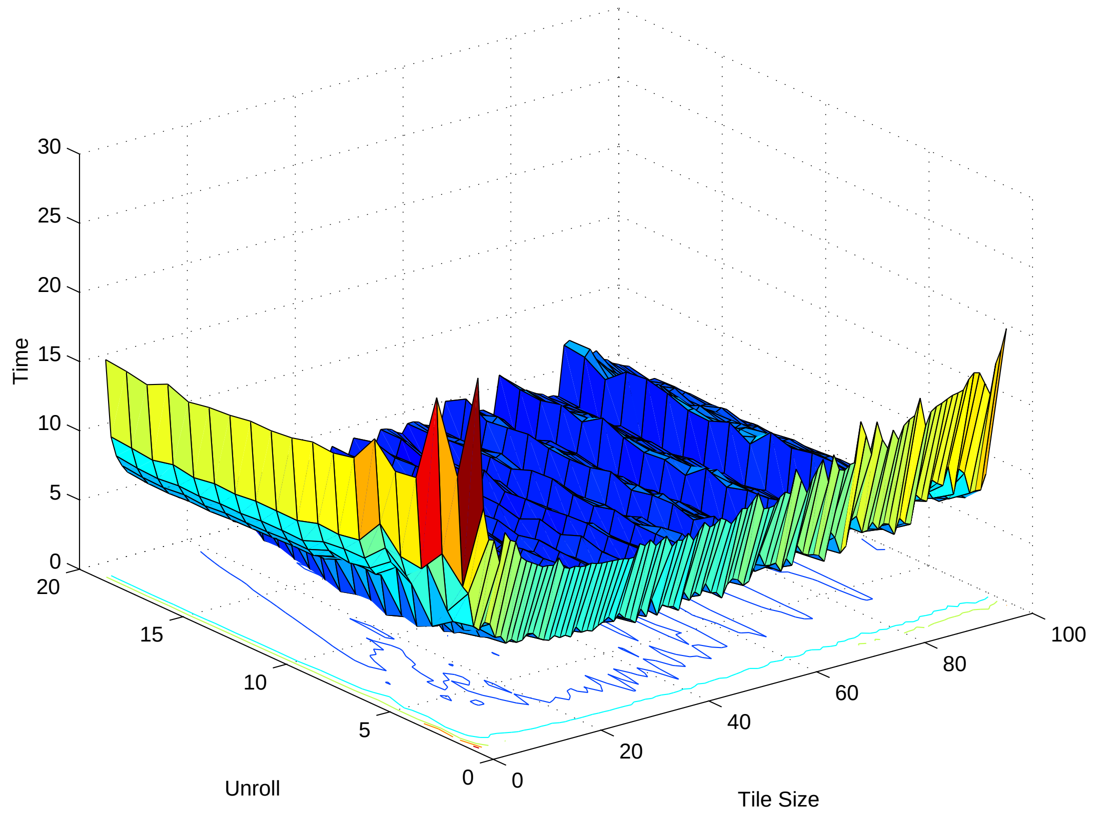
What do the spaces look like. How easy is it to model and find good points.

## Transformations

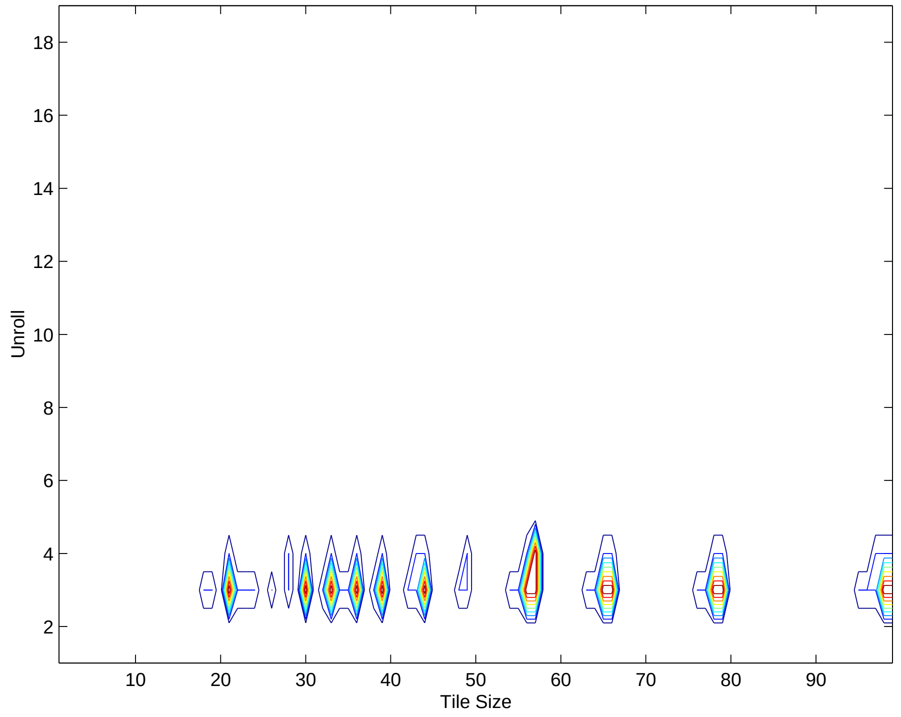
```
REAL a(N,N),B(N,N),C(N,N)
  Do j = 1, N
   Do k = 1, N
    Do i = 1, N
     a(i,j) = a(i,j)
            + b(i,k) * c(k,j)
    Enddo
   Enddo
  Enddo
```

```
REAL a(P+N,N),B(P+N,N),C(P+N,N)
  Do jj = 1, N, TileSize
   Do kk = 1, N, TilseSize
    Do j = jj, jj + TileSize
     Do k = kk, kk + TileSize
      Do i = 1, N, Unroll
       a(i,j) = a(i,j)
          + b(i,k) * c(k,j)
       a(i+1,j) = a(i+1,j)
          + b(i+1,k) * c(k,j)
       a(i+2,j) = a(i+2,j)
          + b(i+,k) * c(k,j)
       ....
```

## Performance UltraSparc for $N = 512$



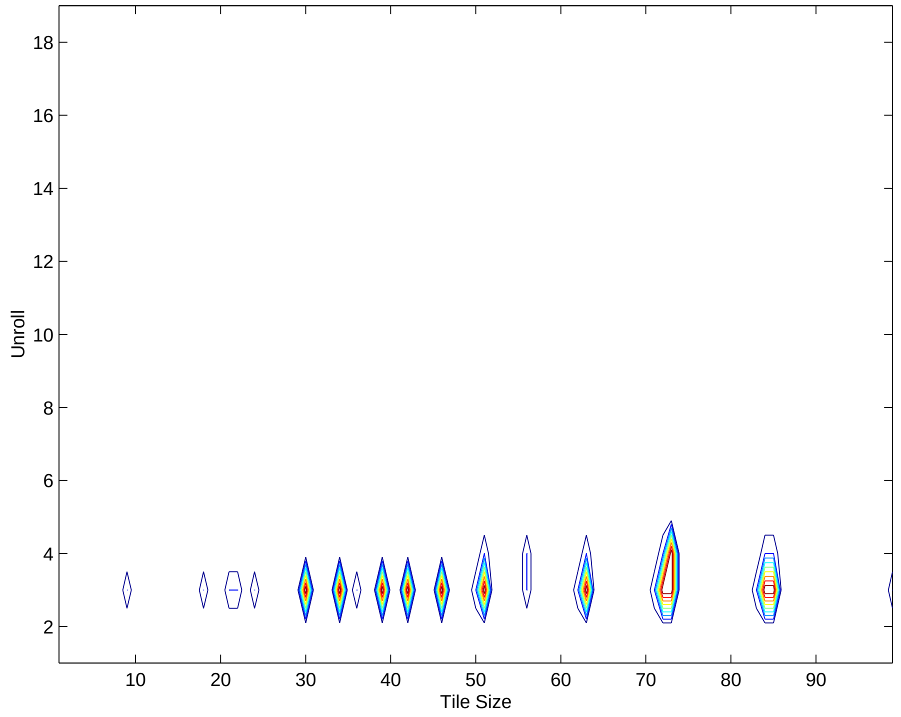
## UltraSparc: space within 20% of minimum $N = 400$



Minimum at: Unroll = 3 and Tile size = 57.

Near minimum: 2.6%. Original 4.99 secs, Minimum 0.56 secs

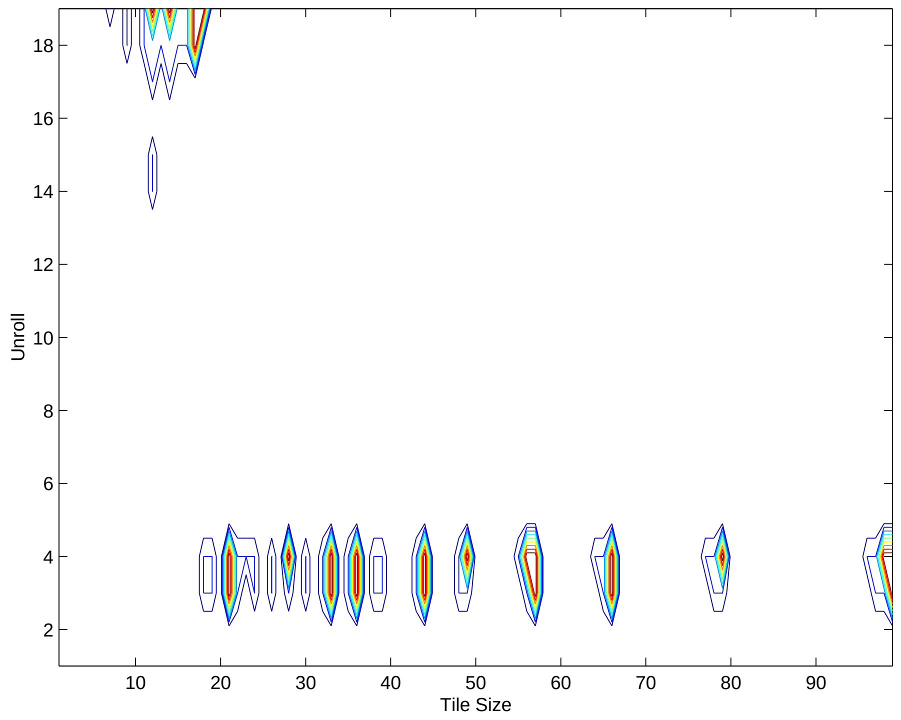
## UltraSparc: space within 20% of minimum $N = 512$



Minimum at: Unroll = 3 and Tile size = 73.

Near minimum: 1.5%. Original 11.53, Minimum 1.54

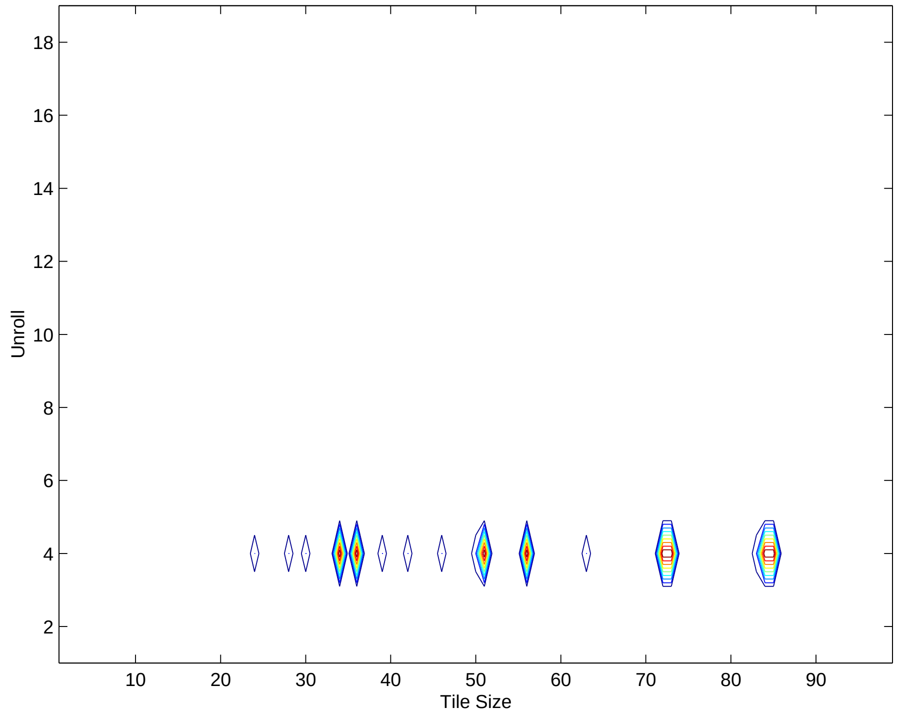
## Alpha: space within 20% of minimum $N = 400$



Minimum at: Unroll = 4 and Tile size = 44.

Near minimum: 4.3%. Original 12.04, Minimum 1.23

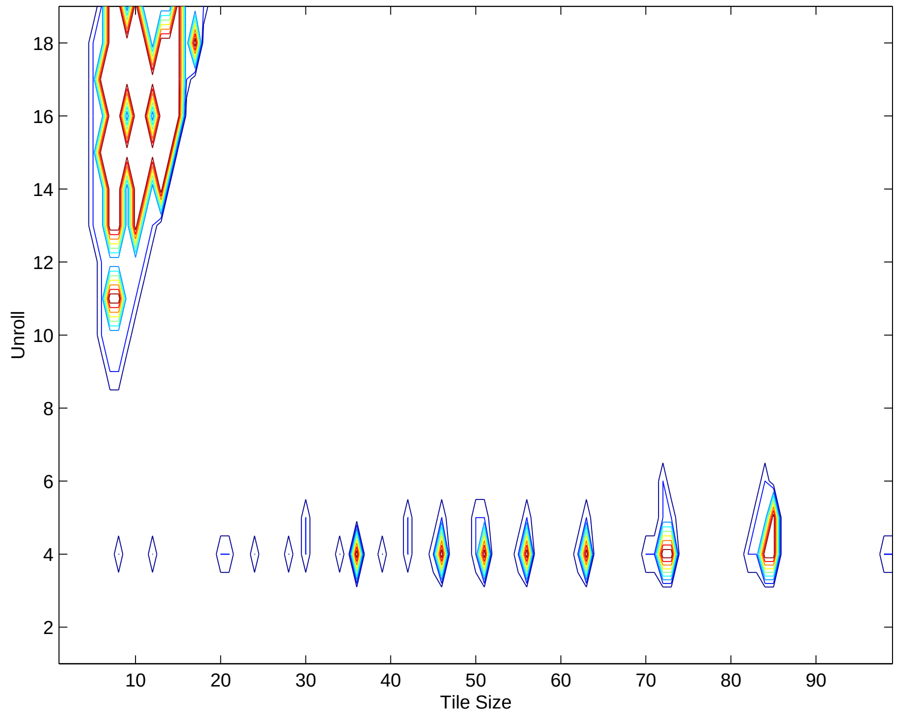
## Alpha: space within 20% of minimum $N = 512$



Minimum at: Unroll = 4 and Tile size = 85.

Near minimum: 0.9%. Original 31.72, Minimum 3.34, Max 81.40 !

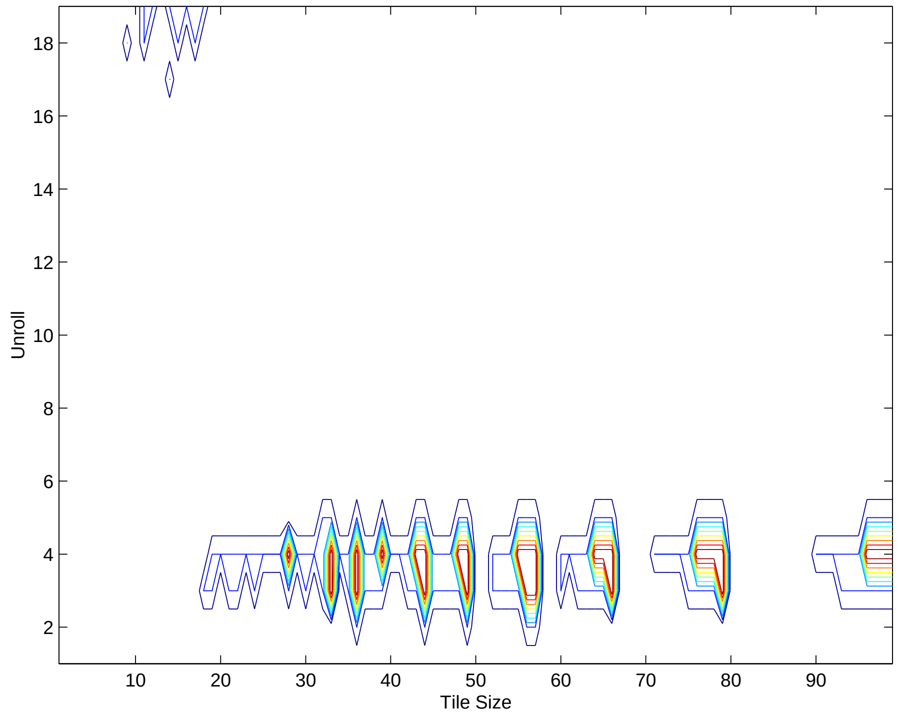
## **R10000:** $N = 512$



Minimum at: Unroll $=$ 4 and Tile size $=$ 85.

Near minimum: 7.2%. Original 2.79, Minimum 1.09

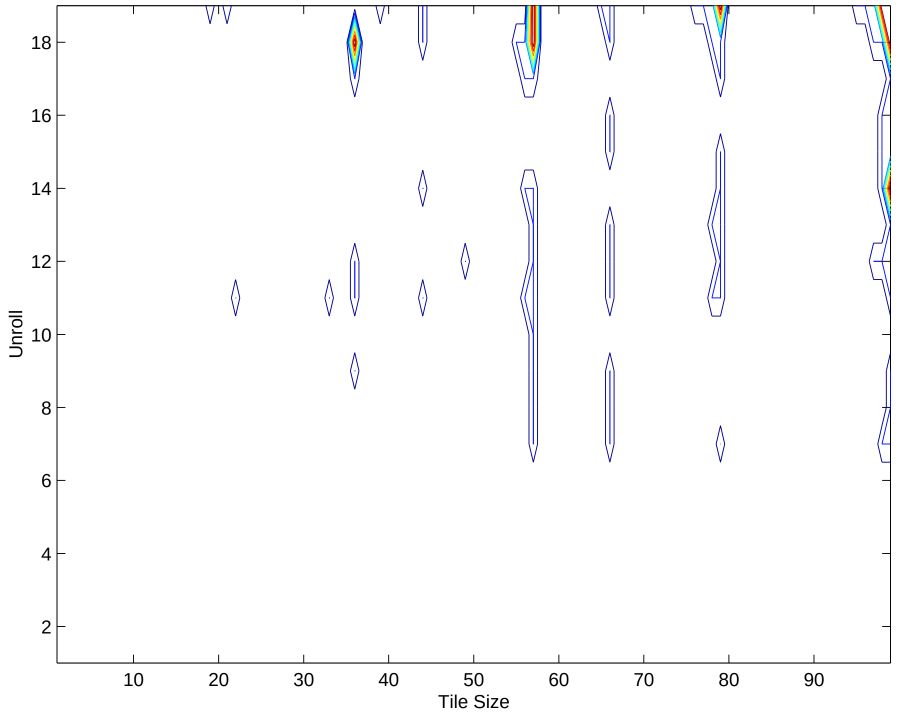
## R10000: $N = 400$



Minimum at: Unroll $=$ 4 and Tile size $=$ 57.

Near minimum: 7.8%. Original 0.71, Minimum 0.41

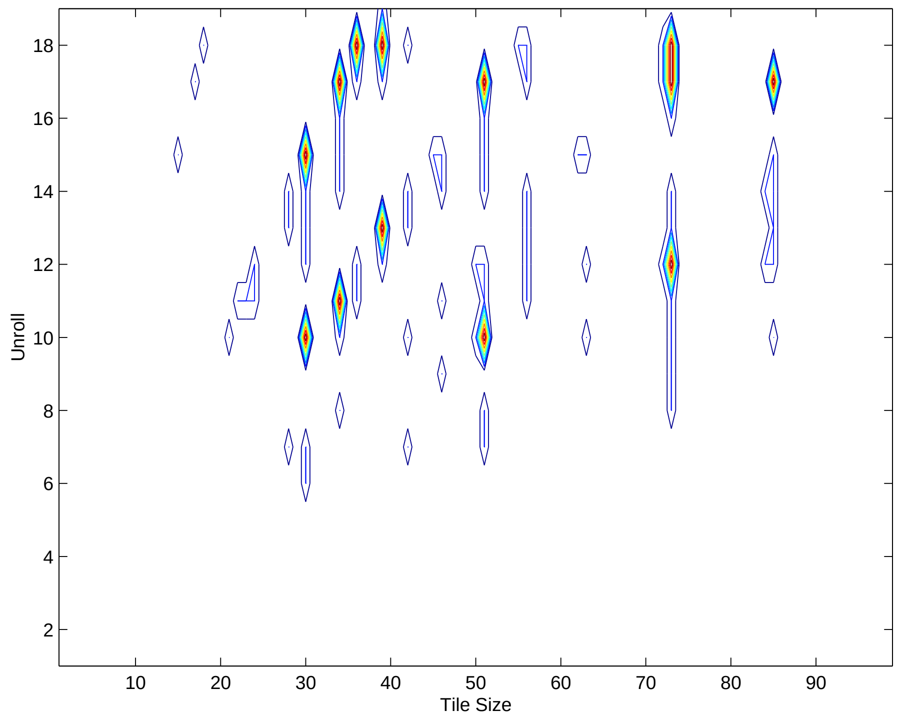
## Pentium Pro: space within 20% of minimum $N = 400$



Minimum at: Unroll = 19 and Tile size = 57.

Near minimum: 4.3%. Original 4.88 Minimum 1.43

## Pentium Pro: space within 20% of minimum $N = 512$



Minimum at: Unroll = 17 and Tile size = 51.

Near minimum: 4.6%. Original 30.93, Minimum 3.34

## Conclusion for Effects

- Graph of execution times is periodic with high frequency oscillations and many local minima. Hence difficult to find absolute minimum.

- Best transformation is highly dependent on type of processor.

- Impossible for static analysis to determine the best optimisation

- Even harder if the processor changes!

## Compilation as Optimisation

- Express as a “formal” optimisation problem. Minimise objective function over a space of options.
- Objective function is execution time, though space and power may be important.
- Optimisation space:All possible equivalent programs
- Objective function is undecidable in general
- Optimisation space: infinite

## Compilation as Optimisation

- Solving an undecidable problem over an infinite space is clearly not feasible so simplification is necessary
- Try to solve undecidable problem in less time than execution! In general undecidable, but in practice, people aim to write programs that terminate
- We know that it is possible to tune performance
- Traditionally have broken the problem into sub-problems based on certain assumptions
- Each of these problems are themselves at least NP-complete

## Intractability

Solve the problem by looking at each prob in isolation

- Code generation - determining the best code for an expression is NP
- Scheduling - determining the best order of instruction is NP
- Register allocation determining the best use of registers to minimise memory traffic - NP

## How do we overcome this

Two main problems

- Complexity of processor architecture, Undecidability of program

Both problems arise from trying to optimise statically at compile time

- Have to guess a tractable model, Have to guess about data input
- Pros and Cons to all approaches .Depends highly on application scenario

## Taxonomy - a roadmap of a new landscape

- 2 main causes program undecidability and processor complexity
- Variables (what): Program (P), Data (D) and Processor (proc)
- Variables (when): design, compile or runtime.
- 2 sides of adaption: portability and specialisation
- Examine all techniques in this light

## Taxonomy

- Program (P), Data (D) and Processor (proc)
- time = f(T(P),D,proc), Pick Transformation T to minimise f
- Standard compilation (SC) typically has a hardwired model of proc built in.
- SC also has an ad hoc view of typical programs (biased by SPEC!) with a *compiler strategy* that is biased to them
- SC applies the strategy at compile time making no reference to data
- Data in no way affects SC behaviour - just guess a "typical" input set

## Taxonomy

Design time:

- Build a compiler: encode compiler optimisation strategy. Typically a time consuming manual process. Takes many person-years. Particular to one processor, data and programs unknown

Compile time:

- Examine program and apply transformations based on design time encoded strategy. Can take a reasonable amount of time. Must be less than accumulated runtime throughout lifetime of program

- Processor assumed, program known, data unknown

## Taxonomy

Runtime:

- Most knowledge about application available: processor, program and data
- Least amount of time available to do anything about it!
- Typically compilers do nothing - leave to independent runtime system/OS

Most room for improvement at runtime - and design time

## Taxonomy: Adaption = Portability + Specialisation

Compiler technology not normally discussed in this manner. Appears an infrastructure rather than optimisation issue.

Portability

- ability to MODIFY behaviour to changing circumstances, changing data, program, processor

Specialisation

- ability to EXPLOIT fixed, known features: processor, program and data

Natural tension between the two . Flexibility vs rigidity

## Taxonomy -current static compilers

- What and when to port/specialise: processor, program, data;design, compile, runtime
- Currently: *specialise* to processor at design time BUT cannot *port* to a new processor
- *Portable* across a wide range of *programs* and *data* at compile and runtime BUT
- Do not *specialise* to *runtime data* or *program/processor* interaction
- Very little exploitation of dynamic *runtime* knowledge/*Adaption* to changing processor or data not considered

## Summary

- Traditional view of compilation
- Compilation as optimisation
- Failure of compiler to deliver and the complexity of the task
- Taxonomy of compiler options as a means to investigate compiler design space
- Next: look at different ways to build compilers that overcome these problems

## Feedback Directed Compilation

## Overview

- Profile Directed Compilation
- Application Tuning
- Iterative Compilation
- Efficient searching
- Critical evaluation and conclusion

## Overcoming the limits of static compilation

Examined limits of current static compilation

- hard optimisation + undecidable problem in general
- complex hardware - processor behaviour massively non-linear

Interested in new techniques that go beyond standard approach

Examine new approaches in terms of

- Adaption: portability vs specialisation
- What: processor, program, data. When: design, compile and runtime.

Focus on exploiting knowledge about data and processor

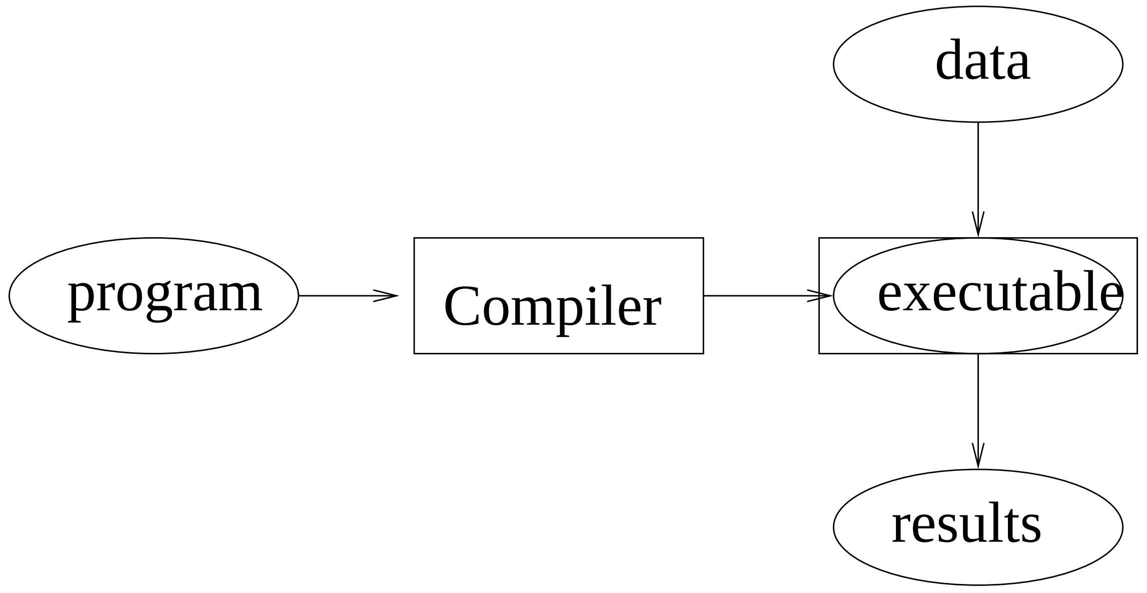
## Profile directed compilation

- Direct addresses problem of compile time unknown data
- Key(simple) idea: run program once and collect some useful information
- Use this runtime information to better improve program performance
- In effect move the first runtime into the compile time phase
- Makes sense if gathering the profile data is cheap and user willing to pay for 2 compiles. Can still use after first compile.
- Allows specialisation to runtime data - pros and cons?

## Offline vs on-line

- Profile directed compilation is one example of off-line optimisation
- Information is gathered and utilised before the the “production” run
- On-line schemes gather information and dynamically change program as it runs.
- Off-line schemes work on basis that costs incurred at compile-time are outweighed by improved runtime. Can be more aggressive than on-line schemes.

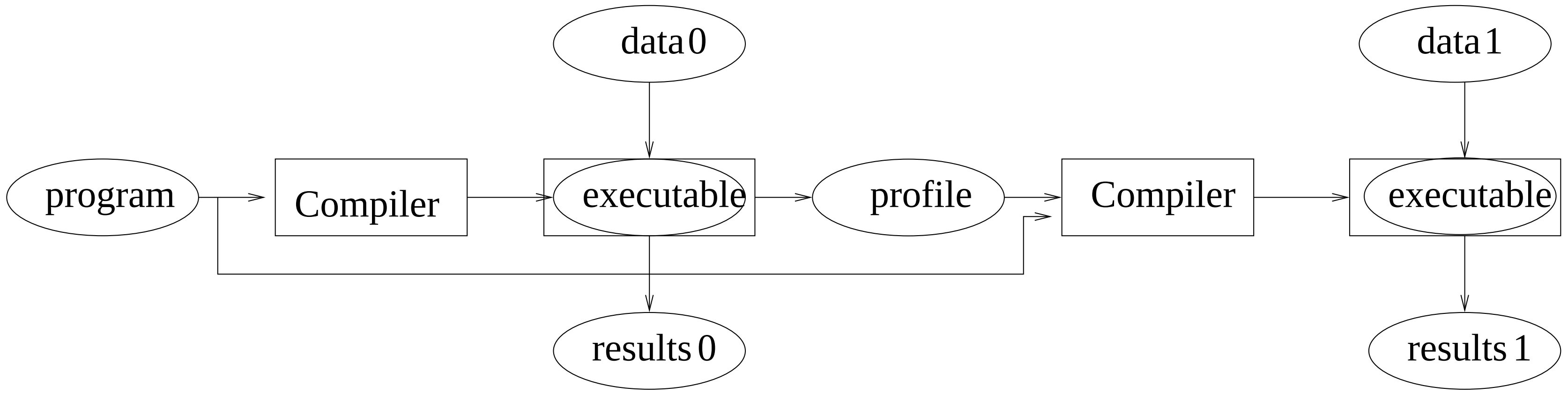
## PDC schematic



Traditional compilation model.

Executable is an output and a process

## PDC schematic



Profile information is an additional output.

Data can change from run to run. Executable still correct.

## Brief History

The use of profiling to aid program performance has been around for a while.

- prof, gprof (1982). A tool to help developers to understand their code. Instrumentation at compile time and then sampled at runtime.

- Hardware analysis (1980s). Monitor program behaviour and adapt. Branch prediction - pipelines means need to guess which branch to take

- Edge/node based profile information for compilers 1990s.

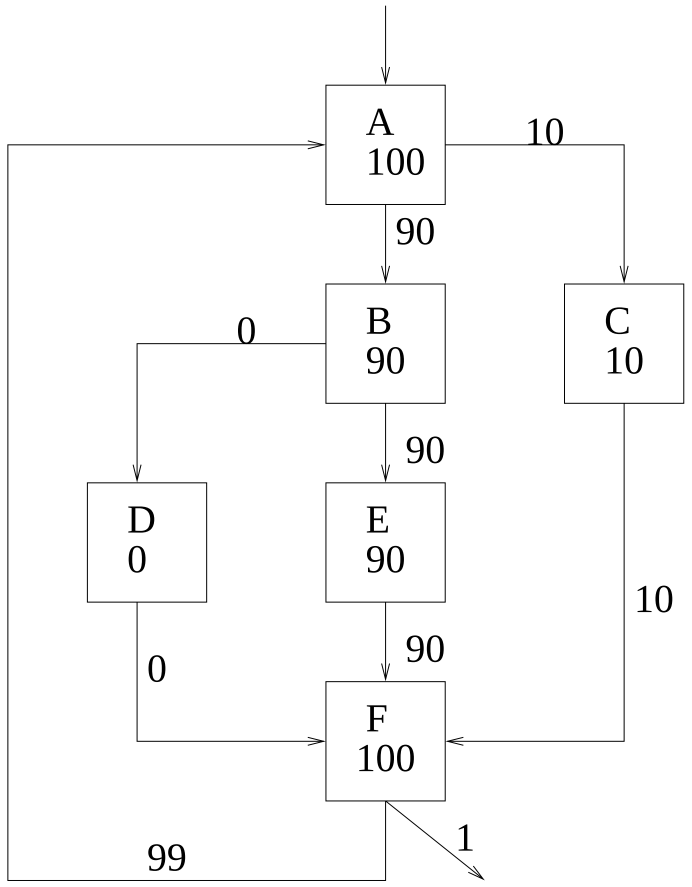
- Path based profiling Larus + Ball late 1990s, Smith 2000

## PDC for classic optimisation

- Record frequently taken edges of program control-flow graph
- IMPACT compiler in 1990s good example of this but also used earlier - Josh Fisher et al, Multiflow.
- Use weight information of edges and paths in graph to restructure control-flow graph to enable greater optimisation
- Main idea: merge frequently executed basic blocks increasing sizes of basic block if possible (superblock/hyperblock) formation. Fix up rest of code.
- Allows improved scheduling of instructions and more aggressive scalar optimisations at expense of code size.

## PDC Example 1

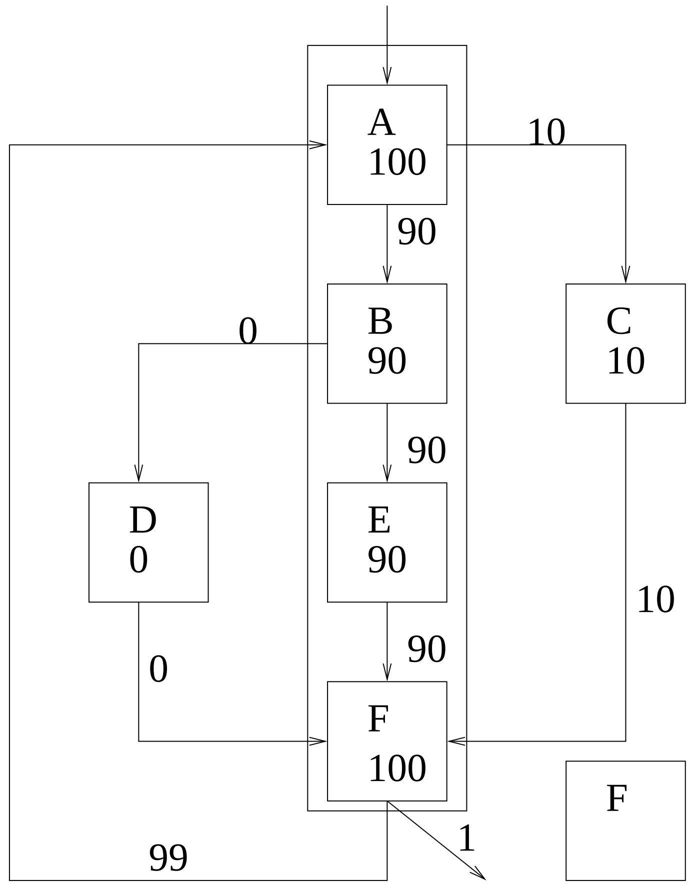
Sequence of basic blocks
Frequency of execution on edges and nodes
Primarily ABEF
Other entry/exit control-flow prevents merging
Super-block - frequently executed path
Merge and tidy-up
Optimise larger unit



## PDC Example 2

Selecting the trace

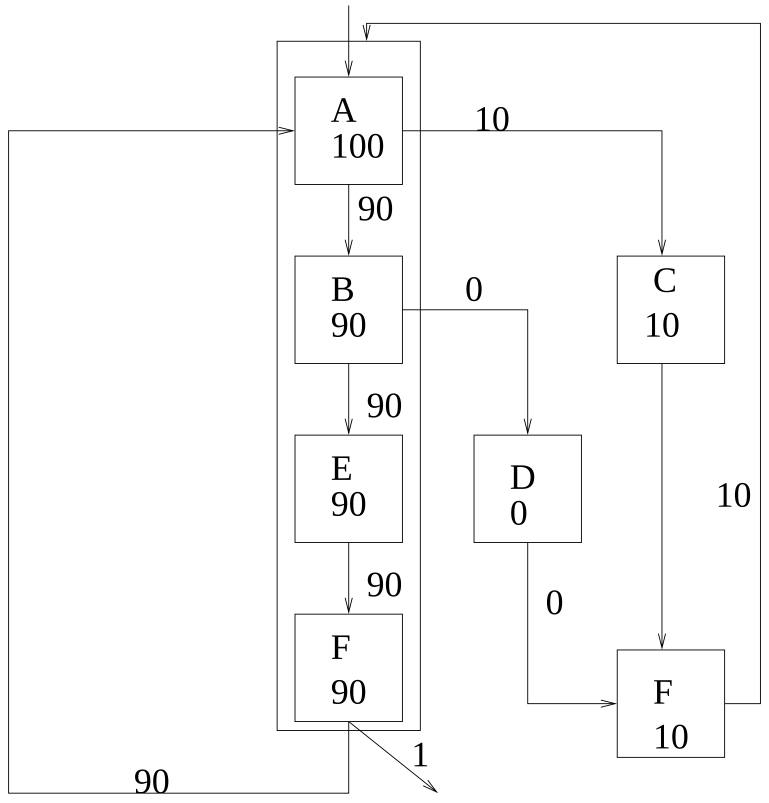
Start at most frequent block
Add blocks on most frequent successors
Repeat on other nodes
Done in both control-flow directions
Do on remaining nodes



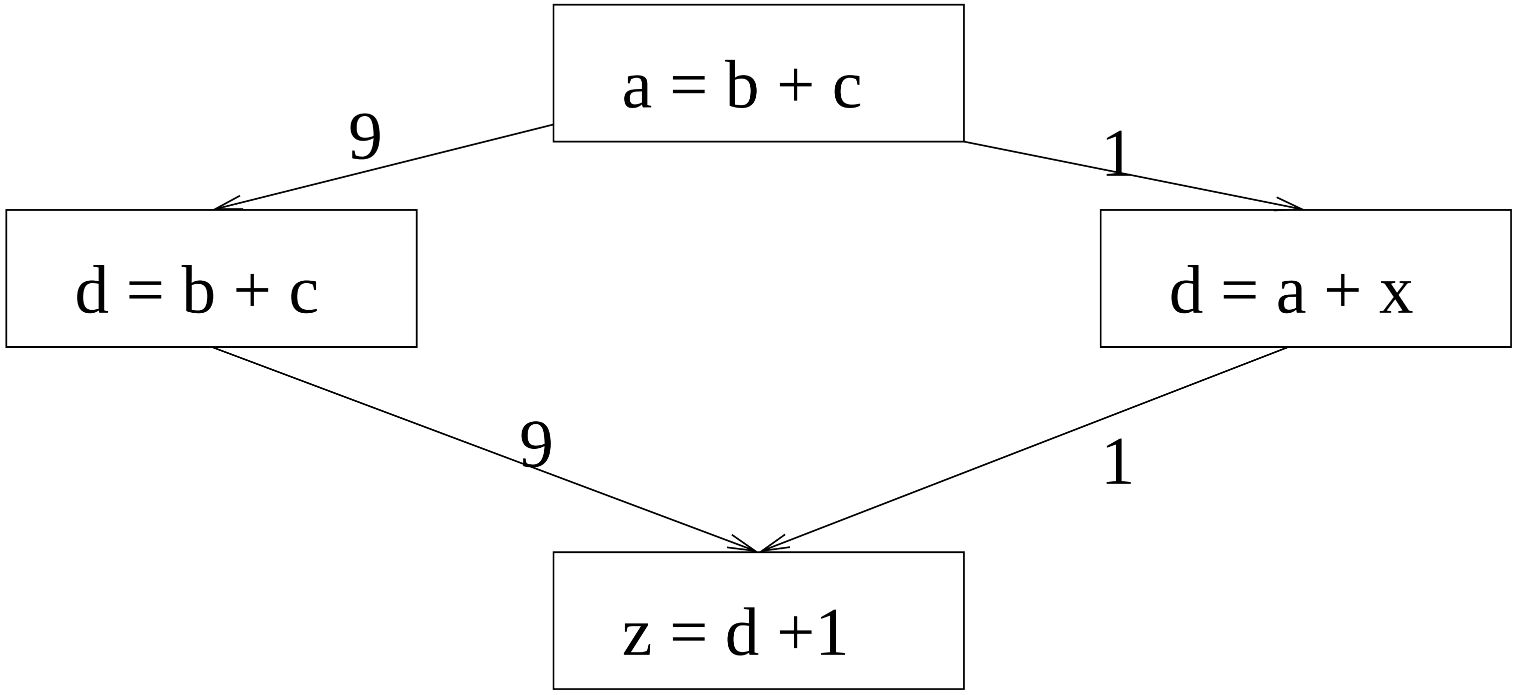
## PDC Example 3

Tail Duplication

Duplicate first block with external entry edges
But not the head
Redirect incoming edges
Duplicate outgoing
Repeat
Much code duplication

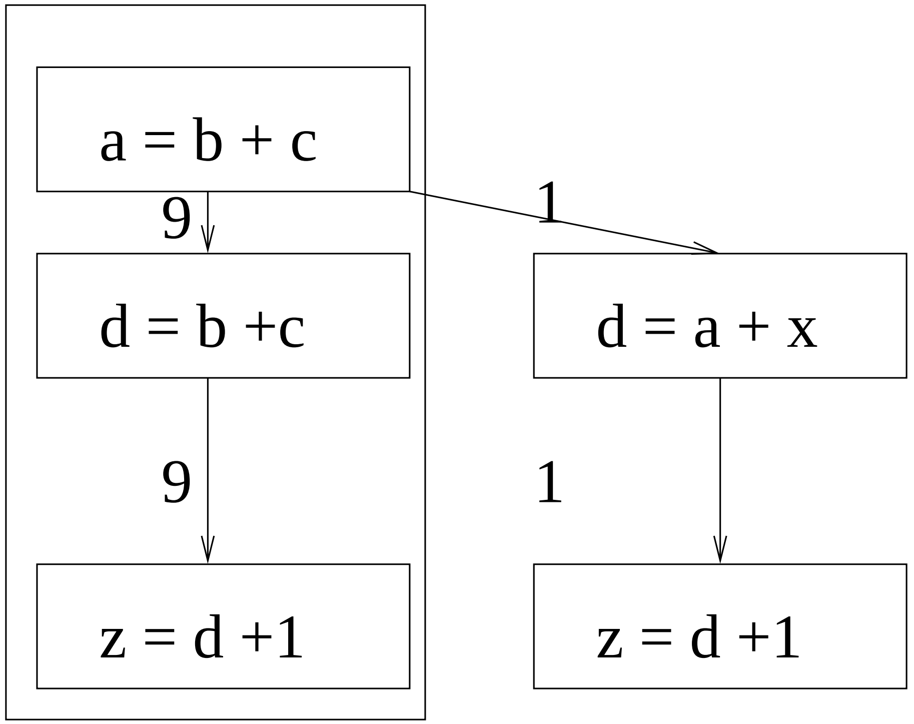


## PDC Example 1



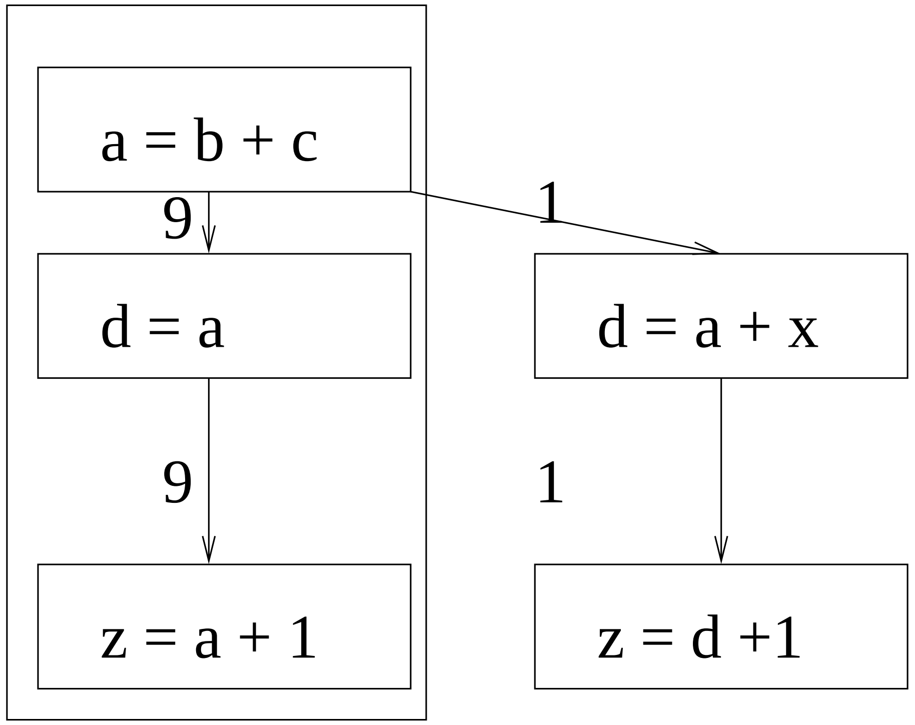
Common b + c on frequently taken path

## PDC Example 2



Replicate first node on main path with external incoming edge
Now separate paths

## PDC Example 3

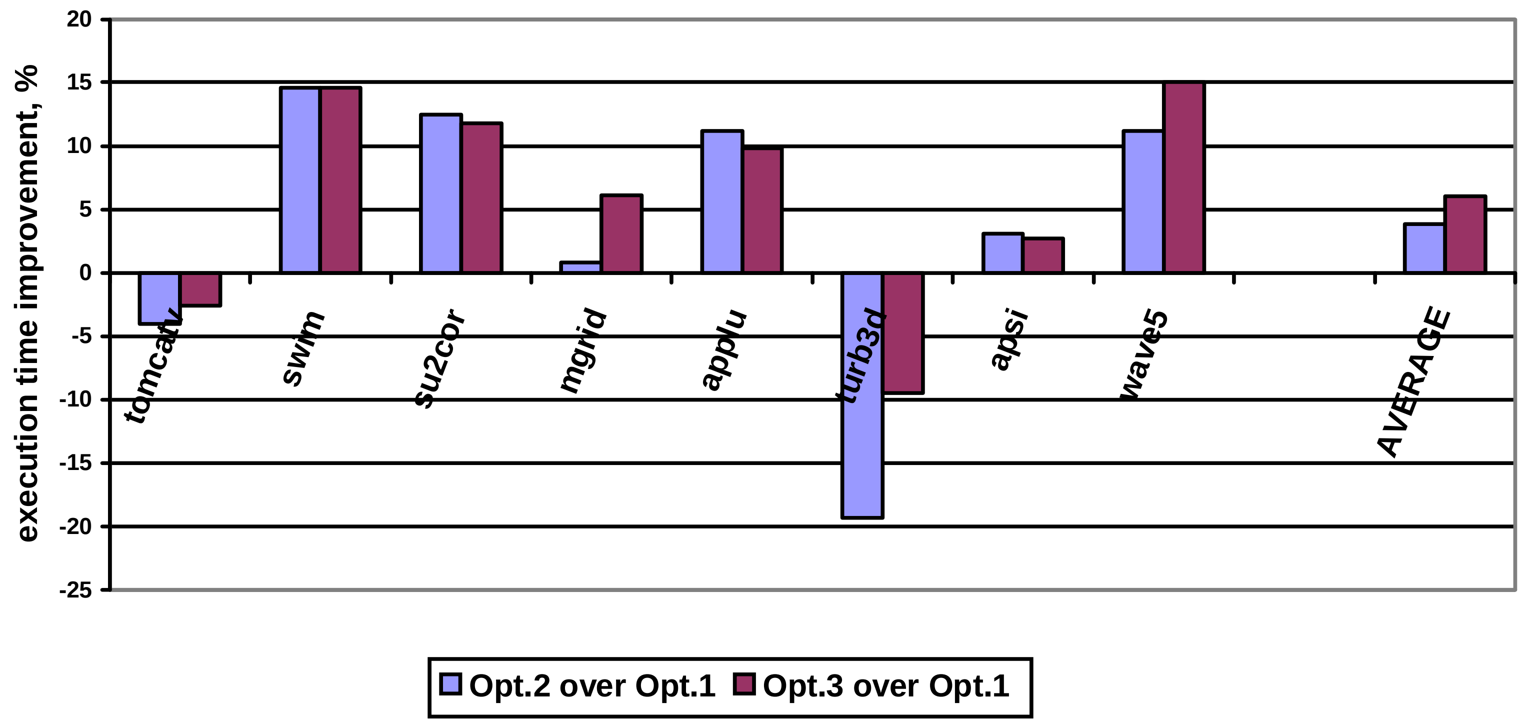


Applying cse eliminates redundant computation at cost of additional code

## Edge vs Path profiling

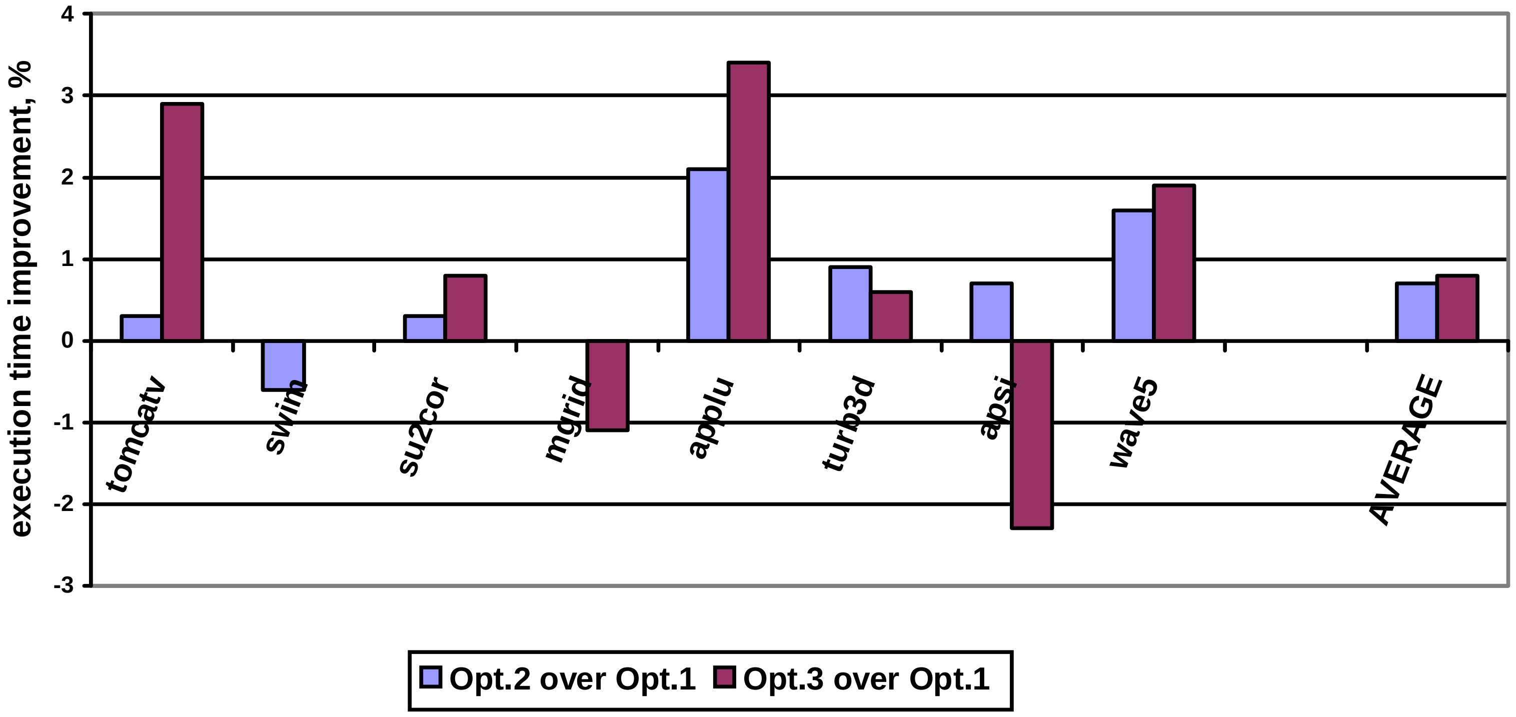
- Overlapping paths cannot be distinguished by edge profiling
- Path profiling allows much greater accuracy
- However, combinatorial explosion in paths. Cycles in graphs leads to potentially unbounded number (e.g.400 + way switch inside a loop in gcc!)
- In practise Edge/node profiling only captures around 40-50
- Larus and Ball '99 developed an efficient path profiler that avoids these problems. In practise the benefit achieved was small though
- Mike Smith at Harvard extended this idea for more targeted optimisation

## So how does this work in practice?



Alpha compiler 21264
Opt 2 feedback directed Opt 3 feedback directed and HLT

## So how does this work in practice?

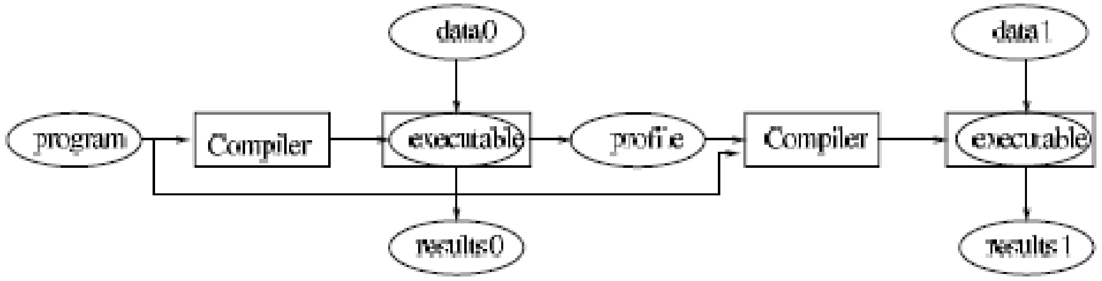


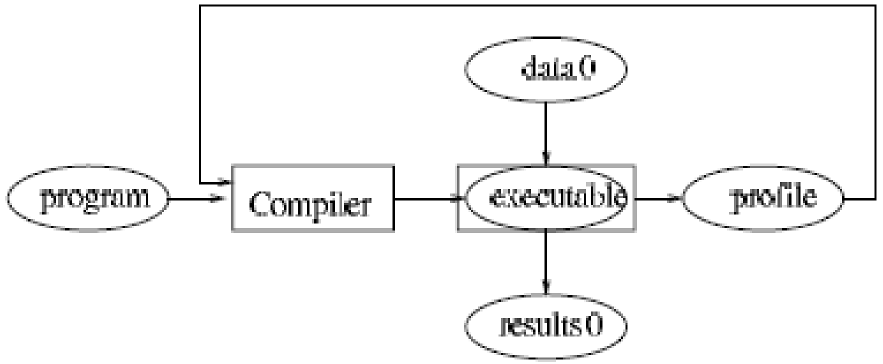
Pentium III on SPEC'95 benchmarks - poor improvement by icc
Extremely well studied benchmarks

## Beyond Path profiling

- Although useful, the performance gains are modest
- Challenge of undecidability and processor behaviour not addressed.
- What happens if data changes on the second run??
- Really focuses on persistent control-flow behaviour
- All other information eg runtime values, memory locations accessed ignored
- Can we get more out of knowing data and its impact on program behaviour?

## Evolution of PDC





PDC with one compile vs. PDC with multiple (iterative) compiles

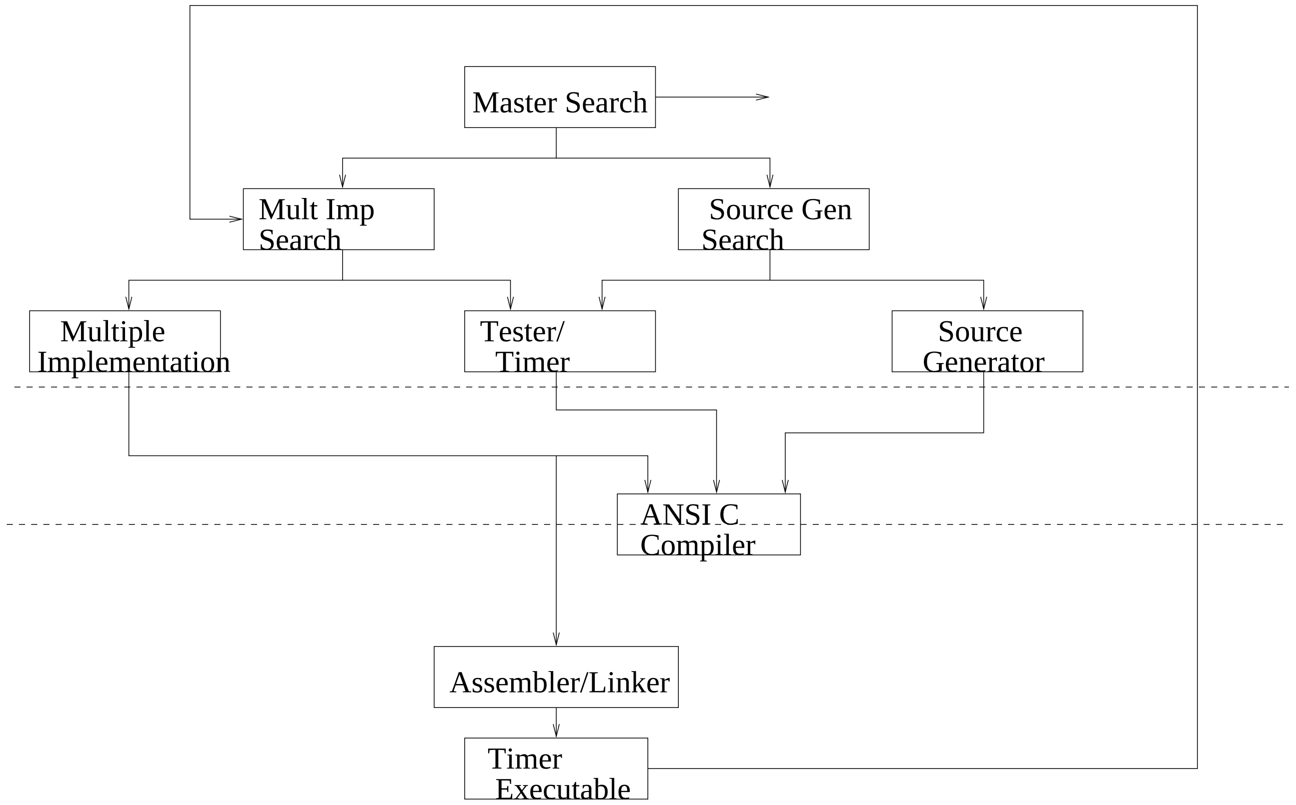
## Automatic Library tuning

- A different off-line approach that exploits knowledge gained by running the program in the optimisation process
- There is a (growing) family of application specific approaches to library tuning
- Rather than recording path information for later optimisation - just record execution time
- Try many different versions of the program and select the best for that machine. Key issue is how different programs are generated.
- In effect move runtime into design time. Main examples ATLAS, PHiPAC and FFTW

## ATLAS

- An automatic method of tuning linear algebraic libraries for differing processors
- It is domain specific and only focuses on tuning the core GEMM routine for a specific processor.
- Takes an ad hoc approach - generate different versions and measure them against anything available - including vendor supplied libraries and pick the best
- It tries different software pipelining and register tiling parameters and enumerates them all, selecting the best. The space of options is derived from explicit knowledge of the application behaviour.

## ATLAS



Broken down into application specific,generic and platform specific sections

## ATLAS

- Regularly outperforms the best existing approaches. Now the standard approach to library generation.
- Adaption?: Very portable - works on any platform AND specialises to the particular processor
- BUT specialised to a particular application - no portability across programs - no exploitation of runtime data as static control-flow
- PHiPAC tries to exploit data patterns in sparse structures by trying simple optimisations off-line and applying them at runtime when data encountered.
- However - domain specific, not generalisable or widely automatable

## Iterative compilation

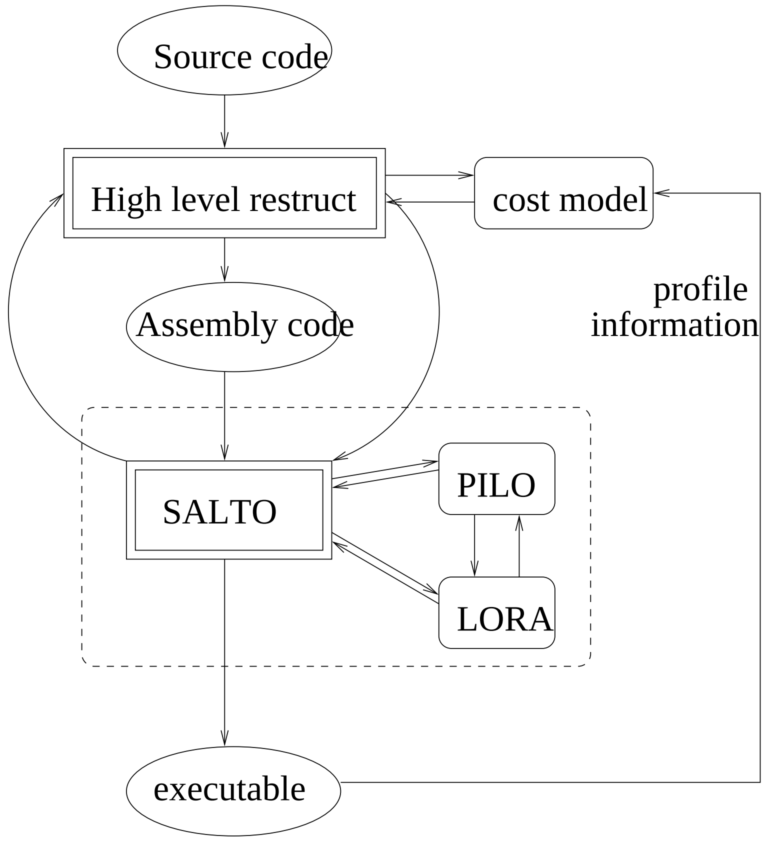
- Iterative compilation really started in 1997 with the OCEANS project
- Similar in spirit to automatic tuning except the space of tuning is in fact the entire program transformation space
- In a sense it is a direct implementation of the formal compiler optimisation problem. Find a transformation T that minimises cost.
- Main ideas was to combine high and low level optimisation and use cost models to guide selection
- Highly ambitious but immature infrastructure prevented much progress

## OCEANS

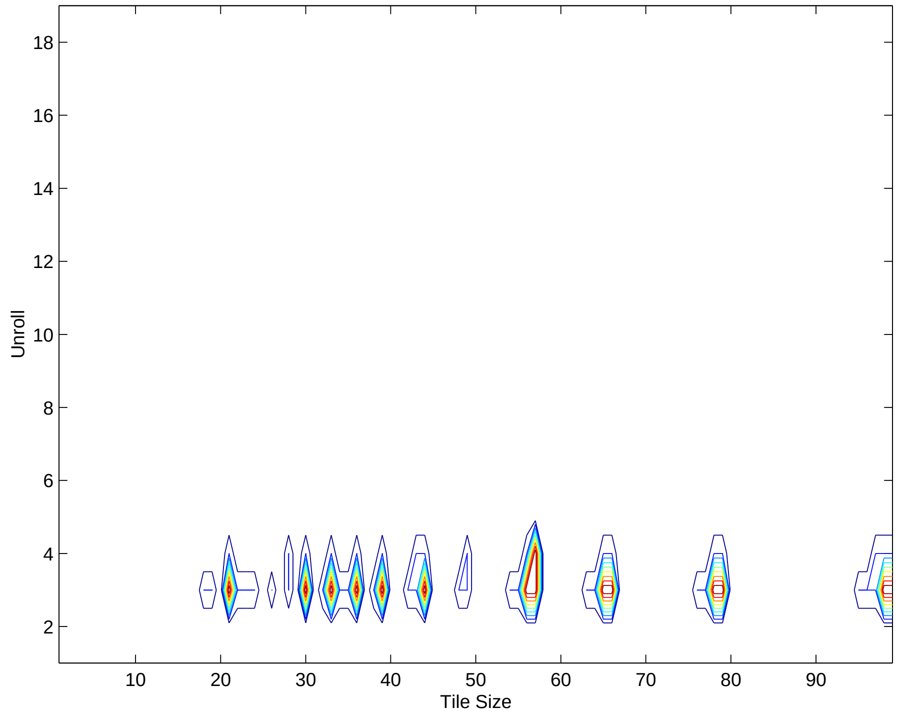
Similar iterative structure to ATLAS.

Novel notion of two communication compiler infrastructures

Main work on searching for best tile and unroll parameters PFDC '98



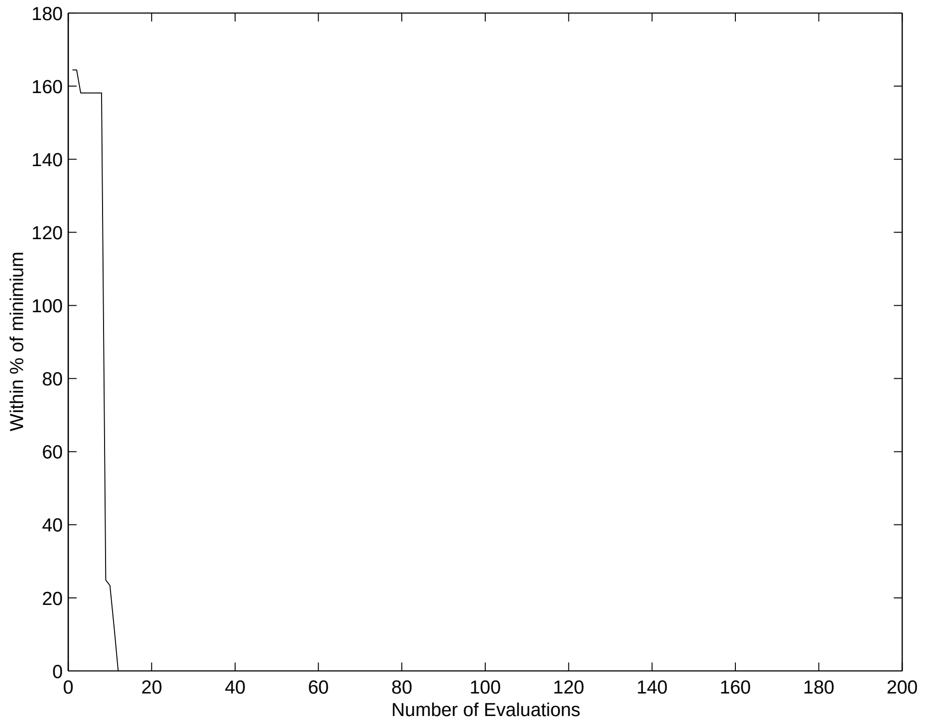
## UltraSparc: space within 20% of minimum $N = 400$



Minimum at: Unroll $=$ 3 and Tile size $=$ 57.

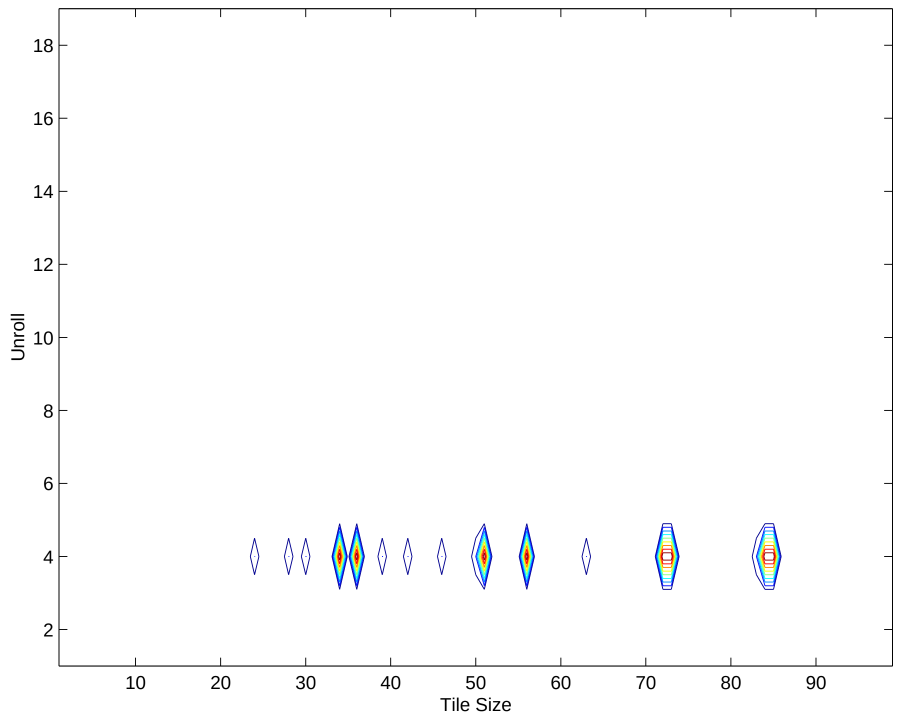
Near minimum: 2.6%. Original 4.99 secs, Minimum 0.56 secs

## UltraSparc $N = 400$



50 steps: within 0.0%. Initially 2.65 times slower than minimum

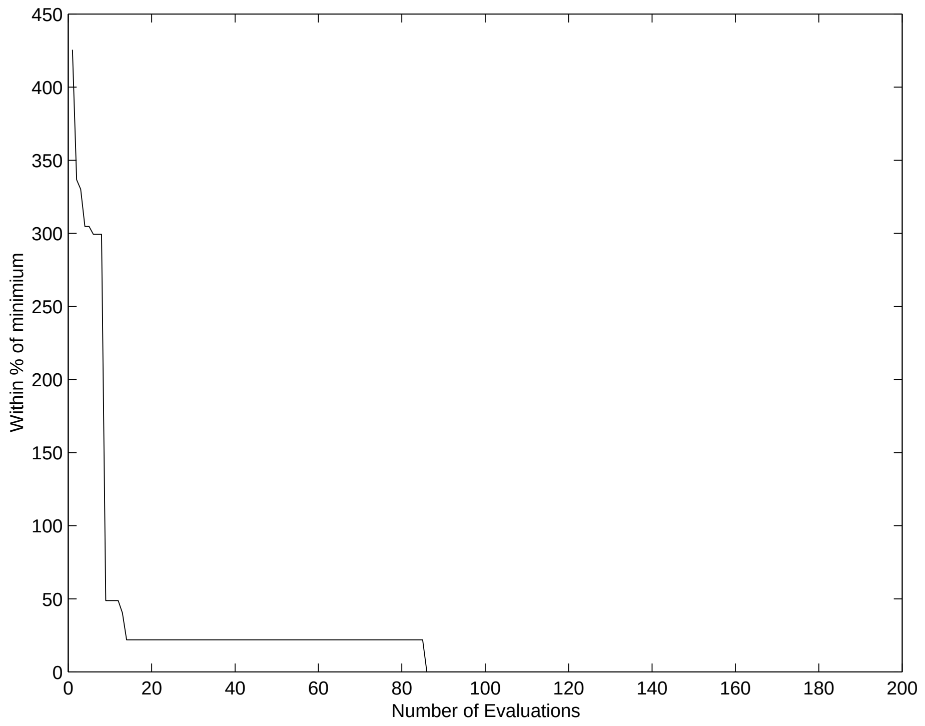
## Alpha: space within 20% of minimum $N = 512$



Minimum at: Unroll $=$ 4 and Tile size $=$ 85.

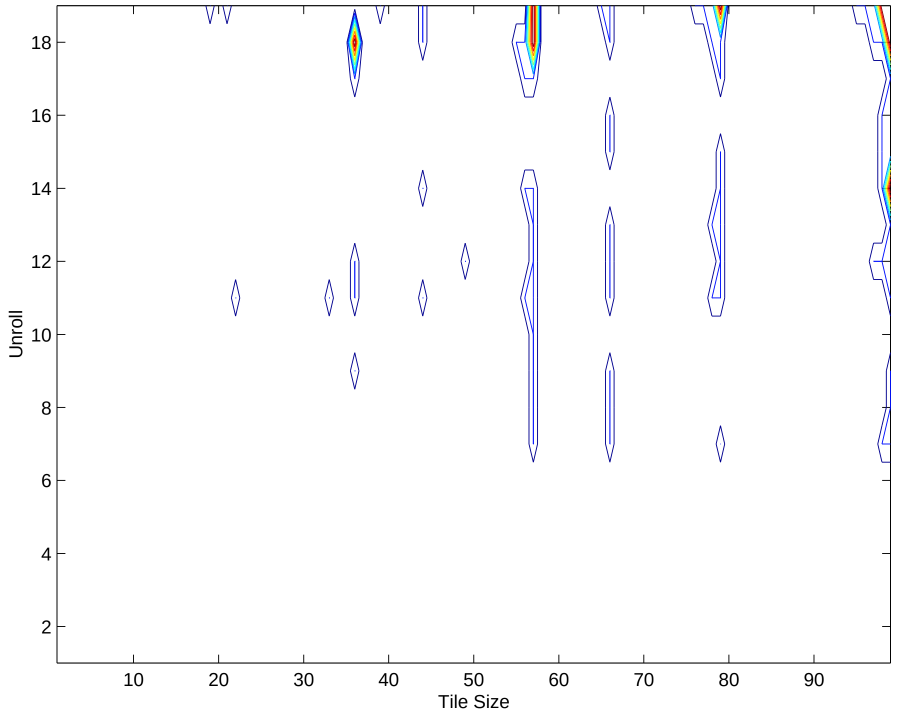
Near minimum: 0.9%. Original 31.72, Minimum 3.34, Max 81.40 !

## Alpha $N = 512$



50 steps: within 21.9%.Originally 5.25 times slower than minimum

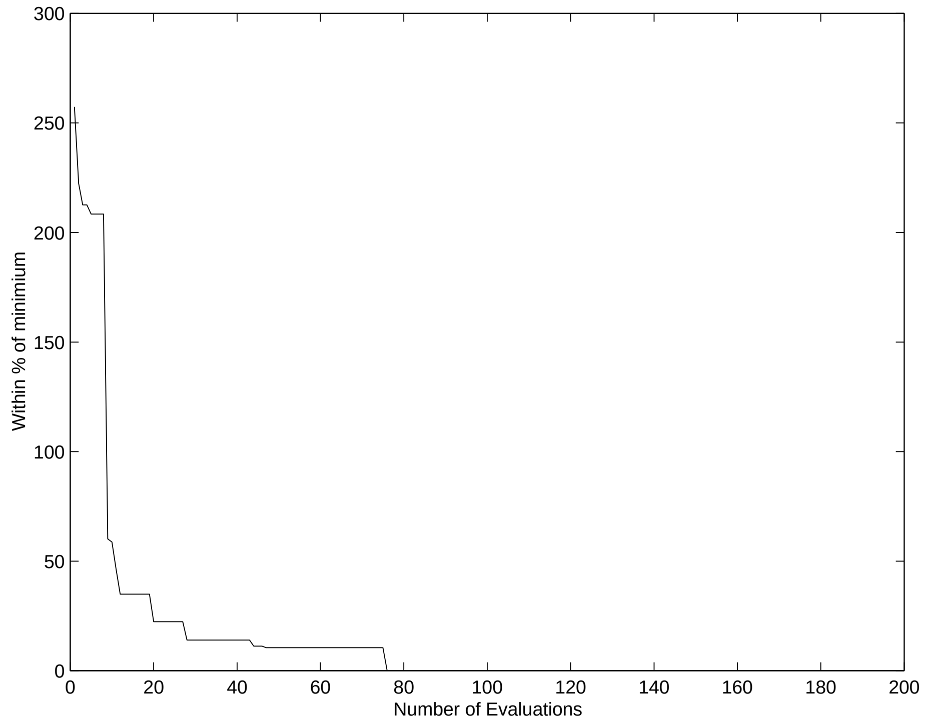
## Pentium Pro: space within 20% of minimum $N = 400$



Minimum at: Unroll = 19 and Tile size = 57.

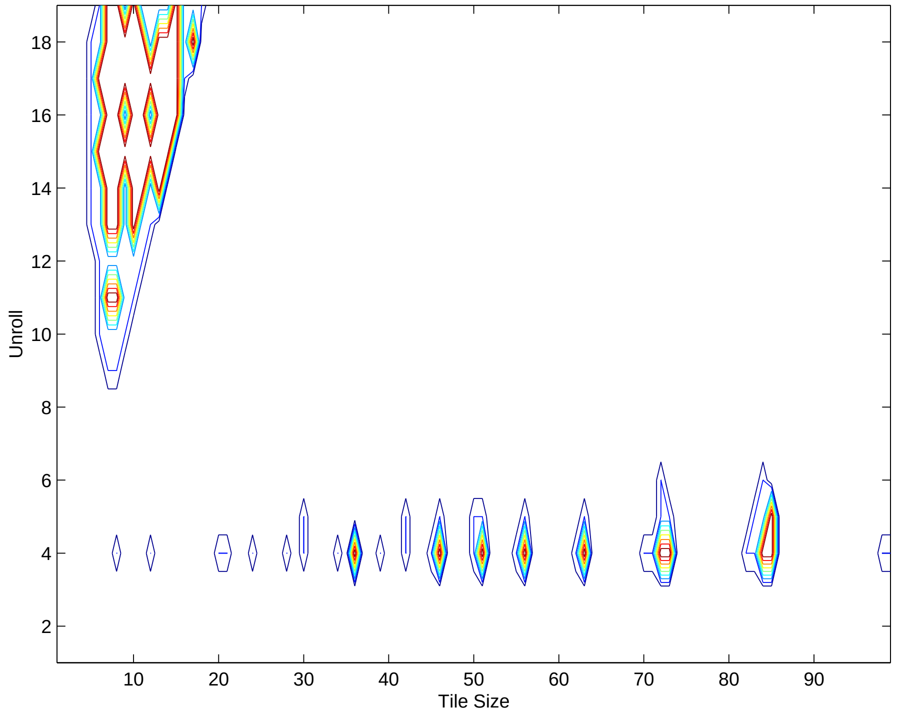
Near minimum: 4.3%. Original 4.88 Minimum 1.43

## **Pentium Pro** $N = 400$



50 steps: within 10.5%.

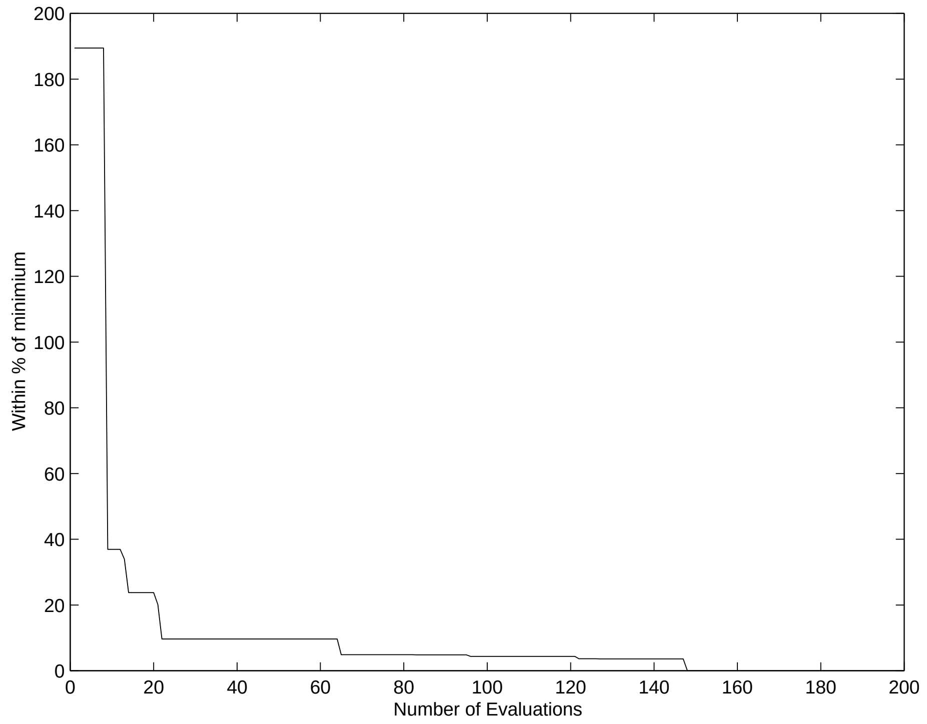
## **R10000:** $N = 512$



Minimum at: Unroll = 4 and Tile size = 85.

Near minimum: 7.2%. Original 2.79, Minimum 1.09

## R10000 $N = 512$

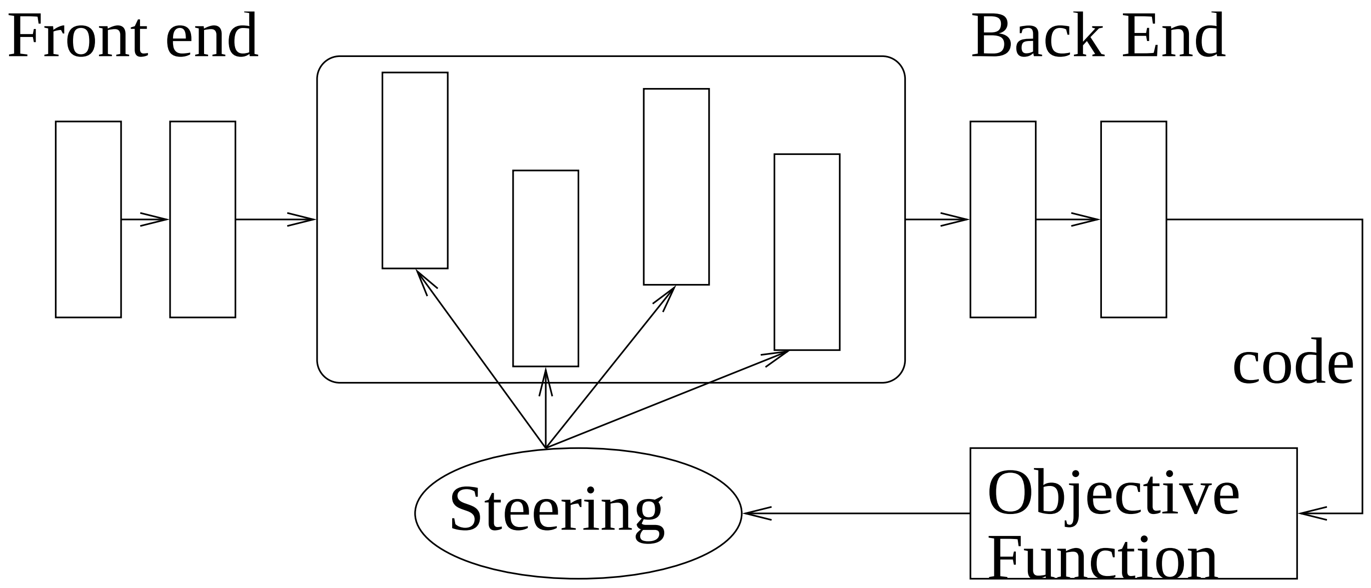


50 steps: within 4.9%.

## Phase Order

- Oceans work looked at parameterised high level search spaces (tiling, unrolling). Restricted by compilers and only small kernel exploration
- Impressive search results due to “tuned” heuristic and small spaces. In practise depends on space shape
- Keith Cooper et al ’99 onwards also looked at iterative compilation
- Cooper’s search space was the orderings of phases within a compiler
- Lower level and not tied to any language. More generic and explores the age-old phase ordering problem more directly

## Phase Order



Cooper has found improvements up to 25% over default sequences.

Examined search heuristics that find good points quickly.

However, evaluation approach is strange and results don't seem portable.

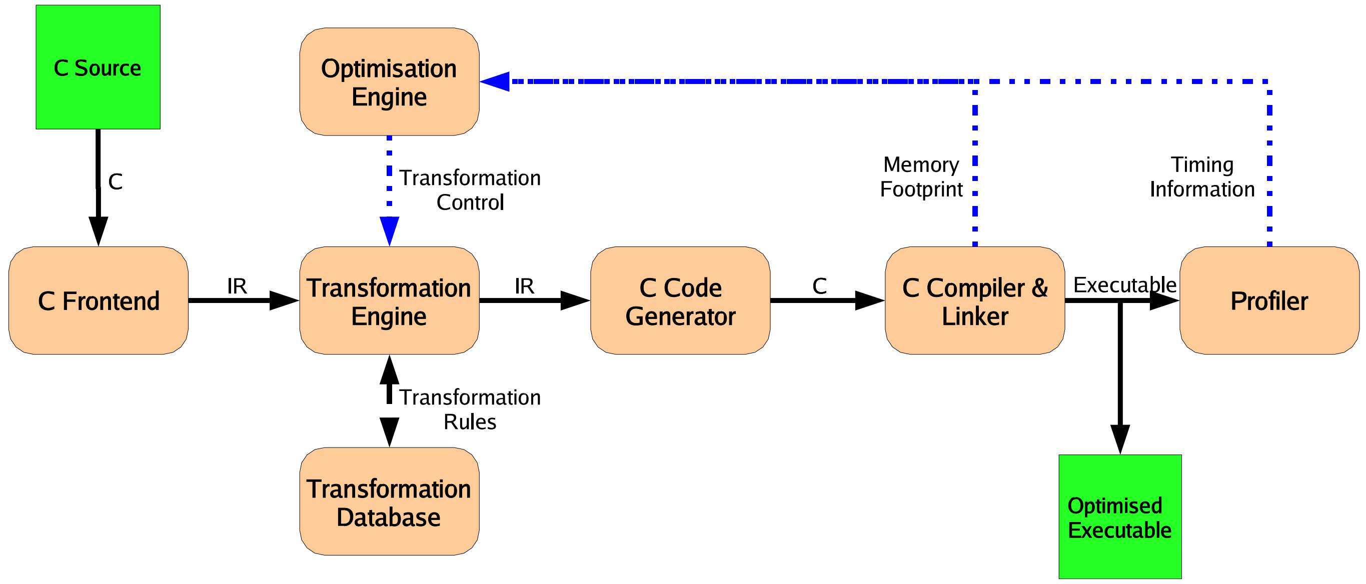
## DSP systems

- Iterative compilation proved to be useful for embedded applications or libraries. Cooper's work presented at embedded forum but not embedded applications

- It is difficult to improve on embedded compilers and hard to get access to internals. HLT is attractive but pointers cause problems

- In Franke et al 2005 we overcome this with a pointer recovery + SUIF based transformation explorer. Uses 2 search strategies.

## DSP Framework

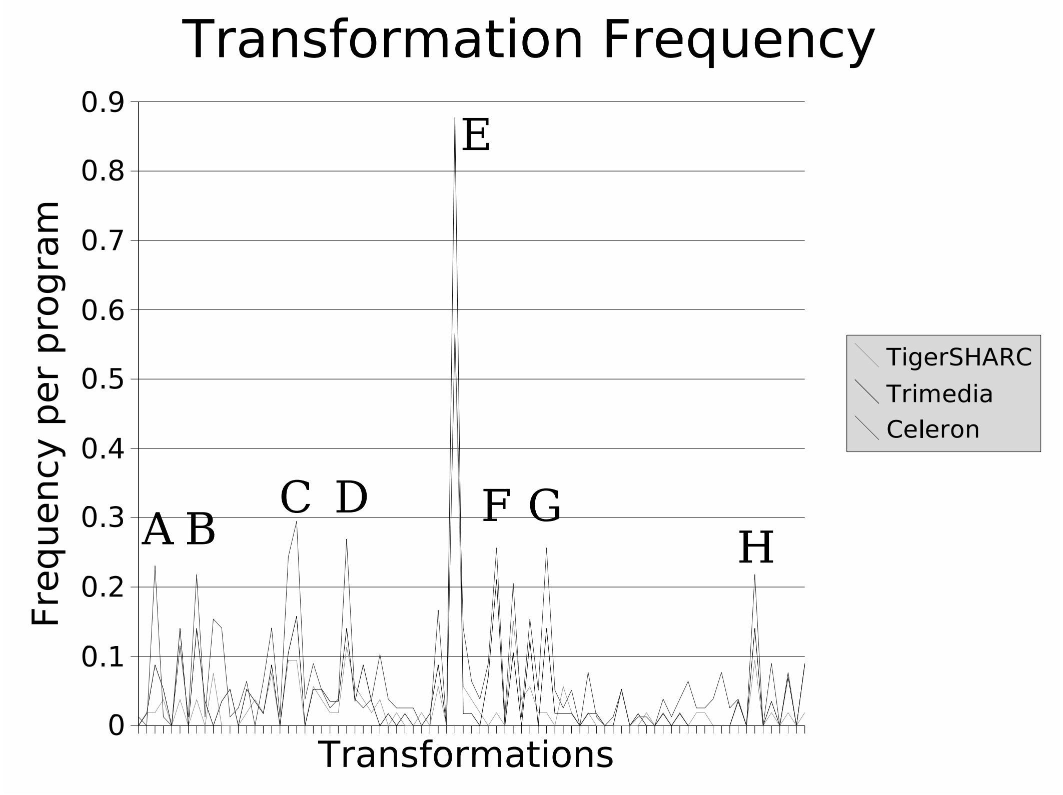


Using this framework to exhaustively explore the space and characterise

## Franke

- Looks through space of $80^{80}$ transformations on 3 platforms for UTDSP benchmark suite. Not feasible to do exhaustively. Really stresses SUIF
- 2 algorithms. Trade-off between coverage and focus.Random - select a random length up to 80. Then randomly select any transformation for each location. Lots of redundant transformations.
- PBIL:Population based inference learning. Modify probability of selecting transformation based on previous trials. Only examine effective transformations
- Average 41% reduction. PBIL finds the best in majority of cases but Random best has a higher speed up.

## Impact of Transformations



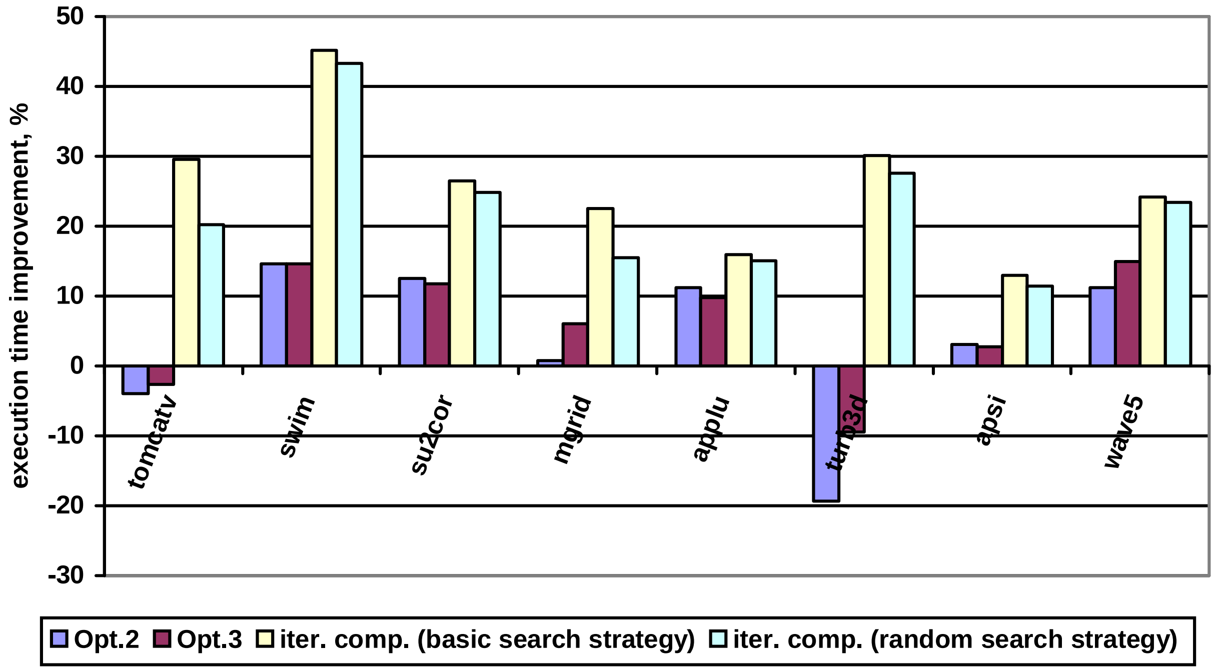
## Results

- Tried 500 runs. On UTDSP benchmark: TriMedia average speedup of 1.43 and 1.73 for TigerSharc
- Shows that HLT can give a big win compared to backend optimisations
- Also compared GCC and ICC on embedded Celeron
- Original: ICC 1.22 faster than GCC
- GCC + it: speedup of 1.54 - better than ICC
- BUT ICC + it: speedup of 2.14

## Search Speed

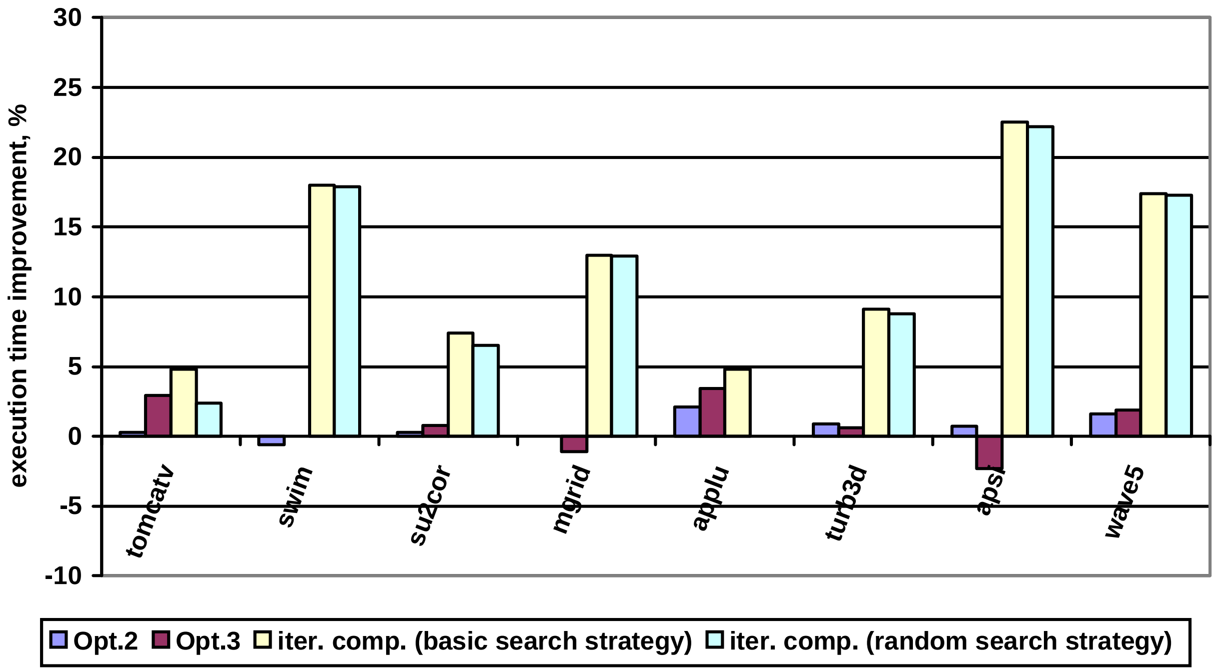
- The main problem is optimisation space size and speed to solution
- Many use a cut down transformation space - but this just imposes ad hoc non portable bias
- Need to have a large interesting transformation space. Orthogonal - no repetition. SUIF is ad hoc. UTF framework from Shun et al 2004 very systematic but doesn't cover everything
- Build search techniques to find good points quickly

## Search Speed Alpha



[Fursin 2002] uses phase order, profile info and potential benefit to reduce search

## Search Speed Pentium



Reduces number of searches by orders of magnitude. Currently investigating increasing number of searches per run

## Using models

- Obvious approach is to use cheap static modes to help reduce number of runs
- Difficulty is to balance savings gained by model against hardwiring strategy
- Wolfe and Mayadan generate many versions of a program and check against an internal cache models rather than generate the best by construction
- Although more successful doesn't address problem of processor complexity.No real feedback (Pugh A* search ). Cannot adapt
- Knijnenburg et al PACT 2000 use simple cache models as filters. Used to eliminate bad options rather than as a substitute for feedback. Significant speed up

## Search space

- Understanding the shape or structure of search space is vital to determining good ways to search it
- Unfortunately little agreement. FDO showed large number of minima with structure. Vuduc '99 shows that minima dramatically vary across processor
- Cooper shows that reasonable minima are very near any given point.
- However, our recent work shows that it strongly depends on scenario. Rich space on a TriMedia -golf green on the TI. Should use structure to aid search
- Vuduc uses distribution of good points as a stopping criteria. Fursin use upper bound of performance as a guide.

## Taxonomy based evaluation

- All techniques move runtime either into compile or design time. Application tuning is not portable across programs.
- Iterative compilation allows great adaption to and specialisation to a processor than PDC.
- However, over specialises to a data set. Makes sense when the behaviour of a program is relatively data independent in the case of linear algebra or DSP programs.
- Excessive design/compile time means only currently suitable for embedded apps or libraries.

## Summary

- Profile directed compilation uses a light weight approach to guide primarily scheduling decisions.
- Automatic tuning used as application specific way to achieve portable high performance
- Iterative compilation broadens this for general purpose programs
- All techniques examined are off-line approaches where optimisation happens prior to 'production' run
- Next focus on on-line dynamic approaches which aim to optimise on the fly

## Dynamic Compilation

## Overview

- Specialisation
- Dynamic code generation - DyC
- Dynamic Binary Translation - DAISY
- Just in time compilation -Jikes
- Critical evaluation and conclusion

## Summary

- Introduced profile directed compilation, program tuning and iterative compilation
- All used runtime behaviour at compile/design time to select better transformations
- Trade-off in number of runs vs eventual performance
- Iterative techniques very good at porting and specialising to new platforms
- However, all rely on eventual on-line runtime data to be same as that visited off-line. Poor at adapting to new data

## Dynamic techniques

- All today's techniques focus on delaying some or all of the optimisations to runtime
- This has the benefit of knowing the exact runtime control-flow, hotspots, data values, memory locations and hence complete program knowledge
- It thus largely eliminates many of the undecidable issues of compile-time optimisation by delaying until runtime
- However, the cost of analysis/optimisation is now crucial as it forms a runtime overhead. All techniques characterised by trying to exploit runtime knowledge with minimal cost

## Background

- Delaying compiler operations until runtime has been used for many years
- Interpreters translates and execute at runtime
- Languages developed in the 60s eg Algol 68 allowed dynamic memory allocation relying on language specific runtime system to mange memory
- Lisp more fundamentally has runtime type checking of objects
- Smalltalk in the 80s deferred compilation to runtime to reduce the amount of compilation otherwise required in the 00 setting
- Java uses dynamic class loading to allow easy upgrading of software

## Runtime specialisation

- For many, runtime optimisation is “adaptive optimisation”
- Although wide range of techniques, all are based around runtime specialisation. Constant propagation is a simple example.
- Specialisation is a technique that has been used in compiler technology for many years especially in more theoretical work
- Specialising an interpreter with respect to a program gives a compiler
- Can we specialise at runtime to gain benefit with minimal overhead? Statically inserted selection code vs parameterised code vs runtime generation.

## Static code selection, parameterised and code generation

```
IF (N<M) THEN
 DO I =1,N
  DO J =1,M
   ...
  ENDDO
 ENDDO
ELSE
 DO J =1,M
  DO I =1,N
   ...
  ENDDO
 ENDDO
ENDIF
```

```
IF (N<M) THEN
 U1 = N
 U2 = M
ELSE
 U1 = M
 U2 = N
ENDIF
DO I1 =1,U1
  DO I2= 1,U2
   ...
  ENDDO
ENDDO
```

```
gen_nest1(fp,N,M)
(*fp)()
```

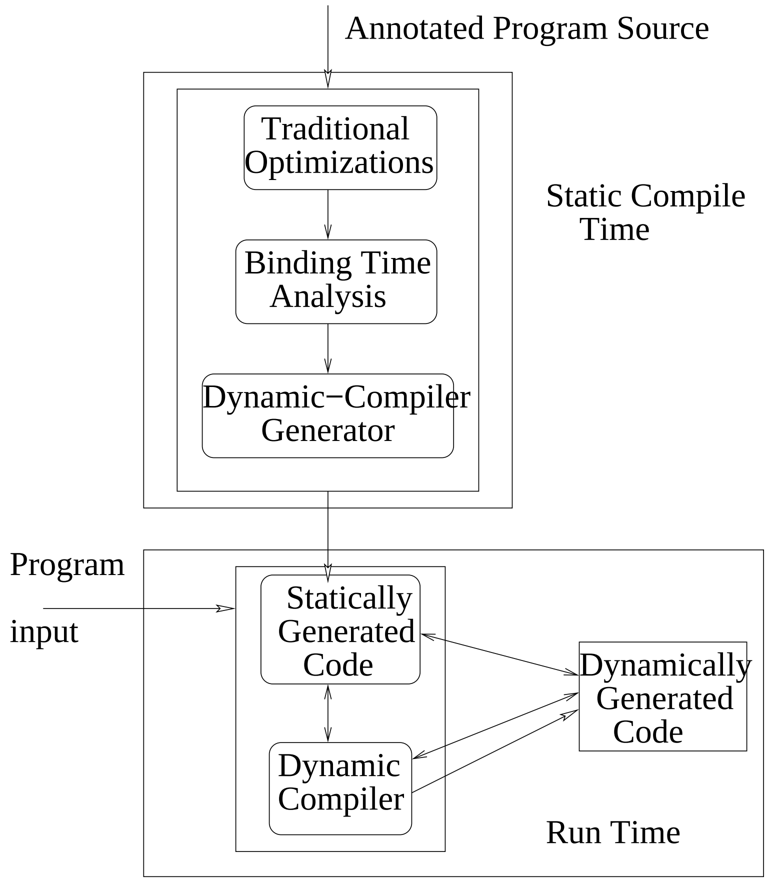
## DyC

- One of the best known dynamic program specialisation techniques based on dynamic code generation.
- The user annotates the program defining where there may be opportunities for runtime specialisation. Marks variables and memory locations that are static within a particular scope.
- The system generates code that checks the annotated values at runtime and regenerates code on the fly.
- By using annotation, the system avoids over-checking and hence runtime overhead. This is at the cost of additional user overhead.

## DyC

Binding analysis
examines all uses
of static variables
within scope

Dynamic compiler
exploits invariancy
and specialises the code
when invoked


Annotated Program Source
Traditional Optimizations
Binding Time Analysis
Dynamic−Compiler Generator
Static Compile Time
Program input
Statically Generated Code
Dynamic Compiler
Dynamically Generated Code
Run Time

## DyC Results

- Asymptotic speedup and a range programs varies from 1.05 to 4.6
- Strongly depends on percentage of time spent in the dynamically compiled region. Varies from 9.9 to 100 %
- Low overhead from 13 cycles to 823 cycles per instruction generated
- However relies on user intervention which may not be realistic in large applications
- Relies on user *correctly* annotating the code

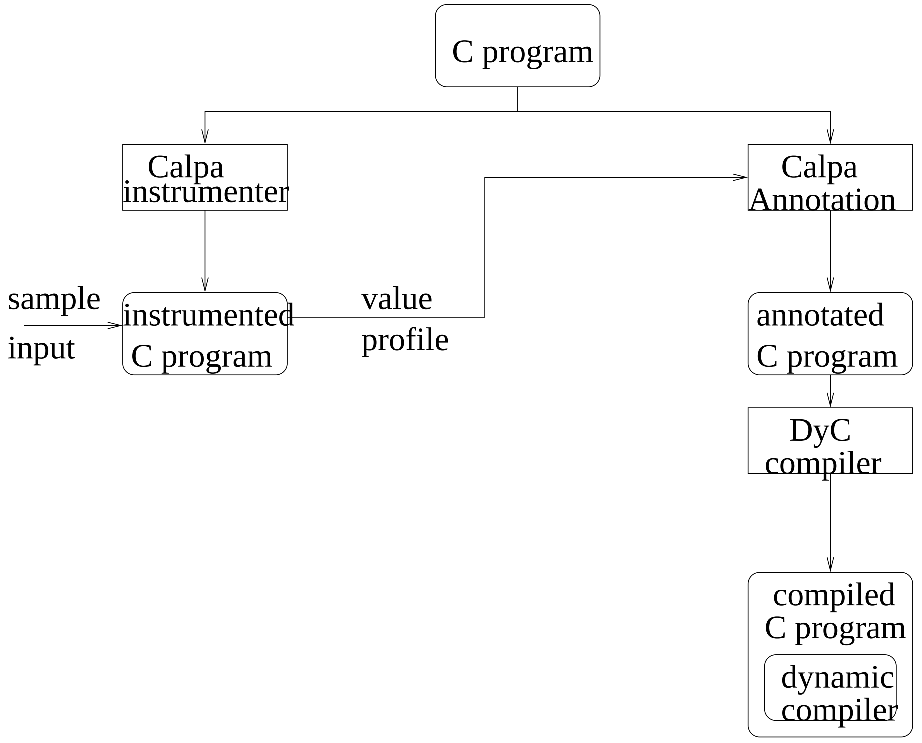
## Calpa for DyC

- Calpa is a system aimed at automatically identifying opportunities for specialisation without user intervention
- It analyses the program for potential opportunities and determines the possible cost vs the potential benefit
- For example if a variable is multiplied by another variable which is known to be constant in a particular scope, then if this is equal to 0 or 1 then cheaper code maybe generated
- If this is inside a deep loop then a quick test for 0 or 1 outside the loop will be profitable

## Calpa for DyC

Calpa is a front end to DyC

It uses instrumentation to guide annotation insertion



## Calpa for DyC

- Instruments code and sees have often variables change value. Given this data determined the cost and benefit for a region of code
- Number of different variants, cost of generating code, cache lookup. Main benefit determined by estimating new critical path
- Explores all specialisation up to a threshold. Widely different overheads 2 seconds to 8 hours. In two cases improves - from 6.6% to 22.6%
- Calpa and DyC utilise selective dynamic code generation. Now look at fully dynamic schemes.

## Dynamic Binary Translation

- The key idea is to take one ISA binary and translate it into another ISA binary at runtime.
- In fact this happens inside Intel processors where x86 is unpacked and translated into an internal RISC opcode which is then scheduled. The TransMeta Crusoe processor does the same. Same with IBM legacy ISAs.
- Why don't we do this statically? Many reasons!
- The source ISA is legacy but the processor internal ISA changes. It is impossible to determine statically what is the program. It is not legal to store a translation. It can be applied to a local ISA for long term optimisation

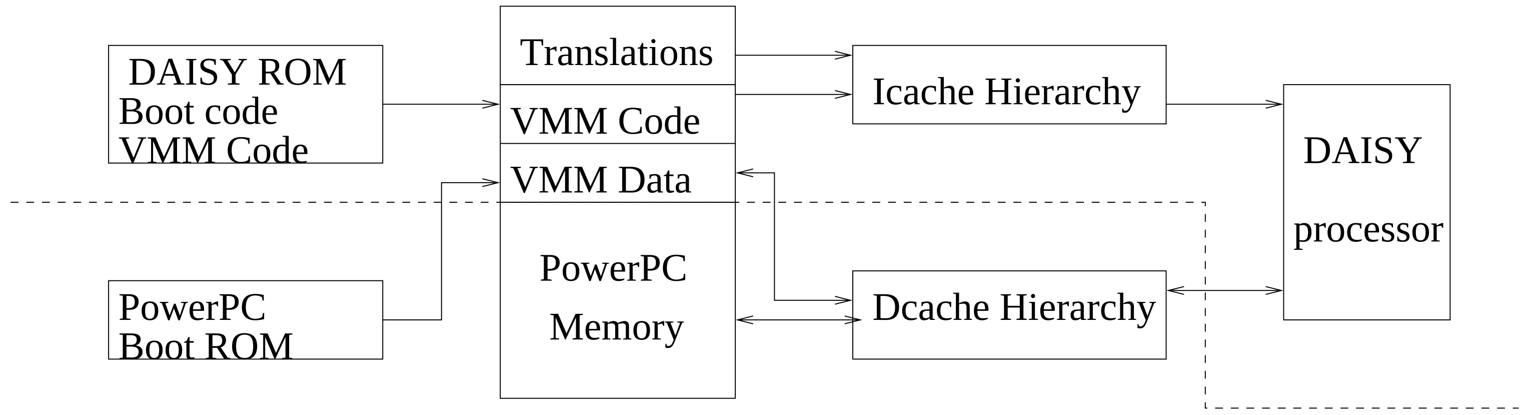
## DAISY

- One of the best known schemes came out of IBM headed by Kemal Ebcioglu
- Aimed at translating PowerPC binaries to the IBM vliw machine
- Idea was to have a simple powerful in-order machine with a software layer handling complexities of PowerPC ISA
- Dynamic translation opens up opportunities for dynamic optimisation.
- Concerned for industrial strength usage. Exceptions, self-modifying code etc.

## DAISY

- At runtime, program path and data known. But need a low overhead scheme to make worthwhile
- Specialisation happens naturally as we know runtime value of variables
- Can bias code generation to check for profitable cases
- DAISY uses a code cache of recently translated code segment.
- Automatic superblock formation and scheduling

## DAISY structure overview



Power PC code runs without modification

DAISY specific additions separated by dotted line.

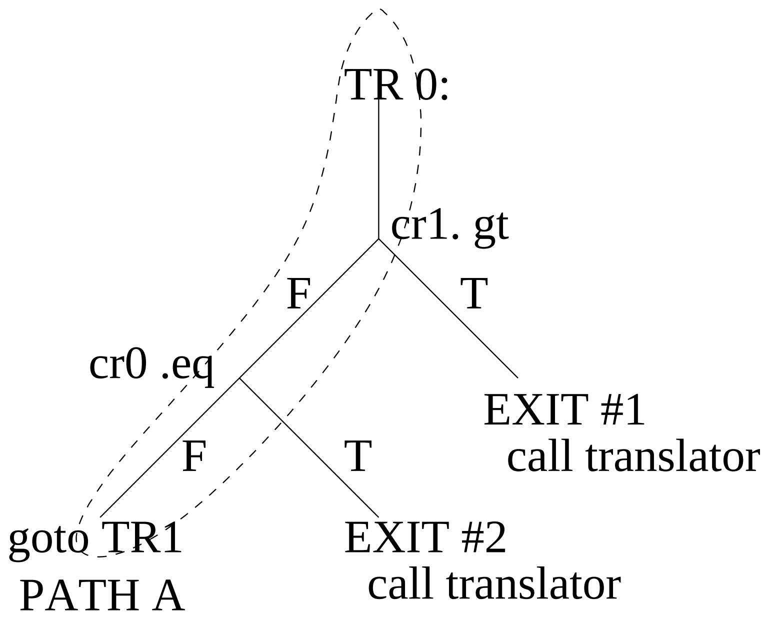
Initially interpret PowerPC instructions and then compile after hitting threshold

Then schedule and save instruction in cache (2-4k). Untaken branches are translated as (unused) calls to the binary translator

## DAISY

Here the group
is expanded
to contain two
conditionals

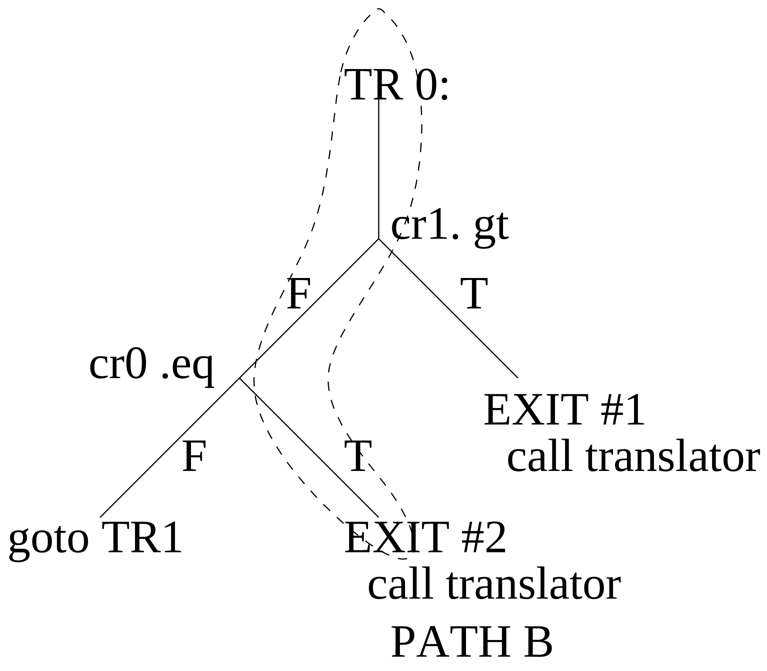
Path A is
encountered and
translated



## DAISY

When Path B
is encountered
for the first
time

Translator is called

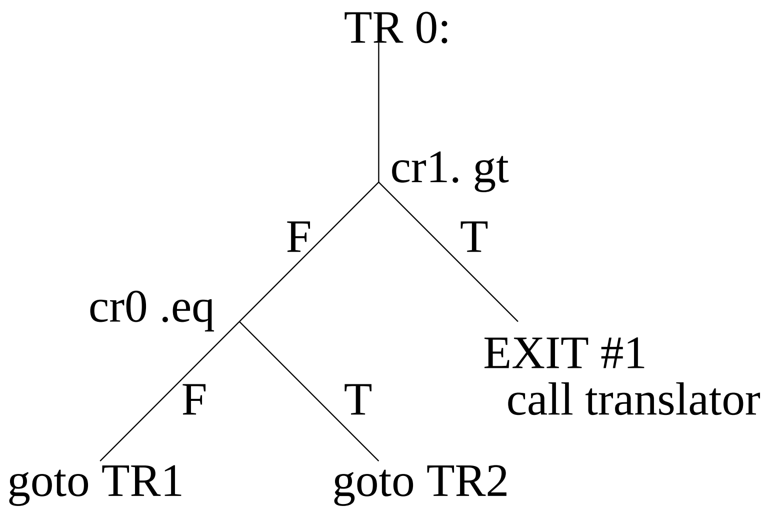


## DAISY

Code in cache
is now updated

Paths A and B
require no further
translation

One untranslated
path remaining



Only translate and store code if needed

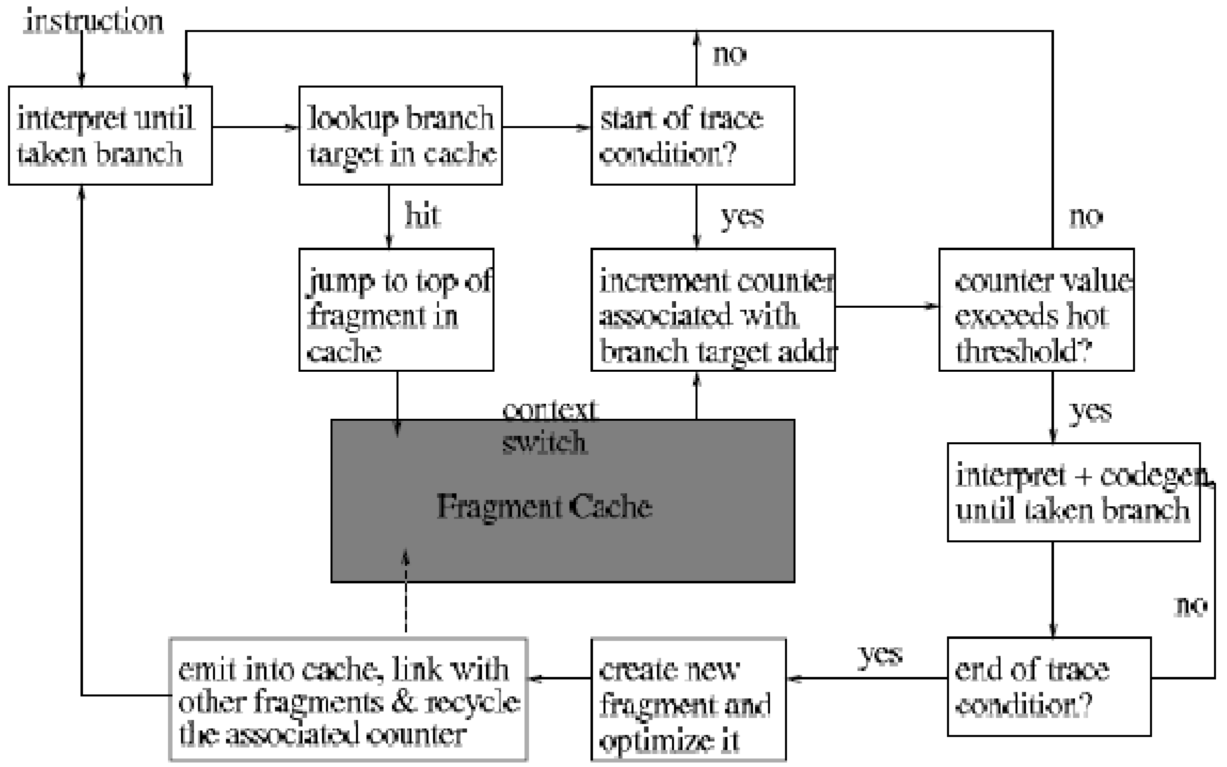
## DYNAMO

- Similar to Daisy though focuses on binary to binary optimisations on the same ISA. One of the claims is that it allows compilation with -01 but overtime provides -03 performance.
- Catches dynamic cross module optimisation opportunities missed by the static compiler. Code layout optimisation allowing improved scheduling due to bigger segments. Branch alignment and partial procedural inlining form part of the optimisations
- Aimed as way of improving performance from a shipped binary overtime
- Unlike DAISY, have to use existing hardware - no additional fragment cache available

## DYNAMO

- Initially interprets code. This is very fast as the code is native. When a branch is encountered check if already translated
- If it has been translated jump and context switch to the fragment cache code and execute. Otherwise if hot translate and put in cache
- Over time the working set forms in the cache and Dynamo overhead reduces - less than 1.5
- Cheap profiling, predictability and few counters are necessary
- Linear code structure in cache makes optimisation cheap. Standard redundancy elimination applied

## Evolution of PDC


instruction
interpret until taken branch
lookup branch target in cache
start of trace condition?
no
hit
yes
no
jump to top of fragment in cache
increment counter associated with branch target addr
counter value exceeds hot threshold?
context switch
yes
Fragment Cache
interpret + codegen until taken branch
no
yes
emit into cache, link with other fragments & recycle the associated counter
create new fragment and optimize it
end of trace condition?

PDC with one compile vs. PDC with multiple (iterative) compiles

## Just in Time Compilation

- Key idea: lazy compilation. Defer compiling a section of high level code until it is encountered during program execution. For OO programs it has been shown that this greatly reduces the amount of code to compile. Krintz'00 shows 14 to 26% reduction in total time.

- Greater knowledge of runtime context allowing optimisation to be focused on important parts of program

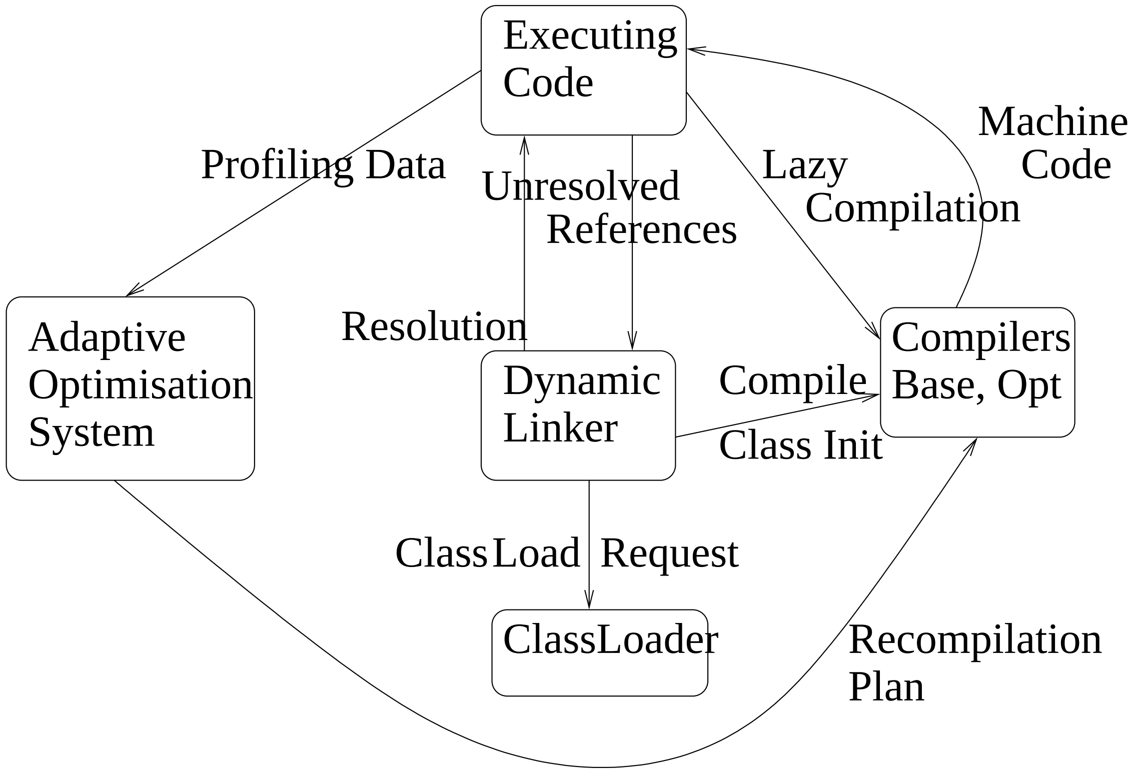
- However is Just in time really Just too late? Why wait until execution time to compile when the code may be lying around on disk for months beforehand.

- Main reason - dynamic linking of code especially in Java. This restricts the optimisations available

## Jikes

- Most Java compilers initially interpret, then compile and finally optimise based on frequency of use
- Normally done on a per method basis -exception Whaley 2000 - very similar to DBT.
- Jikes instead directly compiles code when encountered to native machine code
- Well known robust research compiler freely available
- Much work centred around what level of optimisation to apply and when to apply it.

## Jikes



Main interacting components

## Jikes Example

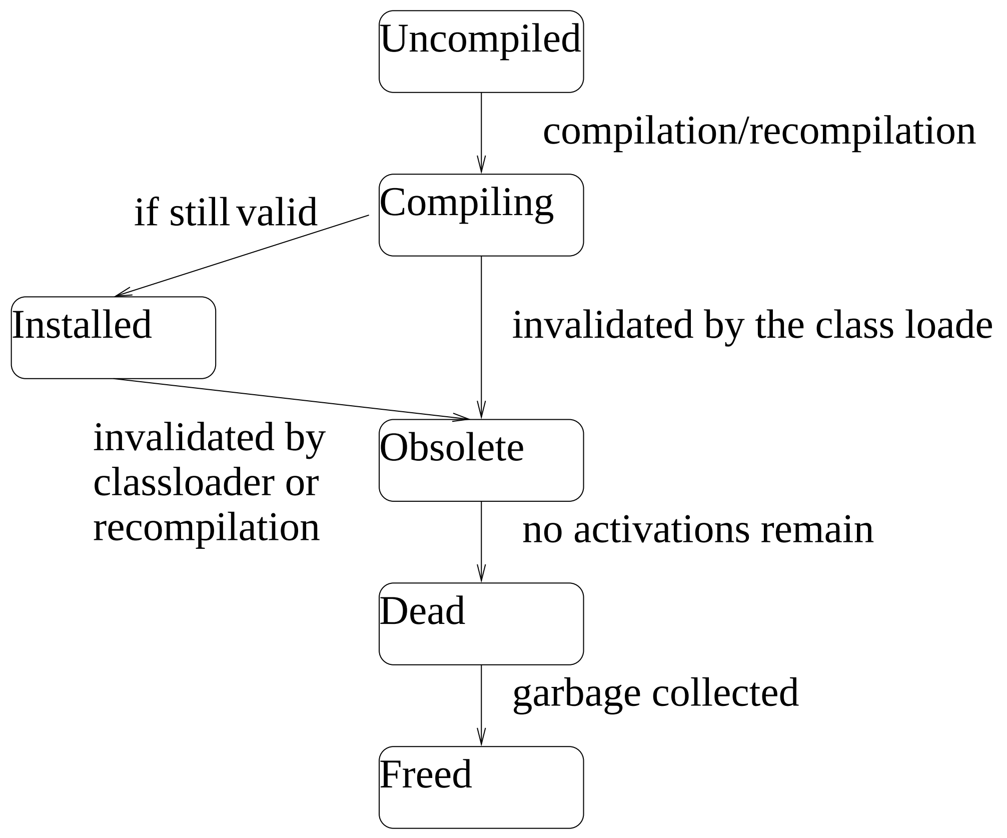
```
iload x          INT_ADD tint,xint,5      INT_ADD yint,xint,5
iconst 5         INT_MOVE yint,tint
iadd
istore y
```

Simple example showing translation of byte code into native code

Simple optimisations to remove redundant temporaries have a significant impact on later virtual to register mapping phases .

First version corresponds to baseline compiler, second to most basic optimising compilation

## Method Life Cycle


Uncompiled
compilation/recompilation
Compiling
if still valid
Installed
invalidated by the class loade
invalidated by classloader or recompilation
Obsolete
no activations remain
Dead
garbage collected
Freed

## Jikes

- Jikes makes use of multiple optimisation levels and uses these to carefully trade off cost vs gain.
- Baseline translates directly into native code simulating operand stack. No IR, no reg alloc. Slightly faster code than interpretation
- Optimising compiler. Translate into an IR with linear register allocation. 3 further optimisation levels.

## Jikes

- Level 0. Effective and cheap optimisations. Simple scalar optimisations and inlining trivial methods. All tend to reduce size of IR
- Level 1 as 0 but with more aggressive speculative inlining. Multiple passes of level 0 opts and some code reorganising algs.
- Level 2 employs simple loop optimisations. Normalisation and unrolling. SSA based flow-sensitive algorithms also employed.

**Jikes**

| Compiler | Bytecodes/ Millisecond | Speed |
|---|---|---|
| Baseline | 377.8 | 1.0 |
| Level 0 | 9.29 | 4.26 |
| Level 1 | 5.69 | 6.07 |
| Level 2 | 1.81 | 6.61 |

Only worthwhile compiling at a higher level if benefit outweighs cost

Adaptive algorithm compares cost of code under current level vs an increased level.

Crucially depends on anticipated future profile which is unavailable. Solution - just guess - currently assume twice as long as now!

## Jikes

Krintz evaluates the adaptive approach

| Compiler | Total Time | Compile Time |
|---|---|---|
| Baseline | 29.24 | 0.44 |
| Opt | 9.98 | 0.36 |
| Adapt | 8.97 | 0.48 |

Figures are time in seconds for SPECjvm98

Total time is better fr Adapt even though it has increased compile-time.

Conclusion - knowing hotspots really helps optimisation

## Java JIT

- JITs suffer from having the necessary info too late. Need to anticipate optimisation opportunities.
- Many different optimisation scenarios available. Adaptive option increases level of optimisation when it recompiles increasingly used hotspots.
- As compile-time is part of runtime, trade-off between two depends on program type

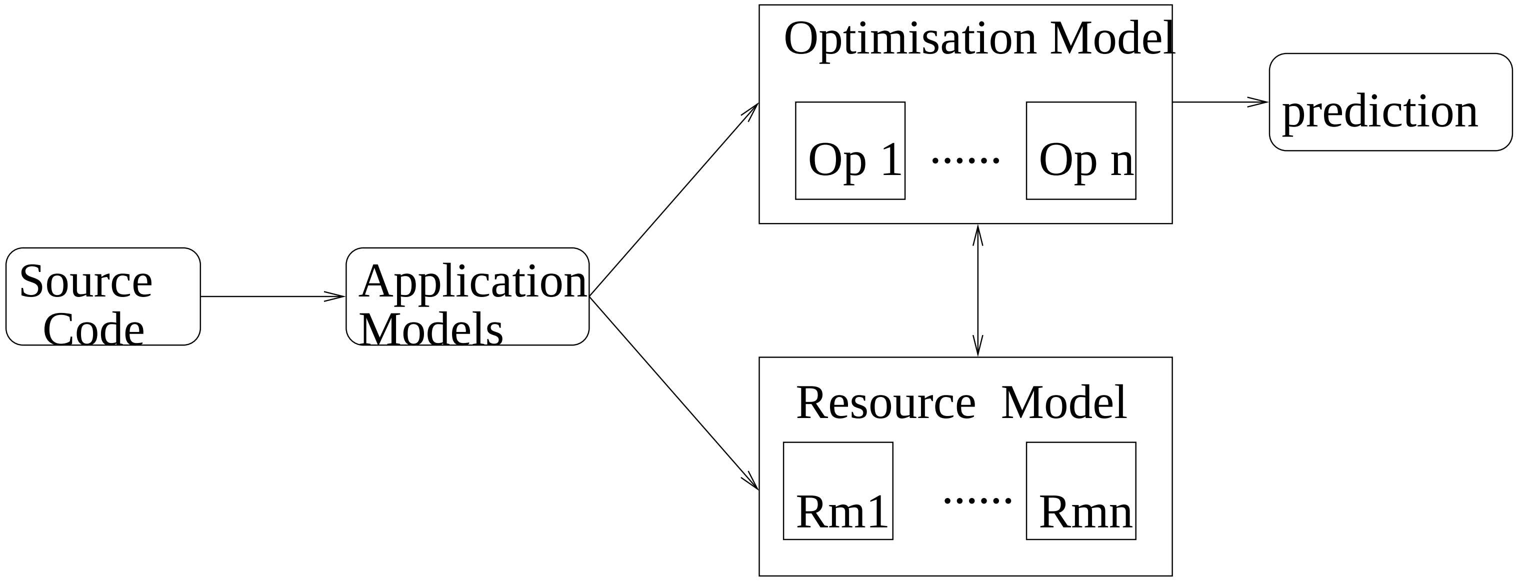
## ADAPT

- ADAPT is a mixed approach to optimisation that combines static and iterative compilation in an on-line manner.
- Basically at runtime different options of a code section are run concurrently and the best-one selected. This is done in parallel on remote servers.
- Really trading space for time making an off-line technique viable as an on-line technique as long as sufficient space available
- Online iterative compilation main contribution.
- Only works for scientific programs with relatively static behaviour

## Continuous compilation

- Obvious question: can we combine off-line and on-line techniques getting the best from both worlds?
- Childer and Soffa propose continuous compilation. Static compilation followed by dynamic compilation.
- Results are stored for off-line analysis and possible recompilation
- Key structure is the use of models to statically determine if it is worthwhile applying an optimisation.
- Allows cheap runtime mechanism to decide what to optimise.

## Continuous compilation



Allows more targeted optimisation. However, models are hardwired and based on compiler writer intuition - not actual behaviour.

## Analysis via taxonomy

- All schemes allow specialisation at runtime to program and data
- Staged schemes such as DyC are more powerful as they only incur runtime overhead for specialisation regions
- JIT and DBT delay everything to runtime leaving little optimisation opportunities
- All except ADAPT have a hardwired heuristic of what the best strategy is. Poor at adapting to new platforms.

## Analysis via taxonomy

- Apart from ADAPT and continuous, none looked at processor specific optimisation. Mainly looked at architecture independent optimisations or standard backend scheduling or register allocation.

- Like PDC only used the data really for limited preconceived optimisation goals rather than overcoming undecidability or processor behaviour

- None of the techniques would adapt their compilation approach in the light of experience

- Design time opportunities (apart from continuous compilation) largely ignored.

## Summary and conclusions

- Examined dynamic on-line approaches that optimise at runtime at various program levels
- Focus of each system depends on its overall setting. DBT fits into a particular niche and is constrained in its scope.
- Selective specialisation at runtime (DyC) seems to be a universal good thing and there is much room there for further work
- None of the systems really tackle processor behaviour
- None tackle portability/specialisation of strategy

## Machine Learning based Compilation

## 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

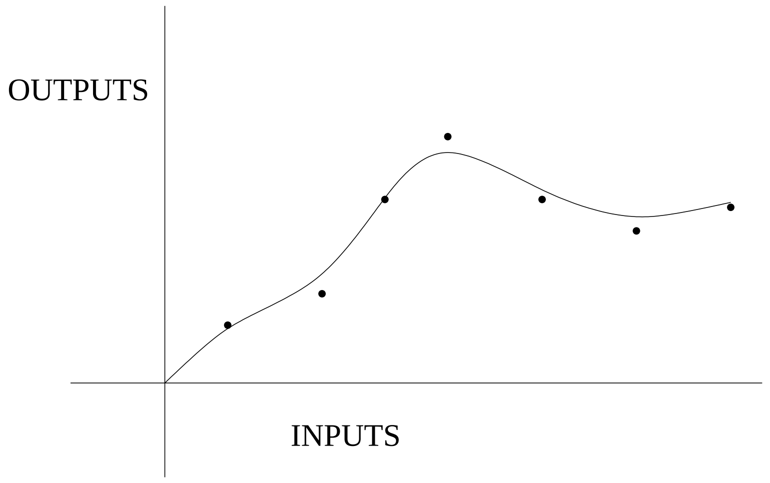
## Failings of previous approaches

- Through we have looked at techniques to overcome data dependent behaviour and and adaption to new processors
- However, we have not looked fundamentally at *process* of *designing* a compiler.
- All rely on a “clever” algorithm inserted into the compiler that determines at compile-time or runtime what optimisations to apply
- Iterative compilation goes beyond this with no a priori knowledge but is not suitable for general compilations and does not adapt to changing data
- What we want is a smart compiler that *adapts* its *strategy* to changes in program, data and processor.

## Machine Learning as a solution

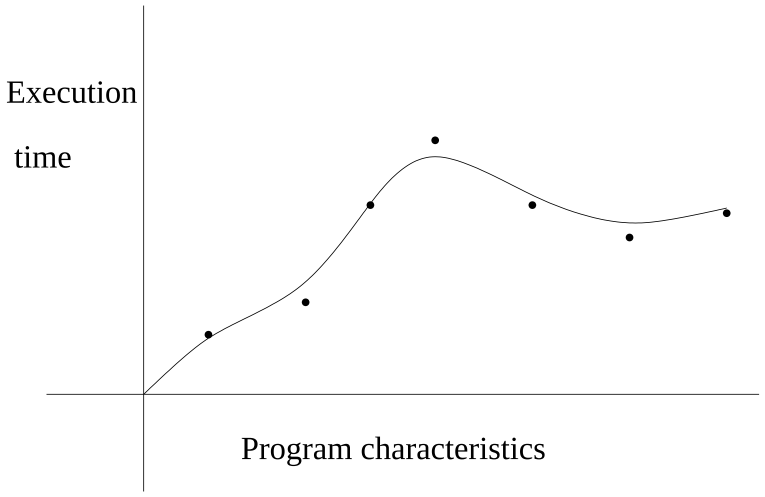
- Well established area of AI, neural networks, genetic algorithms etc. but what has AI got to do with compilation?

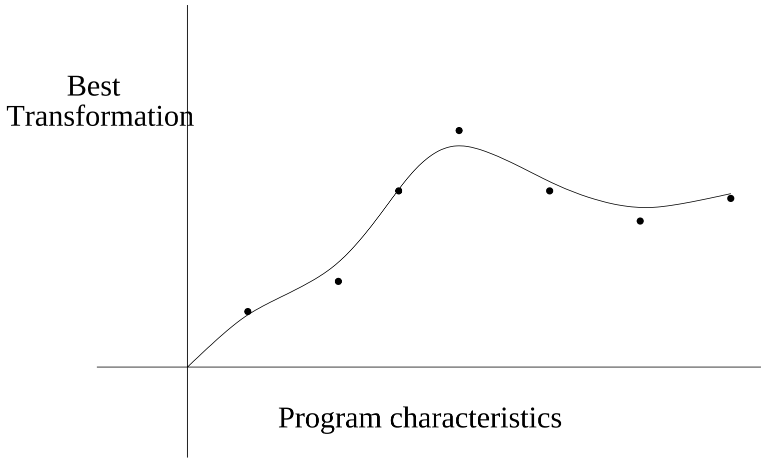
- In a very simplistic sense machine learning can be considered as sophisticated form of curve fitting.



## 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





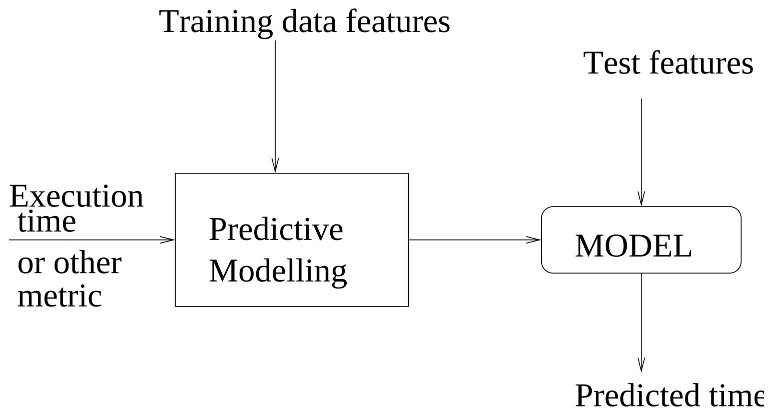
## Hardware based AI

- As in many areas of optimisation, computer architects have been here before us. Architects tend to be more radical then compiler writers.
- Hind (2005) characterises this architect vs compiler distinction as gamblers vs mathematicians.
- Hardware speculation preceded compiler speculation. Again we can learn form them.
- Branch/value prediction is an on-line machine learning process using perceptron based mechanisms (Calder static prediction)

## Global Optimisation and Predictive modelling

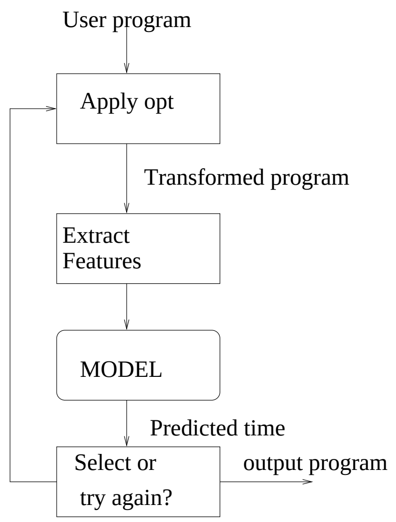
- For our purposes it is possible to consider machine learning as global optimisation and predictive modelling
- *Global optimisation* tries to find the best point in a space. This is achieved by selecting new points, evaluating them and then based on accumulated information selecting a new point as a potential optimum.
- Hill walking and genetic algorithms are obvious examples. Very strong link with iterative compilation
- *Predictive modelling* learns about the optimisation space to build a model. Then uses this model to select the optimum point. Closely related to global 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!

## Predictive Modelling as a proxy



- The model acts as a fast evaluator for program. Automates Soffa's performance prediction framework and speeds up iterative compilation

- Nobody has done this yet!! Feature selection and accuracy main problems

## 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

## Supervised learning

- Building a model based on given inputs and outputs is an example of classical supervised learning. We direct the system to find correlations between selected input features and output behaviour

- In fact unsupervised learning may be more useful in the long run. Generate a large number of examples and features and allow the system to classify them into related groups with shared behaviour.

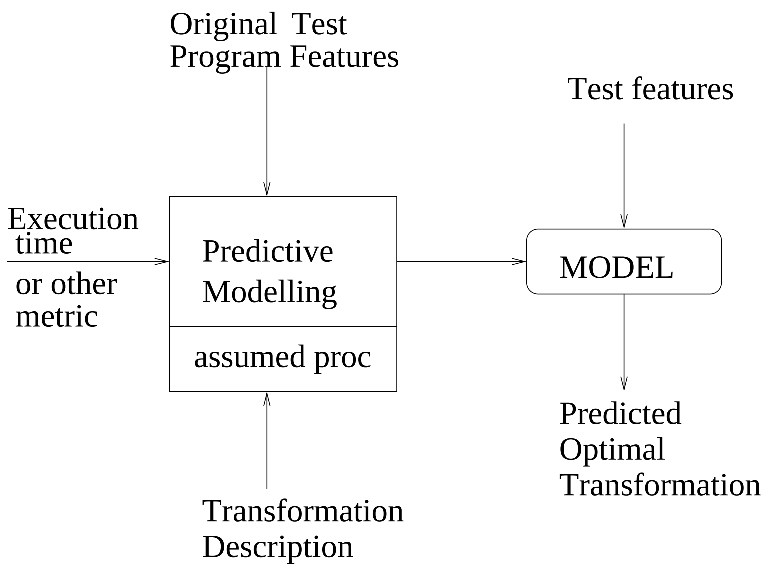
- This prevents missing important features and provide clues as to what aspects of a program are performance determining

- However, we need many combinatorially more programs than features to distinguish between them

## Space to learn over

- Formalisation of compiler optimisation has not been taken really seriously
- However, in order to utilise predictive modelling, we need a descriptions of the program space that allows discrimination between different choices.
- Rather than just having a sophisticated model, what we want is a system that given a program automatically provides the best optimisation
- To do this means that we must have a good description of the transformation space
- The shape of the optimisation space will be critical for learning. Clearly linear regression will not fit the spaces seen before.

## Case studies


Original Program
Test Features
Test features
Execution time or other metric
Predictive Modelling
assumed proc
MODEL
Predicted Optimal Transformation
Transformation Description

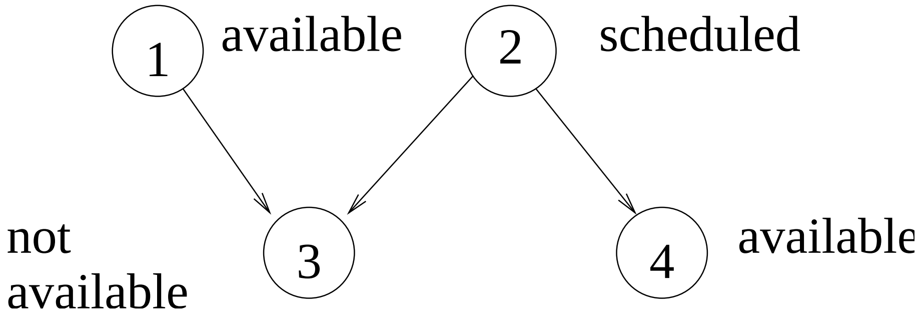
- All of the techniques have the above characterisation
- In fact it is often easier to select a good transformation rather than determine execution time. Relative vs absolute reasoning

## What techniques work and when

- Short answer: Noone knows
- It depends on the structure of the problem space (distribution of minima) and representation of the problem
- One problem particular to compilation is that feature inputs vary in size : length of program, length of transformation sequence
- Also we have no agreed way of representing our problem. Several of the following examples have used different techniques
- Safe to say that the level of ML sophistication is low. Compiler writers tend to try simple things without too much maths or available on the internet!

## 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 $(P, I_j, I_i)$

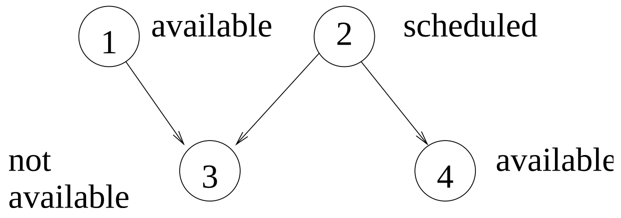
- For each variable size tuple record a fixed length vector summary based on features.

## Learning to schedule

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

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

## Feature extraction

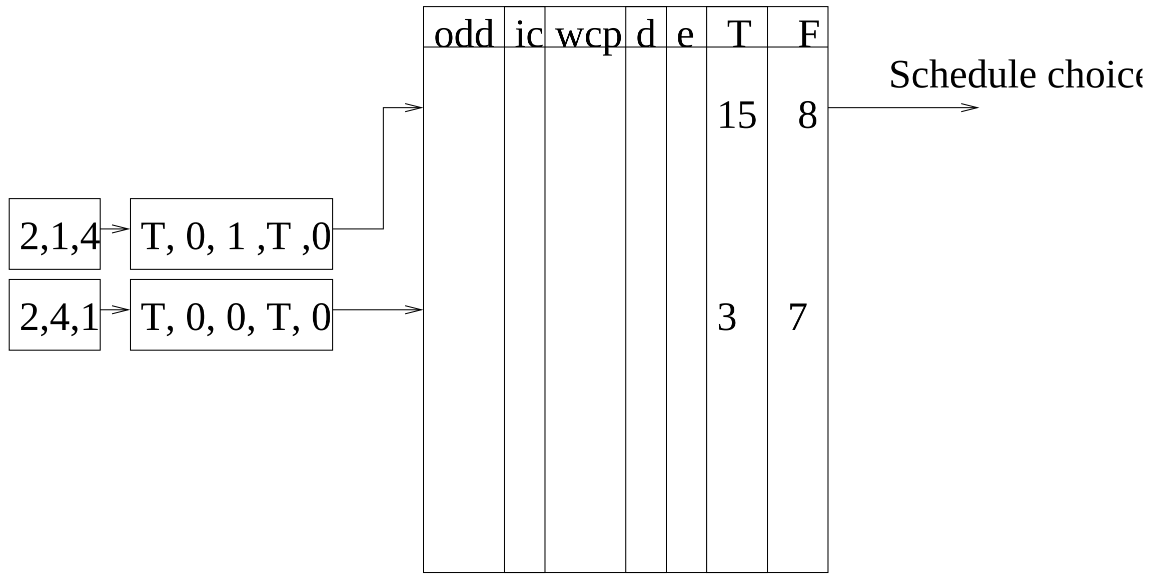


Tuple $(\{2\}, 1, 4)$ : [odd:T, ic:0, wcp:1, d:T, e:0 ]: TRUE,

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 to 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 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

## 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

```
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
- Basic approach - unroll factor not considered.

## Meta-compilation

- Name comes from optimising a heuristic rather than optimising a program.
- Stephenson et al 2003 used genetic programming to tune hyperblock selection, register allocation, and data prefetching within the Trimaran's IMPACT compiler.
- Represent heuristic as a parse tree. Apply mutation and cross over to a population of parse trees and measure fitness.
- Crossover = swap nodes from 2 random parse trees
- Mutate randomly: selected a node and replace with a random expression

## Results

- Two of the pre-existing heuristics were not well implemented .
- For hyperblock selection speedup of 1.09 on test set
- For data prefetching the results are worse - just 1.01 speedup.
- The authors even admit that turning off data prefetching completely is preferable and reduces many of their gains.
- The third optimisation, register allocation is better implemented but only able to achieve on average a 2% increase over the manually tuned heuristic.
- GP is not a focused technique, IMPACT not a commercially quality.

## Learning over UTF

- Shun 2004 uses Pugh's UTF framework to search for good Java optimisations
- Space of optimisation to learn included entire UTF. Training data gathered by using a smart iterative search
- Then using a similar feature extraction to Monsifort classify all found results.
- Uses nearest neighbour based learning able to achieve 70% of the possible performance found using iterative compilation on cross-validated test data
- Larger experimental set needed to validate results. Going beyond loop based transformations for Java

## 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
- Cavazos(2005) 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.

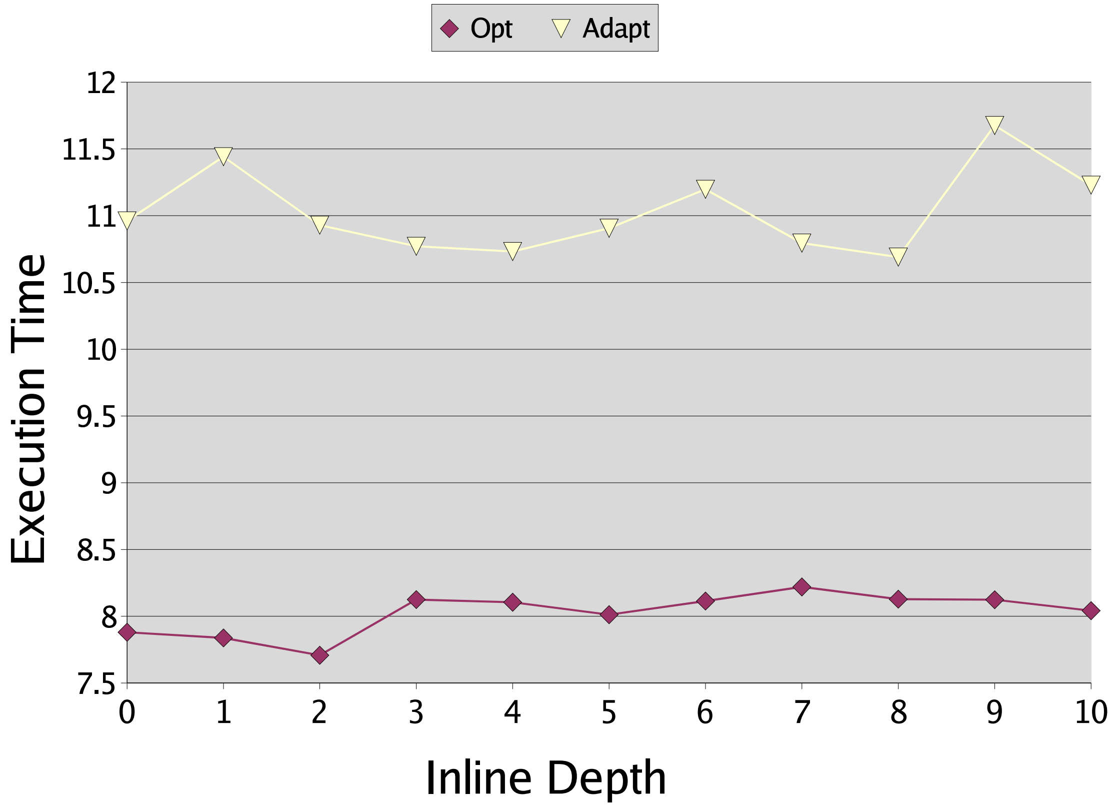
## Learning a heuristic

```
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;
```

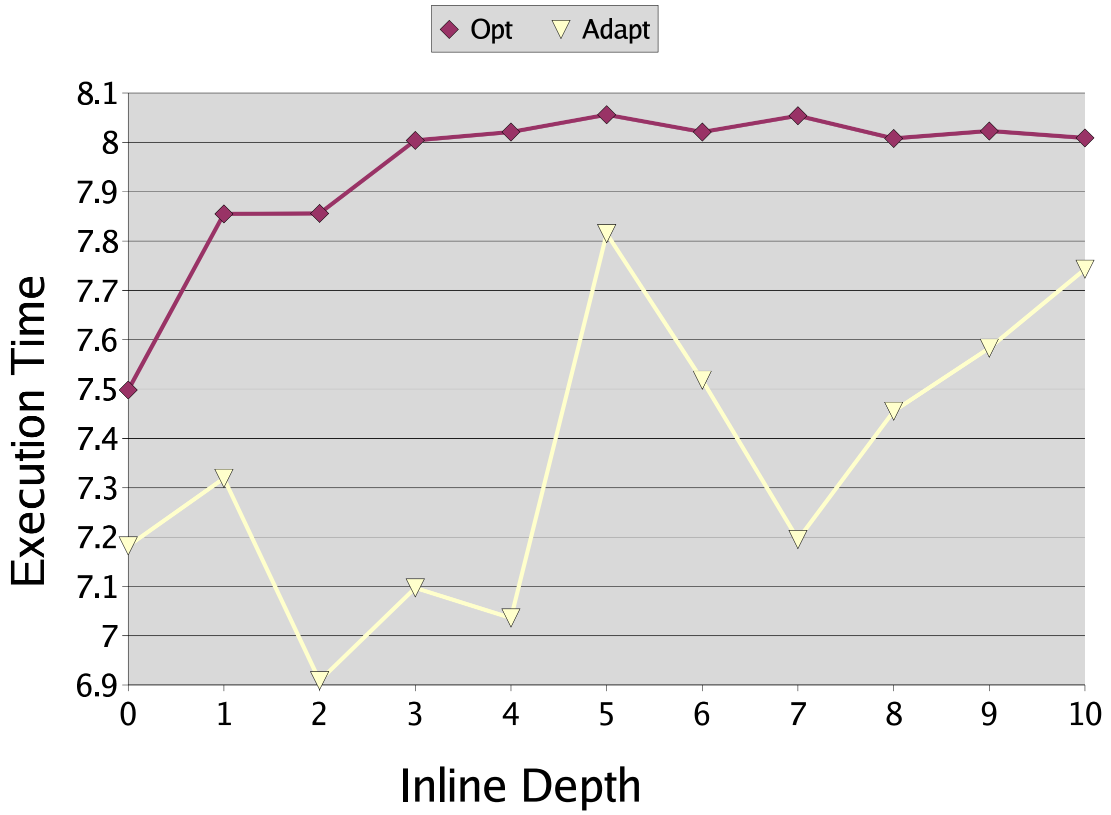
Focus on tuning parameters of an existing heuristic rather than generating a new one from scratch

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

- We 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!

## Future Directions

- Using machine learning for automatic performance prediction is an obvious next step. Can be used as a proxy in dynamic or iterative compilation

- Understanding characteristics of optimisation space to guide algorithm selection

- Learning over data is a challenge. Given a short sample of the input data, predict eventual program behaviour and adapt accordingly

- Given the space of processor, just provide the new features of processor and automatically learn how to optimise for the new machine

- Applying this work to auto-parallelisation.

## Future Directions

- Machine learning will play a key part in design space exploration.
- Given an arch space and limited applications/programs - predict the performance of an optimising compiler - then build it!
- Form part of continuous optimisation. Rather than just learning once, record all information everywhere about program/data/processor/transformation behaviour.
- At suitable intervals learn over this data and update behaviour. At macro level - equivalent to a new compiler release. At micro level dynamic runtime modification of binary

## Summary

- Introduced machine learning as a basic curve fitting/classification approach. Playing catch up with architects.
- It is automatic and can be used to learn compiler strategies which are currently ad hoc and time consuming to develop
- Techniques applied to static and dynamic compilation with promising early results
- Much remains to be done - fertile research area.

## Background (1)

- Hennessy, Patterson: Computer Architecture, a Quantitative Approach, Third Edition. Morgan Kaufmann, 2003.
- Steven Muchnick: Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997
- Randy Allen, Ken Kennedy: Optimizing compilers for modern architectures. Morgan Kaufmann, 2002

## Background (2)

- David Bacon, Susan Graham, Oliver Sharp: Compiler Transformations for High-Performance Computing. ACM Computing Surveys, December 1994, Volume 26 Issue 4. preprint

- Keith Cooper, Linda Torczon: Engineering a Compiler