Adaptive and Profile Directed Compilation

Michael O'Boyle

March, 2011

School of informatics

Overview

- Why we fail to fully optimise
- How to overcome this
- Profile Directed Compilation
- Iterative Compilation
- Critical evaluation and conclusion

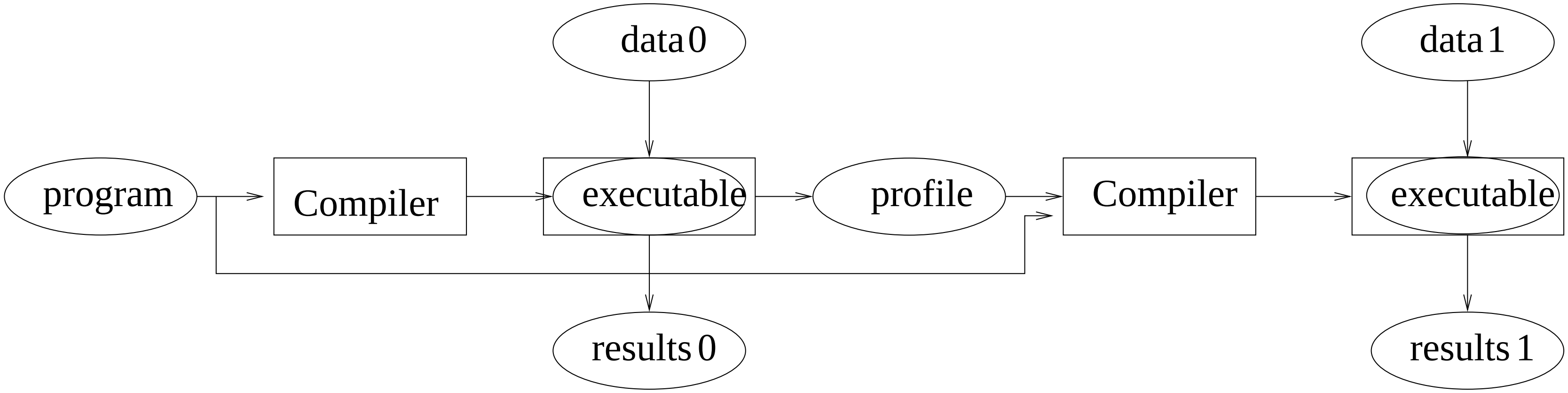
Why fail

- 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!

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?

PDC schematic



Profile information is an additional output.

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

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

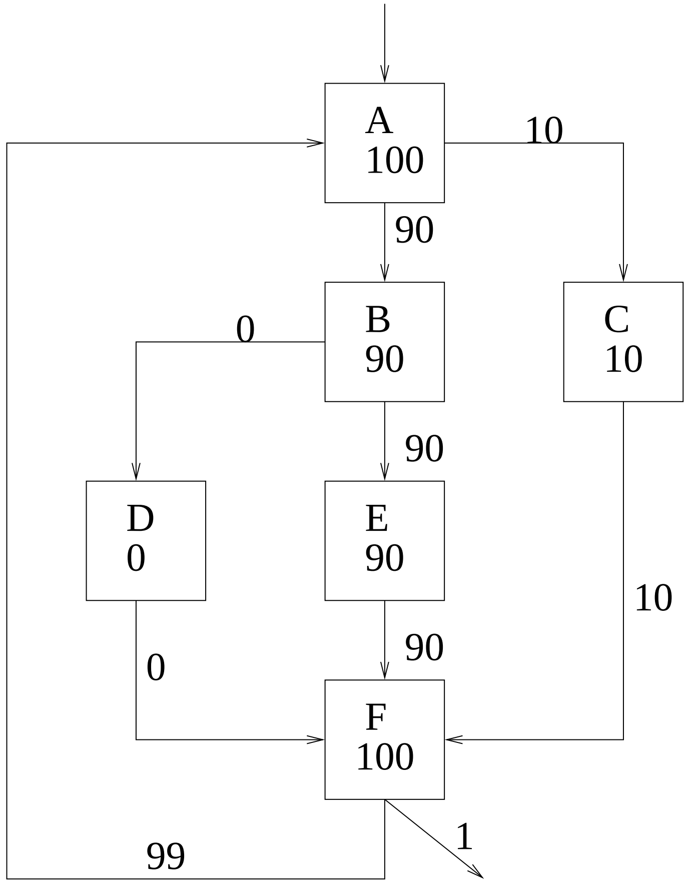
Primarily ABEF

Other entry/exit control-flow prevents merging

Super-block - frequently executed path

Merge and tidy-up

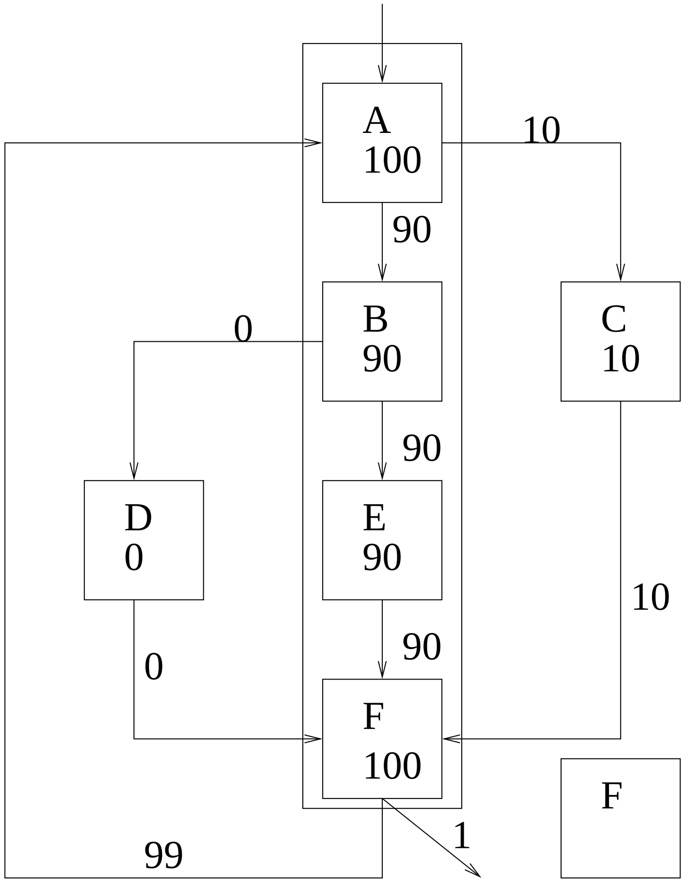
Optimise larger unit



PDC Example 2

Selecting the trace

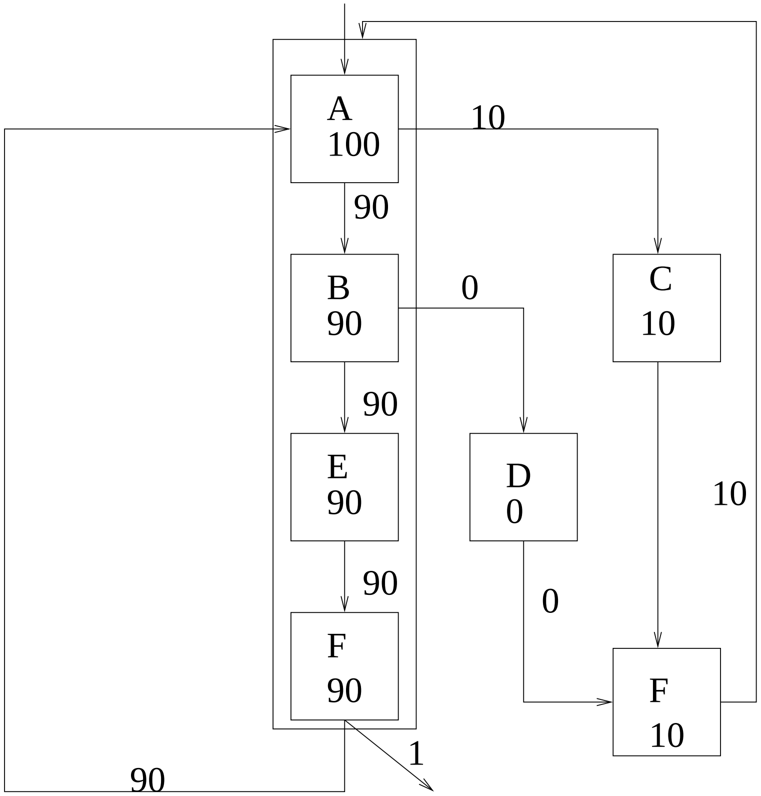
Start at most frequent block
Add blocks on most frequent successors
Repeat on other nodes
Done in both control-flow directions
Do on remaining nodes



PDC Example 3

Tail Duplication

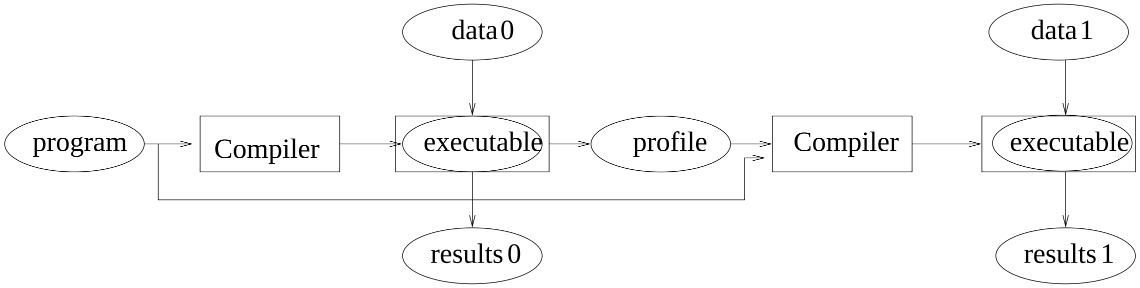
Duplicate first block with external entry edges
But not the head
Redirect incoming edges
Duplicate outgoing
Repeat
Much code duplication



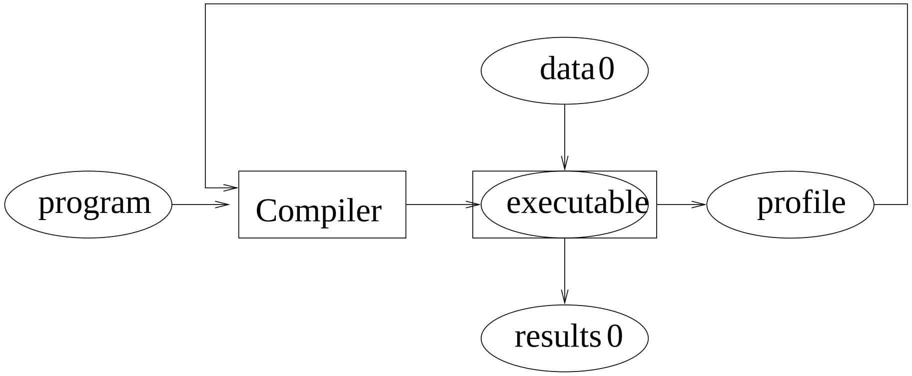
Beyond Path profiling

- 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 one compile



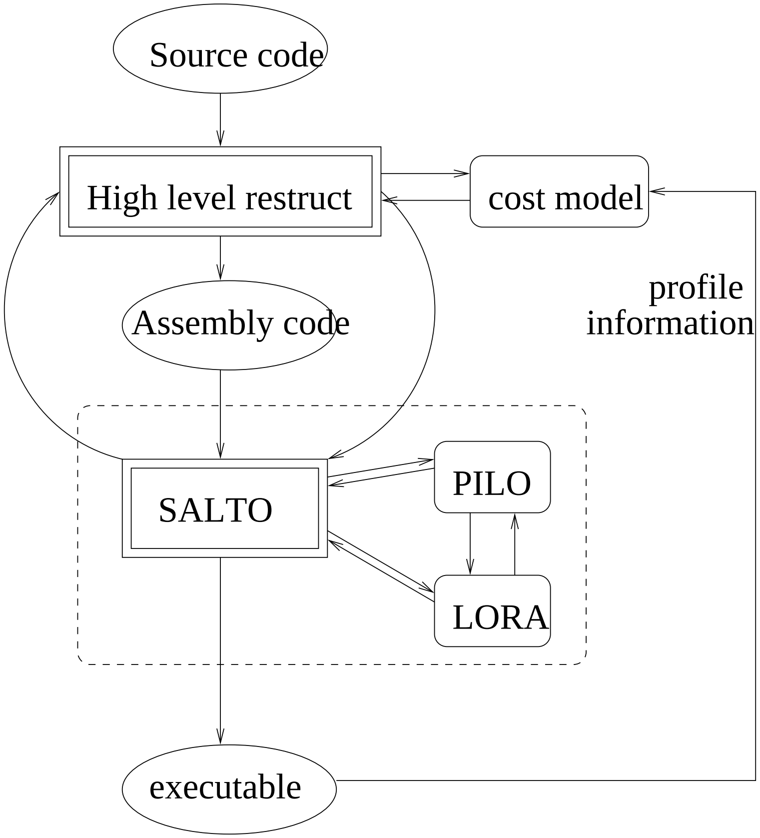
Iterative: multiple compiles

Iterative Compilation : OCEANS

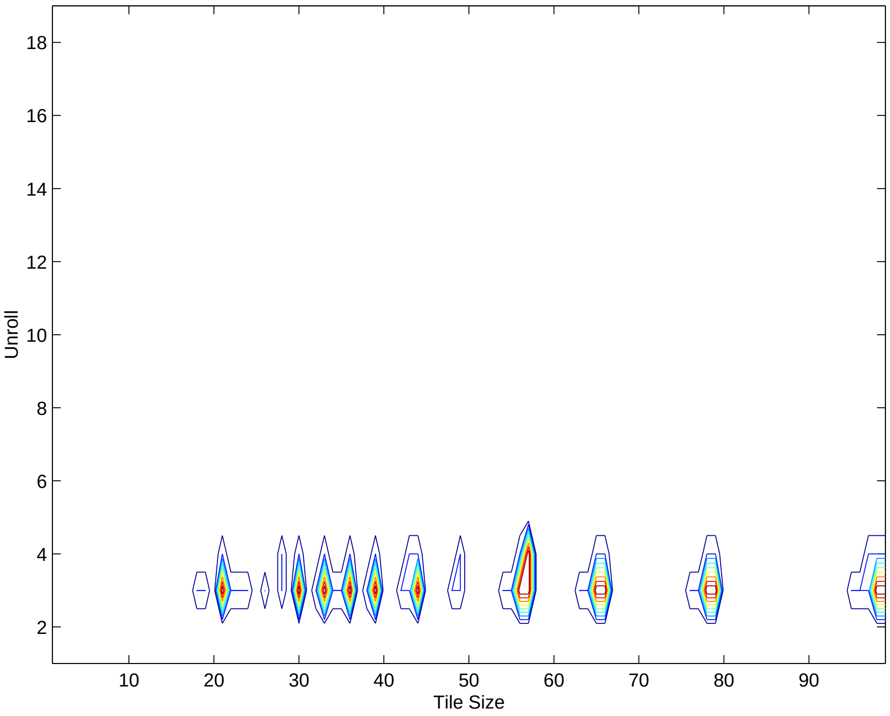
Iterative structure.

Novel notion of two communication compiler infrastructures

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



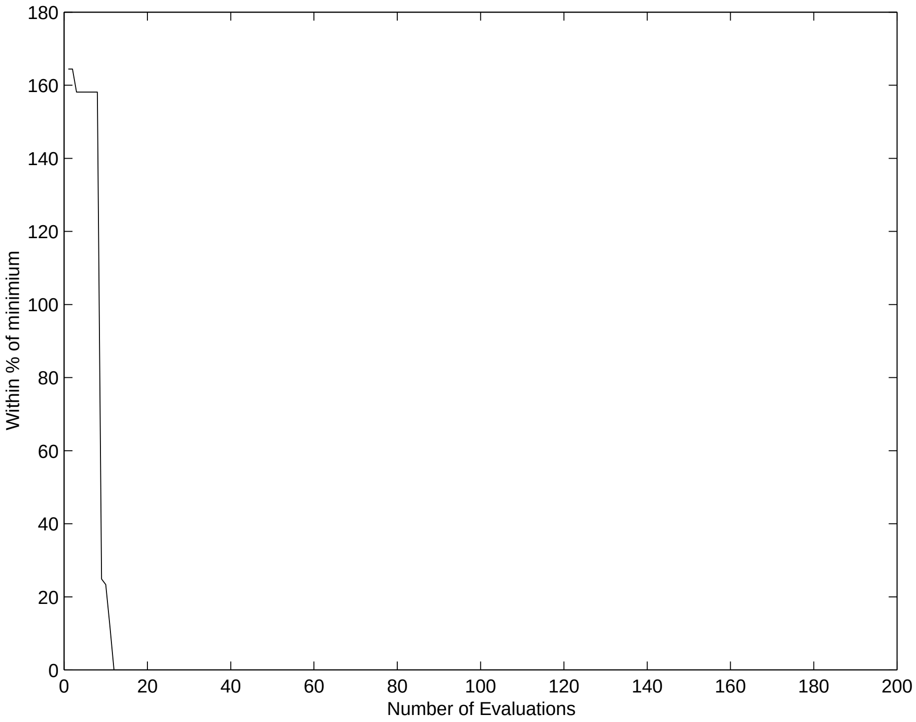
UltraSparc: space within 20% of minimum $N = 400$



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

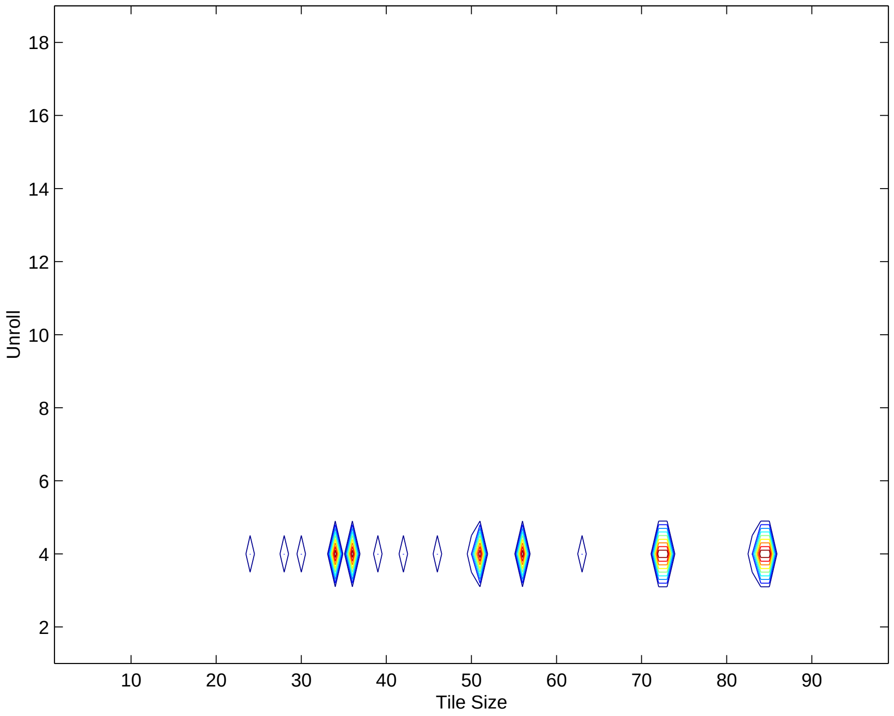
Near minimum: 2.6%. Original 4.99 secs, Minimum 0.56 secs

UltraSparc $N = 400$



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

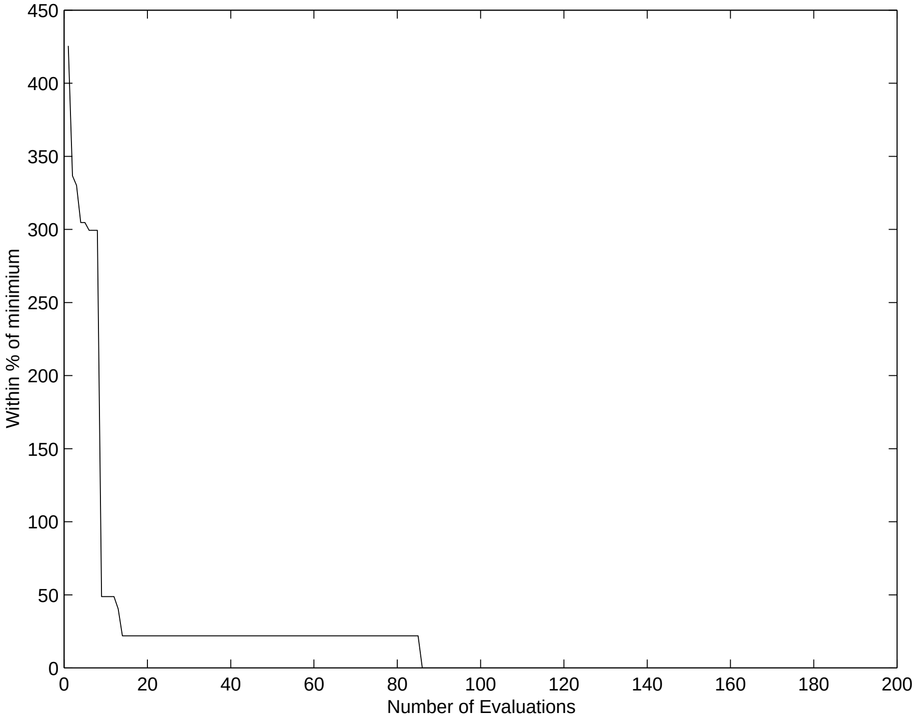
Alpha: space within 20% of minimum $N = 512$



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

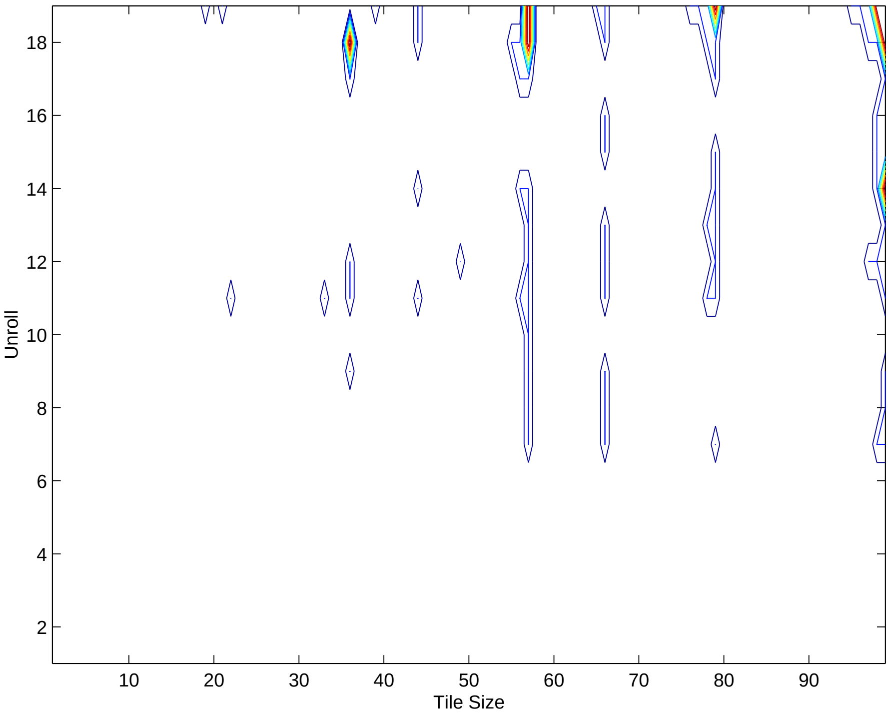
Near minimum: 0.9%. Original 31.72, Minimum 3.34, Max 81.40 !

Alpha $N = 512$



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

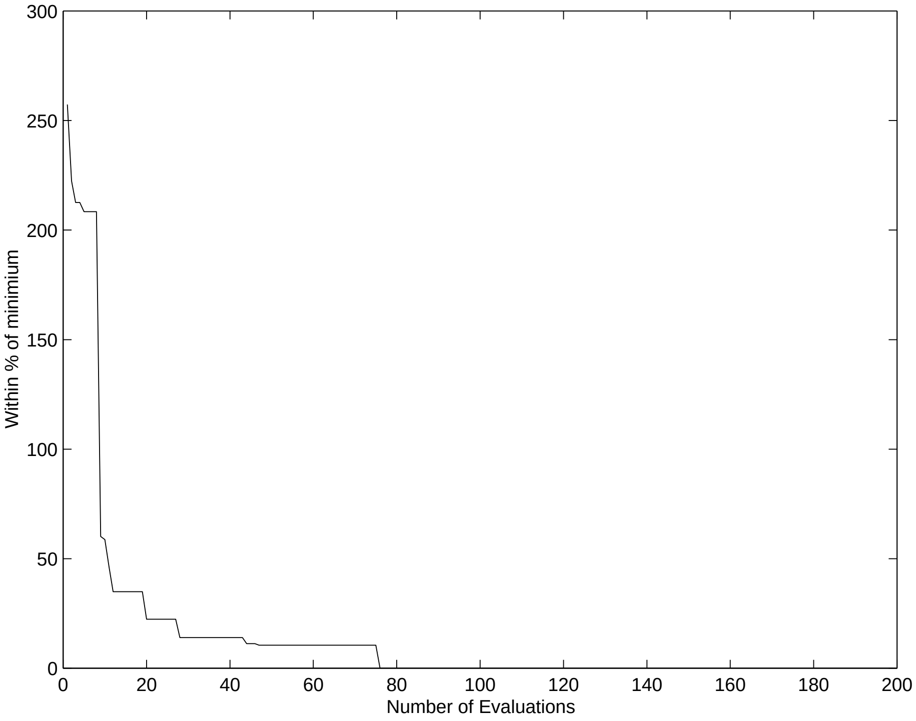
Pentium Pro: space within 20% of minimum $N = 400$



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

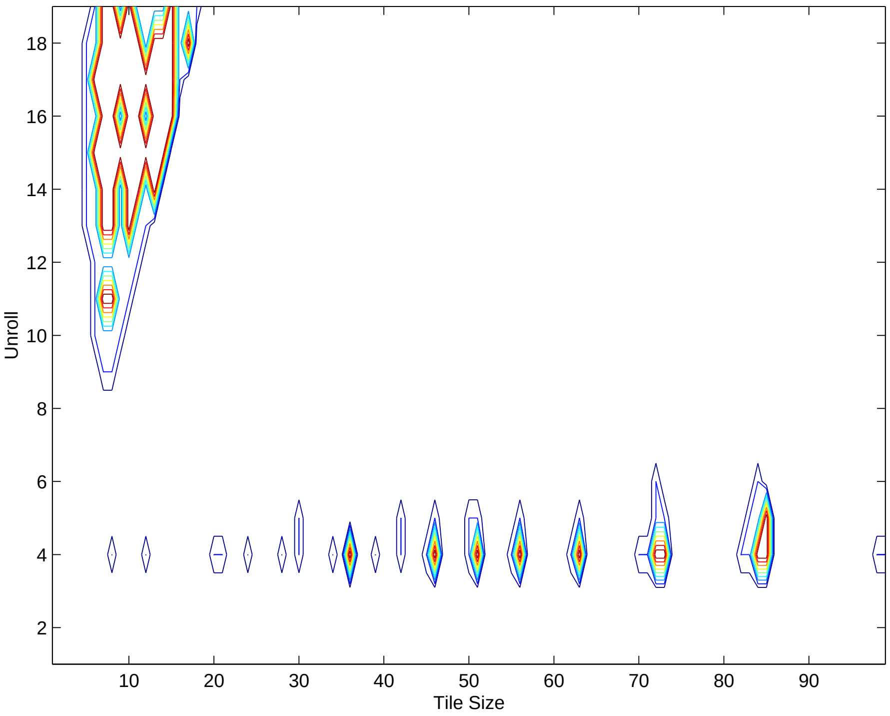
Near minimum: 4.3%. Original 4.88 Minimum 1.43

Pentium Pro $N = 400$



50 steps: within 10.5%.

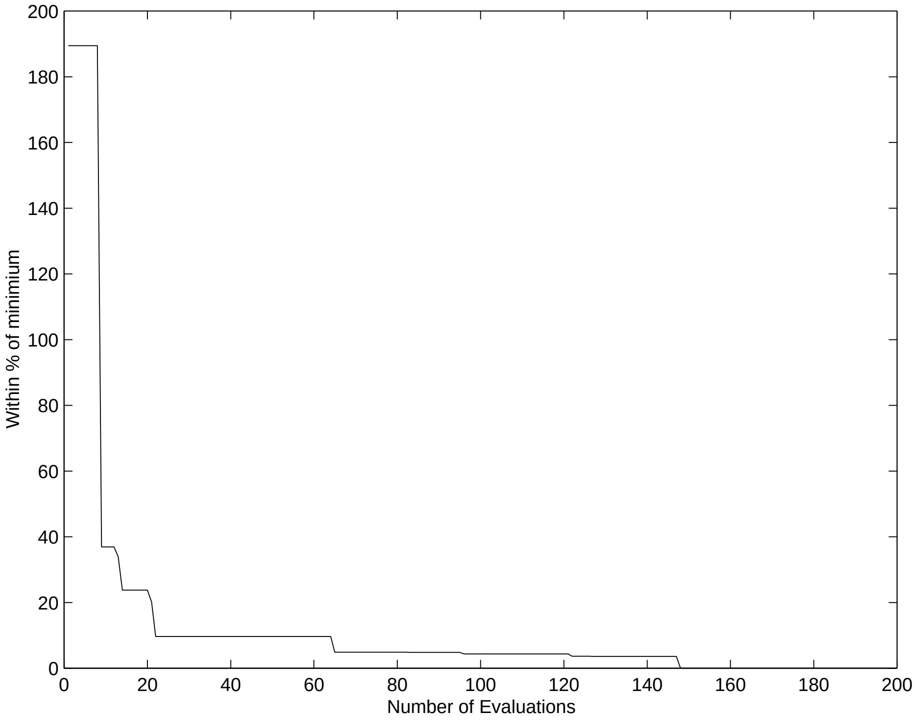
R10000: $N = 512$



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

Near minimum: 7.2%. Original 2.79, Minimum 1.09

R10000 $N = 512$

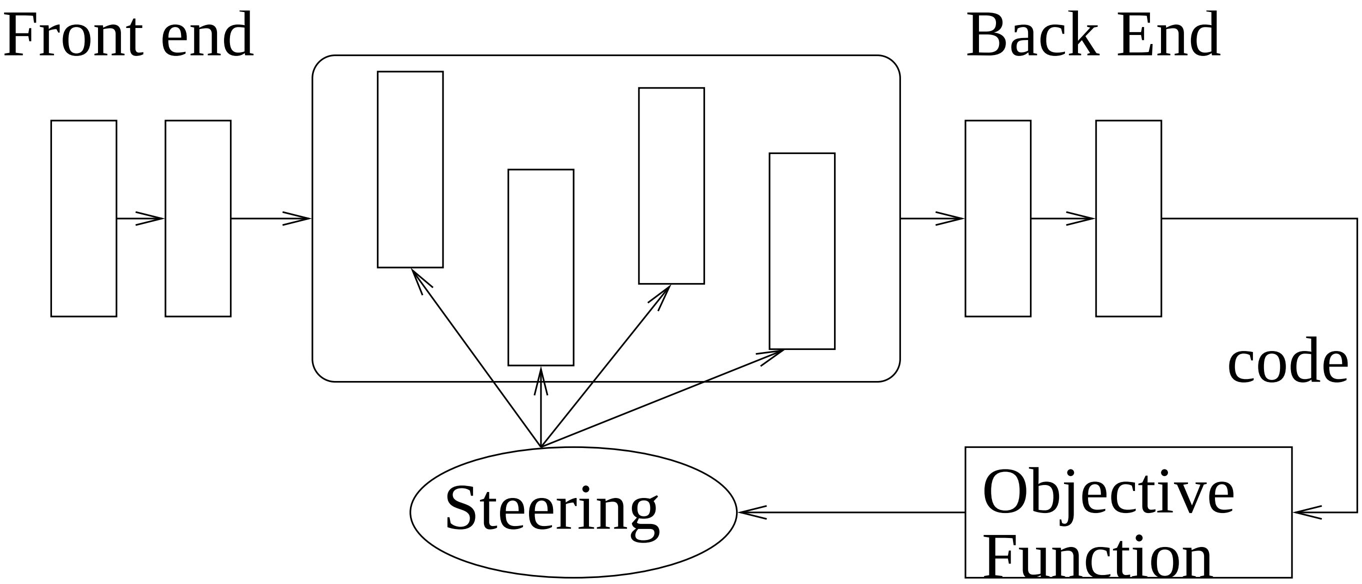


50 steps: within 4.9%.

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.

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.
- Build search techniques to find good points quickly

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.
- Vuduc uses distribution of good points as a stopping criteria. Fursin use upper bound of performance as a guide.
- Using Prior Knowledge in Search space: last lecture on ML

Critical evaluation of iterative compilation

- 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

- 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