Meta Optimization: Improving Compiler Heuristics with Machine Learning

M. Stephenson, S. Amarasinghe, M. Martin, and U. O'Reilly

Presented by Stephen Hines
COP 5622: Advanced Topics in Compilation
October 2003

Presentation Overview

- Basic Principles
- Priority Functions
- Genetic Programming
- Meta Optimization
  - What is it?
  - Case: Hyperblock Formation
  - Case: Register Allocation
  - Case: Data Prefetching
- Conclusions

Basic Principles

- One of the important features of a compiler is the ability to approximate solutions to NP-hard tasks efficiently.
- These solutions are based on heuristics choices that perform well under certain situations but may not be optimal:
  - Heuristics usually require some experimentation to refine (trial and error until adequate performance is achieved).
- Algorithms for solving these problems typically use a priority function (or cost function) to make informed choices.
  - These priority functions are based on heuristics.

Basic Principles (2)

  - Minor modifications to priority functions can drastically affect the performance of the algorithm/optimization since they tend to measure non-uniform non-linear quantities in the program itself.
- Meta Optimization is an attempt to take a step back and look at the actual structure of compiler optimizations:
  - In this case, they are attempting to improve optimizations through the tweaking of heuristics/choices.
  - However rather than using standard iterative and binary search techniques, genetic programming is going to be applied.
- Genetic programming is well-suited to this task due to its ability to adaptively search large spaces using parameters whose interactions are not completely understood.

Priority Functions

- Assist in directing which choices a compiler optimization will make.
- Instruction Scheduling with latency-weighted depths:

$$P(i)=\begin{Bmatrix} latency(i): if\ i\ is\ independent \\ \max_{i\ depends\ on\ j} latency(i)+P(j): otherwise \end{Bmatrix}$$

- P(i) represents the instruction's depth in the dependence graph, and each time an instruction needs to be scheduled the highest priority instruction that is ready is selected from the graph.
- Other examples: Clustered scheduling, Hyperblock formation, Meld scheduling, Modulo scheduling, Register allocation.

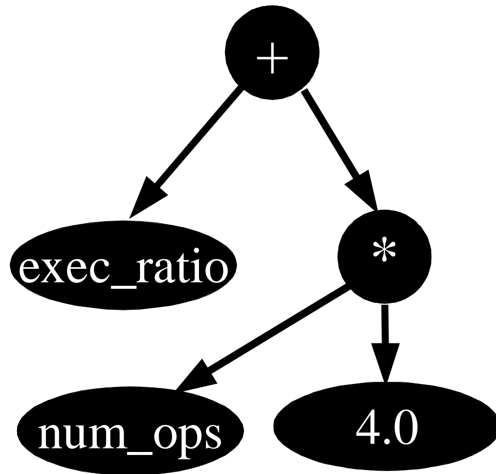
Genetic Programming

- Machine-learning technique with the following characteristics:
  - Effective when working with parameters that have poorly defined interactions (e.g. 'Uncertain tradeoffs').
  - Can be used to search high-dimensional spaces (e.g. Scalable).
  - Distributed algorithm, so work can be parallelized if necessary.
  - Human-readable, so it is possible to obtain a better understanding of what decisions have been made (unlike neural networks).

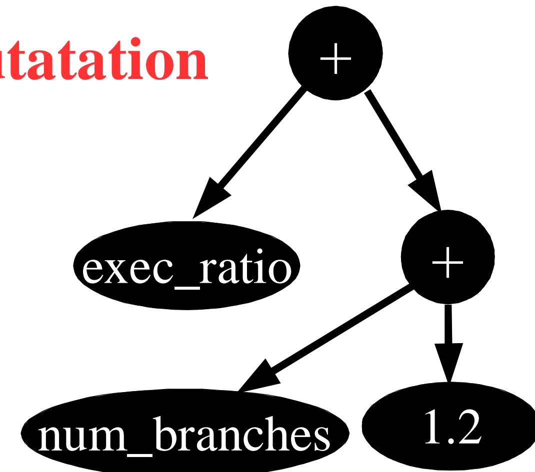
Genetic Programming (2)

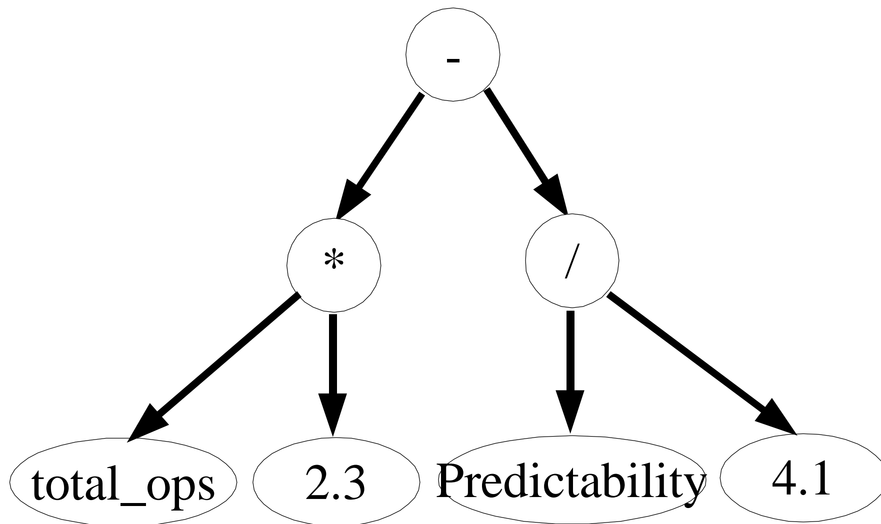
- Modeled after Darwinism (Survival of the Fittest).
  - In this case, the fastest executing compiled code has the fittest priority function.
- Parse trees of operators and operands describe the potential priority functions.
- A population is a collection of parse trees for one generation.
- After testing, several members of the population are selected for reproduction via crossover, which swaps a random node from each of 2 parse trees.
- Other parse trees are selected for mutation, in which a random node is replaced by a random expression.

Genetic Programming (3)

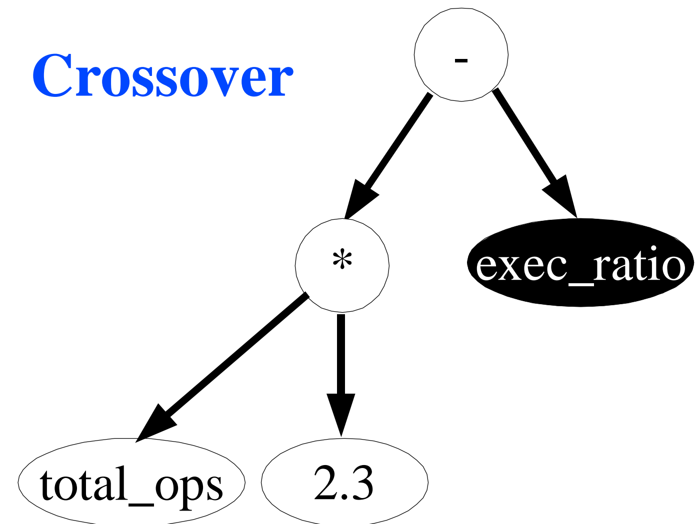


Mutatation





Crossover



Genetic Programming (4)

- The system selects the smaller of several expressions that are equally fit so that the parse trees do not grow exponentially.
- The selection of nodes is done using depth-fair crossover, which weights each depth of node in the tree equally or leaves would be selected more than 50% of the time.
- Tournament selection is used repeatedly to select parse trees for crossover.
  - Choose N expressions at random from the population and select the one with the highest fitness.
- Dynamic subset selection (DSS) is used to reduce the number of priority functions to test for fitness.

Meta Optimization

- What exactly is Meta Optimization?
  - Tuning compiler optimizations to produce better code.
- Scope of paper limits tuning to priority functions, although one could apply this technique to the underlying algorithm.
  - Reduces search space size, hence less time for convergence.
  - Ensures legality of code transformations.
- Initially guess a solution (or use a preexisting priority function).
  - Create 399 random expressions based on parameters.
  - Results indicate that initial guess is many times not a factor of the final solution, so it is viable to use this approach to construct the priority function from scratch (no need for pre-tuning a function).

Meta Optimization (2)

Real-Valued Function	Representation
Real1 + Real2	(add Real1 Real2)
Real1 - Real2	(sub Real1 Real2)
Real1 * Real2	(mul Real1 Real2)
Real1 / Real2 : if Real2 != 0 0 : if Real2 == 0	(div Real1 Real2)
Square root (Real 1)	(sqrt Real1)
Real 1 : if Bool1 Real2 : if not Bool1	(tern Bool1 Real1 Real2)
Real1 * Real2 : if Bool1 Real2 : if not Bool1	(cmul Bool1 Real1 Real2)
Real constant K	(rconst K)
Real value of arg from env	(rarg arg)

Boolean-Valued Function	Representation
Bool1 and Bool2	(and Bool1 Bool2)
Bool1 or Bool2	(or Bool1 Bool2)
Not Bool1	(not Bool1)
Real1 < Real2	(lt Real1 Real2)
Real1 > Real2	(gt Real1 Real2)
Real1 == Real2	(eq Real1 Real2)
Boolean Constant	(bconst {true, false})
Boolean value of arg from env	(barg arg)

Parameter	Setting
Population Size	400 expressions
Number of Generations	50 generations
Generational replacement	22 expressions
Mutation rate	5%
Tournament size	7
Elitism	Best expression is guaranteed survival
Fitness	Average speedup over baseline

Hyperblock Formation

- Uses If-Conversion to convert control dependences into data dependences.
- Single entry multiple exit block with predicated instructions.
- There is a limit to the effectiveness of predicating instructions and forming large hyperblocks.
- Potential criteria involved in forming a hyperblock - (what is important for inclusion in the hyperblock?)
  - Path predictability - unpredictable branches
  - Path frequency - frequent paths
  - Path instruction level parallelism (ILP) - low ILP

Hyperblock Formation (2)

- Number of instructions in path - shorter paths
- Number of branches in path - more branches but not too many
- Compiler optimization considerations (e.g. exceptions)
- Machine-specific considerations (e.g. branch delay penalty)

- Most of the above are fairly intuitive, but their interactions are difficult to understand, hence the need for tuning.
- Experiment performed with Trimaran's IMPACT compiler approximating the Intel Itanium architecture (EPIC).
- Tested with selections from SpecInt, SpecFP, Mediabench and other miscellaneous benchmark programs.

Hyperblock Formation (3)

GP-generated Heuristic

```
(add
 (sub (mul exec_ratio_mean 0.8720) 0.9400)
 (mul 0.4762
    (cmul (not mem_hazard)
     (mul 0.6727 num_paths)
     (mul 1.1609
      (add
       (sub
        (mul
         (div num_ops dep_height) 10.8240)
        exec_ratio)
       (sub (mul (cmul has_unsafe_jsr
                       predict_product_mean
                       0.9838)
                 (sub 1.1039 num_ops_max))
            (sub (mul dep_height_mean
                      num_branches_max)
                 num_paths)))))))
```

Trimaran's Heuristic

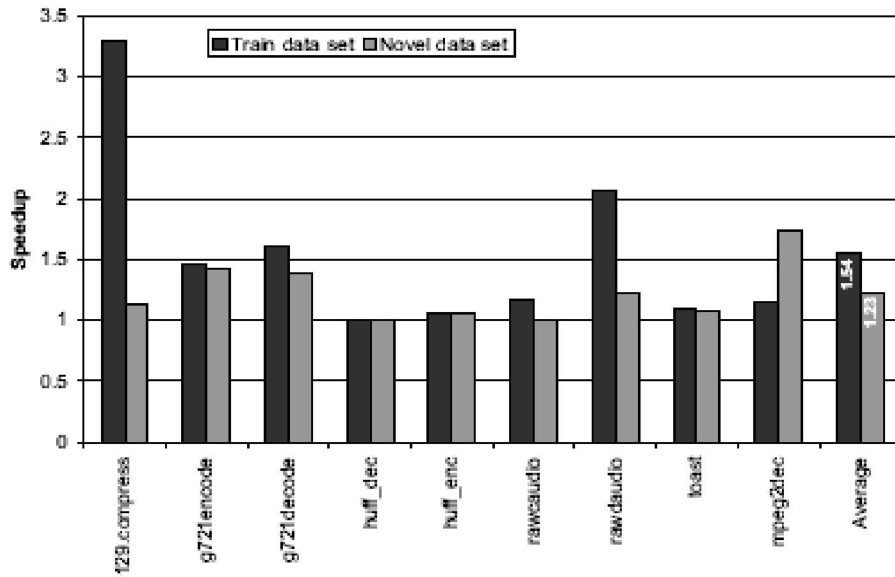
$$h_i = \begin{cases} 0.25 & : \text{ if } path_i \text{ contains a hazard.} \\ 1 & : \text{ if } path_i \text{ is hazard free.} \end{cases}$$

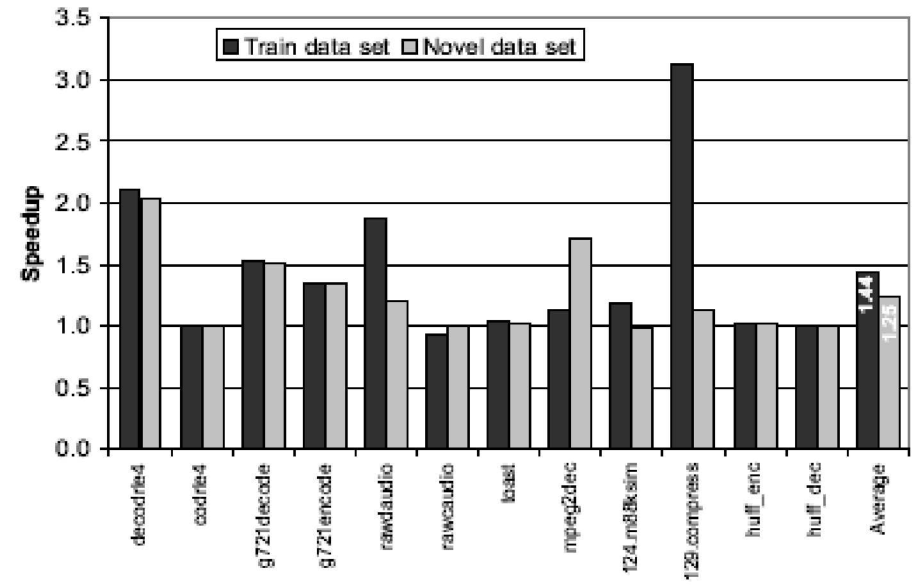
$$d_ratio_i = \frac{dep_height_i}{\max_{j=1 \to N} dep_height_j}$$

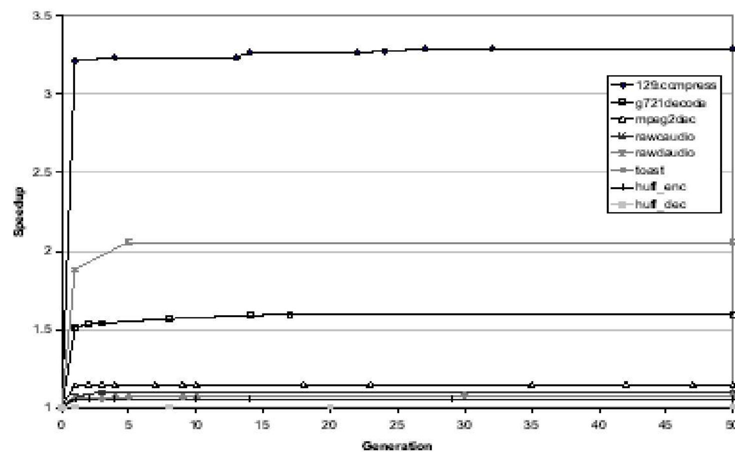
$$o_ratio_i = \frac{num_ops_i}{\max_{j=1 \to N} num_ops_j}$$

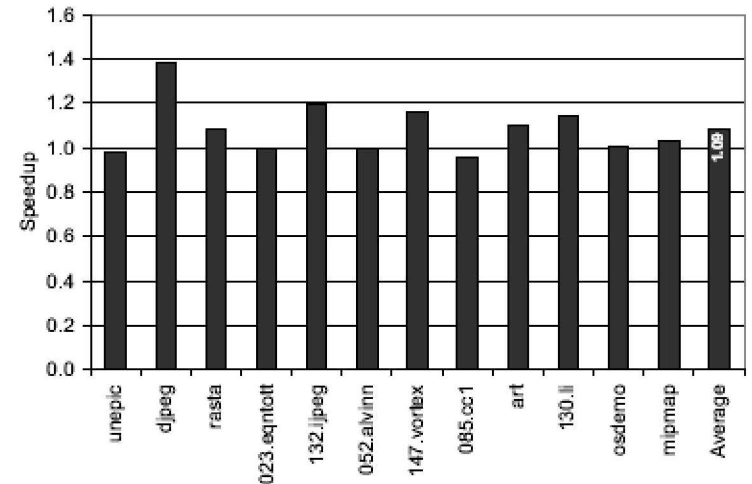
$$priority_i = exec_ratio_i \cdot h_i \cdot (2.1 - d_ratio_i - o_ratio_i)$$

Hyperblock Formation (4)









Hyperblock Formation (5)

- Quantitative Results:
  - Single benchmark results are very impressive: (1.54/1.23)
  - Multiple benchmark results: (1.44/1.25)
  - Results on non-training benchmarks: (1.09)
- Important results:
  - Initial population often has a better performer than starting PF.
  - Useless expressions (Introns) sometimes remain in the solution.
    - Used to preserve good building blocks during variation phase.
  - Solution is easy to read, but hard to understand.

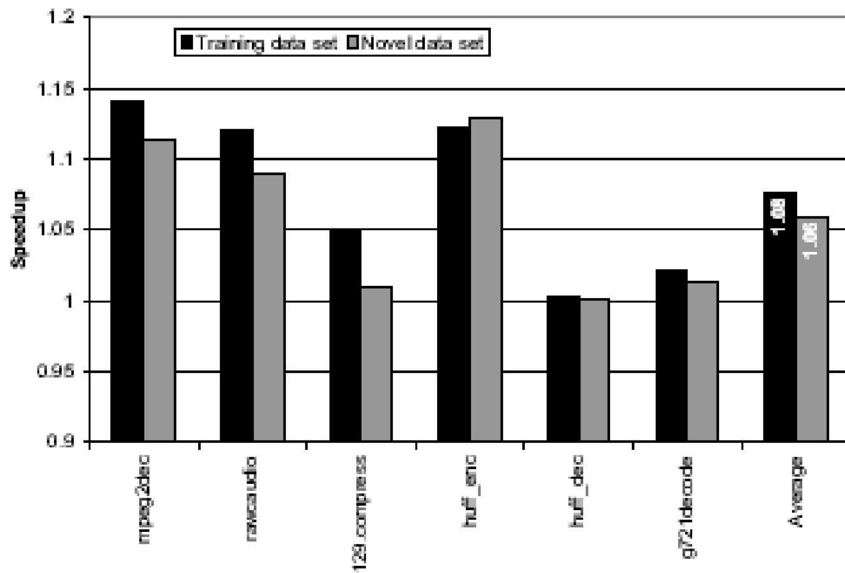
Register Allocation

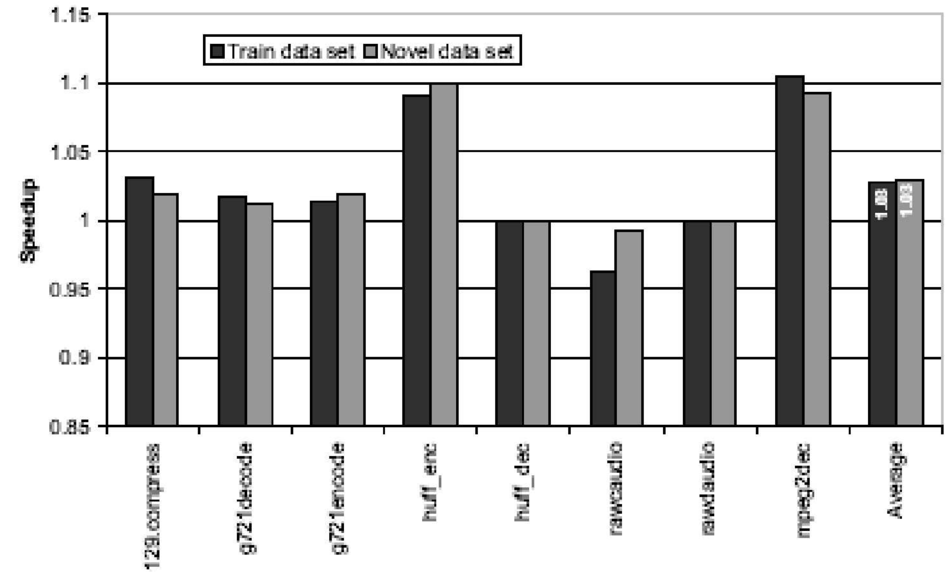
- Determine which variables should be contained in registers, and which registers should be spilled when necessary.
- Live ranges are calculated and then ordered based on a priority function (Remember priority-based graph coloring).
- Most beneficial ranges are then assigned to registers.
- Uses same Itanium architecture setup as Hyperblock Formation with the IMPACT compiler.
- Keep priority(lr) intact (normalization), but change equation for savings.

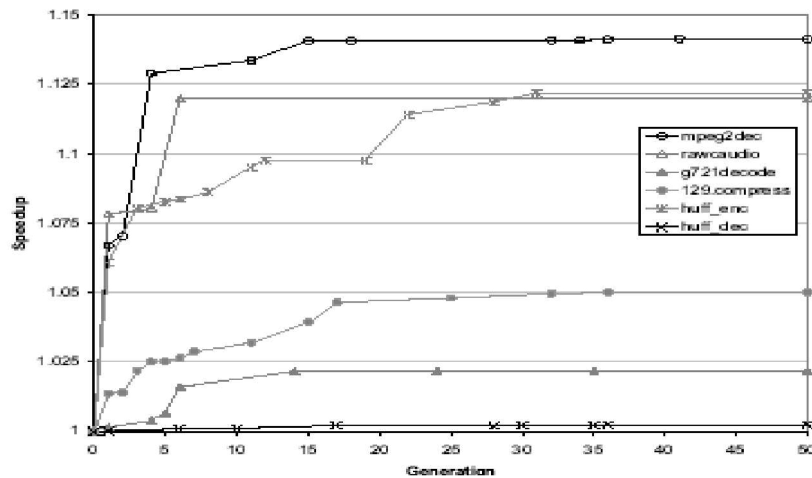
$$savings_i = w_i \cdot (LDsave \cdot uses_i + STsave \cdot defs_i)$$

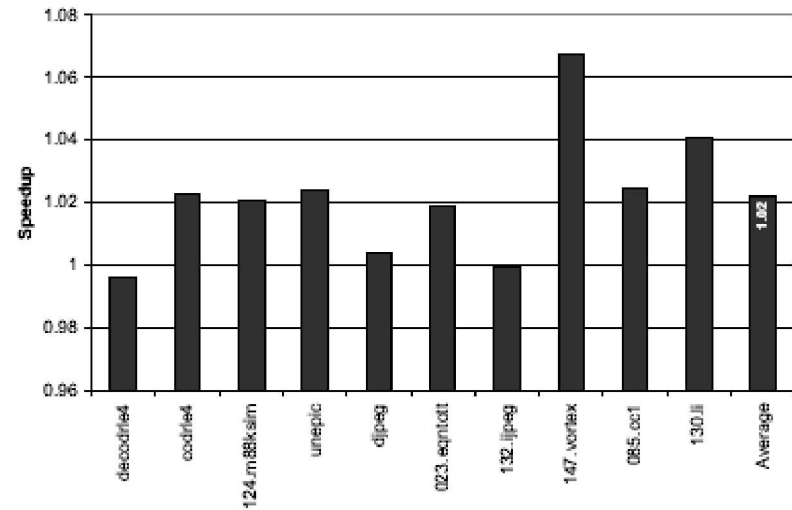
$$priority(lr) = \frac{\sum_{i \in lr} savings_i}{N}$$

Register Allocation (2)









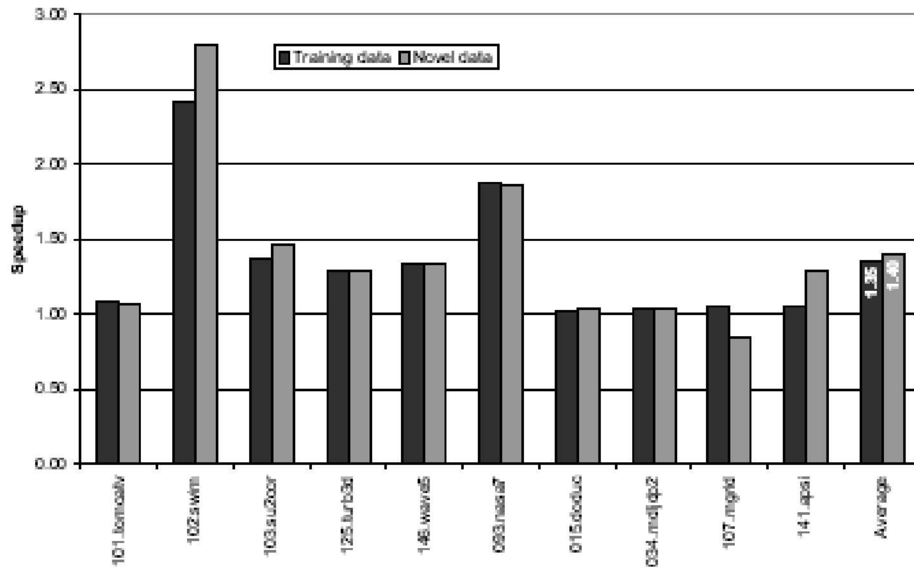
Register Allocation (3)

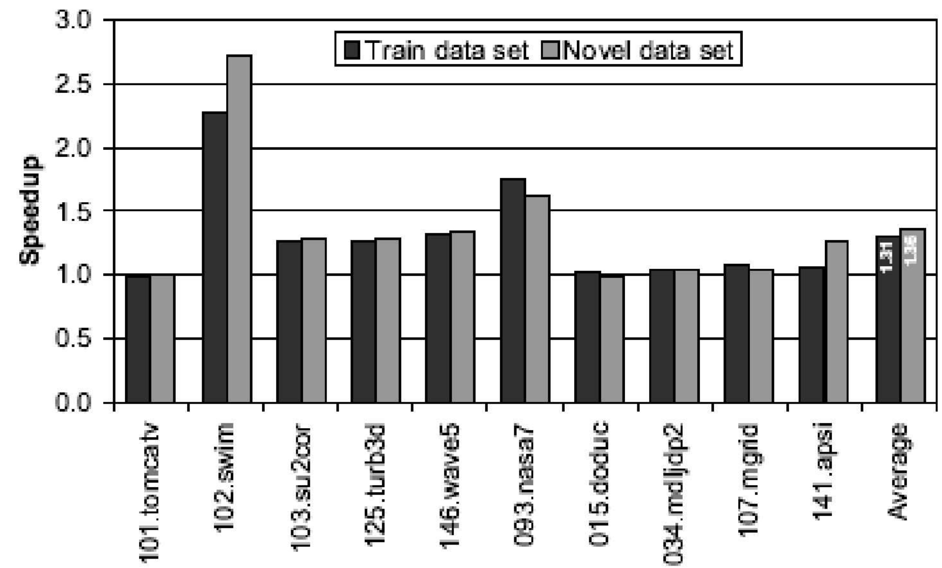
- Quantitative Results:
  - Single benchmark results: (1.08/1.06)
  - Multiple benchmark results: (1.03/1.03)
  - Results on non-training benchmarks: (1.02)
- Important results:
  - Less difference between training/test results since Hyperblock Formation (HF) is much more data-driven.
  - Harder to optimize than HF (less improvement), but still possible.
  - Not as susceptible to variations in input data.
  - Even well-studied optimizations can be improved.

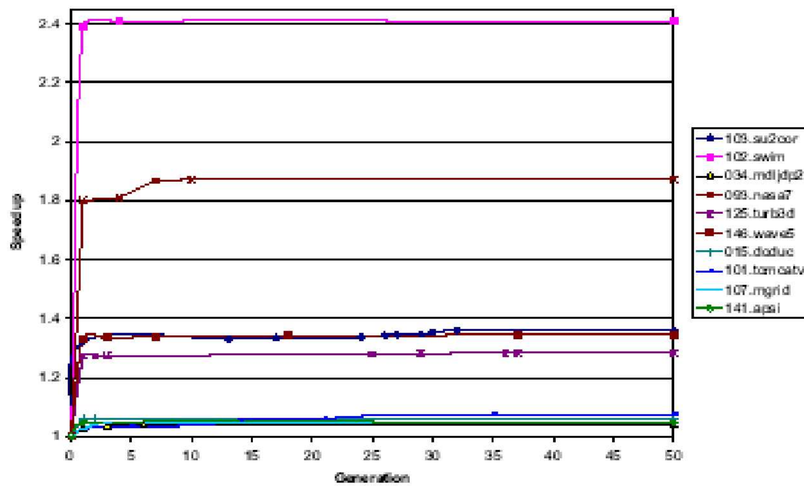
Data Prefetching

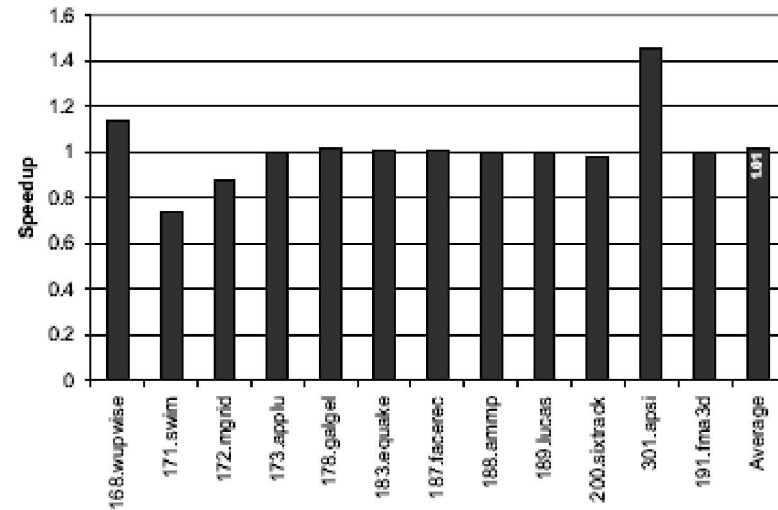
- Optimization that attempts to reduce long-latency memory operations through the insertion of prefetch instructions.
- Dangerous since prefetching too often can evict data from the cache before it is used.
- Experiments done on actual Itanium I systems using the Open Research Compiler (ORC), and code is actually executed and not simulated.

Data Prefetching (2)









Data Prefetching (3)

- Quantitative Results:
  - Single benchmark results: (1.35/1.40)
  - Multiple benchmark results: (1.31/1.36)
  - Results on non-training benchmarks: (1.01)
- Important results:
  - ORC prefetches too much, so turning it off gives a benefit of ~7% to start with. Thus the GP solution cuts down on prefetching.
  - Novel data sets perform better unlike in the other 2 experiments.
  - GP solution fitness saturates after relatively few populations.

Related Research

- Supervised learning is used by Calder et al. to calculate static branch prediction heuristics. Matches inputs with known outcomes (labels), and then can classify every new input based on these labels.
- Cooper et al. use genetic algorithms for solving the phase ordering problem (which order of optimizations provides the greatest benefit).
- COGEN(t) uses GP to map code to irregular DSPs. Although this is a long process, it is worth it for compile-once applications on embedded systems.

Conclusions

- Older architectures were simple and analyzable, newer architectures and compilers are more complex.
- Sometimes it is only possible to make decisions based on empirical evidence, since problems are NP-hard or not well understood.
- Heuristics are used by compilers to obtain good, although not necessarily optimal results.
- Coming up with good heuristics is a very difficult and time-consuming process. Genetic programming takes care of performing this task, obtaining better solutions than trial and error.

Conclusions (2)

- One drawback is that GP needs some adjustment in terms of population size, mutation rate, tournament size, ...
- However results show that this technique does obtain speedups over human-generated heuristics (e.g. priority functions).
- Compiler-writers can focus on providing algorithms with components that may be created or refined using machine learning techniques.