Improving Compiler Heuristics with Machine Learning

Mark Stephenson
Una-May O'Reilly
Martin C. Martin
Saman Amarasinghe
Massachusetts Institute of Technology

System Complexities

- Compiler complexity
  - Open Research Compiler
    - ~3.5 million lines of C/C++ code
  - Trimaran's compiler
    - ~ 800,000 lines of C code
- Architecture Complexity

Features	Pentium® (3M)	Pentium 4 (55M)
Superscalar	✓	✓
Branch prediction		✓
Speculative execution		✓
MMU and improved FPU		✓

NP-Completeness

- Many compiler problems are NP-complete
  - Thus, implementations can't be optimal
- Compiler writers rely on heuristics
  - In practice, heuristics perform well
  - …but, require *a lot* of tweaking
- Heuristics *often* have a focal point
  - Rely on a single priority function

Priority Functions

- A heuristic's Achilles heel
  - A single *priority* or *cost* function often dictates the efficacy of a heuristic
- Priority functions rank the options available to a compiler heuristic
  - List scheduling (identifying instructions in worklist to schedule first)
  - Graph coloring register allocation (selecting nodes to spill)
  - Hyperblock formation (selecting *paths* to include)
- Any priority function is legal

Our Proposal

- Use machine-learning techniques to automatically search the priority function space
  - Increases compiler's performance
  - Reduces compiler design complexity

Qualities of Priority Functions

- Can focus on a small portion of an optimization algorithm
  - Don't need to worry about legality checking
- Small change can yield big payoffs
- Clear specification in terms of input/output
- Prevalent in compiler heuristics

An Example Optimization

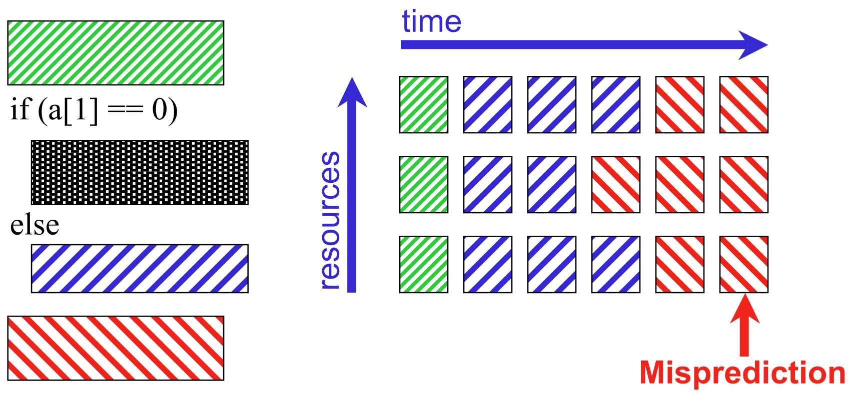
Hyperblock Scheduling

- Conditional execution is potentially *very* expensive on a modern architecture
- Modern processors try to dynamically predict the outcome of the condition
  - This works great for predictable branches…
  - But some conditions can't be predicted
- If they don't predict correctly you waste a lot of time

Example Optimization

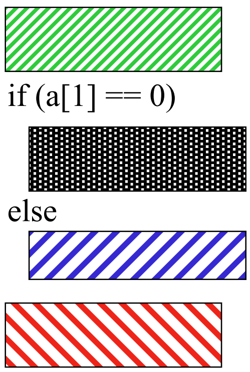
Hyperblock Scheduling

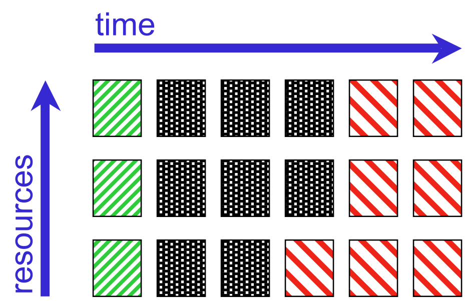
Assume a[1] is 0



Example Optimization
Hyperblock Scheduling

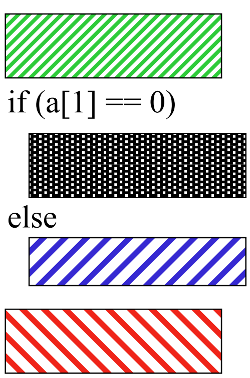
Assume a[1] is 0

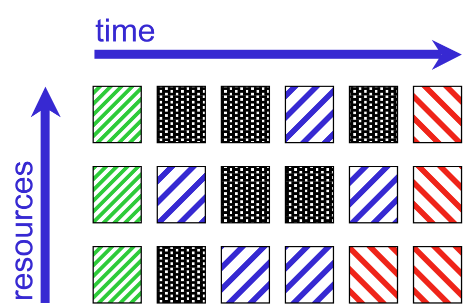




Example Optimization
Hyperblock Scheduling (using predication)

Assume a[1] is 0





Processor simply discards results of instructions that weren't supposed to be run

Example Optimization
Hyperblock Scheduling

- There are unclear tradeoffs
  - In some situations, hyperblocks are faster than traditional execution
  - In others, hyperblocks impair performance
- Many factors affect this decision
  - Accuracy of branch predictor
  - Availability of parallel execution resources
  - Effectiveness of the compiler's scheduler
  - Parallelizability and predictability of the program
- Hard to model

Example Optimization
Hyperblock Scheduling [Mahlke]

- Find predicatable regions of control flow
- Enumerate paths of control in region
  - Exponential, but in practice it's okay
- Prioritize paths based on four path characteristics
  - The priority function we want to optimize
- Add paths to hyperblock in priority order

Trimaran's Priority Function

$$hazard_i = \begin{cases} 0.25 & : \text{if } segment_i \text{ has hazard} \\ 1 & : \text{if } segment_i \text{ is hazard free} \end{cases}$$

$$dep\_ratio_i = \frac{dep\_height_i}{\max_{j=1 \to N} dep\_height_j}$$

$$op\_ratio_i = \frac{op\_height_i}{\max_{j=1 \to N} op\_height_j}$$

$$priority_i = exec\_ratio_i \cdot hazard_i \cdot (2.1 - dep\_ratio_i - op\_ratio_i)$$

Favor frequently executed paths

Penalize paths with hazards

Favor parallel paths

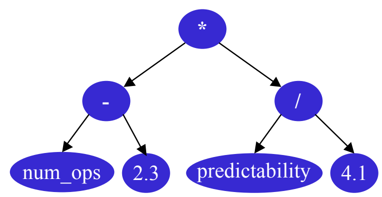
Favor short paths

Our Approach

- Trimaran uses four characteristics
- What are the important characteristics of a hyperblock formation priority function?
- Our approach: Extract all the characteristics you can think of and let a learning technique find the priority function

Genetic Programming

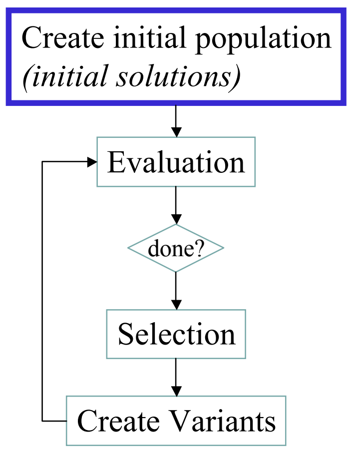
- Searching algorithm analogous to Darwinian evolution
  - Maintain a population of expressions



Genetic Programming

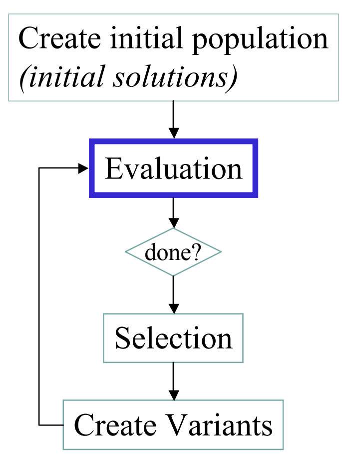
- Searching algorithm analogous to Darwinian evolution
  - Maintain a population of expressions
  - Selection
    - The *fittest* expressions in the population are more likely to reproduce
  - Reproduction
    - Crossing over subexpressions of two expressions
  - Mutation

General Flow



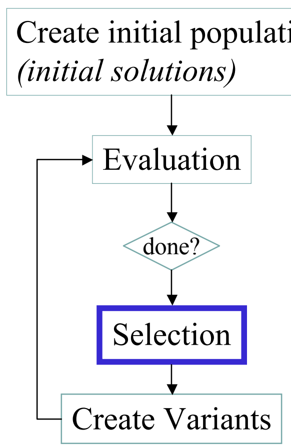
- Randomly generated initial population *seeded* with the compiler writer's best guess

General Flow



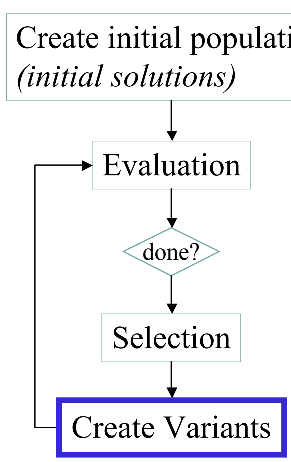
- Compiler is modified to use the given expression as a priority function
- Each expression is evaluated by compiling and running the benchmark(s)
- Fitness is the relative speedup over Trimaran's priority function on the benchmark(s)

General Flow



- Just as with Natural Selection, the fittest individuals are more likely to survive

General Flow



- Use crossover and mutation to generate new expressions
- And thus, generate new compilers

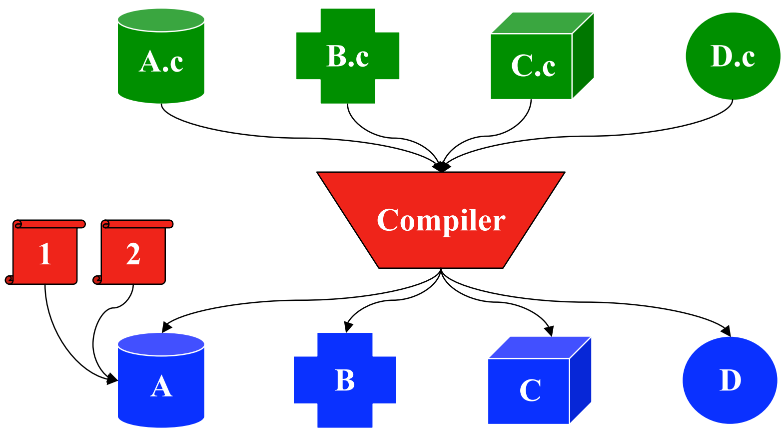
Experimental Setup

- Collect results using Trimaran
  - Simulate a VLIW machine
    - 64 GPRs, 64 FPRs, 128 PRs
    - 4 fully pipelined integer FUs
    - 2 fully pipelined floating point FUs
    - 2 memory units (L1:2, L2:7, L3:35)
- Replace priority functions in IMPACT with our GP expression parser and evaluator

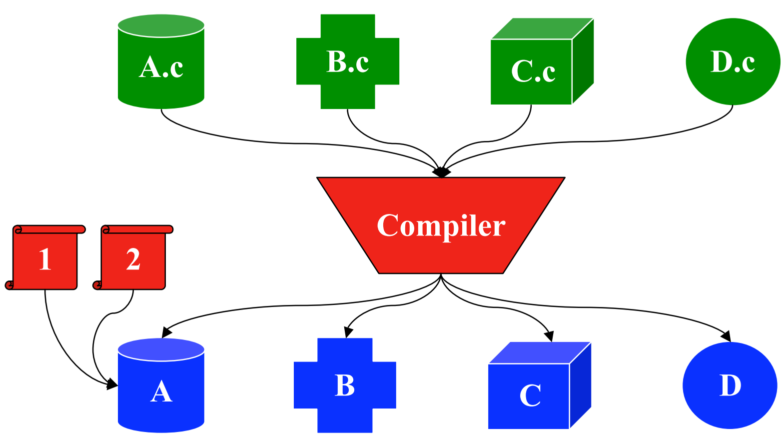
Outline of Results

- High-water mark
  - Create compilers specific to a given application and a given data set
  - Essentially partially evaluating the application
- Application-specific compilers
  - Compiler trained for a given application and data set, but run with an alternate data set
- General-purpose compiler
  - Compiler trained on multiple applications and tested on an unrelated set of applications

Training the Priority Function

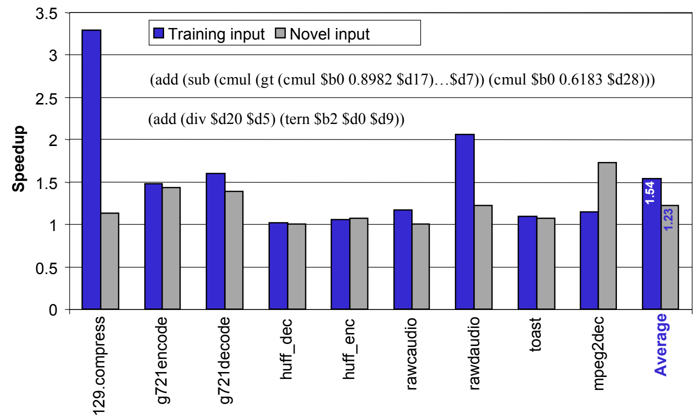


Training the Priority Function
Application-Specific Compilers



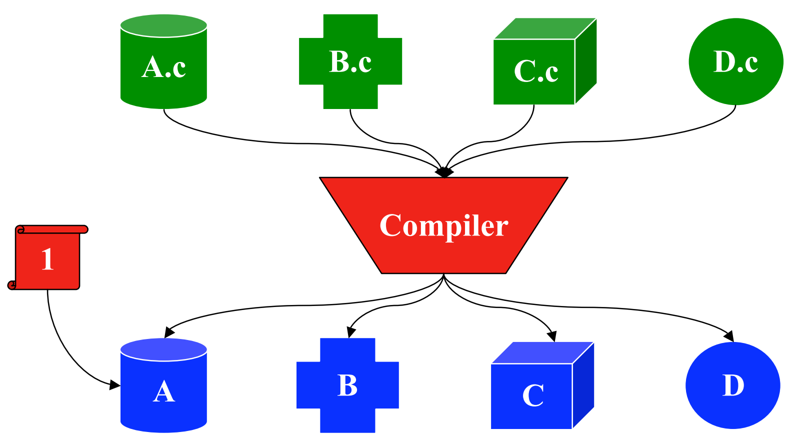
Hyperblock Results

Application-Specific Compilers (High-Water Mark)



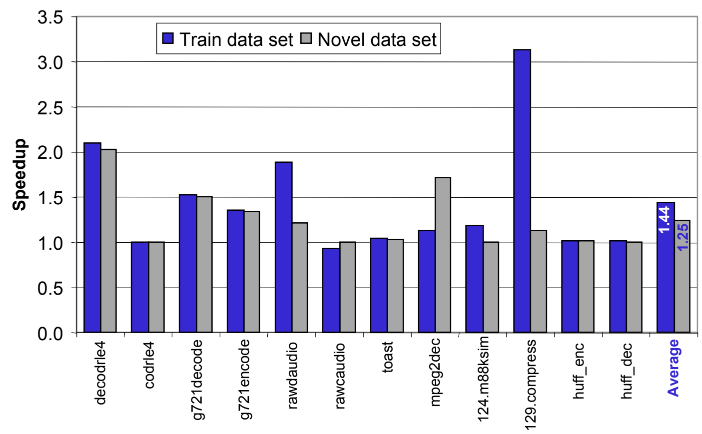
Training the Priority Function

General-Purpose Compilers



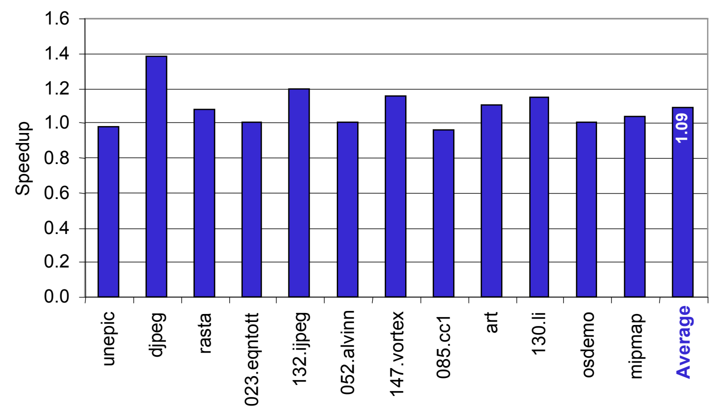
Hyperblock Results

General-Purpose Compiler



Validation of Generality

Testing General-Purpose Applicability



Running Time

- Application specific compilers
  - ~1 day using 15 processors
- General-purpose compilers
  - Dynamic Subset Selection [Gathercole]
    - Run on a subset of the training benchmarks at a time
  - Memoize fitnesses
  - ~1 week using 15 processors
- This is a one time process!
  - Performed by the compiler vendor

GP Hyperblock Solutions

General Purpose

```
(add
 (sub (mul exec_ratio_mean 0.8720) 0.9400)   } Intron that doesn't affect solution
 (mul 0.4762
   (cmul (not has_pointer_deref)
     (mul 0.6727 num_paths)
     (mul 1.1609
       (add (sub
               (mul (div num_ops dependence_height) 10.8240)
               exec_ratio)
            (sub (mul (cmul has_unsafe_jsr predict_product_mean 0.9838)
                      (sub 1.1039 num_ops_max))
                 (sub (mul dependence_height_mean num_branches_max)
                      num_paths)))))))
```

Intron that doesn't affect solution

GP Hyperblock Solutions

General Purpose

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

Favor paths that don't have pointer dereferences

GP Hyperblock Solutions

General Purpose

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


Favor highly parallel (fat) paths

GP Hyperblock Solutions

General Purpose

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

If a path calls a subroutine that may have side effects, penalize it

Our Proposal

- Use machine-learning techniques to automatically search the priority function space
  - Increases compiler's performance
  - Reduces compiler design complexity

Eliminate the Human from the Loop

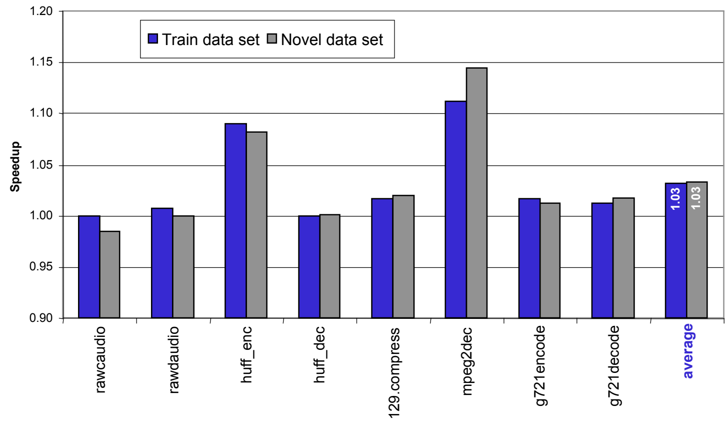
- So far we have tried to improve *existing* priority functions
  - Still a lot of person-hours were spent creating the initial priority functions
- Observation: the human-created priority functions are often eliminated in the 1st generation
- What if we start from a completely random population (no human-generated seed)?

Another Example
Register Allocation

- An old, established problem
  - Hundreds of papers on the subject
- Priority-Based Register Allocation [Chow,Hennessey]
  - Uses a priority function to determine the worth of allocating a register
- Let's throw our GP system at the problem and see what it comes up with

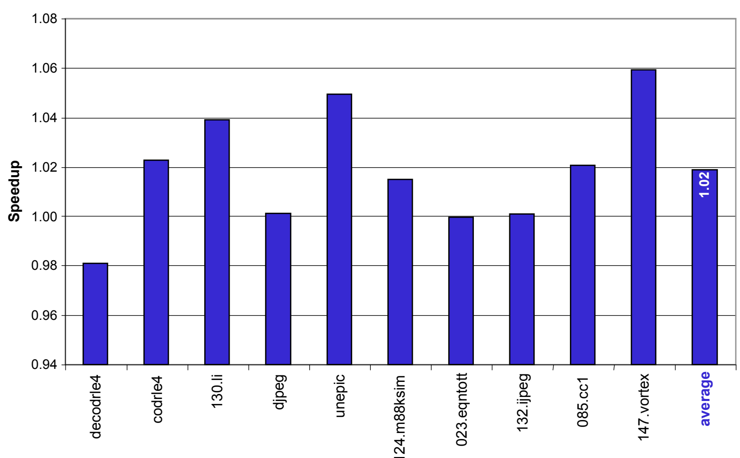
Register Allocation Results

General-Purpose Compiler



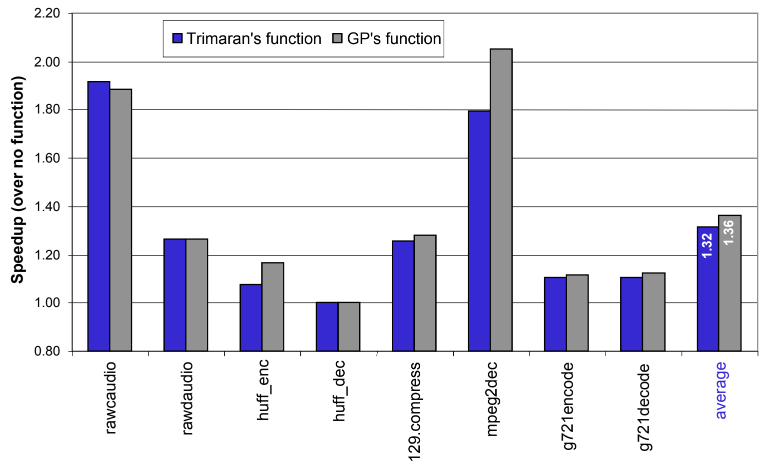
Validation of Generality

Testing General-Purpose Applicability



Importance of Priority Functions

Speedup over a constant priority function



Advantages our System Provides

- Engineers can focus their energy on more important tasks
- Can quickly retune for architectural changes
- Can quickly retune when compiler changes
- Can provide compilers catered toward specific application suites
  - e.g., consumer may want a compiler that excels on scientific benchmarks

Related Work

- Calder et al. [TOPLAS-19]
  - Fine tuned static branch prediction heuristics
  - Requires a priori classification by a supervisor
- Monsifrot et al. [AIMSA-02]
  - Classify loops based on amenability to unrolling
  - Also used a priori classification
- Cooper et al. [Journal of Supercomputing-02]
  - Use GAs to solve phase ordering problems

Conclusion

- Performance talk and a complexity talk
- Take a huge compiler, optimize one priority function with GP and get exciting speedups
- Take a well-known heuristic and create priority functions for it *from scratch*
- There's a lot left to do

Why Genetic Programming?

- Many learning techniques rely on having pre-classified data (*labels*)
  - e.g., statistical learning, neural networks, decision trees
- Priority functions require another approach
  - Reinforcement learning
  - Unsupervised learning
- Several techniques that might work well
  - e.g., hill climbing, active learning, simulated annealing

Why Genetic Programming

- Benefits of GP
  - Capable of searching high-dimensional spaces
  - It is a distributed algorithm
  - The solutions are human readable
- Nevertheless…there are other learning techniques that may also perform well

Genetic Programming

Reproduction