Genetic algorithms and code optimization

Devika Subramanian
Rice University

Work supported by DARPA and the USAF Research Labs

A quiet revolution

- (May 1997) Deep Blue vs Kasparaov
  - first match won against world champion.
  - intelligent and creative play.
  - Kasparov: "I could feel -- I could smell -- a new kind of intelligence across the table".

This was a qualitative difference from previous brute-force results.
A quiet revolution (contd)

- Deep Space One
  - Rich models combined with significant offline computation on those models can provide fast, real-time responses to autonomously steer complex spacecraft millions of miles from Earth.

A quiet revolution (contd)

- TD gammon
  - World champion backgammon player, learns from the rules of the game alone by playing millions of games against itself!
A quiet revolution (contd)

- Planning and scheduling
  - large scale problems in Hubble telescope scheduling and logistics planning.
  - do better than the best special-purpose algorithms for planning and scheduling.

What’s fueling the revolution

- Faster hardware, and hardware implementations of core computations.
- New stochastic search algorithms: GAs, simulated annealing, GSAT
- A fundamental shift from knowledge-intensive to compute-intensive approaches, i.e. a move from special-purpose algorithms to general search techniques for solving optimization problems.
Compilers and the revolution

Compilers are not using their "share" of available cycles. Can compilers benefit from the compute-intensive revolution?

How do compilers work?

Front end handles source-language & generates IR
- Problems are mostly solved, mostly $O(n)$ time
- Rely heavily on automation

Middle end (optimizer) improves the IR program
- Problems change with architecture & language
- Most methods take $O(n)$ to $O(n^2)$ time

Back end maps IR program onto the target machine
- Allocation, scheduling, placement
- Features of idiosyncratic processors

Lots of hard optimization problems in the middle and back end!
Optimizing for reduced code space

- What is the best order of application for the optimizations?
  > Long-standing open question
- Is the notion of “best” program-specific?
  > Might produce better code \[\text{better} \Rightarrow \text{smaller or faster}\]

Why is problem difficult?

- Many optimizations available.
- Interactions between optimizations not well understood.
  - Difficult to analytically predict the impact of an optimization sequence on a program.
- Optimization sequences affect different programs differently.
The solution space

Large solution space. Discrete, non-linear objective function. How do we intelligently sample the space to get a good solution?

Genetic algorithms

- Search algorithms based on the mechanics of natural selection.
- A highly simplified computational model of biological evolution.
- Developed by John Holland in the 60s.
A genetic algorithm at work

Find setting of switches that maximizes reward.

Outline of a GA

- Set up initial population of solutions.
- Generate successive populations using
  - selection
  - crossover
  - mutation
- Repeat generation until no further improvement in reward.
Generating initial population

- Start with a number of random guesses.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- The population size is 4.

Safety in numbers.

Generating successive populations

- Selection
  - a solution is retained for the next generation in proportion to its reward (fitness).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Reward</th>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>169</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>576</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>361</td>
</tr>
</tbody>
</table>

- Analog of "survival of the fittest".
Generating successive populations

- The mating pool

1  0 1 1 0 1  169  
2  1 1 0 0 0  576  
3  1 1 0 0 0  576  
4  1 0 0 1 1  361  

- 2 copies of the best solution in mating pool, and worst solution is dropped!

Generating successive populations

- Crossover
  - Pairs of solutions are chosen randomly from mating pool and crossed over at a randomly selected crossover point.
  - Analog of sexual reproduction.

parents  0 1 1 0 1  0 1 1 0 1  offspring  
          1 1 0 0 0  1 1 0 0 0  

Crossover point
Generating successive populations

- Crossover combines elements in two good solutions to generate even better ones.

<table>
<thead>
<tr>
<th>Mating pool</th>
<th>mate</th>
<th>New population</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 1</td>
<td>169</td>
<td>2 0 1 1 0 0</td>
</tr>
<tr>
<td>2 1 1 0 0 0</td>
<td>576</td>
<td>1 1 1 0 0 1</td>
</tr>
<tr>
<td>3 1 1 0 0 0</td>
<td>576</td>
<td>4 1 1 0 1 1</td>
</tr>
<tr>
<td>4 1 0 0 1 1</td>
<td>361</td>
<td>3 1 0 0 0 0</td>
</tr>
</tbody>
</table>

Average fitness of new population = 439
Average fitness of initial population = 293

Generating successive populations

- Mutation
  - each bit in each solution is flipped with a very small probability.
  - Analog of mutation in nature.
  - Insurance policy against premature loss of important subparts of a solution.
Generating successive populations

Why do GAs work?

- Independent sampling is provided by large populations that are initialized randomly.
- High fitness solutions are preserved through selection, and this biases the sampling process toward regions of high fitness.
- Crossover combines partial solutions, called “building blocks”, thus exploiting the parallelism provided by maintaining a population of solutions.
- Mutation guards against premature loss of diversity in population.
When are GAs inappropriate?

- When exact global optima are needed.
- When any guarantee on quality of solution or convergence time is needed.
- When "appropriate" representations of solutions are not available.

Extensions to GAs

- **Messy GAs**: individual solutions represented as variable length strings.
- **Genetic programming**: individual solutions represented as s-expressions (programs in Scheme or Lisp).
- This field now goes by the name Evolutionary Computation.
A partial list of GA applications

- Designing jet engines (GE)
- Designing walking strategies for legged robots.
- Scheduling job shop.
- Classifying news stories for Dow Jones.
- Creating art, jazz improvisations.
- TSP.
- Drug design.
- Etc. etc...

GAs & code space optimization

- Formulation
  - 10 optimizations, population of 20 solutions (of length 12)
  - Test fitness by measuring code size, number of operations executed as secondary fitness criteria

Appears in LCTES 99
**Optimizations used**

- Constant propagation (cprop) \( c \)
- dead code elimination (dead) \( d \)
- empty basic block removal (clean) \( n \)
- global value renumbering (valnum) \( v \)
- Lazy code motion (lazy) \( z \)
- Partial redundancy elimination (partial) \( l \)
- Peephole optimization (combine) \( o \)
- Reassociation (shape) \( r \)
- register coalescing (coalesce) \( s \)
- operator strength reduction (strength) \( t \)

**GAs & code space optimization**

- The algorithm
  - compute fitness values & rank the 20 solutions.
  - Discard the worst + 3 chosen at random from 11–19.
  - Generate 4 new chromosomes from crossover using 1–10.
  - Mutate survivors from 2 — 19. (elitism excludes top)
- Run 1000 generations (6-8 hours on a $6K workstation). [But we observed convergence in 200-300 generations]
Experiments

- We ran the GA to find optimization sequences for several benchmark programs
  - Fortran: fmin, rkf45, seval, solve, svd, urand, zeroin (FMM benchmarks), tomcatv (SPEC).
  - C: adpcm, compress, fft, dfa, dhrystone, nsieve.

Experimental results

<table>
<thead>
<tr>
<th>Code</th>
<th>GA/unoptimized</th>
<th>Gen found</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>19.90%</td>
<td>6</td>
</tr>
<tr>
<td>compress</td>
<td>24.80%</td>
<td>77,79</td>
</tr>
<tr>
<td>dfa</td>
<td>36.50%</td>
<td>806</td>
</tr>
<tr>
<td>dhrystone</td>
<td>29.50%</td>
<td>22,920</td>
</tr>
<tr>
<td>fft</td>
<td>27.20%</td>
<td>2</td>
</tr>
<tr>
<td>fmin</td>
<td>50%</td>
<td>32</td>
</tr>
<tr>
<td>nsieve</td>
<td>42.80%</td>
<td>0,189</td>
</tr>
<tr>
<td>rkf45</td>
<td>51.10%</td>
<td>74</td>
</tr>
<tr>
<td>seval</td>
<td>72.90%</td>
<td>39</td>
</tr>
<tr>
<td>solve</td>
<td>57.30%</td>
<td>33,58</td>
</tr>
<tr>
<td>svd</td>
<td>53.40%</td>
<td>26</td>
</tr>
<tr>
<td>tomcatv</td>
<td>75.50%</td>
<td>90</td>
</tr>
<tr>
<td>urand</td>
<td>54.40%</td>
<td>0,18</td>
</tr>
<tr>
<td>zeroin</td>
<td>45.10%</td>
<td>239,270</td>
</tr>
<tr>
<td>average</td>
<td>45.70%</td>
<td></td>
</tr>
</tbody>
</table>
Experimental results

<table>
<thead>
<tr>
<th>Code</th>
<th>GA/old default</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>3.00%</td>
</tr>
<tr>
<td>compress</td>
<td>6.70%</td>
</tr>
<tr>
<td>dfa</td>
<td>5.20%</td>
</tr>
<tr>
<td>dhrystone</td>
<td>6.60%</td>
</tr>
<tr>
<td>fft</td>
<td>10.90%</td>
</tr>
<tr>
<td>fmin</td>
<td>8%</td>
</tr>
<tr>
<td>nsieve</td>
<td>11.00%</td>
</tr>
<tr>
<td>rk45</td>
<td>10.50%</td>
</tr>
<tr>
<td>seval</td>
<td>8.00%</td>
</tr>
<tr>
<td>solve</td>
<td>28.20%</td>
</tr>
<tr>
<td>svd</td>
<td>40.80%</td>
</tr>
<tr>
<td>tomcatv</td>
<td>28.40%</td>
</tr>
<tr>
<td>urand</td>
<td>0.00%</td>
</tr>
<tr>
<td>zeroin</td>
<td>5.10%</td>
</tr>
<tr>
<td>average</td>
<td>12.30%</td>
</tr>
</tbody>
</table>

Old default sequence = rvzcodtvzcod

Experimental results

<table>
<thead>
<tr>
<th>Code</th>
<th>GA/new default</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>1.40%</td>
</tr>
<tr>
<td>compress</td>
<td>0.50%</td>
</tr>
<tr>
<td>dfa</td>
<td>3.30%</td>
</tr>
<tr>
<td>dhrystone</td>
<td>1.50%</td>
</tr>
<tr>
<td>fft</td>
<td>0.00%</td>
</tr>
<tr>
<td>fmin</td>
<td>6%</td>
</tr>
<tr>
<td>nsieve</td>
<td>0.00%</td>
</tr>
<tr>
<td>rk45</td>
<td>0.80%</td>
</tr>
<tr>
<td>seval</td>
<td>3.00%</td>
</tr>
<tr>
<td>solve</td>
<td>0.20%</td>
</tr>
<tr>
<td>svd</td>
<td>0.10%</td>
</tr>
<tr>
<td>tomcatv</td>
<td>2.50%</td>
</tr>
<tr>
<td>urand</td>
<td>0.00%</td>
</tr>
<tr>
<td>zeroin</td>
<td>2.60%</td>
</tr>
<tr>
<td>average</td>
<td>1.50%</td>
</tr>
</tbody>
</table>

New default sequence = nodvcoevs
Bottom line

- **GA did better than any fixed sequence.**
  - Beat the compiler's default string (used for five years).
- **GA showed us how to construct a better fixed sequence.**
  - Beats the old default sequence in code size (12.3% on average)
  - produces faster compilations and smaller code.
- **Program specific solutions beat both fixed sequences.**
  - beat new default by up to 7%, and old default by up to 41%.

So what?

- Use GAs to "tune up" your compiler. The GA can develop better fixed optimization sequences.
- If your code is within 10% of fitting in ROM, let the GA crunch on it!
- Use GA to develop customized optimization sequences for specific application code.
Current work

- Exploring other objective functions
  - power consumption (battery powered systems)
  - multi-objective optimizations.
- Program specific optimizations (parallelization of GAs).

Instruction scheduling

A classic compiler solves problems by making a series of local decisions.
It constructs a solution and declares victory.
If we gave it more time, it would plant the flag and finish early!

How can we use intensive computation to improve instruction scheduling?
Stochastic search algorithms

- Start with an approximate solution & improve it.
- Use randomization & restart to explore the solution space.
- Gaining two kinds of knowledge
  - New techniques for scheduling
  - Understanding both the problem & the solution space

Iterative repair algorithm

The IR technique starts with an approximate solution.
It improves the solution in a small number of steps.
It uses randomization & restart to explore different parts of the solution space.
Given more time, it plants the flag in more places.
It can often use more time constructively.
Results

- Built a series of schedulers based on iterative repair.
- Relatively small improvements over list scheduling on available benchmarks with respect to running time.
- Found better schedules with respect to other dimensions (register pressure).

Results

Modest performance of IR led us to a detailed study of space of scheduling problems
- what class of scheduling problems are hard?
- we developed a metric for determining when more complex IR technique will yield substantial improvements over greedy list scheduling.
- yielded new technique for coupling scheduling with register allocation with very promising initial results.
Conclusions

- Compute intensive techniques (GAs) can generate custom tailored sequences for code space optimization that are significant improvements over what current algorithms can offer.
  - Next stop: optimizing power consumption.
- Compute intensive techniques (Iterative repair) can generate instruction schedules that combine constraints of scheduling and register allocation better than known special-purpose algorithms.