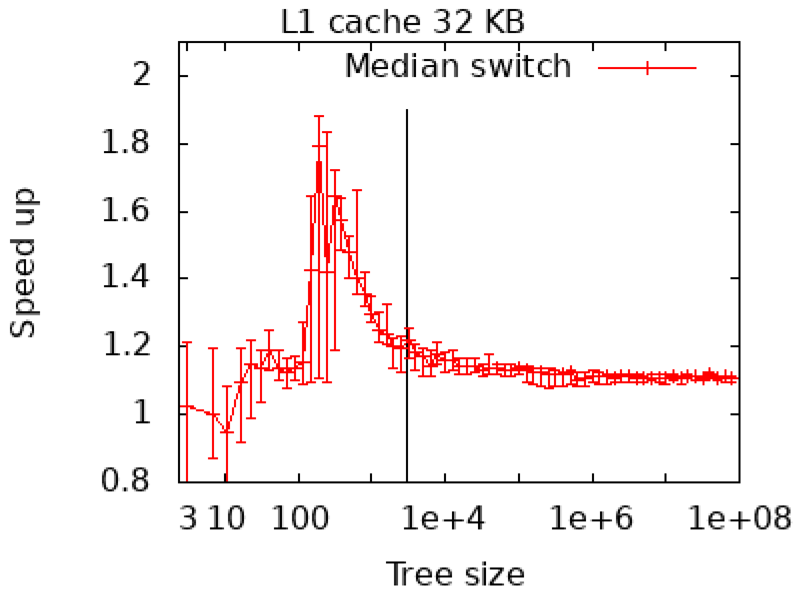
Genetic Improvement of Genetic Programming

GI @ WCCI 2020 special session [1]

W. B. Langdon

Genetic Improvement



slide 1 of 21

28.6.2020

Genetic Improvement of Genetic Programming

- Applying Genetic Improvement to own parallel C++ Genetic Programming system
- Intel AVX parallel vector instructions
  - AVX-512 does 16 float operations in parallel
- GPavx[2] based on Singleton's GPquick[1994]

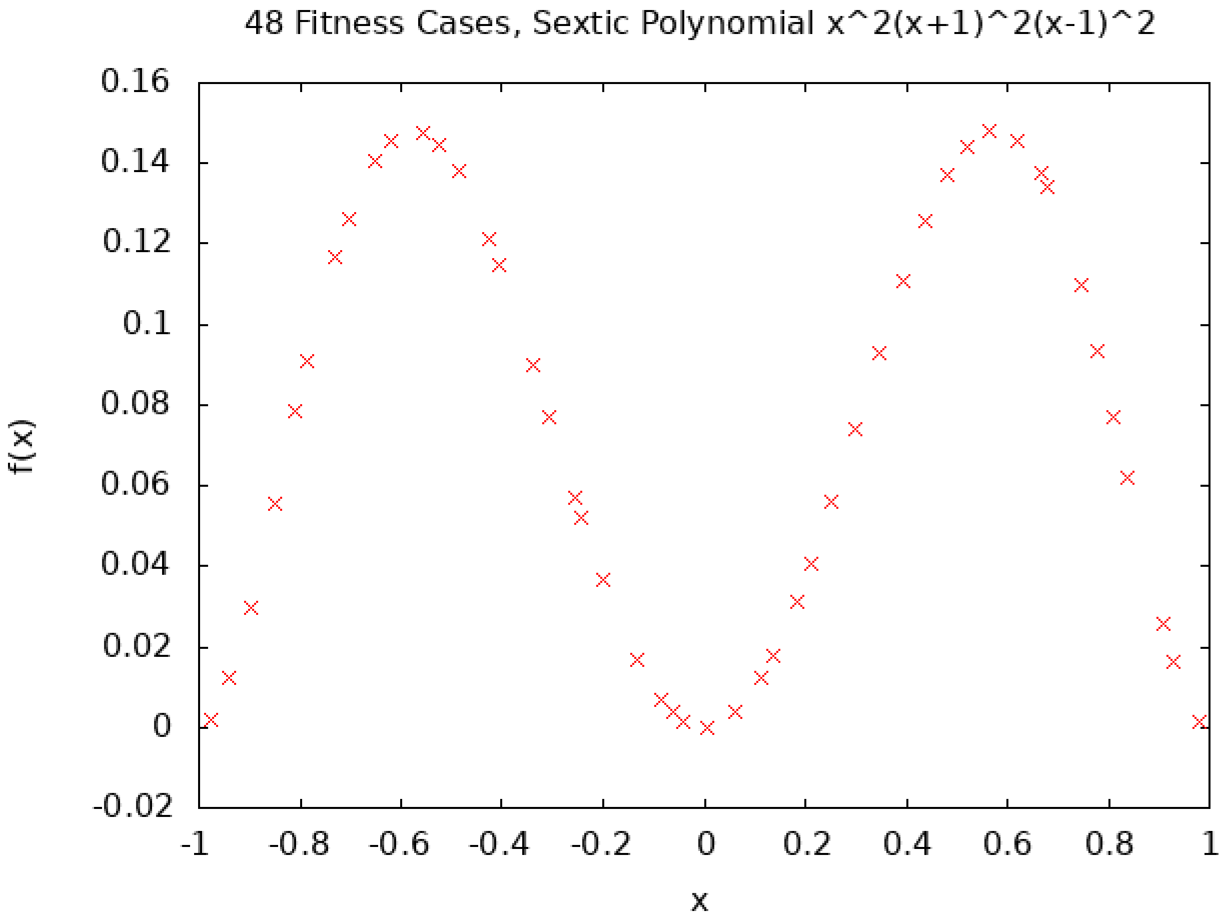
  http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/GPavx.tar.gz
- GPavx can evolve trees of 100 million [3]
- Takes days. Overhead is AVX interpreter
- Can GI on interpreter do better?

GI’s Target: Big Genetic Programming

- Demo problem
- No tree size limit
- Run up to 1 million generations
- Trees evolve greater 100 million

  (GP continues to find improvements [3])
- Run on 46GB multiple core Intel server
- Parallel
  - multiple trees: one pthread per core
  - multiple fitness cases: 3×AVX = 48 eval in parallel

Big Genetic Programming Benchmark



Sextic polynomial: match curve at 48 points

(48 chosen since multiple of 16)

GPquick Eval Interpret GP tree

- Recursive Eval. Start at root node. Examples:

1. AddEval binary function

   return EVAL + EVAL;

2. XEval leaf

   return x[i]; value of x at ith fitness test case

- Implicit use of system stack
- Eval tree once per fitness test case

GPquick Eval interpret GP tree

- GP tree flattened into linear string
- One byte per tree node, 255 opcodes
- For efficiency [1994a], opcode byte is index of table of <256 functions, call the function
  - – function table occupies 32 cache lines

GPavx Interpret Opcode

- GPavx explicit stack, vector of 48 floats
  - Eval tree in one pass (not 48)
  - 3 cache lines per tree depth
- E.g. AddEval() args top and next on stack

```
EVAL;
EVAL;
push(*sp1 + *sp0); add 48 pairs of floats
                   48 results left on top of stack
```

- 98 lines of C++ code

Applying Genetic Improvement to GPavx

- CEC-2020 paper [1] has lots of experiments, concentrate on last one.
- Evolve fast mutant for random tree of size of about twenty million
- Show mutant generalises to trees of size 3 to 100 million

Applying Genetic Improvement to GPavx

- Automatically convert C++ to grammar
- Evolve grammar: mutation and crossover
- Fitness:
  - Does mutant compile, run, return right answers
  - Test on random tree (change each generation)
  - Compare with original code
    - Are all 48 answers the same?
    - Run time?
- Select better half of population as parents

Fitness Function, wall clock time

- Each mutation run independently (own exe)
- Run on multi-user AVX-512 server
  – Load varies with other users
  – Server dynamically changes each CPU core's clock frequency (1.00-3.00 GHz nominal 2.30)
  – Operating system sometimes moves process between cores
- Noise!

(less with unix perf stat instruction:u ?)

Combating Fitness Noise

- Use single thread
- Performance relative to original code
  - run original & mutant close together (tight loop)
  - small fast (≈0.1μS) trees usually on same core
  - usually same clock frequency
- 11 small runs. Difference in 1st quartile time
- Only run fast mutants on big trees[4] (≈0.5sec)
  - Noise proportionately less, so run just once

Six EVAL environments

3 conditional compilation –D macros

- New switch code v. Jump table
- Internal(system) v. External stack
- (if internal) process 16 test cases together and so interpret tree 3 times v. All 48

Gives 6 options

Six GPavx EVAL grammars

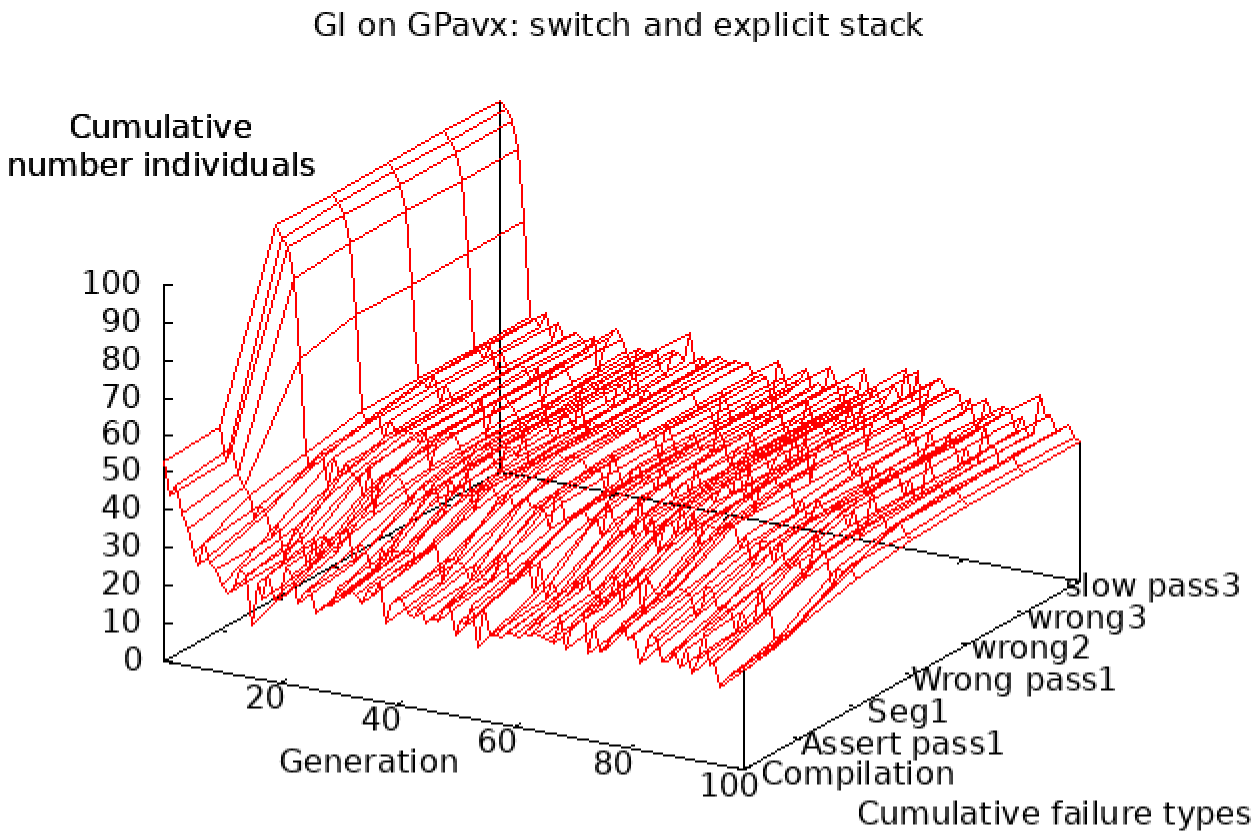
- Each option has own grammar
  - 5694 functions from Intel Intrinsics library
  - Most variable rules are type line
  - Few other types
- Concentrate upon option 010 (switch and explicit stack) as most successful.
- All six run ok

	000	001	010	100	101	110
Line	43	45	58	43	45	58
Others	6	12	28	6	12	28
Total	49	57	86	49	57	86

6 runs

- Population 100, up to generation 100.
- Takes ≈11 hours (3.8 seconds per mutant)

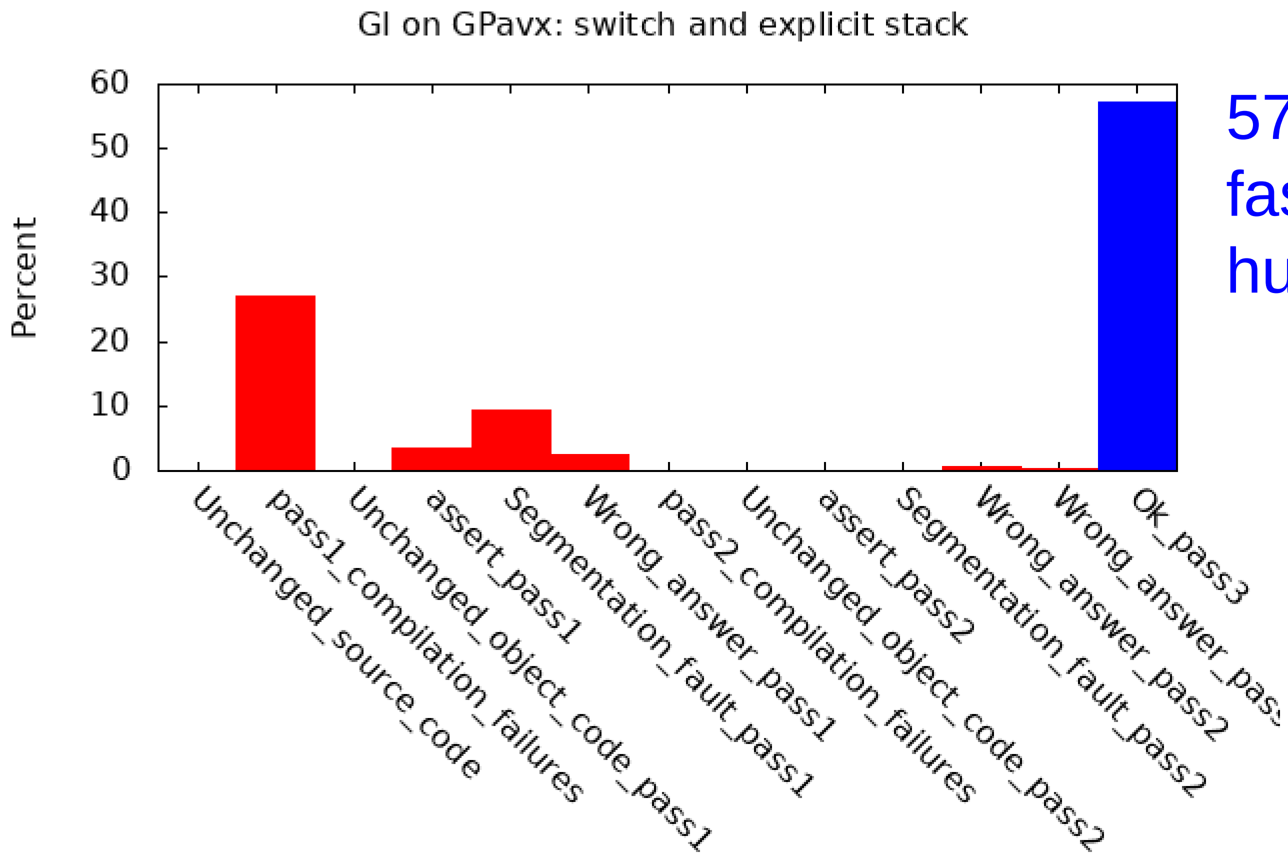
Number of mutants which fail at each generation



Overall 57% give pass3 speedup, 27% fail to compile, 9% segfault in pass1.

Pass 2+3 errors < 1%

Percentage failure type



5744 mutants faster than human code

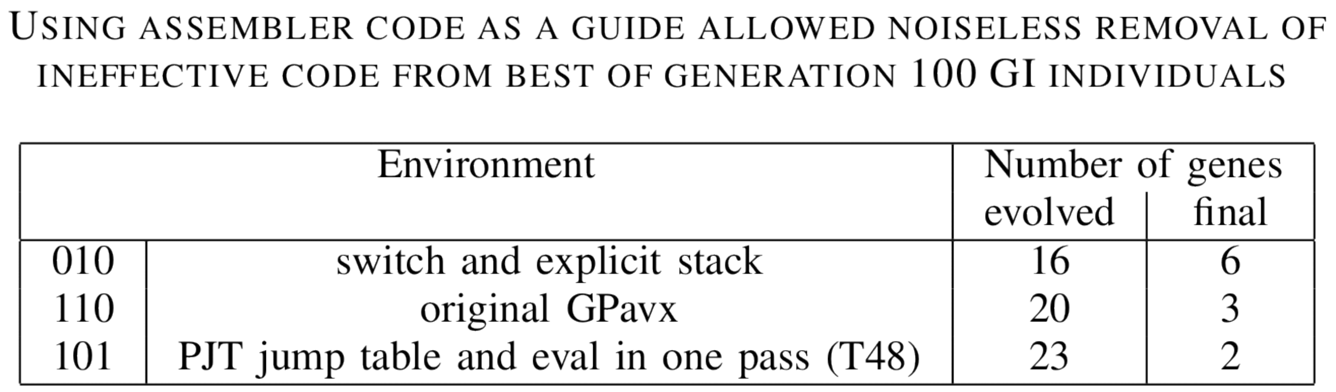
Overall 57% give pass3 speedup, 27% fail to compile, 9% segfault in pass1.

Pass 2+3 errors < 1%

Mutant Clean Up via g++ output

- Use best in last generation.
- Typically bloated (i.e. BNF grammar changes with no external impact)
- Use assembler code from compiler
- Scan evolved mutant one change at a time
  - Remove each change one at a time
  - If assembler code is different restore
    else remove permanently
- Scan again, in case can now remove more

Mutant Clean Up

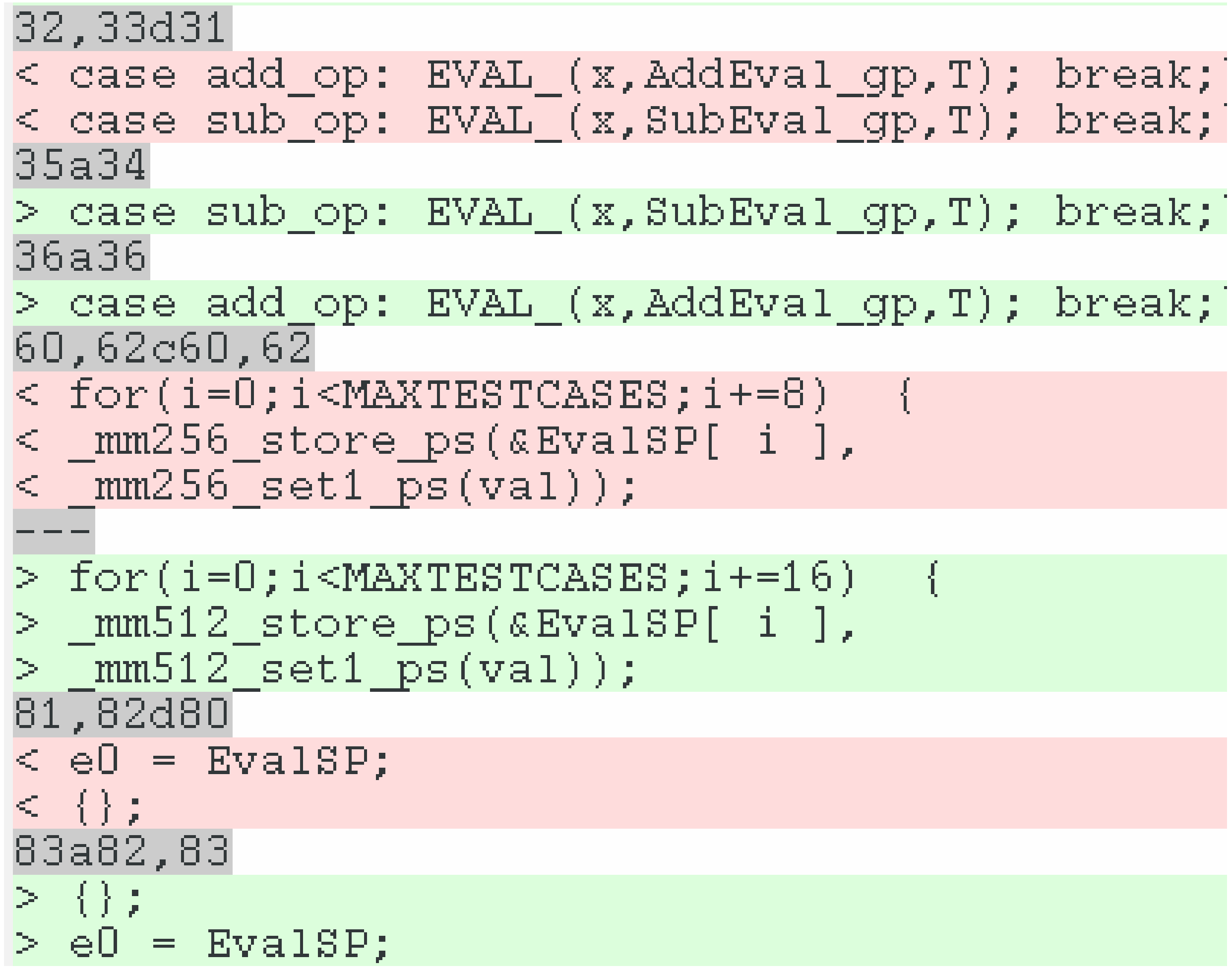


USING ASSEMBLER CODE AS A GUIDE ALLOWED NOISELESS REMOVAL OF INEFFECTIVE CODE FROM BEST OF GENERATION 100 GI INDIVIDUALS

Environment		Number of genes	
		evolved	final
010	switch and explicit stack	16	6
110	original GPavx	20	3
101	PJT jump table and eval in one pass (T48)	23	2

- Only 6 of 16 genes impact assembler code generated by g++ compiler
- No speed up in other three runs 000, 001, 100

Switch Mutant code changes



```
32,33d31
< case add_op: EVAL_(x,AddEval_gp,T); break;
< case sub_op: EVAL_(x,SubEval_gp,T); break;
35a34
> case sub_op: EVAL_(x,SubEval_gp,T); break;
36a36
> case add_op: EVAL_(x,AddEval_gp,T); break;
60,62c60,62
< for(i=0;i<MAXTESTCASES;i+=8)  {
< _mm256_store_ps(&EvalSP[ i ],
< _mm256_set1_ps(val));
---
> for(i=0;i<MAXTESTCASES;i+=16)  {
> _mm512_store_ps(&EvalSP[ i ],
> _mm512_set1_ps(val));
81,82d80
< e0 = EvalSP;
< {};
83a82,83
> {};
> e0 = EvalSP;
```

< Original

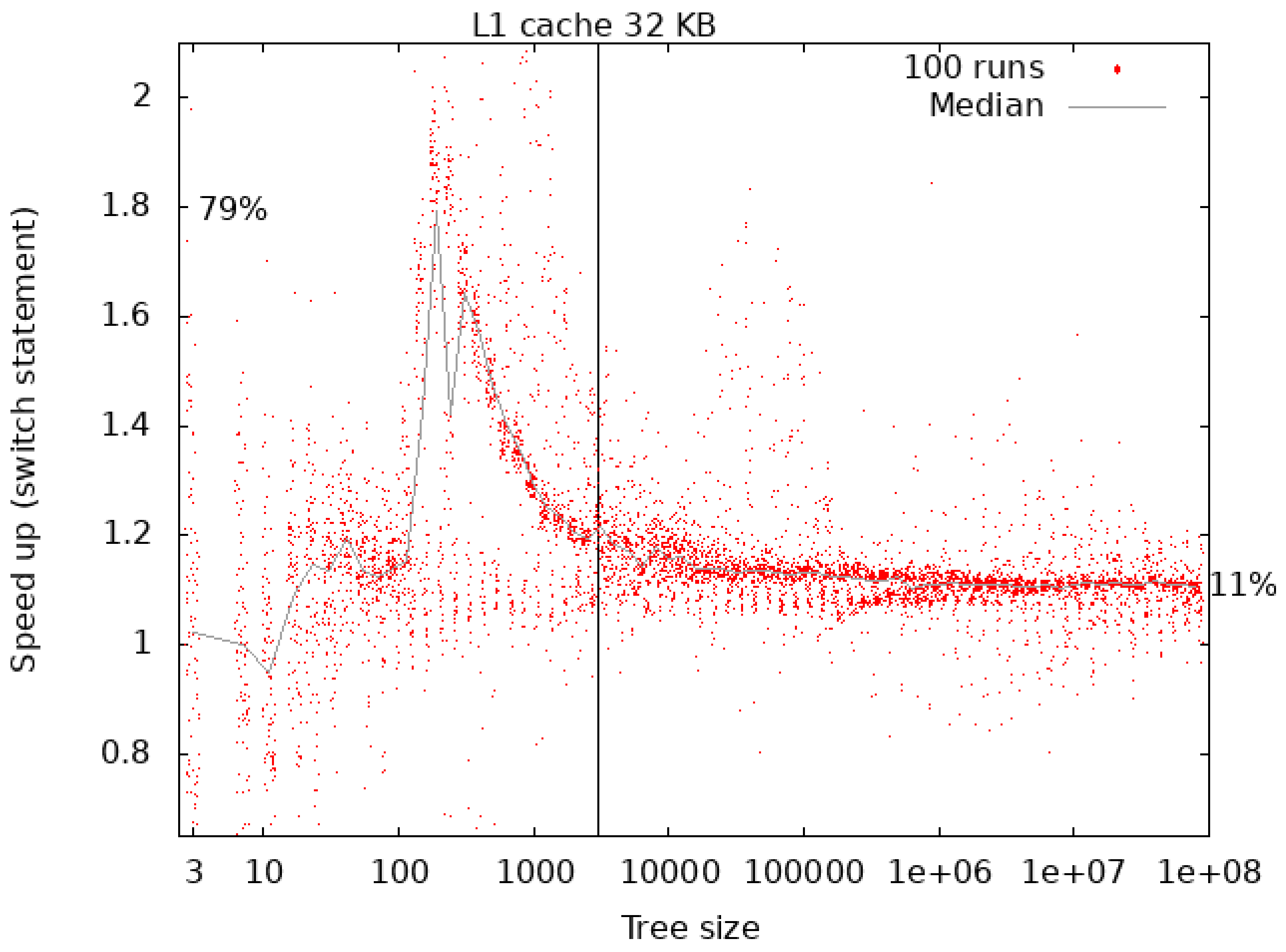
> Mutated code

3 CASE swaps reorder switch

vecsize=16 forces use of AVX-512

Swapping lines 139 and 139 probably no effect

After clean up: Out of Sample Generalisation



Run on 100 random trees of each length, which the evolved code never saw. Gray line is average speed up

Summary: Genetic Improvement of GP

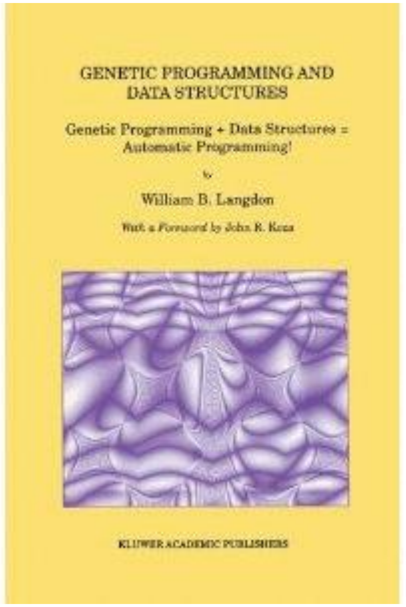
- Have applied GI to own code.
- GI can mutant C++ Genetic Programming interpreter (speed up to 2.1 fold)
- 20+ years established wisdom overturned
  - Jump table forced out of cache so switch faster
- Can use real runtime as fitness
  - even on multiple user cluster with dynamic power management.
- Software is not fragile

END

W. B. Langdon YouTube videos
https://www.youtube.com/channel/UChebBvv66dPOcElWk6ht4OA

Genetic Programming

W. B. Langdon



GENETIC PROGRAMMING AND DATA STRUCTURES

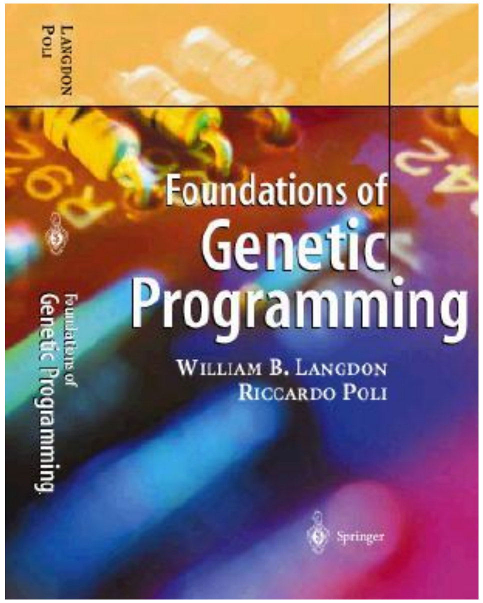
Genetic Programming + Data Structures = Automatic Programming!

by

William B. Langdon

With a Foreword by John R. Koza

KLUWER ACADEMIC PUBLISHERS



LANGDON POLI

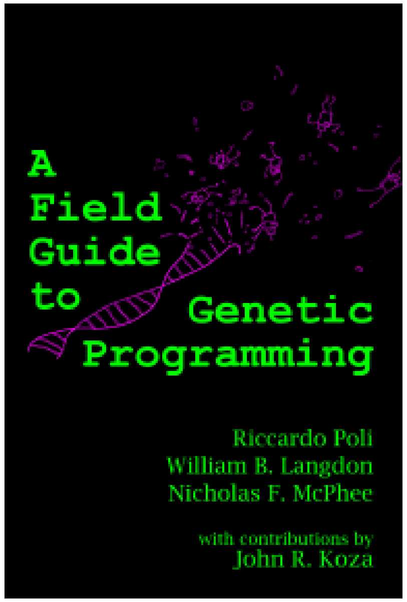
Foundations of Genetic Programming

Foundations of Genetic Programming

WILLIAM B. LANGDON

RICCARDO POLI

Springer



A Field Guide to Genetic Programming

Riccardo Poli

William B. Langdon

Nicholas F. McPhee

with contributions by John R. Koza

Alternative Fast GPs

- GPavx was fastest tree interpreter
- Avoid trees: Linear GP, Cartesian GP
- Avoid interpreting
  - Compile tree to machine code[4]
  - Evolve machine code: Discipulus
- Avoid interpreting whole tree, changes only
  - evolving population of trees as evolving directed acyclic graph holding partial fitness. Eval O(depth) not O(size) (no side effects)[5,6]
- Avoid interpreting dead code (introns)

Typical Genetic Programming

- Random initial population of trees

1. Test each tree, give it fitness score
2. Select better trees to be parents
3. Create next population from parents

- Loop (1.) until done

- Most time is taken by fitness evaluation
  Often use multiple fitness cases.
- Parallel: multiple trees, multiple(48) fitness cases

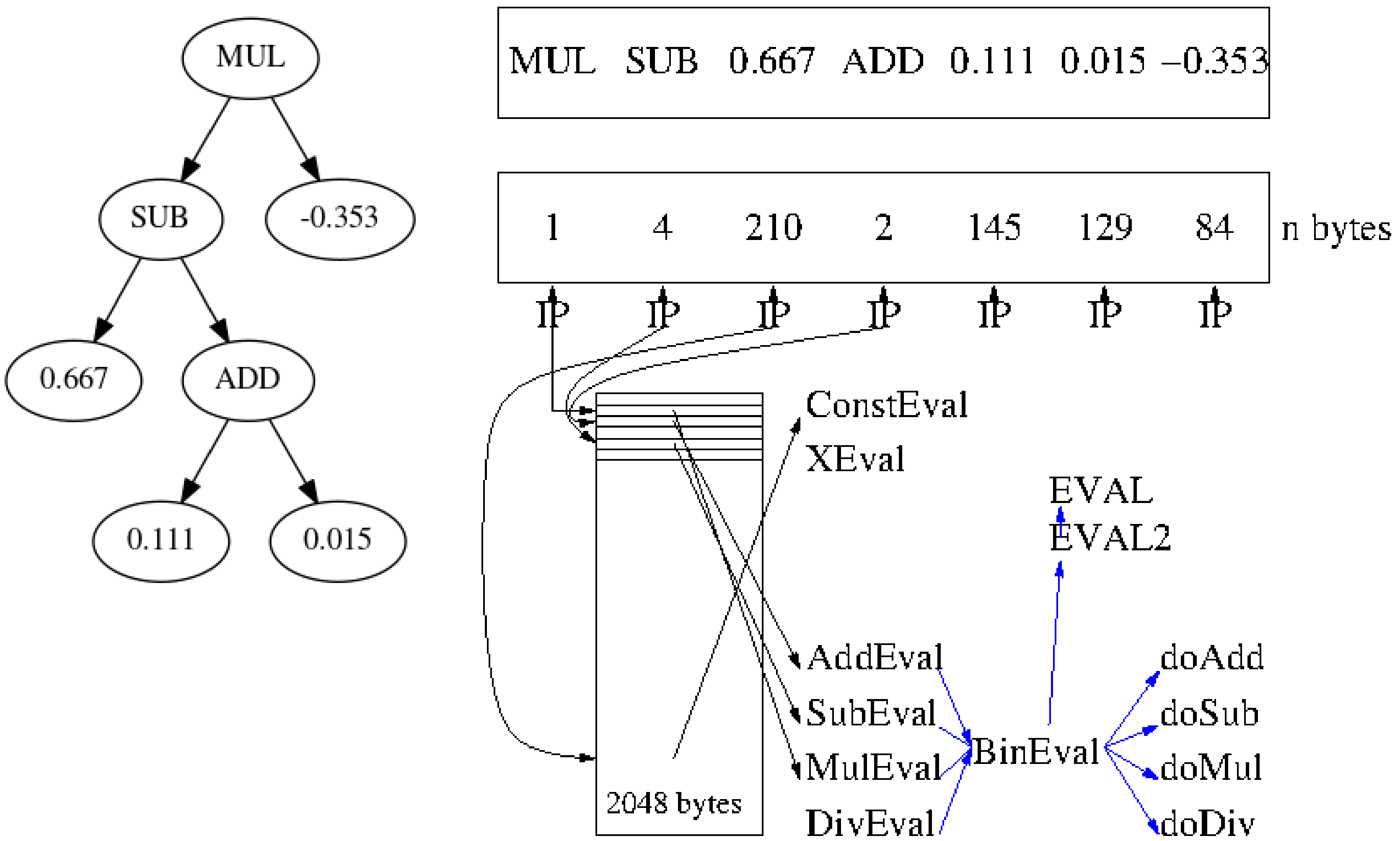
GPquick Eval Interpret GP tree

- GP tree flattened into linear string
- One byte per tree node, 255 opcodes:
  - a few used for functions and for inputs (eg x)
  - most used for leafs (constants, eg -0.995)
- Recursive Eval (once per test case)
  - Extract opcode from linear string
  - Increment pointer into array ++IP
  - Interpret opcode

EVAL

- EVAL (Evalfunc[(++IP)->op])(ip,sp)
- GPquick stores tree in linear array[IP]
- GPquick uses array indexed by op code to call interpreter code for op code.
  - Not switch.
- Tree 1 byte per node
- Up to 255 opcodes, Evalfunc[256] addresses = 256×8bytes = 2048 bytes = 32 cache lines

EVAL (Evalfunc[(++IP)->op])(ip,sp)



Evalfunc[256] array of function addresses

Genetic Improvement on GPavx

- GPavx Eval 98 lines of C++ code
- Written in style of GPquick interpreters
  - OPDEF EVAL EVAL2 BINEVAL macros
- Supports + - × / (protected division) x 0.24
  - DivEval recursively EVAL both arguments
    - dodiv if(arg2==0) return 1.0f else return arg1/arg2
    - Use AVX to do 16 float operations in parallel
- Tight code
- Eval contains six OPDEF(function)

Multi-threaded re-entrant EVAL(ip,sp)

- To support reentrant multi-threading, replaced original global instruction pointer (i.e. point to active tree node) by passing IP as (hidden) argument to EVAL.
- Similarly pass stack pointer as EVAL arg sp
- Explicit stack
- Each thread has own IP and stack

Recursive Evaluation of 48 floats

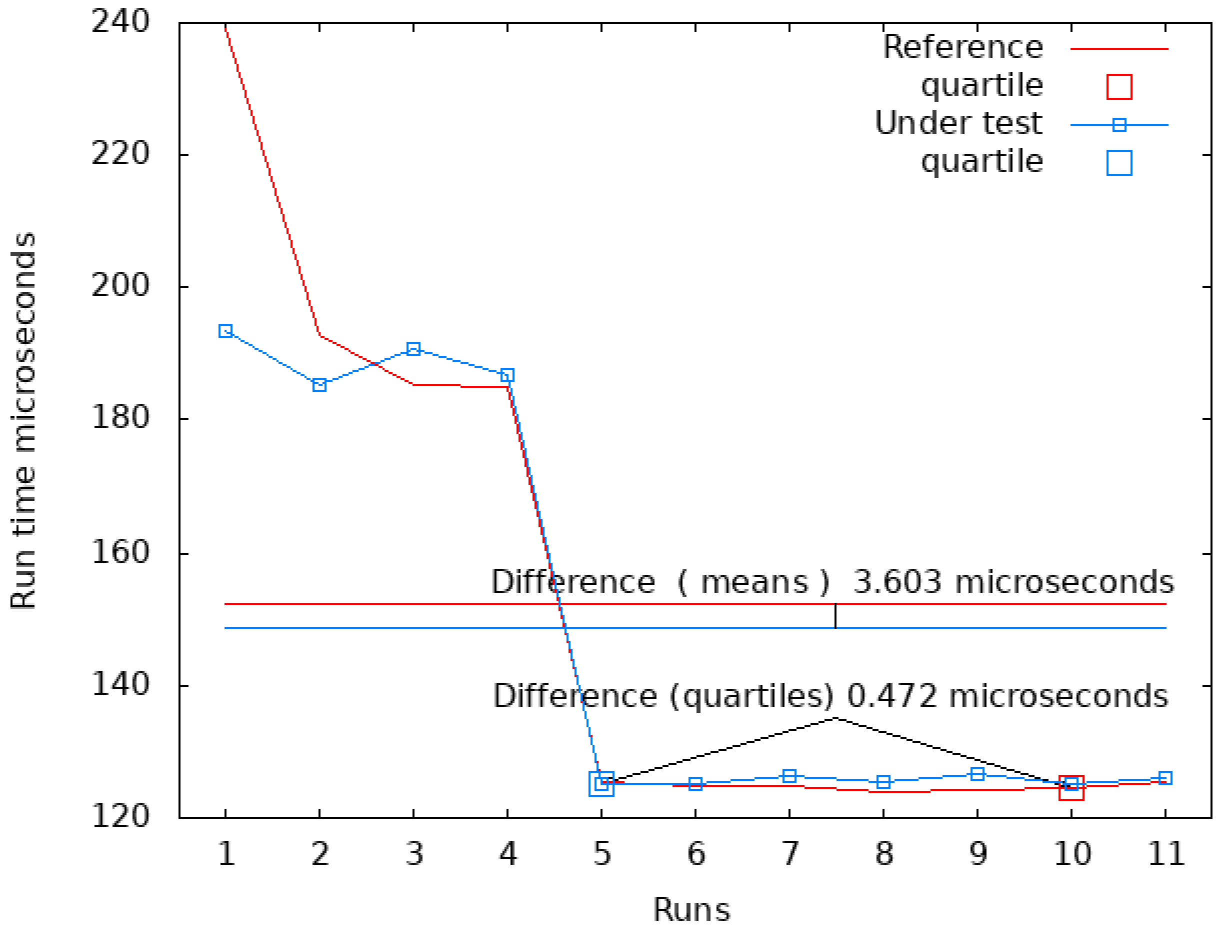
- Each function recursively calls EVAL2.
- EVAL2 calls EVAL twice.
- Each EVAL leaves its answer on the (explicit) stack.
- Function, e.g. AddEval, pops twice, doAdd adds values, AddEval pushes result.
- XEval push value of x onto stack
- ConstEval pushes 48 copies of const
- Outermost Eval returns vector 48 floats

Multiple Test Cases Evaluate 48 in parallel

- Typically tree evaluated once per test case
- AVX do 16 test cases simultaneously.
- By making stack 48 floats wide, can eval whole tree in one pass by doing three AVX (sequentially).
- Eval returns vector 48 floats
- Fitness = for(i=0;i<48;i++) sum += $|\text{error}_i|$

  (not reduction, GPavx gives identical answers)

Use Quartile to combat runtime noise



Example of using quartile difference in run time. Reference code in red. Effectively only use faster half of runs, then take robust average (median). Fitness = 472

Mean not used as dominated by outliers.

avx.cc EVAL with switch macro

```
#define EVAL() \
switch ((++IP)->op) {\
 case mul_op: MulEval_gp(ip,sp); break;         \
 case div_op: DivEval_gp(ip,sp); break;         \
 case sub_op: SubEval_gp(ip,sp); break;         \
 case x_op:   XEval_gp(ip,sp); break;           \
 case add_op: AddEval_gp(ip,sp); break;         \
 default:     ConstEval_gp(ip,sp); break;       \
}
```

```
29              ::= "#define EVAL() "
30              ::= "switch ((++IP)->op) {"
31              ::= <CASE_31>
<CASE_31>       ::= "case add_op: AddEval_gp(ip,sp); break;"
32              ::= <CASE_32>
<CASE_32>       ::= "case sub_op: SubEval_gp(ip,sp); break;"
33              ::= <CASE_33>
<CASE_33>       ::= "case mul_op: MulEval_gp(ip,sp); break;"
34              ::= <CASE_34>
<CASE_34>       ::= "case div_op: DivEval_gp(ip,sp); break;"
35              ::= <CASE_35>
<CASE_35>       ::= "case x_op:   XEval_gp(ip,sp); break;"
36              ::= <CASE_36>
<CASE_36>       ::= "default:     ConstEval_gp(ip,sp); break;"
37              ::= "}"
```

Swap mutation, eg <CASE_32>x<CASE_34>, allows easy re-ordering of case and default statements.

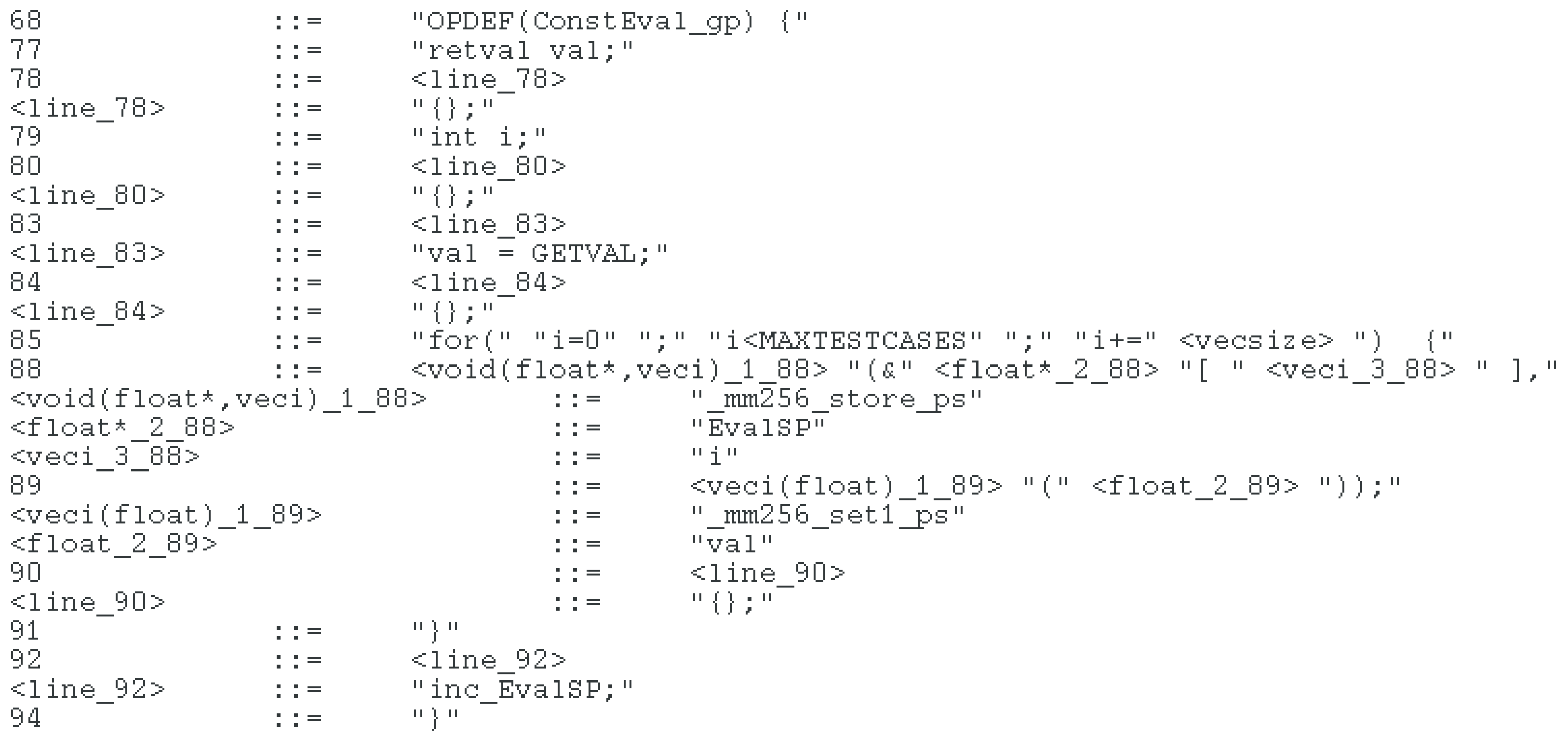
avx.cc Eval Leafs const, x

```
OPDEF(ConstEval_gp) {
  retval val;
  int i;
  val = GETVAL;
  for(i=0;i<MAXTESTCASES;i+=8) {
    _mm256_store_ps(&EvalSP[ i ],
                    _mm256_set1_ps(val));
  }
  inc_EvalSP;
}
OPDEF(XEval_gp){
  int i;
  for(i=0;i<MAXTESTCASES;i+=16)  {
    _mm512_store_ps(&EvalSP[ i ],
                    _mm512_load_ps(&dataX[ i ]));
  }
  inc_EvalSP;
}
```

Can not mutate: declarations, for and }

Comments removed.

avx.cc switch ConstEval grammar



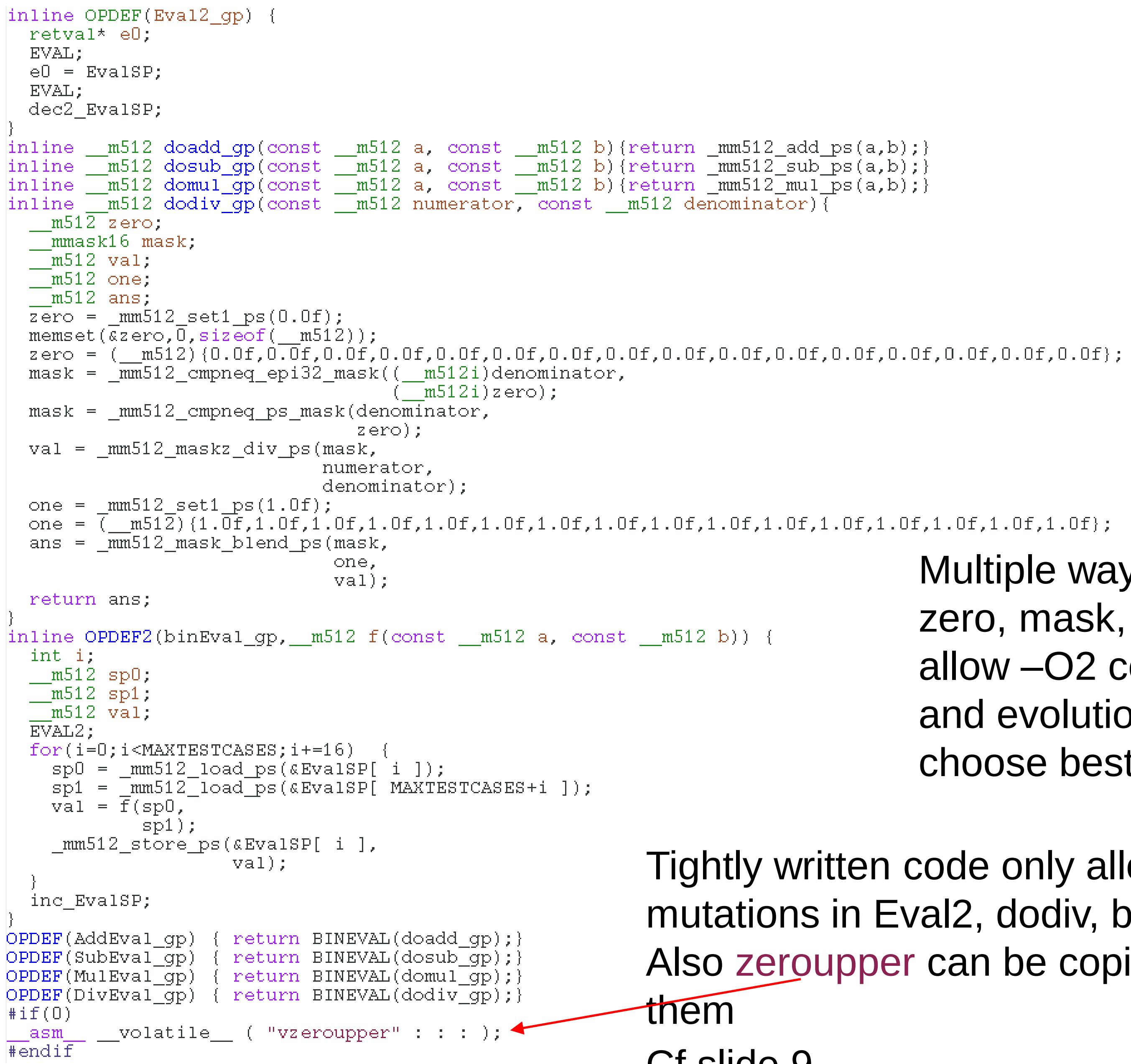
```
68              ::=     "OPDEF(ConstEval_gp) {"
77              ::=     "retval val;"
78              ::=     <line_78>
<line_78>       ::=     "{};"
79              ::=     "int i;"
80              ::=     <line_80>
<line_80>       ::=     "{};"
83              ::=     <line_83>
<line_83>       ::=     "val = GETVAL;"
84              ::=     <line_84>
<line_84>       ::=     "{};"
85              ::=     "for(" "i=0" ";" "i<MAXTESTCASES" ";" "i+=" <vecsize> ")  {"
88              ::=     <void(float*,veci)_1_88> "(&" <float*_2_88> "[ " <veci_3_88> " ],"
<void(float*,veci)_1_88>        ::=     "_mm256_store_ps"
<float*_2_88>                   ::=     "EvalSP"
<veci_3_88>                     ::=     "i"
89                              ::=     <veci(float)_1_89> "(" <float_2_89> "));"
<veci(float)_1_89>              ::=     "_mm256_set1_ps"
<float_2_89>                    ::=     "val"
90                              ::=     <line_90>
<line_90>                       ::=     "{};"
91              ::=     "}"
92              ::=     <line_92>
<line_92>       ::=     "inc_EvalSP;"
94              ::=     "}"
```

Variable rules <type_etc>, e.g. type line, float*

Mutate rules of same type <line_92>x<line_84>

veci rules depend on GI vecsize (4,8 or 16)

avx.cc Eval functions



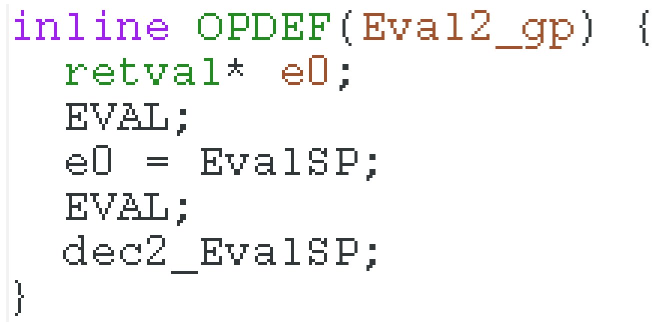
```
inline OPDEF(Eval2_gp) {
  retval* e0;
  EVAL;
  e0 = EvalSP;
  EVAL;
  dec2_EvalSP;
}
inline __m512 doadd_gp(const __m512 a, const __m512 b){return _mm512_add_ps(a,b);}
inline __m512 dosub_gp(const __m512 a, const __m512 b){return _mm512_sub_ps(a,b);}
inline __m512 domul_gp(const __m512 a, const __m512 b){return _mm512_mul_ps(a,b);}
inline __m512 dodiv_gp(const __m512 numerator, const __m512 denominator){
  __m512 zero;
  __mmask16 mask;
  __m512 val;
  __m512 one;
  __m512 ans;
  zero = _mm512_set1_ps(0.0f);
  memset(&zero,0,sizeof(__m512));
  zero = (__m512){0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f};
  mask = _mm512_cmpneq_epi32_mask((__m512i)denominator,
                                  (__m512i)zero);
  mask = _mm512_cmpneq_ps_mask(denominator,
                               zero);
  val = _mm512_maskz_div_ps(mask,
                            numerator,
                            denominator);
  one = _mm512_set1_ps(1.0f);
  one = (__m512){1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f};
  ans = _mm512_mask_blend_ps(mask,
                             one,
                             val);
  return ans;
}
inline OPDEF2(binEval_gp,__m512 f(const __m512 a, const __m512 b)) {
  int i;
  __m512 sp0;
  __m512 sp1;
  __m512 val;
  EVAL2;
  for(i=0;i<MAXTESTCASES;i+=16)  {
    sp0 = _mm512_load_ps(&EvalSP[ i ]);
    sp1 = _mm512_load_ps(&EvalSP[ MAXTESTCASES+i ]);
    val = f(sp0,
            sp1);
    _mm512_store_ps(&EvalSP[ i ],
                    val);
  }
  inc_EvalSP;
}
OPDEF(AddEval_gp) { return BINEVAL(doadd_gp);}
OPDEF(SubEval_gp) { return BINEVAL(dosub_gp);}
OPDEF(MulEval_gp) { return BINEVAL(domul_gp);}
OPDEF(DivEval_gp) { return BINEVAL(dodiv_gp);}
#if(0)
__asm__ __volatile__ ( "vzeroupper" : : : );
#endif
```

Multiple ways to set zero, mask, one, to allow –O2 compiler and evolution to choose best.

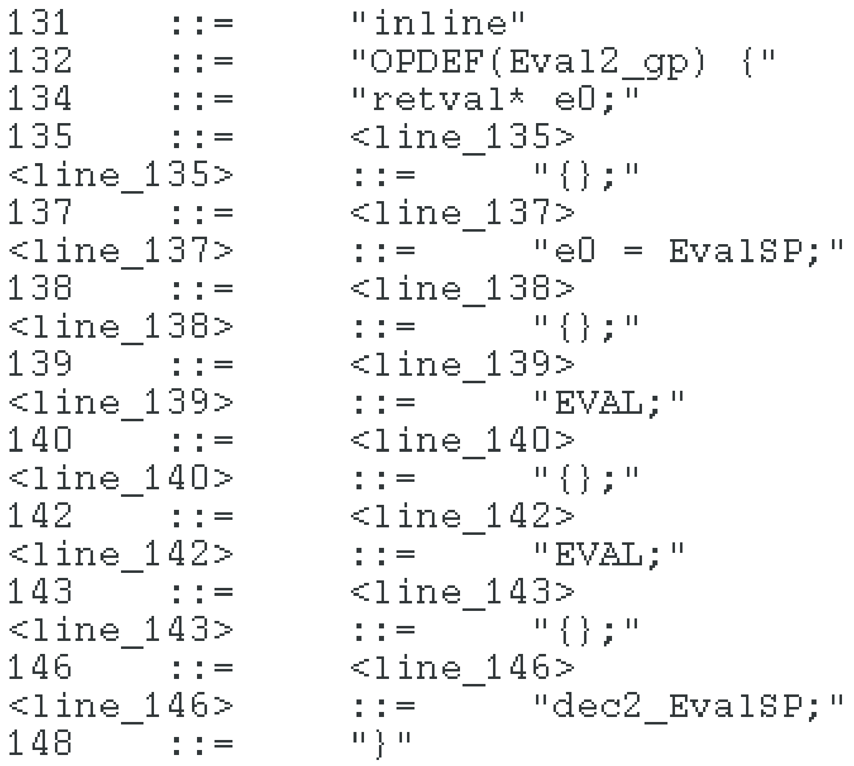
Tightly written code only allows mutations in Eval2, dodiv, binEval. Also zeroupper can be copied into them

Cf slide 9.

avx.cc switch Eval2 grammar



```
inline OPDEF(Eval2_gp) {
  retval* e0;
  EVAL;
  e0 = EvalSP;
  EVAL;
  dec2_EvalSP;
}
```



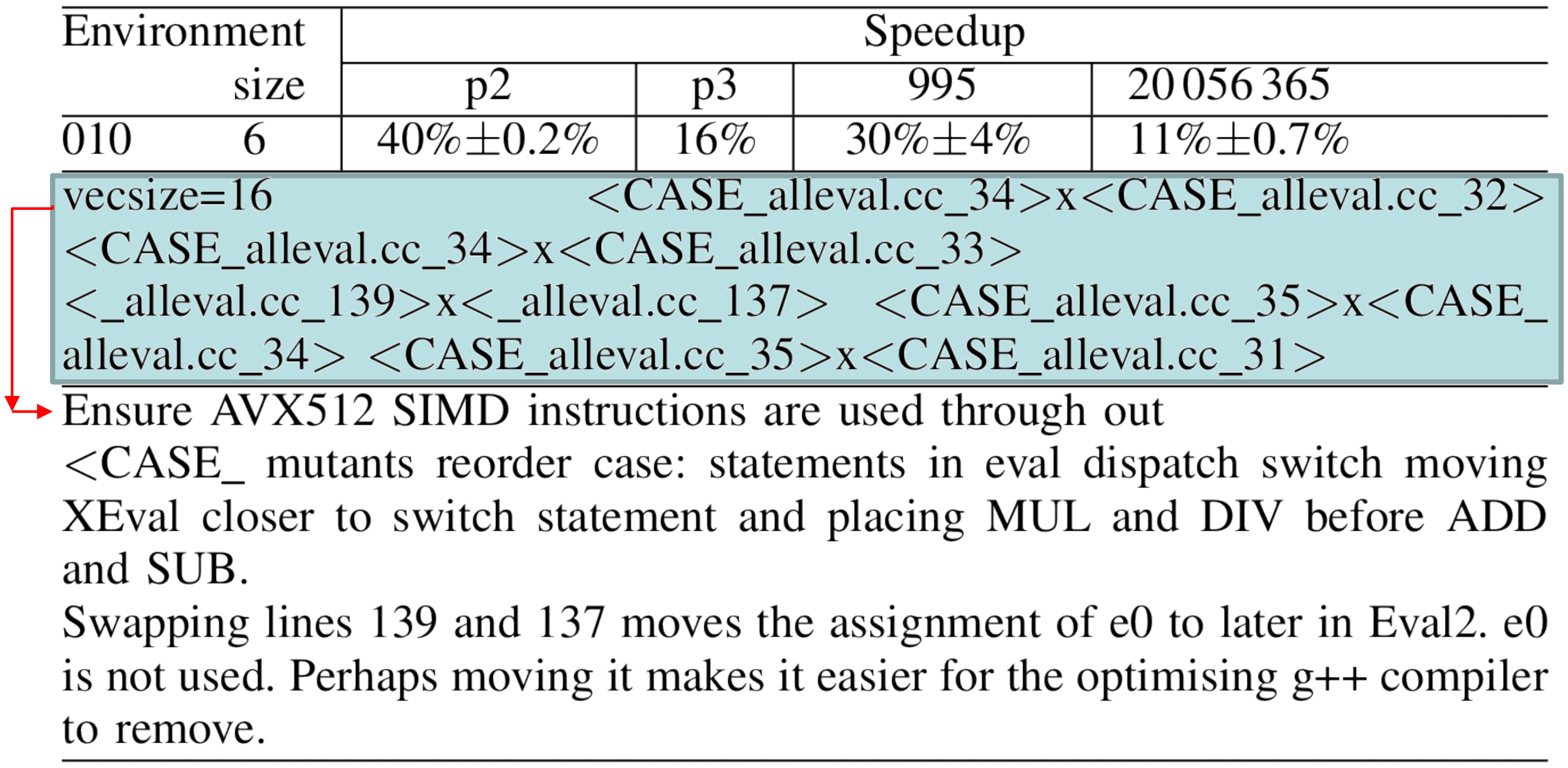
```
131         ::=        "inline"
132         ::=        "OPDEF(Eval2_gp) {"
134         ::=        "retval* e0;"
135         ::=        <line_135>
<line_135>             ::=        "{};"
137         ::=        <line_137>
<line_137>             ::=        "e0 = EvalSP;"
138         ::=        <line_138>
<line_138>             ::=        "{};"
139         ::=        <line_139>
<line_139>             ::=        "EVAL;"
140         ::=        <line_140>
<line_140>             ::=        "{};"
142         ::=        <line_142>
<line_142>             ::=        "EVAL;"
143         ::=        <line_143>
<line_143>             ::=        "{};"
146         ::=        <line_146>
<line_146>             ::=        "dec2_EvalSP;"
148         ::=        "}"
```

- All variable rules for Eval2 are of type line
  - Can be deleted, inserted, replaced or swapped with any other grammar rule of type line
- Variable e0 left over from earlier debug version

Default Genetic Improvement Parameters

Represent-ation:	variable list of replacements, deletions, swaps and insertions into BNF grammar comprised of Eval() C++ code specific rules, plus 5694 rules for the Intel Intrinsics library
Fitness:	compile with g++ 9.2.0 -O3 -DNDEBUG -pthread -march=skylake-avx512 Run on random tree changed every generation. log distribution tree size (half lie below 1000 and half above). Fitness is difference in first quartiles of elapsed time between original and evolved C++ code
Population:	100 (500 in Section V-C), panmictic, no elite, generational.
Parameters:	Initial population of random single mutants. Best half selected to be parents. 50% two point crossover,50% mutation. No size limit. Stop after 100 generations.

Switch Mutant



Environment	size	Speedup p2	p3	995	20 056 365
010	6	40%±0.2%	16%	30%±4%	11%±0.7%

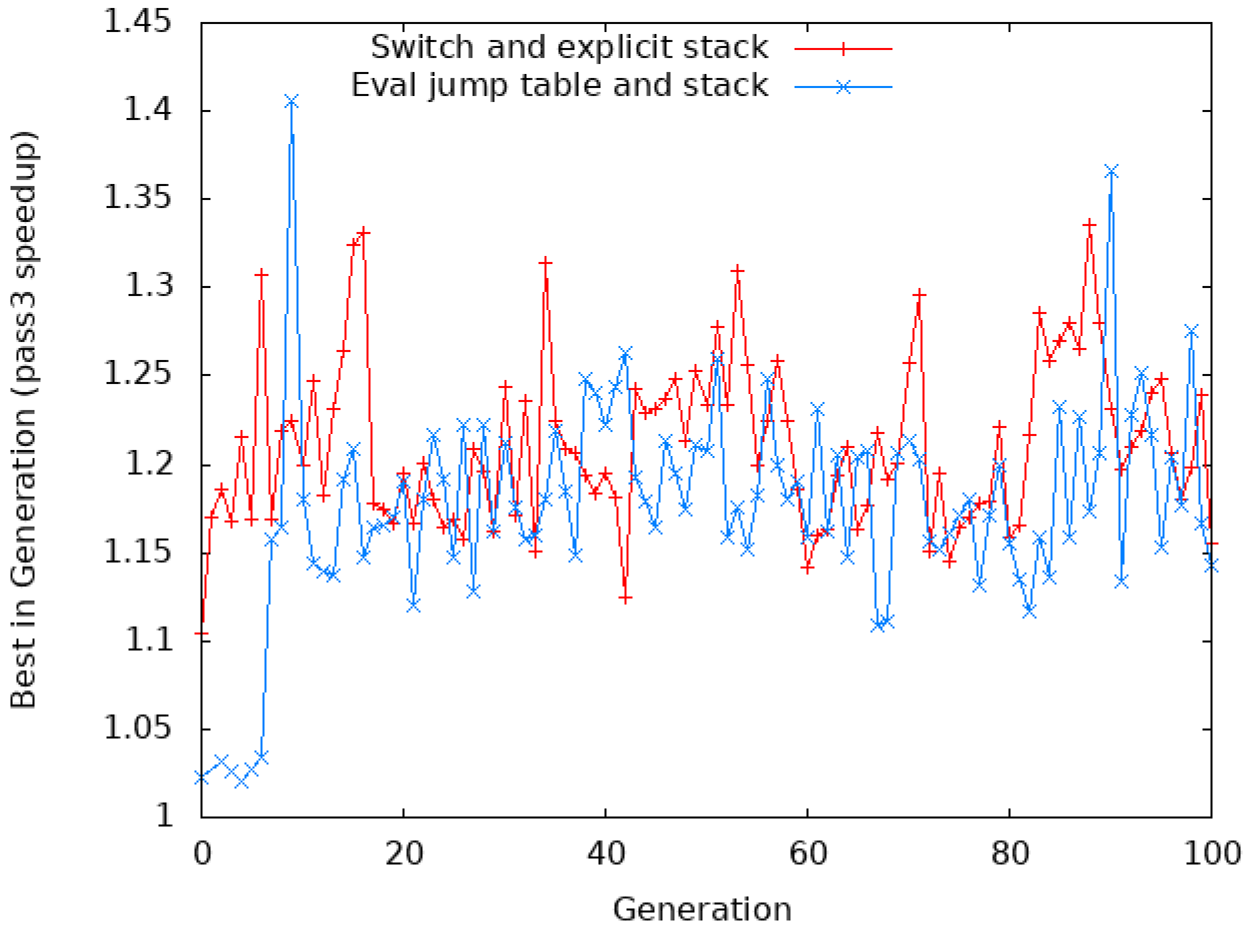
vecsize=16 <CASE_alleval.cc_34>x<CASE_alleval.cc_32> <CASE_alleval.cc_34>x<CASE_alleval.cc_33> <_alleval.cc_139>x<_alleval.cc_137> <CASE_alleval.cc_35>x<CASE_alleval.cc_34> <CASE_alleval.cc_35>x<CASE_alleval.cc_31>

Ensure AVX512 SIMD instructions are used through out

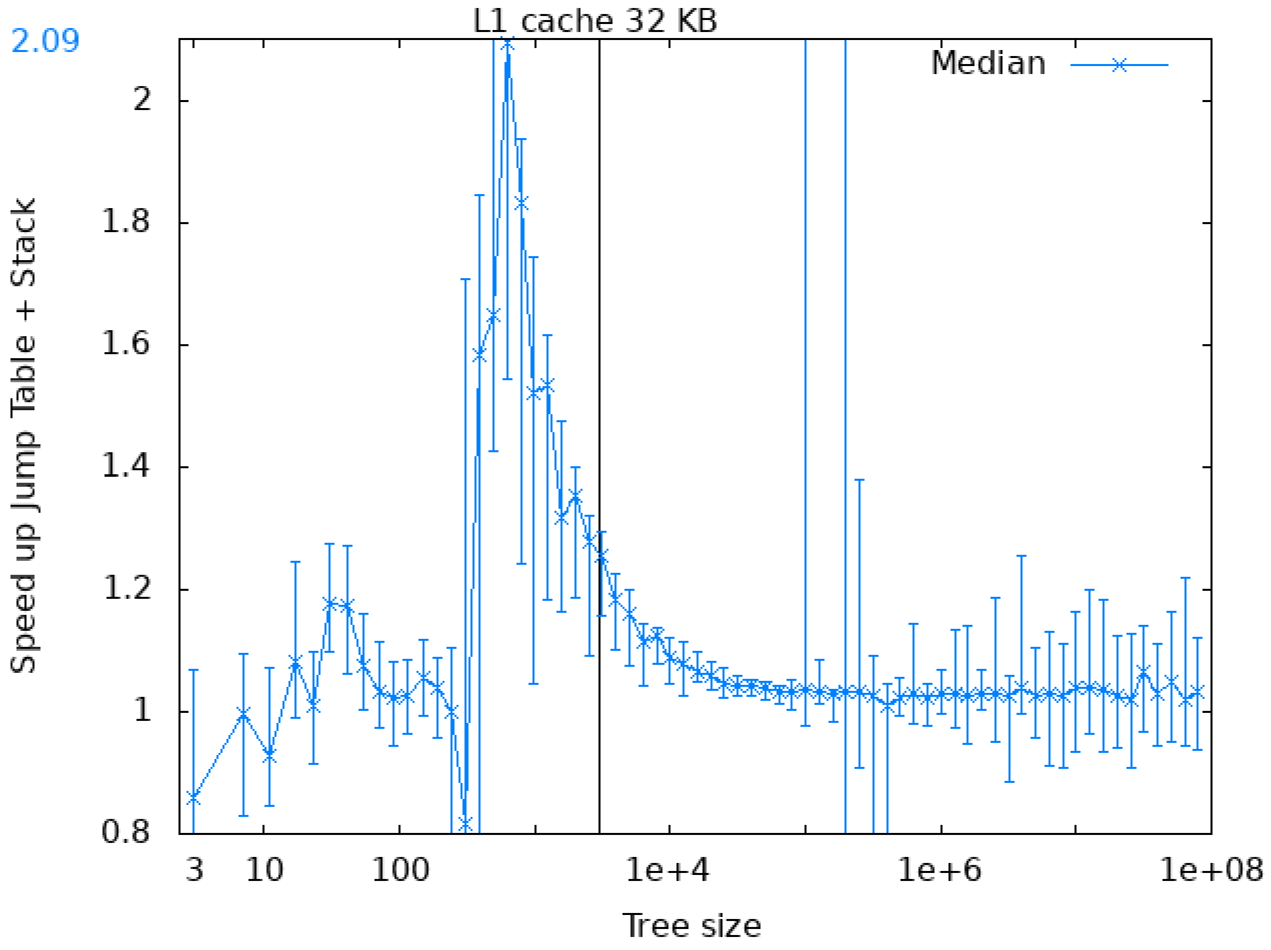
<CASE_ mutants reorder case: statements in eval dispatch switch moving XEval closer to switch statement and placing MUL and DIV before ADD and SUB.

Swapping lines 139 and 137 moves the assignment of e0 to later in Eval2. e0 is not used. Perhaps moving it makes it easier for the optimising g++ compiler to remove.

Evolution of Fitness



After clean up: Out of Sample Generalisation



The Genetic Programming Bibliography

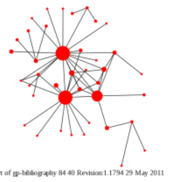
http://gpbib.cs.ucl.ac.uk/

13607 references, 12000 authors

Make sure it has all of your papers!

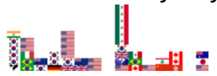
E.g. email W.Langdon@cs.ucl.ac.uk or use | Add to It | web link

RSS Support available through the XML RSS Collection of CS Bibliographies.

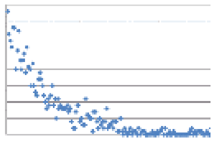


Co-authorship community.
Downloads

Downloads by day



A personalised list of every author's GP publications.



Your papers

blog

Googling GP Bibliography, eg:
ocean waves site:gpbib.cs.ucl.ac.uk