A Systematic Study of Automated Program Repair:
Fixing 55 out of 105 Bugs for $8 Each

Claire Le Goues (Virginia), Michael Dewey-Vogt (Virginia), Stephanie Forrest (New Mexico) , Westley Weimer (Virginia)

International Conference on Software Engineering (ICSE) 2012

Presented by Paul Wood

Other Papers

Claire Le Goues, Stephanie Forrest, Westley Weimer: The Case for Software Evolution. *Foundations of Software Engineering Working Conference on the Future of Software Engineering* (FoSER) 2010: 205-209

Westley Weimer, Stephanie Forrest, Claire Le Goues, ThanhVu Nguyen: Automatic Program Repair With Evolutionary Computation. *Communications of the ACM* Vol. 53 No. 5, May 2010, Pages 109-116.

http://dijkstra.cs.virginia.edu/genprog/

Paper's Purpose

- Evaluate the Genetic Programming ("GenProg") method proposed by the authors (in previous papers) to determine:
  - What fraction of bugs can be repaired
  - How much does it cost
- Because the author's method of repair uses an extensive search space, there is a computation cost
  - How much cluster time, cost (ie AWS) is there to correct a bug
- Approach is to
  - Use GP to generate candidate repairs and evaluate them
  - Distribute the process to bring down the wall time for repairs

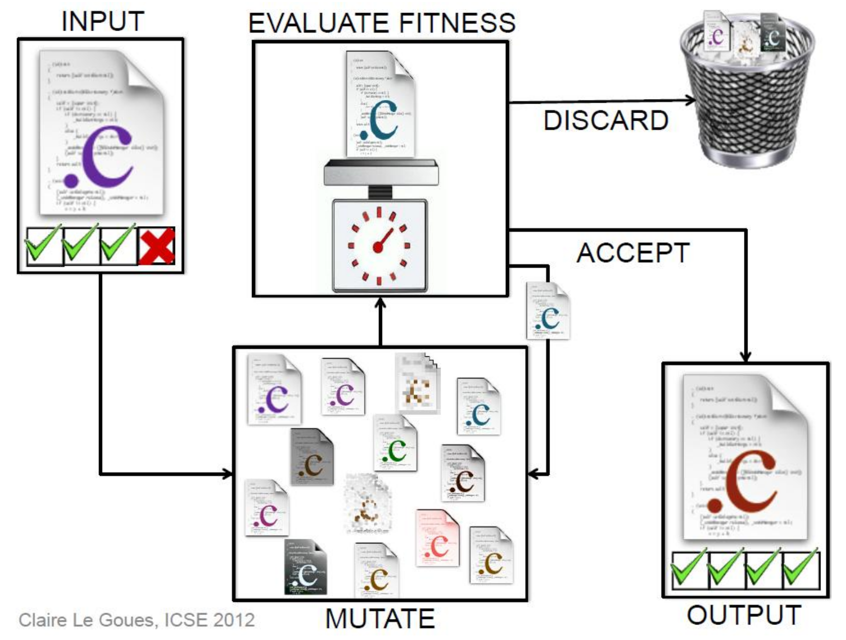
---

Genetic Programming

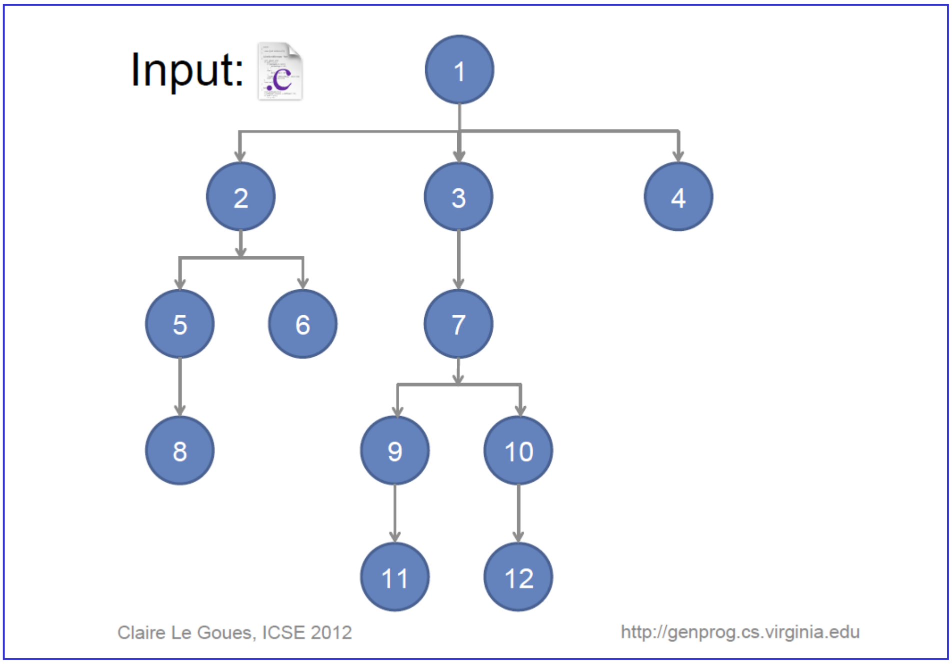
- In artificial intelligence, genetic programming (GP) is an evolutionary algorithm-based methodology inspired by biological evolution to find computer programs that perform a user-defined task. (Wikipedia)
- Programs can be represented as abstract syntax trees for example and nodes selected/swapped/deleted/replaced
- Benefits:
  - Novel solutions can be found to some problems
- Downsides:
  - Very large (infinite) mutation spaces possible

GP Operations

- Fitness
  - A program is scored by a test
  - Example: 4 test cases, a program passes 3 and fails 1: 75% score
  - GenProg: test cases provided by programmer
- Selection
  - Programs in a population are selected, usually probabilistically based on fitness (natural selection)
  - GenProg: Fitness-weighted selection, some filtering based on computability
- Cross-Over
  - Parts of selected programs are merged (example: lines 1:50 of program 1 and lines 51:100 of program 2)
  - GenProg: Uniform cross-over is performed, but only on the code edits
- Mutation
  - Some part of the program is changed randomly
  - GenProg: Delete, Insert, or Replace (Delete & Insert) nodes near fault/fix locality

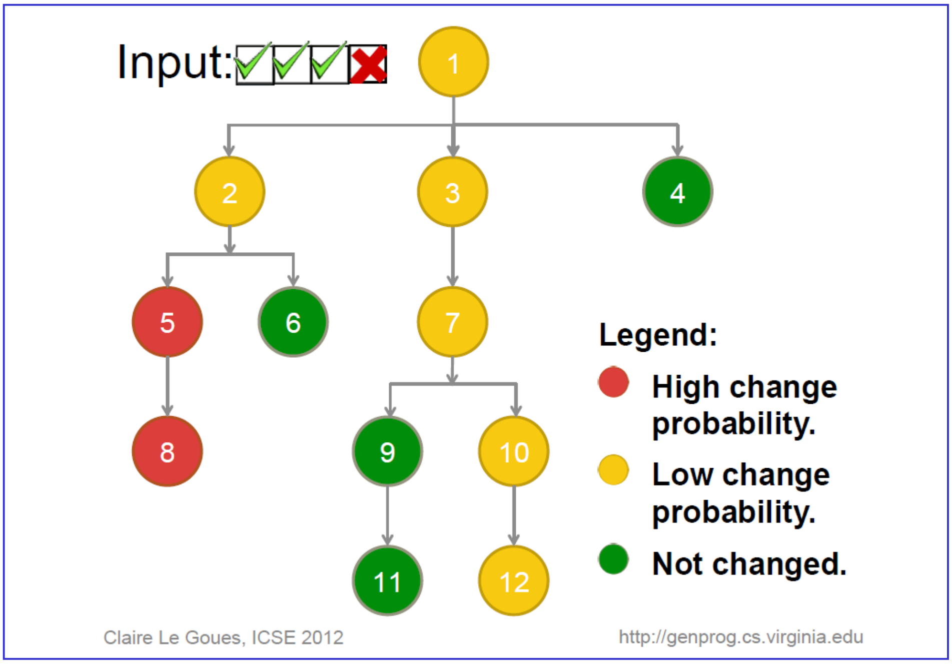


Claire Le Goues, ICSE 2012



Claire Le Goues, ICSE 2012

http://genprog.cs.virginia.edu

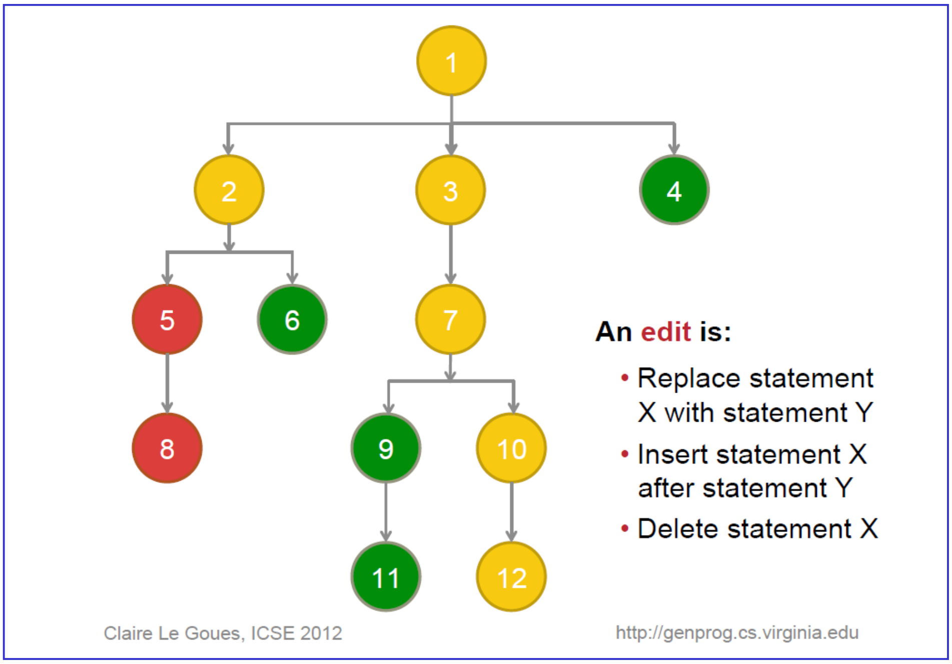


Legend:

- 🔴 **High change probability.**
- 🟡 **Low change probability.**
- 🟢 **Not changed.**

Claire Le Goues, ICSE 2012

http://genprog.cs.virginia.edu

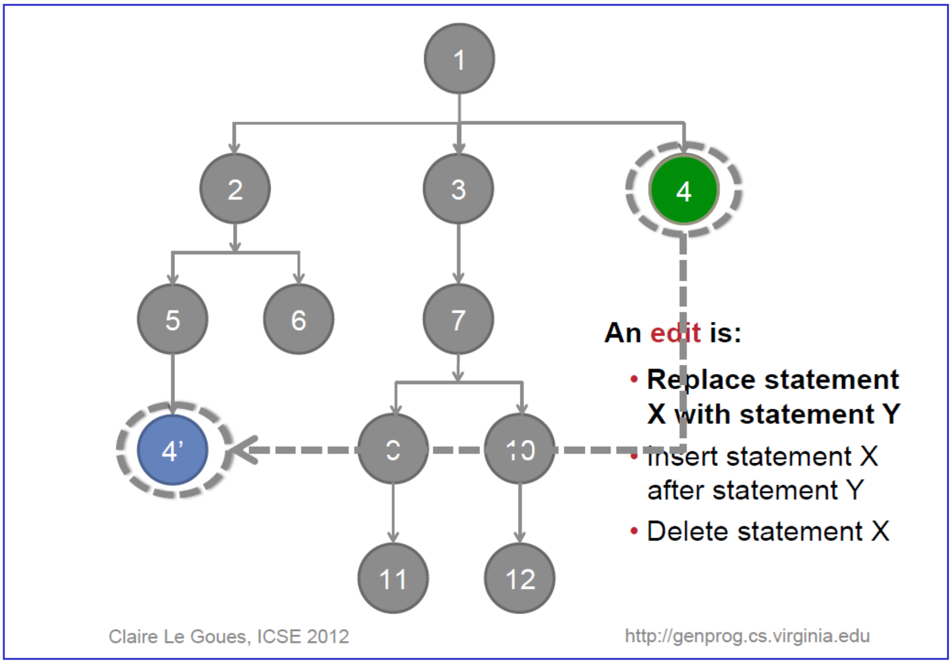


An edit is:

- Replace statement X with statement Y
- Insert statement X after statement Y
- Delete statement X

Claire Le Goues, ICSE 2012

http://genprog.cs.virginia.edu



An edit is:

- **Replace statement X with statement Y**
- Insert statement X after statement Y
- Delete statement X

Claire Le Goues, ICSE 2012

http://genprog.cs.virginia.edu

GP Problems

- Infinite Monkey Theorem
  - Monkeys hitting keys at random for an infinite amount of time will almost surely write the complete works of William Shakespeare
- This paper on genetic programming attempts to solve this problem by showing:
  - Constraints can be used to limit the mutation space without compromising too many valid solutions
  - The time (and money) required to solve a problem is comparable to some other approach
- GenProg limits the search space by:
  - Using fault/fix localization
  - Mutating with existing code in the program
    - Assumption is made that a program that contains an error in one area likely implements the correct behavior elsewhere

---

Patch Representation

- GenProg represents patches as node edits
  - Similar to a diff output
- Previous work used the entire AST, but memory usage was too high
- A patch consists of edits like:
  - Delete(81)
  - Replace(23,44)
- Contains no redundant code

Fitness Evaluation

- The pass percentage of the test suite provided for each program is used for the fitness evaluation
- A random subset of tests is used to screen candidates without overburdening resources running test cases
- Fails are weighted twice as much as passes on the test cases, and a weighted sum is used for selection

---

Fault Localization

- The fault is localized by observing the statements visited by statements visited by a failing test and not a passing test
- Statements never visited have 0 weight, statements visited only on failed tests have 1.0 weight, and statements visited by both have 0.1

$$\mathit{faultloc}(s) = \begin{cases} 0 & \forall t \in T.\ s \notin \mathit{Visited}(t) \\ 1.0 & \forall t \in T.\ s \in \mathit{Visited}(t) \implies \neg \mathit{Pass}(t) \\ 0.1 & \text{otherwise} \end{cases}$$

Fix/Mutation Source Localization

- To limit the choice of statements to mutate, some “fix localization” is done
- The source statements must have in scope variables (can compile)
- It must also be visited by at least one of the test cases

$$\mathit{fixloc}(d) = \left\{ s \;\middle|\; \begin{array}{l} \exists t \in T.\ s \in \mathit{Visited}(t) \wedge \\ \mathit{VarsUsed}(s) \subseteq \mathit{InScope}(d) \end{array} \right\}$$

Mutation and Crossover

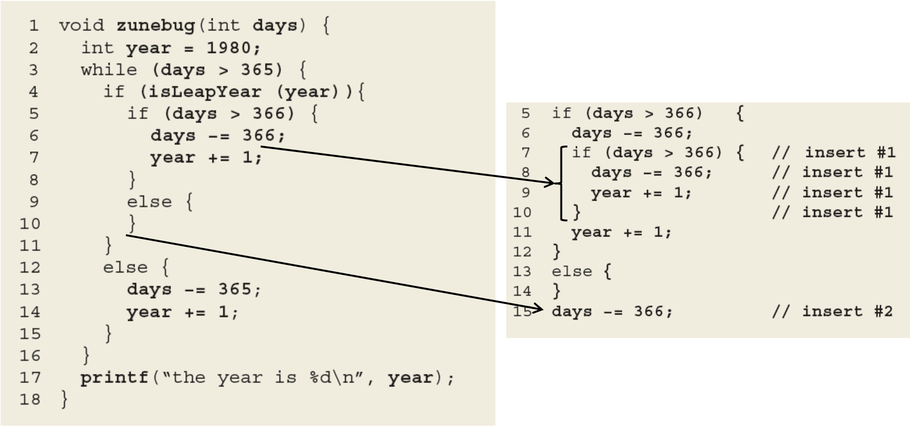
- Mutation is done by replacing a statement, inserting a statement, or deleting a statement
- Each operator is selected with equal probability
- Each source statement for insert/replace is randomly selected from the fix locality
  - This requires the fix statement to already be present somewhere in the source code
- Crossover is done by combining two parents (a list of edits) and then removing edits with a probability 0.5
  - Result average is the same size as the parent

Bug Repair Example

```c
void zunebug(int days) {
  int year = 1980;
  while (days > 365) {
    if (isLeapYear (year)){
      if (days > 366) {
        days -= 366;
        year += 1;
      }
      else {
      }
    }
    else {
      days -= 365;
      year += 1;
    }
  }
  printf(“the year is %d\n”, year);
}
```

Infinite loop possible if days = 366

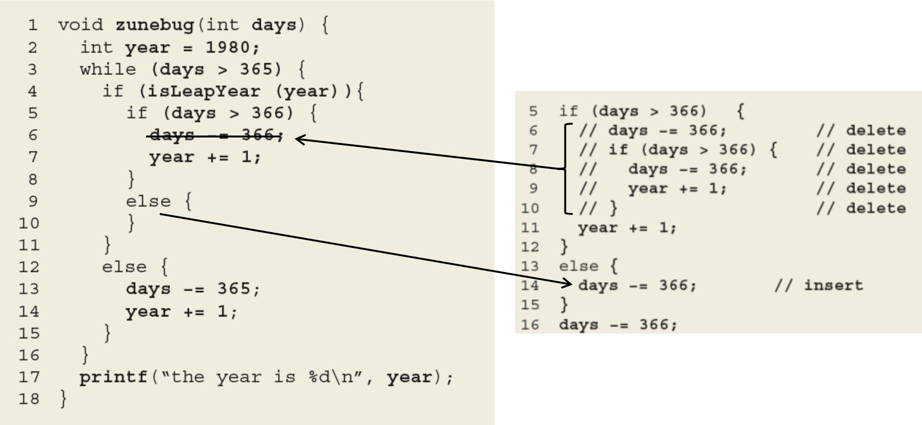
Mutation 1&2



```c
1   void zunebug(int days) {
2     int year = 1980;
3     while (days > 365) {
4       if (isLeapYear (year)){
5         if (days > 366) {
6           days -= 366;
7           year += 1;
8         }
9         else {
10        }
11      }
12      else {
13        days -= 365;
14        year += 1;
15      }
16    }
17    printf(“the year is %d\n”, year);
18  }
```

```c
5   if (days > 366)   {
6     days -= 366;
7     if (days > 366) {    // insert #1
8       days -= 366;       // insert #1
9       year += 1;         // insert #1
10    }                    // insert #1
11    year += 1;
12  }
13  else {
14  }
15  days -= 366;           // insert #2
```

Final Mutation



```
void zunebug(int days) {
  int year = 1980;
  while (days > 365) {
    if (isLeapYear (year)){
      if (days > 366) {
        ~~days -= 366;~~
        year += 1;
      }
      else {
      }
    }
    else {
      days -= 365;
      year += 1;
    }
  }
  printf("the year is %d\n", year);
}
```

```
if (days > 366)   {
  // days -= 366;          // delete
  // if (days > 366) {     // delete
  //   days -= 366;        // delete
  //   year += 1;          // delete
  // }                     // delete
  year += 1;
}
else {
  days -= 366;         // insert
}
days -= 366;
```

Final Repair

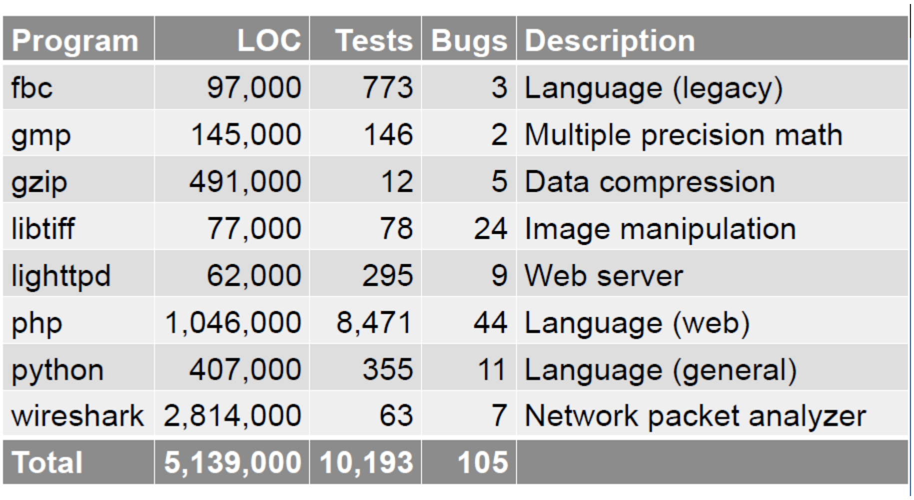
```
void zunebug_repair (int days) {
  int year = 1980;
  while (days > 365) {
    if (isLeapYear (year)) {
      if (days > 366) {
        // days -= 366;   // deleted
        year += 1;
      }
      else {
      }
      days -= 366;        // inserted
    } else {
      days -= 365;
      year += 1;
    }
  }
  printf ("the year is %dn", year);
}
```

GenProg Benchmark

- Programs from SourceForge, Google Code, etc are taken
- Pairs of versions where test cases transition from fail to pass are considered
  - A human-written repair caused the test case to pass
- The most recent test cases are used and then older versions of the source code are taken
  - The idea is that a test case was added to validate a bug fix, subsequently it can be used to find the bug
- To determine the cost of a repair, Amazon’s EC2 is used and the cost of finding a bug is the cost of the EC2 resource

---

GenProg Benchmark



Program	LOC	Tests	Bugs	Description
fbc	97,000	773	3	Language (legacy)
gmp	145,000	146	2	Multiple precision math
gzip	491,000	12	5	Data compression
libtiff	77,000	78	24	Image manipulation
lighttpd	62,000	295	9	Web server
php	1,046,000	8,471	44	Language (web)
python	407,000	355	11	Language (general)
wireshark	2,814,000	63	7	Network packet analyzer
Total	**5,139,000**	**10,193**	**105**	

GenProg Benchmark

Program	Defects Repaired	Cost per non-repair		Cost per repair	
		Hours	US$	Hours	US$
fbc	1/3	8.52	5.56	6.52	4.08
gmp	1/2	9.93	6.61	1.60	0.44
gzip	1/5	5.11	3.04	1.41	0.30
libtiff	17/24	7.81	5.04	1.05	0.04
lighttpd	5/9	10.79	7.25	1.34	0.25
php	28/44	13.00	8.80	1.84	0.62
python	1/11	13.00	8.80	1.22	0.16
wireshark	1/7	13.00	8.80	1.23	0.17
Total	**55/105**	**11.22h**		**1.60h**	

$403 for all 105 trials, leading to 55 repairs; $7.32 per bug repaired.

GenProg Benchmark

JBoss issue tracking: median 5.0, mean 15.3 hours.[1]

IBM: $25 per defect during coding, rising at build, Q&A, post-release, etc.[2]

Tarsnap.com: $17, 40 hours per non-trivial repair.[3]

Bug bounty programs in general:

- At least $500 for security-critical bugs.
- One of our php bugs has an associated security CVE.

[1] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller, “How long will it take to fix this bug?” in *Workshop on Mining Software Repositories*, May 2007.

[2] L. Williamson, “IBM Rational software analyzer: Beyond source code,” in *Rational Software Developer Conference*, Jun. 2008.

[3] http://www.tarsnap.com/bugbounty.html

Conclusion

- GenProg repaired 55 of 105 defects from programs spanning 5.1 MLOC and 10,193 tests
- Repairs are generated using reasonable resources ($7.32/patch)
  - The programs must have test suites available
  - Not all faults can be repaired
- The cost of computational resources when a repair is generated are lower than the human cost
  - The patches generated still require developer validation
  - Evaluating costs is complex, and the conclusion is not absolute

Paper Critique

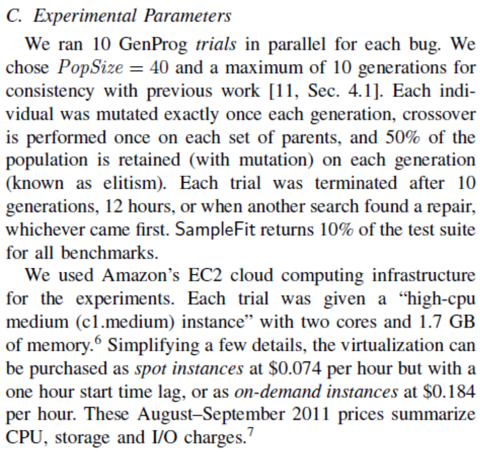
- The authors make a strong case for their work by using cost analysis
- It is loaded with impressive statistics about performance
  - “Our improved algorithm finds repairs 68% more often”
- The experiment space is gigantic (100x larger) compared with other automated repair publications

Future Work / Improvements

- Data structure manipulation
  - GenProg only uses statement insert/delete/replace
- Performance considerations
  - The test suites do not consider performance, the generated results can leave orphaned variables, etc., or be inefficient in some ways
- Repair Method Inefficiency
  - Genetic Programming is inefficient by nature, requires too much CPU time to be useful on current embedded systems or in a real time setting (it still requires hours on EC2)
- Automated Repair for High Availability

Backup

Parameters



C. Experimental Parameters

We ran 10 GenProg *trials* in parallel for each bug. We chose $PopSize = 40$ and a maximum of 10 generations for consistency with previous work [11, Sec. 4.1]. Each individual was mutated exactly once each generation, crossover is performed once on each set of parents, and 50% of the population is retained (with mutation) on each generation (known as elitism). Each trial was terminated after 10 generations, 12 hours, or when another search found a repair, whichever came first. SampleFit returns 10% of the test suite for all benchmarks.

We used Amazon's EC2 cloud computing infrastructure for the experiments. Each trial was given a "high-cpu medium (c1.medium) instance" with two cores and 1.7 GB of memory.[6] Simplifying a few details, the virtualization can be purchased as *spot instances* at $0.074 per hour but with a one hour start time lag, or as *on-demand instances* at $0.184 per hour. These August–September 2011 prices summarize CPU, storage and I/O charges.[7]