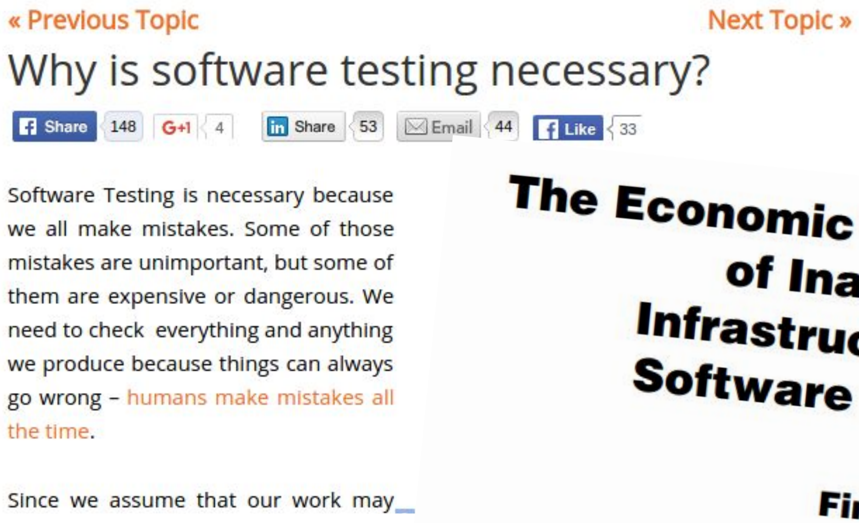
Generation of Test Cases for Languages with Pointer Arithmetics

Demontiê Junior
Advisor: Mariza Bigonha
Co-advisor: Fernando Pereira

EXAME
2000
1999
Por que bilhões de dólares estão sendo gastos para evitar que os computadores parem o mundo no ano 2000
O BUG DO MILÊNIO
Por Helio Gurovitz

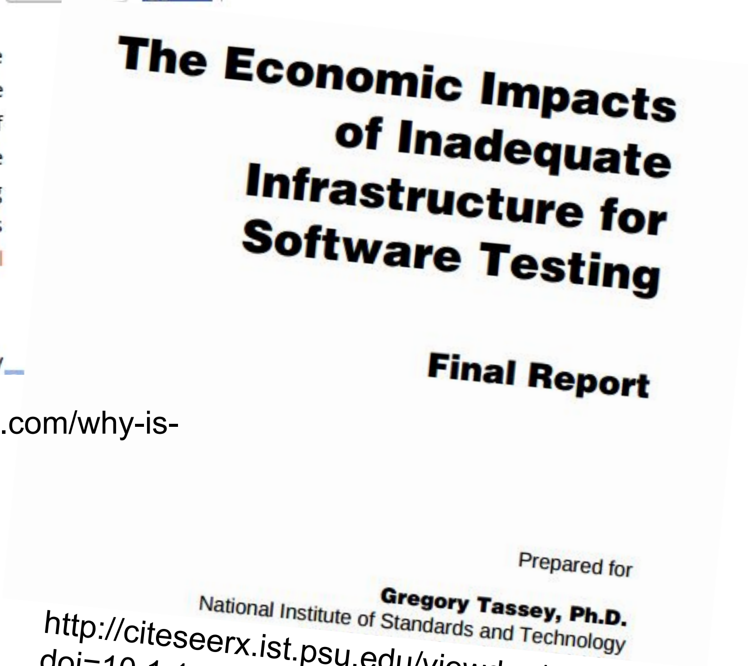

« Previous Topic Next Topic »

Why is software testing necessary?

Software Testing is necessary because we all make mistakes. Some of those mistakes are unimportant, but some of them are expensive or dangerous. We need to check everything and anything we produce because things can always go wrong – humans make mistakes all the time.

Since we assume that our work may

http://istqbexamcertification.com/why-is-testing-necessary/

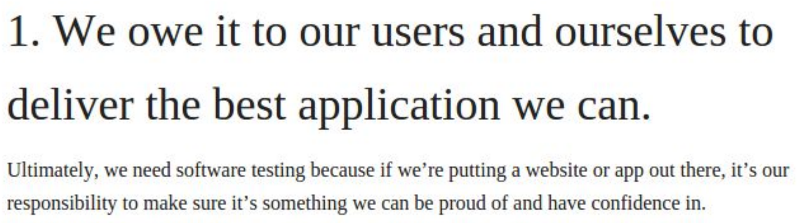

The Economic Impacts of Inadequate Infrastructure for Software Testing

Final Report

Prepared for

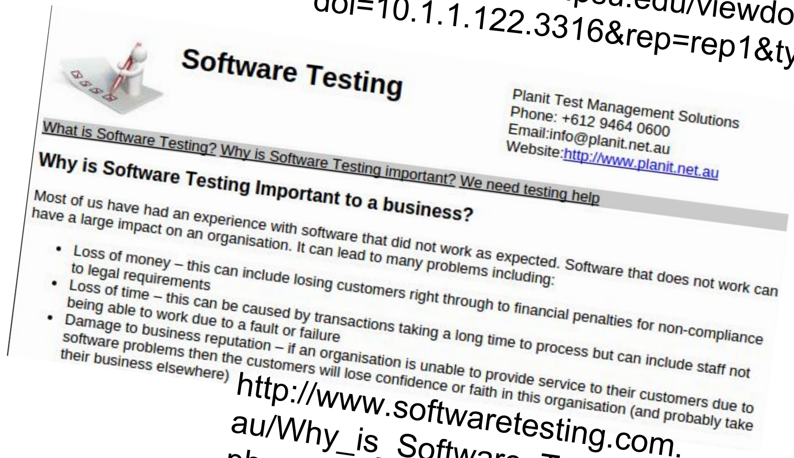
Gregory Tassey, Ph.D.
National Institute of Standards and Technology

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.122.3316&rep=rep1&type=pdf


1. We owe it to our users and ourselves to deliver the best application we can.

Ultimately, we need software testing because if we're putting a website or app out there, it's our responsibility to make sure it's something we can be proud of and have confidence in.

http://www.te52.com/testtalk/2014/08/07/5-reasons-we-need-software-testing/


Software Testing

Planit Test Management Solutions
Phone: +612 9464 0600
Email:info@planit.net.au
Website:http://www.planit.net.au

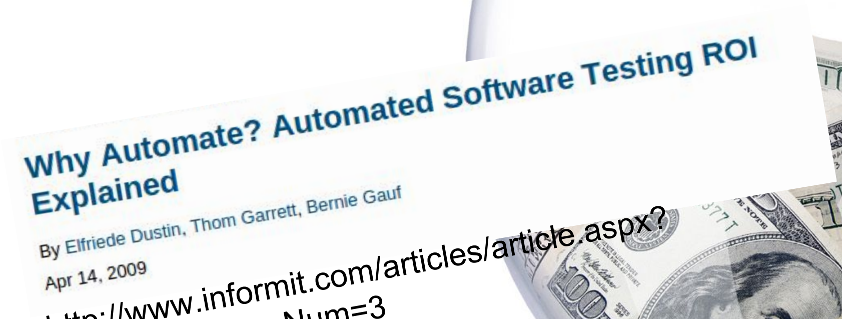
What is Software Testing? Why is Software Testing important? We need testing help

Why is Software Testing Important to a business?

Most of us have had an experience with software that did not work as expected. Software that does not work can have a large impact on an organisation. It can lead to many problems including:

- Loss of money – this can include losing customers right through to financial penalties for non-compliance to legal requirements
- Loss of time – this can be caused by transactions taking a long time to process but can include staff not being able to work due to a fault or failure
- Damage to business reputation – if an organisation is unable to provide service to their customers due to software problems then the customers will lose confidence or faith in this organisation (and probably take their business elsewhere)

http://www.softwaretesting.com.au/Why_is_Software_Testing_important.php



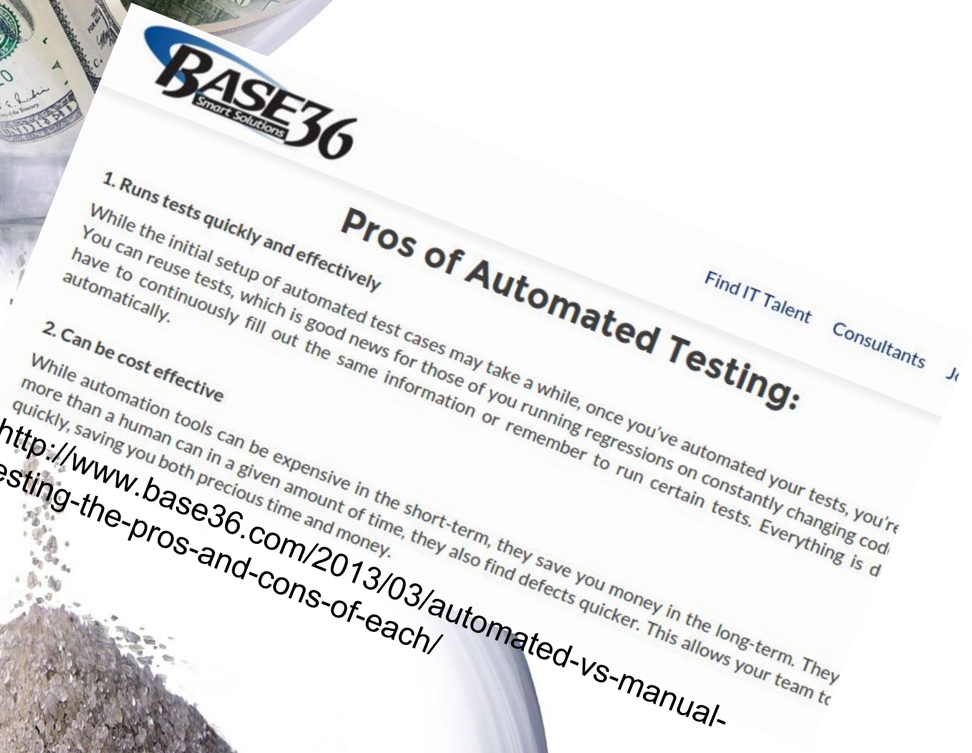
Why Automate? Automated Software Testing ROI Explained

By Elfriede Dustin, Thom Garrett, Bernie Gauf

Apr 14, 2009

http://www.informit.com/articles/article.aspx?p=1332758&seqNum=3

http://www.softwaretestingclass.com/why-how-and-when-to-automate-software-testing/



BASE36 Smart Solutions

Find IT Talent Consultants J

Pros of Automated Testing:

1. Runs tests quickly and effectively

While the initial setup of automated test cases may take a while, once you've automated your tests, you're You can reuse tests, which is good news for those of you running regressions on constantly changing cod have to continuously fill out the same information or remember to run certain tests. Everything is d automatically.

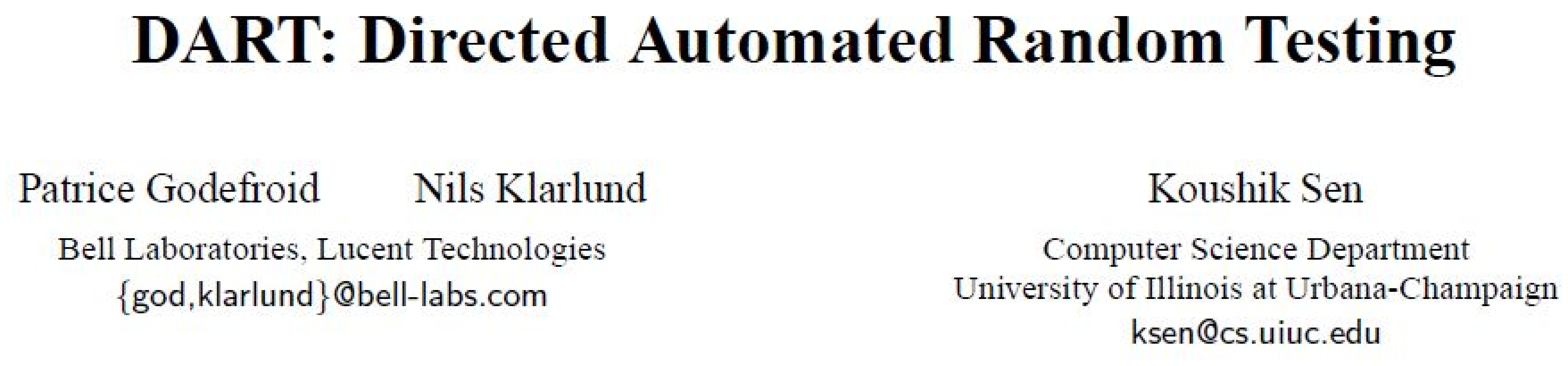
2. Can be cost effective

While automation tools can be expensive in the short-term, they save you money in the long-term. They more than a human can in a given amount of time, they also find defects quicker. This allows your team to quickly, saving you both precious time and money.

http://www.base36.com/2013/03/automated-vs-manual-testing-the-pros-and-cons-of-each/

The goal of this work is to improve automatic test generation for type unsafe languages, such as C, C++ and assembly.

Previous Work

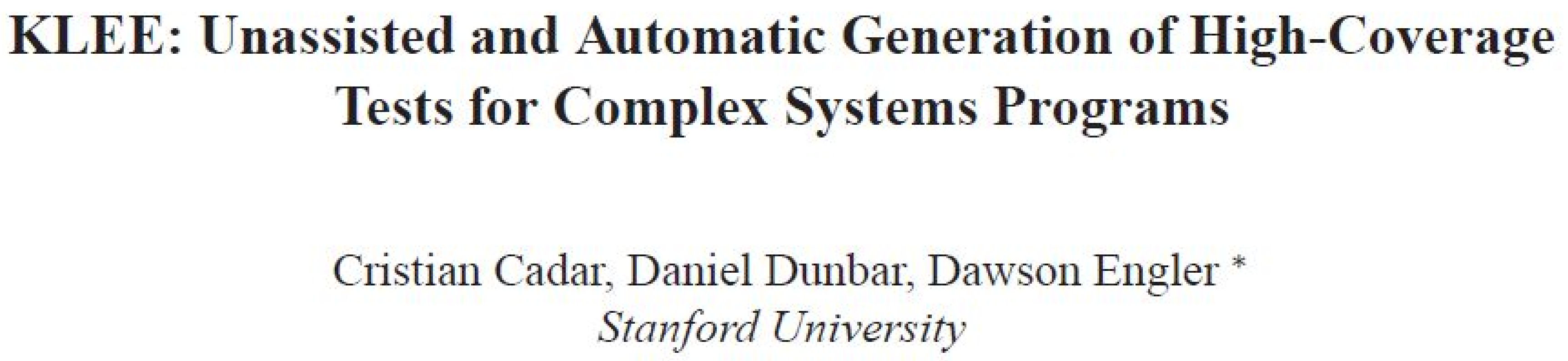

DART: Directed Automated Random Testing

Patrice Godefroid Nils Klarlund
Bell Laboratories, Lucent Technologies
{god,klarlund}@bell-labs.com

Koushik Sen
Computer Science Department
University of Illinois at Urbana-Champaign
ksen@cs.uiuc.edu

- **Infers** the program’s **interface**
- **Randomly generates values** for the function’s arguments
- Analyzes the execution to find new input values

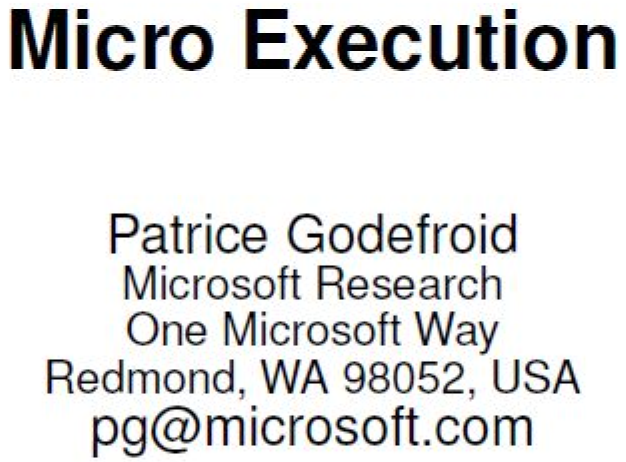
Previous Work


KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler *
Stanford University

- **Symbolic Execution**
- Deals with the **external environment**

Previous Work


Micro Execution

Patrice Godefroid
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
pg@microsoft.com

- **Dynamic** input generation
- **Virtual Machine**
- Generates **random values** for **uninitialized memory locations** that are inputs

Problem

- Unlike some strongly typed languages, allocated **memory in C has no meta information**

```
Java: for (int i=0; i < array.length; i++)
Python: for i in xrange(len(array))
C#: foreach (int i in array)
```

C

```
void foo(int *array, int size) {
    for (int i=0; i < size; i++)
        array[i];
    ...
}
```

Problem

- Unlike some strongly typed languages, allocated **memory in C has no meta information**

```c
void foo(int *array, int size) {
    for (int i=0; i < size; i++)
        array[i];
    ...
}
```

Problem

- Unlike some strongly typed languages, allocated **memory in C has no meta information**

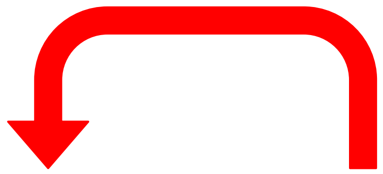
int[10]   13

```
void foo(int *array, int size) {
    for (int i=0; i < size; i++)
        array[i];
    ...
}
```

Invalid memory accesses for i ≥ 10

Solution

- We use **static analyses to bind** function parameters that represent **meta information of memory regions** and improve the generation of test cases



```
void foo(int *array, int size) {
    for (int i=0; i < size; i++)
        array[i];
    ...
}
```

Background

Basic Block

- Maximum set of consecutive instructions
    - The **execution** of a basic block always **starts with its first instruction**
    - The **execution** of a basic block always **ends in its last instruction**

Basic Block

- Maximum set of consecutive instructions
  - The **execution** of a basic block always **starts with its first instruction**
  - The **execution** of a basic block always **ends in its last instruction**

```
...
while (i < n) {
  o = a[i];
  ...
  i++;
}
return ...;
```

Basic Block

- Maximum set of consecutive instructions
  - The **execution** of a basic block always **starts with its first instruction**
  - The **execution** of a basic block always **ends in its last instruction**

```
...
while (i < n) {
  o = a[i];
  ...
  i++;
}
return ...;
```

```
l0: i = load i.addr
    cmp = icmp sge i, n
    brz cmp, l1, l2
```

Basic Block

- Maximum set of consecutive instructions
  - The **execution** of a basic block always **starts with its first instruction**
  - The **execution** of a basic block always **ends in its last instruction**

```
...
while (i < n) {
  o = a[i];
  ...
  i++;
}
return ...;
```

```
l0: i = load i.addr
    cmp = icmp sge i, n
    brz cmp, l1, l2
l1: tmp = add a.addr, i
    a = load tmp
    ...
```

Basic Block

- Maximum set of consecutive instructions
  - The **execution** of a basic block always **starts with its first instruction**
  - The **execution** of a basic block always **ends in its last instruction**

```
...
while (i < n) {
  o = a[i];
  ...
  i++;
}
return ...;
```

```
l0: i = load i.addr
    cmp = icmp sge i, n
    brz cmp, l1, l2
l1: tmp = add a.addr, i
    a = load tmp
    ...
    inc = add i, 1
    store inc, i.addr
```

Basic Block

- Maximum set of consecutive instructions
  - The **execution** of a basic block always **starts with its first instruction**
  - The **execution** of a basic block always **ends in its last instruction**

```
...
while (i < n) {
  o = a[i];
  ...
  i++;
}
return ...;
```

```
l0: i = load i.addr
    cmp = icmp sge i, n
    brz cmp, l1, l2
l1: tmp = add a.addr, i
    a = load tmp
    ...
    inc = add i, 1
    store inc, i.addr
    br l0
```

Basic Block

- Maximum set of consecutive instructions
    - The **execution** of a basic block always **starts with its first instruction**
    - The **execution** of a basic block always **ends in its last instruction**

```
...
while (i < n) {
  o = a[i];
  ...
  i++;
}
return ...;
```

```
l0: i = load i.addr
    cmp = icmp sge i, n
    brz cmp, l1, l2
l1: tmp = add a.addr, i
    a = load tmp
    ...
    inc = add i, 1
    store inc, i.addr
    br l0
l2: ret ...
```

Basic Block

- Maximum set of consecutive instructions
  - The **execution** of a basic block always **starts with its first instruction**
  - The **execution** of a basic block always **ends in its last instruction**

```
...
while (i < n) {
  o = a[i];
  ...
  i++;
}
return ...;
```

```
l0: i = load i.addr
    cmp = icmp sge i, n
    brz cmp, l1, l2
l1: tmp = add a.addr, i
    a = load tmp
    ...
    inc = add i, 1
    store inc, i.addr
    br l0
l2: ret ...
```

Basic Block

- Maximum set of consecutive instructions
  - The **execution** of a basic block always **starts with its first instruction**
  - The **execution** of a basic block always **ends in its last instruction**

```
...
while (i < n) {
  o = a[i];
  ...
  i++;
}
return ...;
```

```
l0: i = load i.addr
    cmp = icmp sge i, n
    brz cmp, l1, l2
l1: tmp = add a.addr, i
    a = load tmp
    ...
    inc = add i, 1
    store inc, i.addr
    br l0
l2: ret ...
```

Basic Block

- Maximum set of consecutive instructions
    - The **execution** of a basic block always **starts with its first instruction**
    - The **execution** of a basic block always **ends in its last instruction**

```
...
while (i < n) {
  o = a[i];
  ...
  i++;
}
return ...;
```

```
l0: i = load i.addr
    cmp = icmp sge i, n
    brz cmp, l1, l2
l1: tmp = add a.addr, i
    a = load tmp
    ...
    inc = add i, 1
    store inc, i.addr
    br l0
l2: ret ...
```

Basic Block

- Maximum set of consecutive instructions
  - The **execution** of a basic block always **starts with its first instruction**
  - The **execution** of a basic block always **ends in its last instruction**

```
...
while (i < n) {
  o = a[i];
  ...
  i++;
}
return ...;
```

```
l0: i = load i.addr
    cmp = icmp sge i, n
    brz cmp, l1, l2
```

```
l1: tmp = add a.addr, i
    a = load tmp
    ...
    inc = add i, 1
    store inc, i.addr
    br l0
```

```
l2: ret ...
```

Control Flow Graph

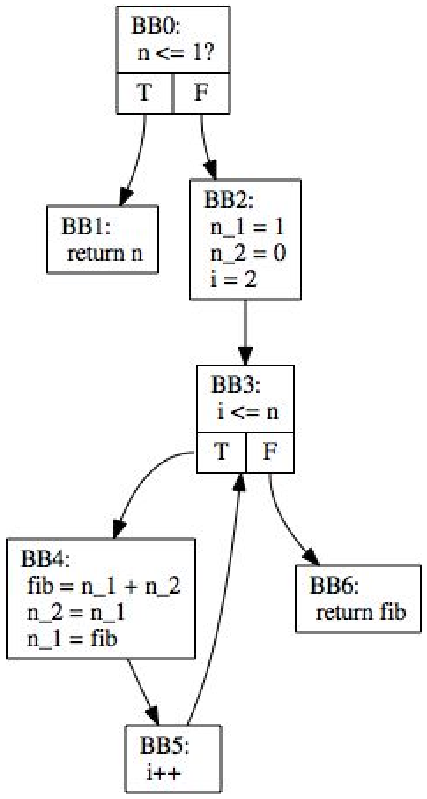
- Directed graph where nodes are basic block
- There is an edge between two blocks if the execution can flow from one block to the other

Control Flow Graph

- Directed graph where nodes are basic block
- There is an edge between two blocks if the execution can flow from one block to the other

```
int fibonacci(int n) {
  if (n <= 1)
    return n;

  int n_1 = 1, n_2 = 0, fib;
  for (int i=2; i <= n; i++) {
    fib = n_1 + n_2;
    n_2 = n_1;
    n_1 = fib;
  }
  return fib;
}
```



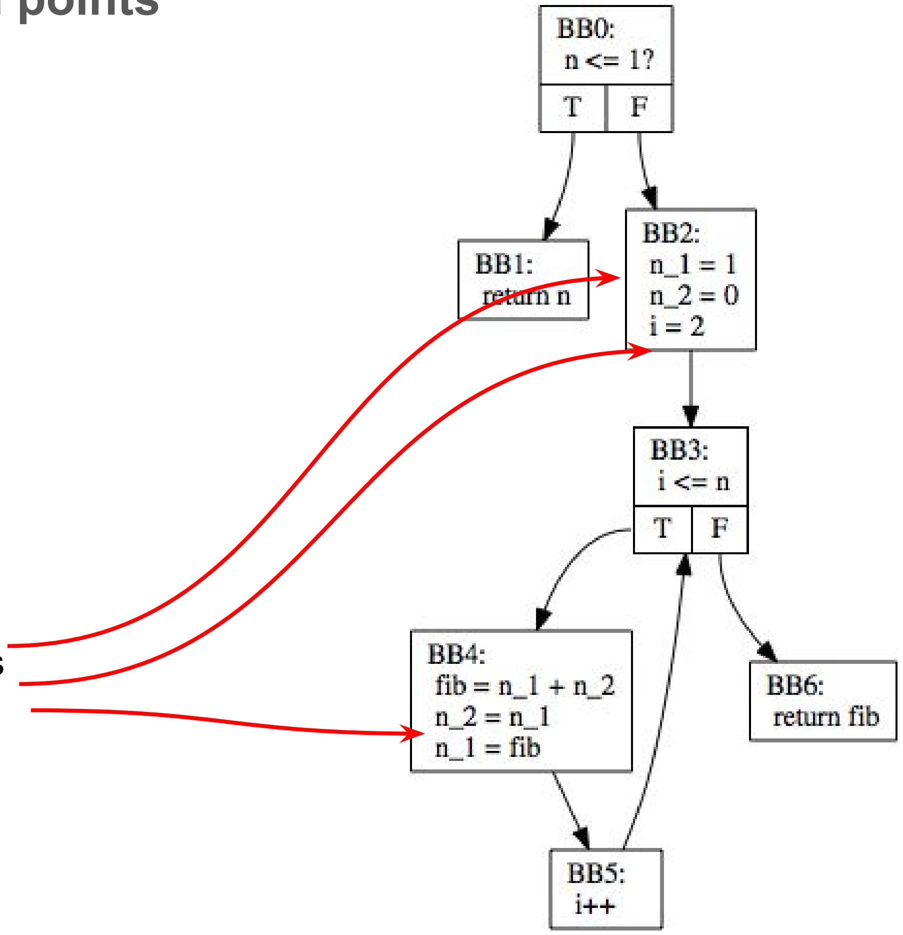
BB0: n <= 1?
T
F
BB1: return n
BB2: n_1 = 1 n_2 = 0 i = 2
BB3: i <= n
T
F
BB4: fib = n_1 + n_2 n_2 = n_1 n_1 = fib
BB6: return fib
BB5: i++

Data-flow Analysis

- Approximates the dynamic behavior of a program **regarding a property of interest** among several **program points**

Data-flow Analysis

- Approximates the dynamic behavior of a program **regarding a property of interest** among several **program points**
- Makes use of the program's **CFG**



BB0: n <= 1?
T
F
BB1: return n
BB2: n_1 = 1 n_2 = 0 i = 2
BB3: i <= n
T
F
BB4: fib = n_1 + n_2 n_2 = n_1 n_1 = fib
BB5: i++
BB6: return fib

How many registers do we need here?

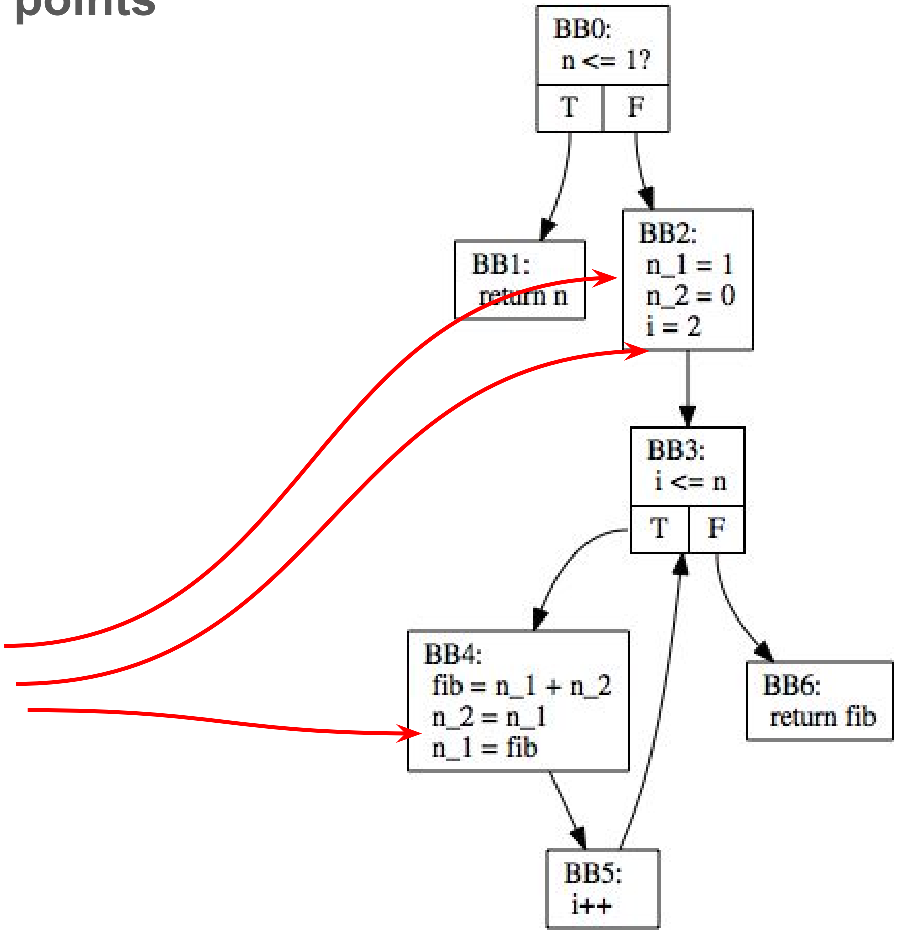
Data-flow Analysis

- Approximates the dynamic behavior of a program **regarding a property of interest** among several **program points**
- Makes use of the program's **CFG**
- Expressed as transfer functions

$$[\![p]\!]_{in} = ([\![p]\!]_{out} - \{v\}) \cup vars(E)$$

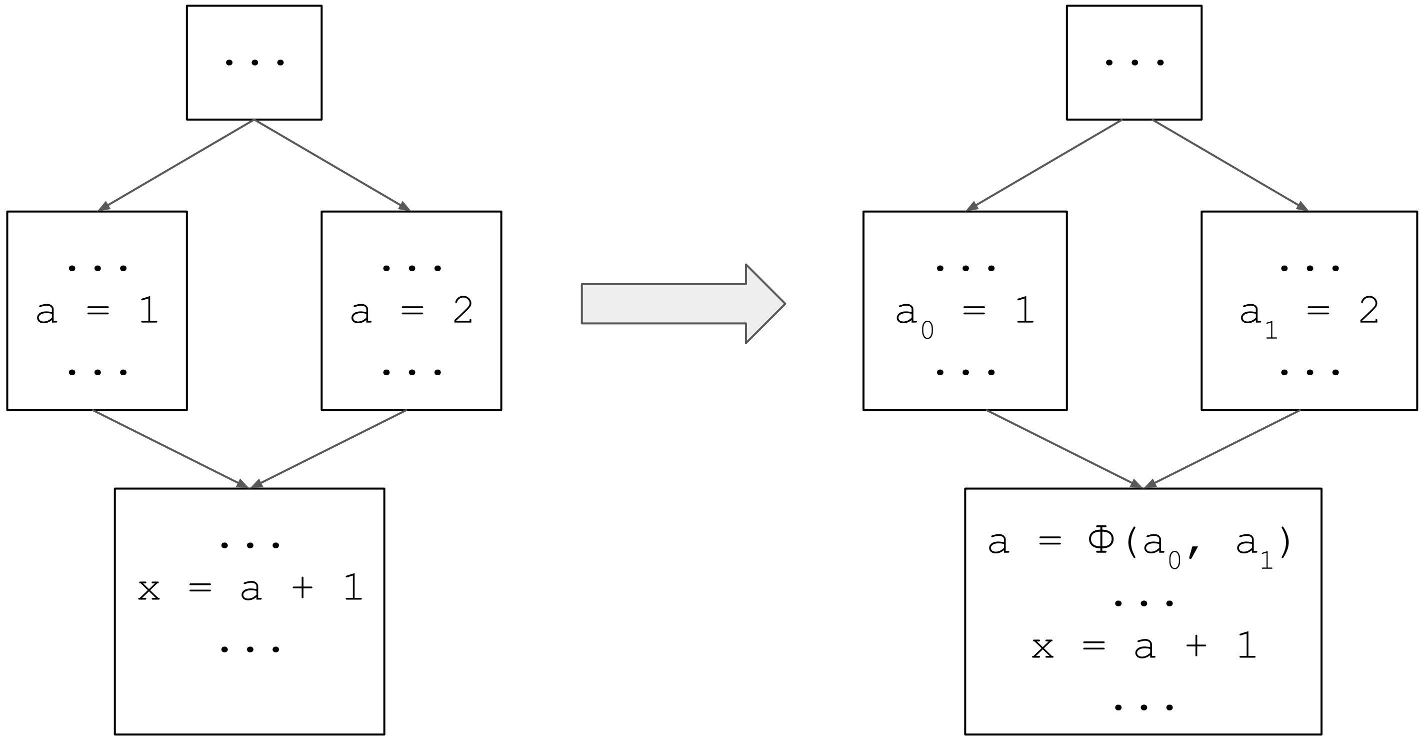
$$[\![p]\!]_{out} = \bigcup_{p' \in succ(p)} [\![p']\!]_{in}$$

How many registers do we need here?



SSA: Static Single Assignment

- Each variable is only **assigned once**
- Φ-functions
- Useful to make data-flow analyses sparse



Our Solution

Forward Array Size Analysis

- Forward data-flow analysis
- The **information starts** being propagated from **memory allocation instructions**

$$v = malloc(c) \ \vdash \ [\![v]\!] = \{c\}$$

$$v = malloc(a) \ \vdash \ [\![v]\!] = \{a\}, [\![a]\!]\cup = \{v\}$$

$$v_1 = v_2 + c \ \vdash \ \frac{v_2' \in [\![v_2]\!]}{[\![v_1]\!] = [\![v_2]\!], [\![v_2']\!]\cup = \{v_1\}}$$

$$v_1 = v_2 \odot v_3 \ \vdash \ \frac{v_2' \in [\![v_2]\!], v_3' \in [\![v_3]\!]}{[\![v_1]\!] = [\![v_2]\!] \cup [\![v_3]\!], [\![v_2']\!]\cup = \{v_1\}, [\![v_3']\!]\cup = \{v_1\}}$$

$$v = \phi(v_1, \ldots, v_n) \ \vdash \ \frac{v_1' \in [\![v_1]\!], \ldots, v_n' \in [\![v_n]\!]}{[\![v]\!] = \bigcup_{1 \le i \le n} [\![v_i]\!], \ [\![v_1']\!]\cup = \{v\}, \ldots, [\![v_n']\!]\cup = \{v\}}$$

Forward Array Size Analysis

- Forward data-flow analysis
- The **information starts** being propagated from **memory allocation instructions**

$$v = malloc(c) \ \vdash \ [\![v]\!] = \{c\}$$

$$v = malloc(a) \ \vdash \ [\![v]\!] = \{a\}, [\![a]\!]\cup = \{v\}$$

$$v'_2 \in [\![v_2]\!]$$

$$v_1$$

$$\}$$

$$v_1 = v_2 \odot v_3$$

$$v'_3]\!]\cup = \{v_1\}$$

$$]\!]$$

$$v = \phi(v_1, \ldots, v_n) \ \vdash$$

$$[\![v]\!] = \bigcup_{1 \leq i \leq n} [\![v_i]\!], \ [\![v'_1]\!]\cup = \{v\}, \ldots, [\![v'_n]\!]\cup = \{v\}$$

```
int *v = (int*) malloc(10);
```

Forward Array Size Analysis

- Forward data-flow analysis
- The **information starts** being propagated from **memory allocation instructions**

$$v = malloc(c) \ \vdash \ [\![v]\!] = \{c\}$$

$$v = malloc(a) \ \vdash \ [\![v]\!] = \{a\}, [\![a]\!]\cup = \{v\}$$

$v_2' \in [\![v_2]\!]$

v_1 }

$v_1 = v_2 \odot v_3$ $v_3']\!]\cup = \{v_1\}$

$$v = \phi(v_1, \ldots, v_n) \ \vdash \ \frac{\quad}{[\![v]\!] = \bigcup_{1 \leq i \leq n} [\![v_i]\!], \ [\![v_1']\!]\cup = \{v\}, \ldots, [\![v_n']\!]\cup = \{v\}}$$

```
int *v = (int*) malloc(n);
```

Forward Array Size Analysis

- Forward data-flow analysis
- The **information starts** being propagated from **memory allocation instructions**

$$v = malloc(c) \;\vdash\; [\![v]\!] = \{c\}$$

$$v = malloc(a) \;\vdash\; [\![v]\!] = \{a\}, [\![a]\!]\cup = \{v\}$$

$$v_1 = v_2 + c \;\vdash\; \frac{v_2' \in [\![v_2]\!]}{[\![v_1]\!] = [\![v_2]\!], [\![v_2']\!]\cup = \{v_1\}}$$

$$v_2' \in [\![v_2]\!], v_3' \in [\![v_3]\!]$$

$$v_1 = v_2 \odot v_3$$

$$v_3']\!]\cup = \{v_1\}$$

$$]\!]$$

$$v = \phi(v_1, \ldots, v_n)$$

$$[\![v_n']\!]\cup = \{v\}$$

```
int *v2 = ...
int *v1 = v2 + 1;
```

Forward Array Size Analysis

- Forward data-flow analysis
- The **information starts** being propagated from **memory allocation instructions**

v $\}$

v_1 $\}$

```
int *v = (int*) malloc(n2);
      int n1 = n2 + x;
```

$$v_1 = v_2 \odot v_3 \;\vdash\; \frac{v_2' \in [\![v_2]\!], v_3' \in [\![v_3]\!]}{[\![v_1]\!] = [\![v_2]\!] \cup [\![v_3]\!], [\![v_2']\!]\cup = \{v_1\}, [\![v_3']\!]\cup = \{v_1\}}$$

$$v = \phi(v_1, \ldots, v_n) \;\vdash\; \frac{v_1' \in [\![v_1]\!], \ldots, v_n' \in [\![v_n]\!]}{[\![v]\!] = \bigcup_{1 \le i \le n} [\![v_i]\!],\; [\![v_1']\!]\cup = \{v\}, \ldots, [\![v_n']\!]\cup = \{v\}}$$

Forward Array Size Analysis

- Forward data-flow analysis
- The **information starts** being propagated from **memory allocation instructions**

$$v = malloc(c) \;\vdash\; [\![v]\!] = \{c\}$$

$$v = malloc(a) \;\vdash\; [\![v]\!] = \{a\}, [\![a]\!]\cup = \{v\}$$

$$v_1 = v_2 + c \;\vdash\; \frac{v_2' \in [\![v_2]\!]}{[\![v_1]\!] = [\![v_2]\!], [\![v_2']\!]\cup = \{v_1\}}$$

$$v_1 = v_2 \odot v_3 \;\vdash\; \frac{v_2' \in [\![v_2]\!], v_3' \in [\![v_3]\!]}{[\![v_1]\!] = [\![v_2]\!] \cup [\![v_3]\!], [\![v_2']\!]\cup = \{v_1\}, [\![v_3']\!]\cup = \{v_1\}}$$

$$v = \phi(v_1, \ldots, v_n) \;\vdash\; \frac{v_1' \in [\![v_1]\!], \ldots, v_n' \in [\![v_n]\!]}{[\![v]\!] = \bigcup_{1 \le i \le n} [\![v_i]\!],\; [\![v_1']\!]\cup = \{v\}, \ldots, [\![v_n']\!]\cup = \{v\}}$$

Forward Array Size Analysis

```
void f() {
  int size = ...;
  ...
  int *array = (int*) malloc(sizeof(int)*size);
  ...
  foo(array, size);
  ...
}

void foo(int *array, int size) {
    for (int i=0; i < size; i++)
       array[i];
    ...
}
```

Forward Array Size Analysis

```
void f() {
  int size = ...;
  ...
  int *array = (int*) malloc(sizeof(int)*size);
  ...
  foo(array, size);
  ...
}

void foo(int *array, int size) {
    for (int i=0; i < size; i++)
       array[i];
    ...
}
```

```
%size = ...
...
%mul = mul 4, %size
%array = call @malloc(%mul)
...
call @foo(%array, %size);
...
```

Forward Array Size Analysis

```
%size = ...
...
%mul = mul 4, %size
%array = call @malloc(%mul)
...
call @foo(%array, %size);
...
```

$[\![\mathit{array}]\!] = \{\mathrm{mul}\}$

Forward Array Size Analysis

```
%size = ...
...
%mul = mul 4, %size
%array = call @malloc(%mul)
...
call @foo(%array, %size);
...
```

Does **size** have any relation to **mul**?

$[\![array]\!] = \{\mathrm{mul}\}$

Forward Array Size Analysis

```
%size = ...
...
%mul = mul 4, %size
%array = call @malloc(%mul)
...
call @foo(%array, %size);
...
```

Does **size** has any relation with **mul**?

$[\![array]\!] = \{\mathrm{mul}\}$

Forward Array Size Analysis

- Our analysis is interprocedural
  - We analyze the function in a topological order of the call-graph
- It lets us find user-defined memory allocation functions

```
void *xmalloc(size_t n) {
  void *p = malloc(n);
  if (p == 0)
    xalloc_die();
  return p;
}
```

$[\![p]\!] = \{n\}$

Results

- The forward analysis only found sizes for two benchmarks in **SPEC**
  - 6.9% for GCC
  - 0.5% for HMMER
  - **1.3% total**
- For the **single-source** and **multi-source** benchmarks present in LLVM's test-suite, the forward analysis found about **3%** of array sizes
  - The **maximum** was **66.6%** for FreeBench's *analyzer*

Results

- We have randomly selected **12** SPEC **functions** for **manual inspection**
- Found 4 main code patterns for which our forward analysis is ineffective

Results

- We have randomly selected **12** SPEC **functions** for **manual inspection**
- Found 4 main code patterns for which our forward analysis is ineffective
  1. The size is calculated with no relation to the array allocation

```
...
int *a = (int*) malloc(sizeof(int)*n);
...
foo(a, 10);
...
```

Results

- We have randomly selected **12** SPEC **functions** for **manual inspection**
- Found 4 main code patterns for which our forward analysis is ineffective
  1. The size is calculated with no relation to the array allocation
  2. The array (and maybe its size) is encapsulated inside a struct

```c
struct my_vector {
  int *array;
  size_t size;
};

void f(struct my_vector v) {
  ...
  foo(v.array, v.size);
  ...
}
```

Results

- We have randomly selected **12** SPEC **functions** for **manual inspection**
- Found 4 main code patterns for which our forward analysis is ineffective
  1. The size is calculated with no relation to the array allocation
  2. The array (and maybe its size) is encapsulated inside a struct
  3. The array is statically allocated

```
...
int a[128];
...
for (int i=0; i < n, i++)
  foo(&a[10], i);
...
```

Results

- We have randomly selected **12** SPEC **functions** for **manual inspection**
- Found 4 main code patterns for which our forward analysis is ineffective
  1. The size is calculated with no relation to the array allocation
  2. The array (and maybe its size) is encapsulated inside a struct
  3. The array is statically allocated
  4. The array is received as a function argument

```
void f(int *a, int size) {
  ...
  foo(a, size);
  ...
}
```

Backward Array Size Analysis

- The backward analysis doesn't need allocation information
- The **information starts** being propagated, backwards, from **array accesses**
- This analysis was proposed by Alves *et al*. and uses the symbolic range analysis proposed by Nazaré *et al.*

Backward Array Size Analysis

- The backward analysis doesn't need allocation information
- The **information starts** being propagated, backwards, from **array accesses**
- This analysis was proposed by Alves *et al*. and uses the symbolic range analysis proposed by Nazaré *et al.*

```
void foo(int *array, int size) {
    for (int i=0; i < size; i++)
       array[i];
    ...
}
```

Backward Array Size Analysis

- The backward analysis doesn’t need allocation information
- The **information starts** being propagated, backwards, from **array accesses**
- This analysis was proposed by Alves *et al*. and uses the symbolic range analysis proposed by Nazaré *et al.*

```
void foo(int *array, int size) {
    for (int i=0; i < size; i++)
        array[i];
    ...
}
```

Gather array access expressions (only “i” in this case)

Backward Array Size Analysis

- The backward analysis doesn't need allocation information
- The **information starts** being propagated, backwards, from **array accesses**
- This analysis was proposed by Alves *et al*. and uses the symbolic range analysis proposed by Nazaré *et al.*

```
void foo(int *array, int size) {
    for (int i=0; i < size; i++)
        array[i];
    ...
}
```

Symbolic range analysis on each variable of the array access expressions.
i: [0, max(0, size-1)]

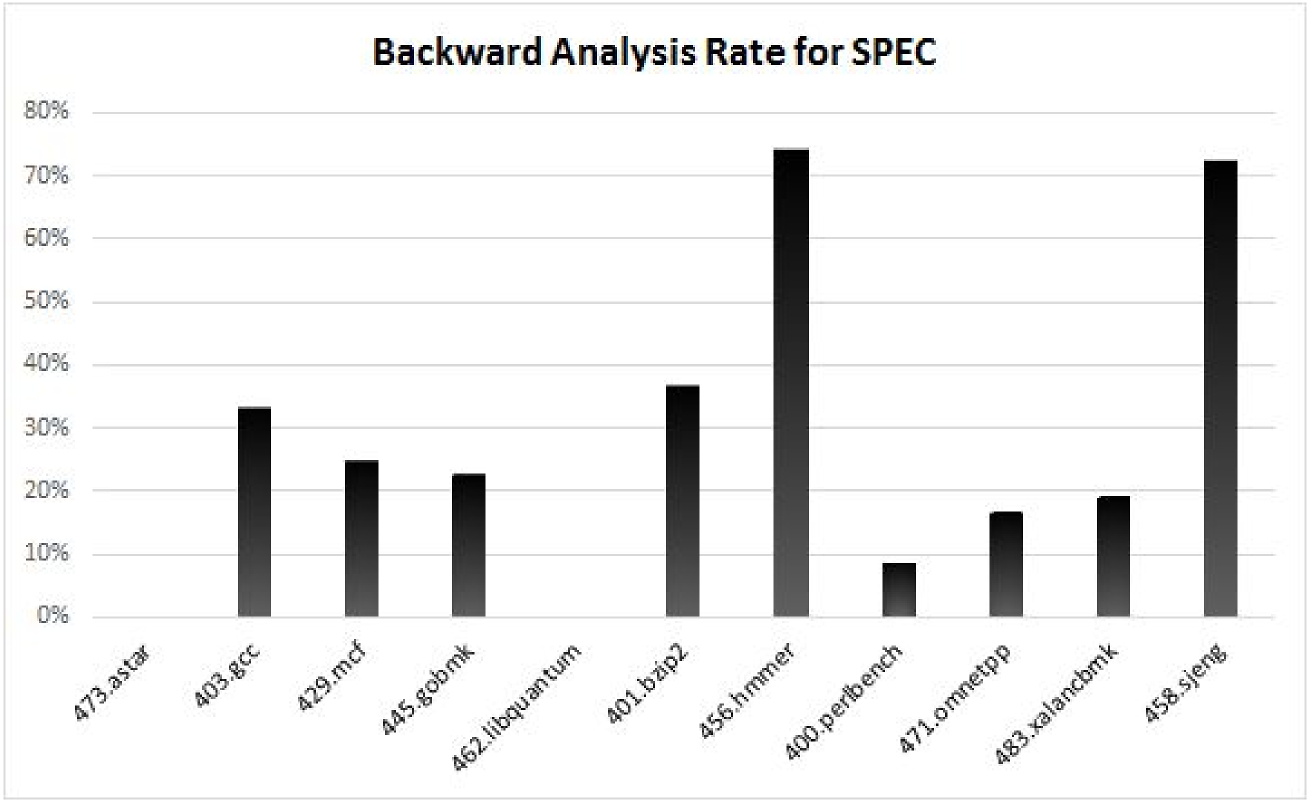
Backward Array Size Analysis

- The backward analysis doesn't need allocation information
- The **information starts** being propagated, backwards, from **array accesses**
- This analysis was proposed by Alves *et al*. and uses the symbolic range analysis proposed by Nazaré *et al.*

```
void foo(int *array, int size) {
    for (int i=0; i < size; i++)
        array[i];
    ...
}
```

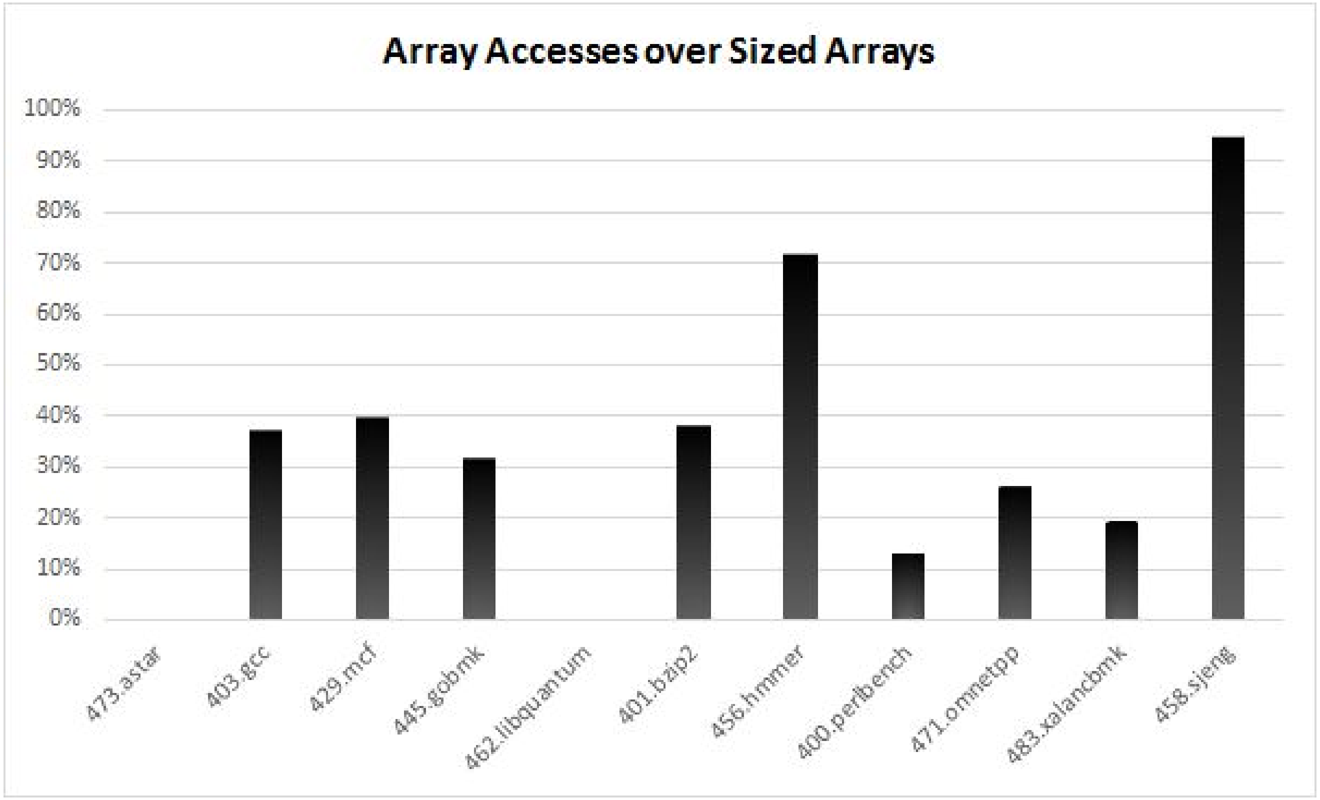
We then check whether the upper bound (max(0, size-1)) has a relation with any parameter

Results



The **backward** analysis found **33.9%** of the array sizes for SPEC

Results



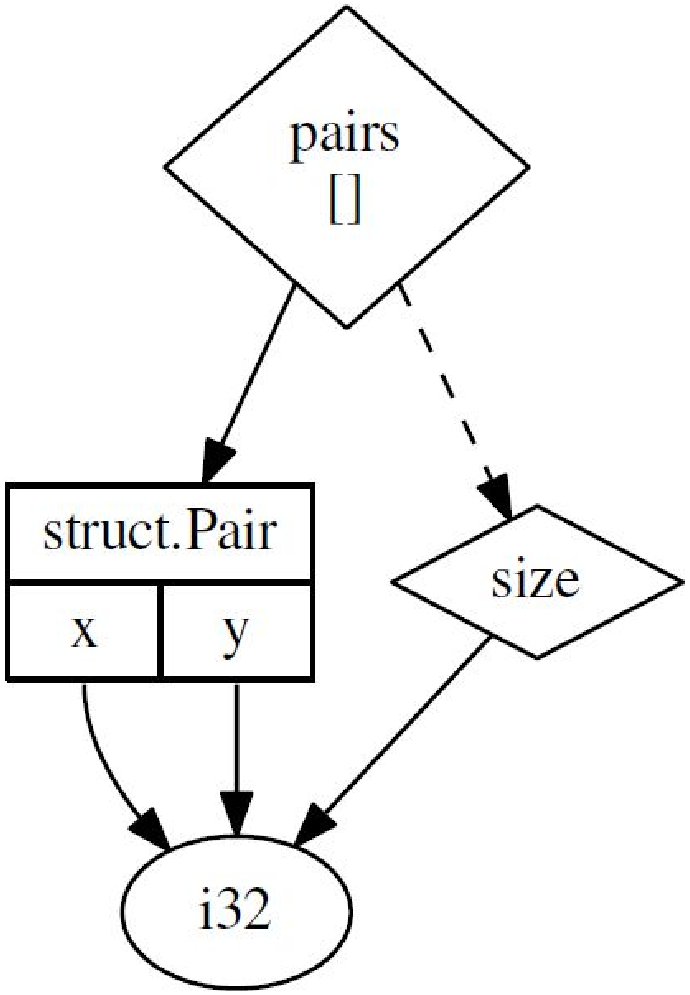
33% of all array accesses in SPEC are performed over sized arrays

Input Generator

Data Structure Graph

```
struct {
  int x;
  int y;
} Pair;

void closestPoint(Pair *pairs, int size) {
  int min = INFINITY;
  Pair closest;
  for (int i=0; i < size; i++) {
    Pair p = pairs[i];
    int dist = sqrt(pow(p.x, 2)
                 + pow(p.y, 2));
    if (dist < min) {
      min = dist;
      closest = p;
    }
  }
  printf("Closest = (%d, %d)\n", closest.x,
          closest.y);
}
```



Input Generation

```
function generate_value(field):
  if field is pointer && not field is array:
    if flip_coin():
      allocate memory and store the result of a recursive call
    else:
      return NULL
  else if field is array:
    create a function to create arrays of type field.node.type
    if forward analysis found size:
      get the slice to calculate the size
    else if backward analysis found size:
      get the size from the backward analysis
    else:
      size = random_int()
    return call to the created function passing size as argument
  else if field.node.type is struct:
    allocate memory
    for each struct field f’:
      generate_value(f’)
  else:
    return random_value(field.node.type)
```

Input Generation

```
function generate_value(field):
  if field is pointer && not field is array:
    if flip_coin():
      allocate memory and store the result of a recursive call
    else:
      return NULL
  else if field is array:
    create a function to create arrays of type field.node.type
    if forward analysis found size:
      get the slice to calculate the size
    else if backward analysis found size:
      get the size from the backward analysis
    else:
      size = random_int()
    return call to the created function passing size as argument
  else if field.node.type is struct:
    allocate memory
    for each struct field f':
      generate_value(f')
  else:
    return random_value(field.node.type)
```

Input Generation

```
function generate_value(field):
  if field is pointer && not field is array:
    if flip_coin():
      allocate memory and store the result of a recursive call
    else:
      return NULL
  else if field is array:
    create a function to create arrays of type field.node.type
    if forward analysis found size:
      get the slice to calculate the size
    else if backward analysis found size:
      get the size from the backward analysis
    else:
      size = random_int()
    return call to the created function passing size as argument
  else if field.node.type is struct:
    allocate memory
    for each struct field f':
      generate_value(f')
  else:
    return random_value(field.node.type)
```

Input Generation

```
function generate_value(field):
  if field is pointer && not field is array:
    if flip_coin():
      allocate memory and store the result of a recursive call
    else:
      return NULL
  else if field is array:
    create a function to create arrays of type field.node.type
    if forward analysis found size:
      get the slice to calculate the size
    else if backward analysis found size:
      get the size from the backward analysis
    else:
      size = random_int()
    return call to the created function passing size as argument
  else if field.node.type is struct:
    allocate memory
    for each struct field f':
      generate_value(f')
  else:
    return random_value(field.node.type)
```

Input Generation

```
function generate_value(field):
  if field is pointer && not field is array:
    if flip_coin():
      allocate memory and store the result of a recursive call
    else:
      return NULL
  else if field is array:
    create a function to create arrays of type field.node.type
    if forward analysis found size:
      get the slice to calculate the size
    else if backward analysis found size:
      get the size from the backward analysis
    else:
      size = random_int()
    return call to the created function passing size as argument
  else if field.node.type is struct:
    allocate memory
    for each struct field f':
      generate_value(f')
  else:
    return random_value(field.node.type)
```

Case Study

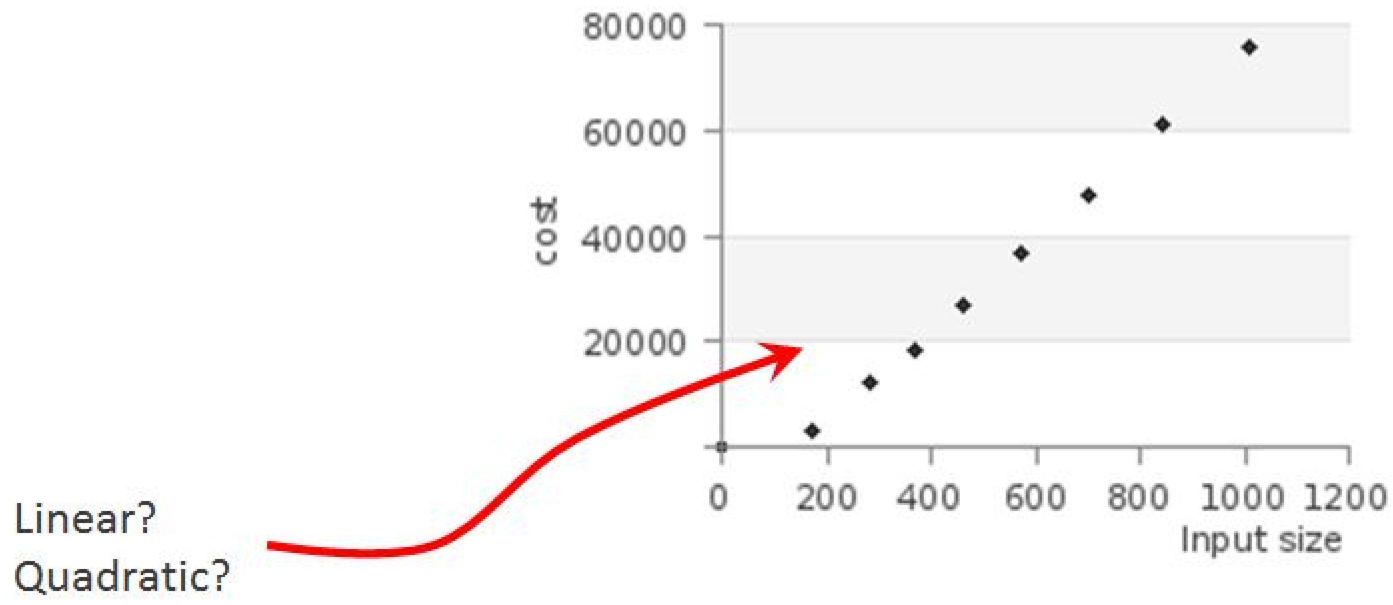
Context

- Previous approaches on automatic inference of program complexity have shortcomings:

Context

- Previous approaches on automatic inference of program complexity have shortcomings:
    - Imprecise

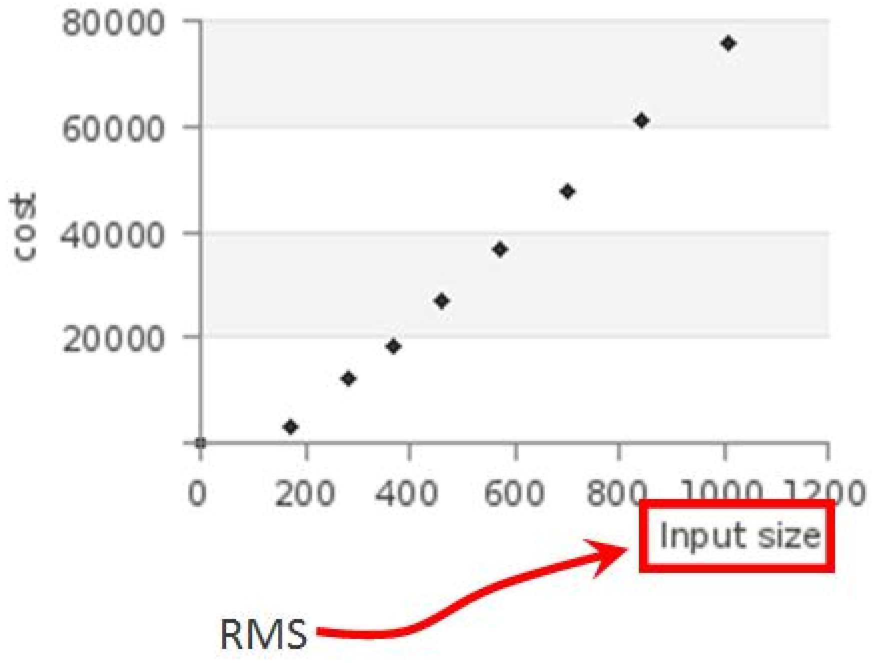
Aprof's output



Context

- Previous approaches on automatic inference of program complexity have shortcomings:
  - Imprecise
  - Hard to read

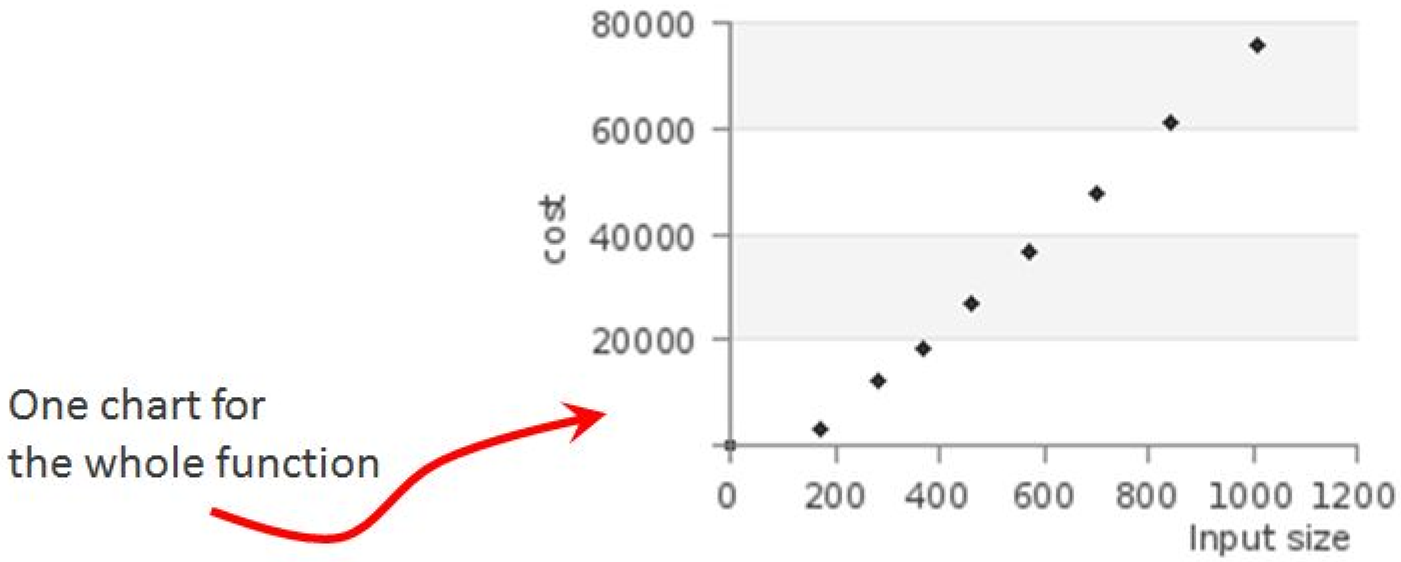
Aprof's output



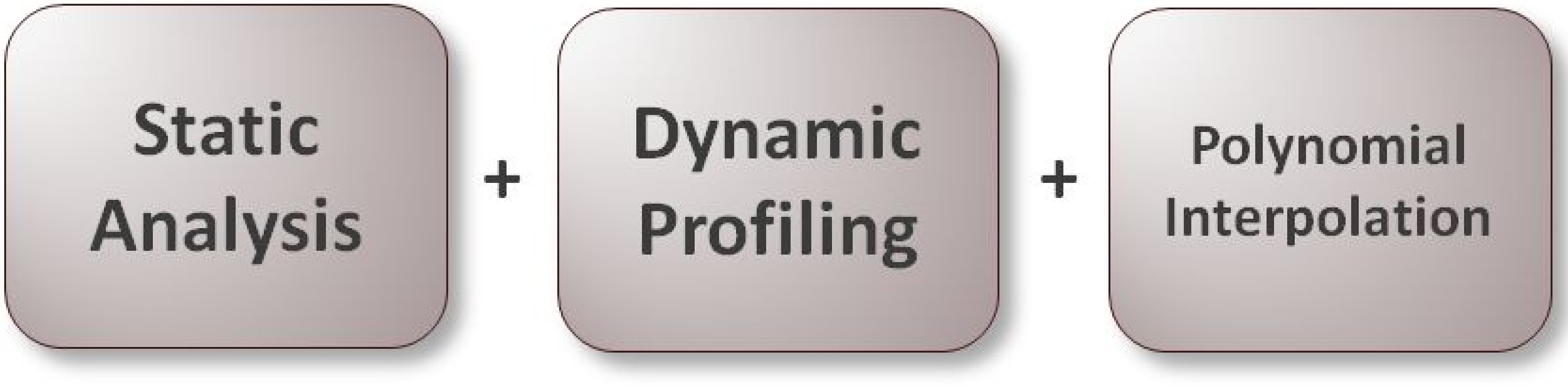
Context

- Previous approaches on automatic inference of program complexity have shortcomings:
  - Imprecise
  - Hard to read
  - Too coarse

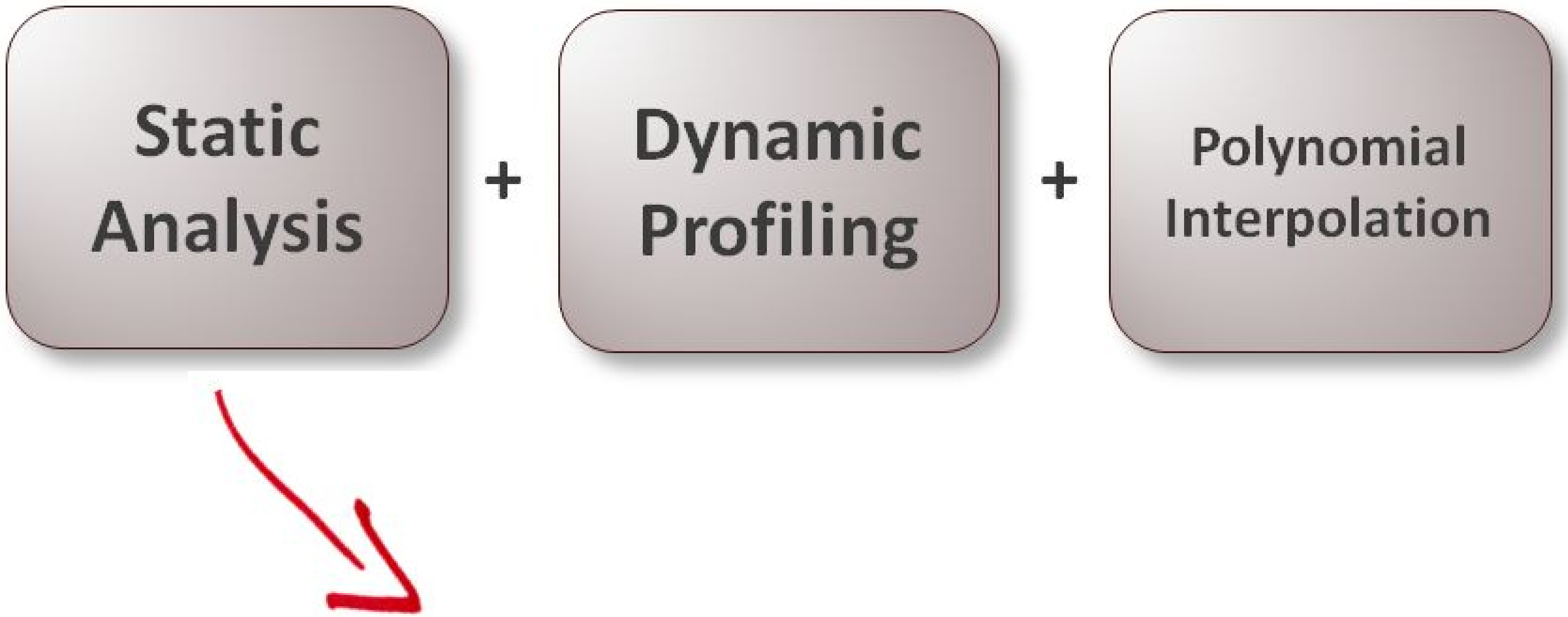
Aprof's output



Asymptus

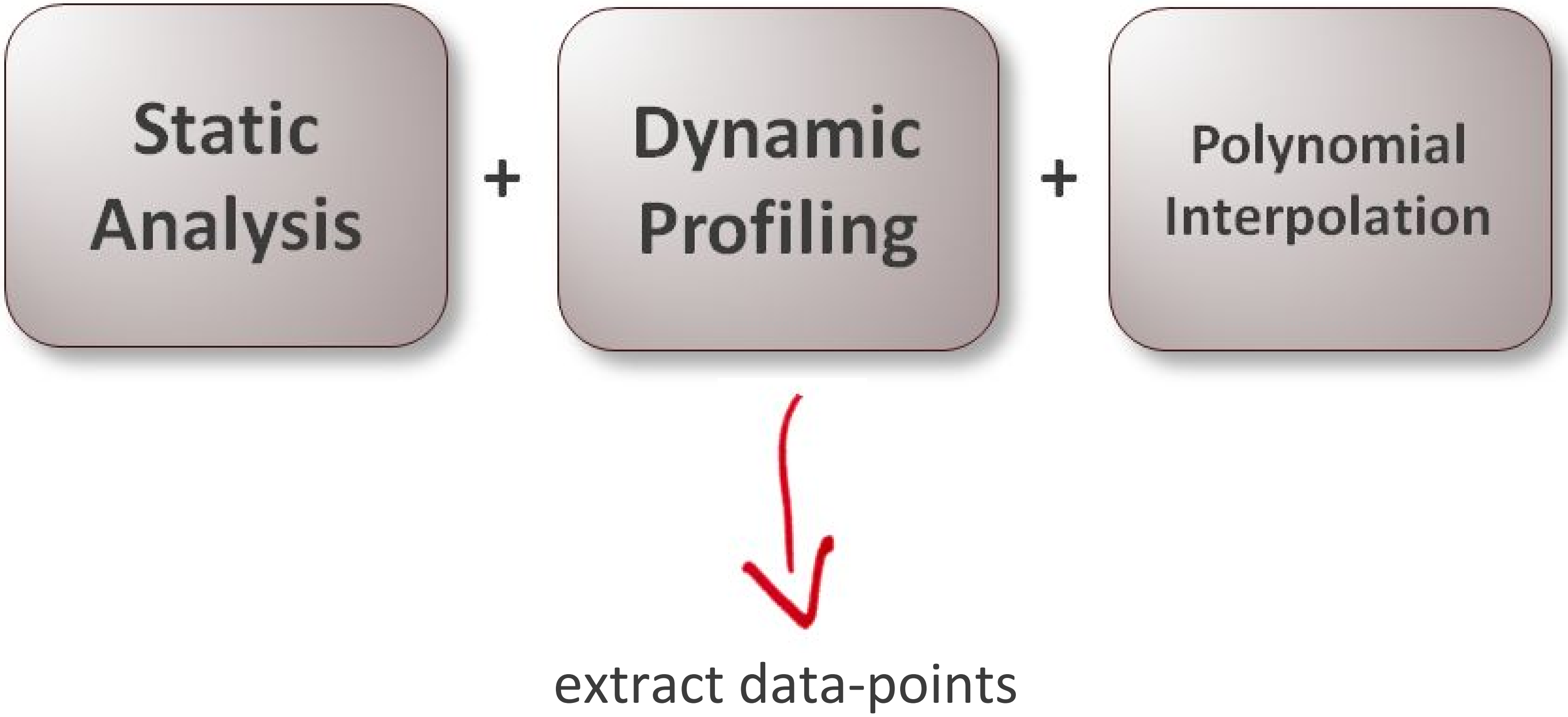

Static Analysis
+
Dynamic Profiling
+
Polynomial Interpolation

Asymptus



identify loop **inputs** and **relations** between loops

Asymptus


Static Analysis
+
Dynamic Profiling
+
Polynomial Interpolation
extract data-points

Asymptus

Static Analysis + **Dynamic Profiling** + **Polynomial Interpolation**

$\swarrow$

find a cost function

Example

```c
int** multiply(int **matA, int **matB, int side){
  int i, j, k, sum;
  int **result = (int**) malloc(side*sizeof(int*));
  for (i = 0; i < side; i++)
    result[i] = (int*) malloc(side*sizeof(int));

  for (i=0; i < side; i++) {
    for (j=0; j < side; j++) {
      sum = 0;
      for (k=0; k < side; k++) {
        sum += matA[i][k] * matB[k][j];
      }
      result[i][j] = sum;
    }
  }
  return result;
}
```

Example

```
 1: int** multiply(int **matA, int **matB, int side){
 2:   int i, j, k, sum;
 3:   int **result = (int**) malloc(side*sizeof(int*));
 4:   for (i = 0; i < side; i++)
 5:     result[i] = (int*) malloc(side*sizeof(int));
 6:
 7:   for (i=0; i < side; i++) {
 8:     for (j=0; j < side; j++) {
 9:       sum = 0;
10:       for (k=0; k < side; k++) {
11:         sum += matA[i][k] * matB[k][j];
12:       }
13:       result[i][j] = sum;
14:     }
15:   }
17:   return result;
18: }
```

Example

```
int** multiply(int **matA, int **matB, int side){
  int i, j, k, sum;
  int **result = (int**) malloc(side*sizeof(int*));
  for (i = 0; i < side; i++)
    result[i] = (int*) malloc(side*sizeof(int));

  for (i=0; i < side; i++) {
    for (j=0; j < side; j++) {
      sum = 0;
      for (k=0; k < side; k++) {
        sum += matA[i][k] * matB[k][j];
      }
      result[i][j] = sum;
    }
  }
  return result;
}
```

Inputs

Example

```
 1: int** multiply(int **matA, int **matB, int side){
 2:   int i, j, k, sum;
 3:   int **result = (int**) malloc(side*sizeof(int*));
 4:   for (i = 0; i < side; i++)
 5:     result[i] = (int*) malloc(side*sizeof(int));
 6:
 7:   for (i=0; i < side; i++) {
 8:     for (j=0; j < side; j++) {
 9:       sum = 0;
10:       for (k=0; k < side; k++) {
11:         sum += matA[i][k] * matB[k][j];
12:       }
13:       result[i][j] = sum;
14:     }
15:   }
17:   return result;
18: }
```

Multiply

Line 7 x Line 8 x Line 10

Example

```
 1: int** multiply(int **matA, int **matB, int side){
 2:   int i, j, k, sum;
 3:   int **result = (int**) malloc(side*sizeof(int*));
 4:   for (i = 0; i < side; i++)
 5:     result[i] = (int*) malloc(side*sizeof(int));
 6:
 7:   for (i=0; i < side; i++) {
 8:     for (j=0; j < side; j++) {
 9:       sum = 0;
10:       for (k=0; k < side; k++) {
11:         sum += matA[i][k] * matB[k][j];
12:       }
13:       result[i][j] = sum;
14:     }
15:   }
17:   return result;
18: }
```

Sum

Multiply

Line 4 + (Line 7 x Line 8 x Line 10)

Example

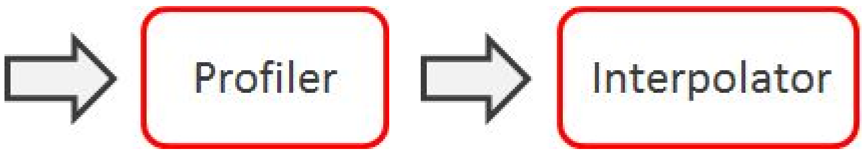
```
int** multiply(int **matA, int **matB, int side){
  int i, j, k, sum;
  int **result = (int**) malloc(side*sizeof(int*));
  for (i = 0; i < side; i++)
    result[i] = (int*) malloc(side*sizeof(int));

  for (i=0; i < side; i++) {
    for (j=0; j < side; j++) {
      sum = 0;
      for (k=0; k < side; k++) {
        sum += matA[i][k] * matB[k][j];
      }
      result[i][j] = sum;
    }
  }
  return result;
}
```

Example

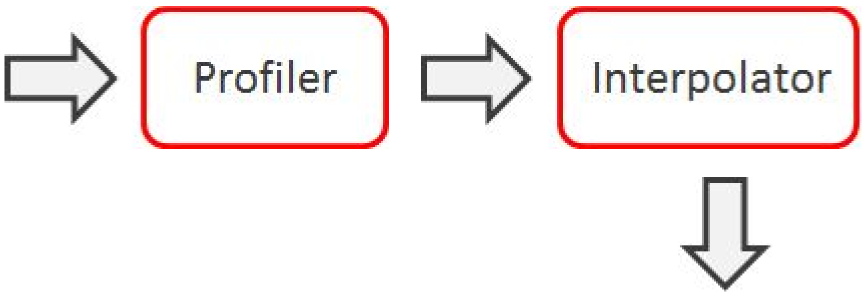
```c
int** multiply(int **matA, int **matB, int side){
  int i, j, k, sum;
  int **result = (int**) malloc(side*sizeof(int*));
  for (i = 0; i < side; i++)
    result[i] = (int*) malloc(side*sizeof(int));

  for (i=0; i < side; i++) {
    for (j=0; j < side; j++) {
      sum = 0;
      for (k=0; k < side; k++) {
        sum += matA[i][k] * matB[k][j];
      }
      result[i][j] = sum;
    }
  }
  return result;
}
```



Example

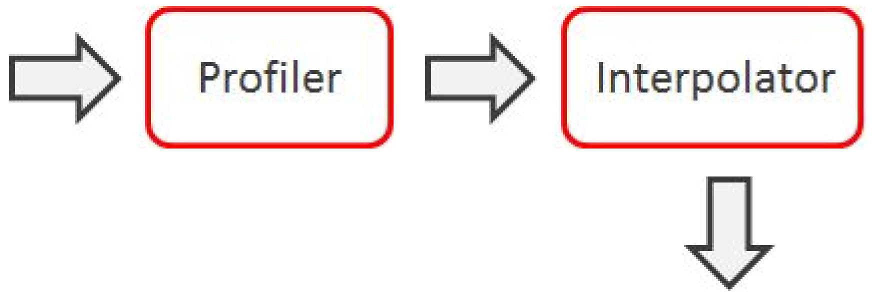
```
 1: int** multiply(int **matA, int **matB, int side){
 2:   int i, j, k, sum;
 3:   int **result = (int**) malloc(side*sizeof(int*));
 4:   for (i = 0; i < side; i++)
 5:     result[i] = (int*) malloc(side*sizeof(int));
 6:
 7:   for (i=0; i < side; i++) {
 8:     for (j=0; j < side; j++) {
 9:       sum = 0;
10:       for (k=0; k < side; k++) {
11:         sum += matA[i][k] * matB[k][j];
12:       }
13:       result[i][j] = sum;
14:     }
15:   }
17:   return result;
18: }
```



Line 4 + (Line 7 x Line 8 x Line 10)

Example

```
 1: int** multiply(int **matA, int **matB, int side){
 2:   int i, j, k, sum;
 3:   int **result = (int**) malloc(side*sizeof(int*));
 4:   for (i = 0; i < side; i++)
 5:     result[i] = (int*) malloc(side*sizeof(int));
 6:
 7:   for (i=0; i < side; i++) {
 8:     for (j=0; j < side; j++) {
 9:       sum = 0;
10:       for (k=0; k < side; k++) {
11:         sum += matA[i][k] * matB[k][j];
12:       }
13:       result[i][j] = sum;
14:     }
15:   }
17:   return result;
18: }
```



Line 4 + (Line 7 x Line 8 x Line 10)

Function ‘multiply(int**, int**, int)':

Loop at line 4: 1.00 * side + 1.00

Loop at line 7: 1.00 * side + 1.00

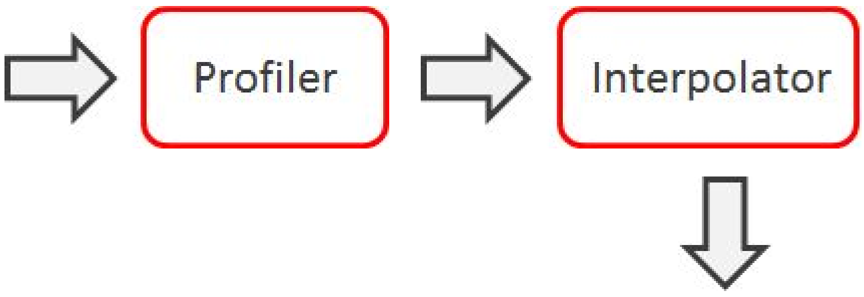
Loop at line 8: 1.00 * side + 1.00

Loop at line 10: 1.00 * side + 1.00

Complexity: O(side^3)

Example

```
 1: int** multiply(int **matA, int **matB, int side){
 2:   int i, j, k, sum;
 3:   int **result = (int**) malloc(side*sizeof(int*));
 4:   for (i = 0; i < side; i++)
 5:     result[i] = (int*) malloc(side*sizeof(int));
 6:
 7:   for (i=0; i < side; i++) {
 8:     for (j=0; j < side; j++) {
 9:       sum = 0;
10:       for (k=0; k < side; k++) {
11:         sum += matA[i][k] * matB[k][j];
12:       }
13:       result[i][j] = sum;
14:     }
15:   }
17:   return result;
18: }
```



Line 4 + (Line 7 x Line 8 x Line 10)

Function ‘multiply(int**, int**, int)':

Loop at line 4: 1.00 * side + 1.00

Loop at line 7: 1.00 * side + 1.00

Loop at line 8: 1.00 * side + 1.00

Loop at line 10: 1.00 * side + 1.00

Complexity: O(side^3)

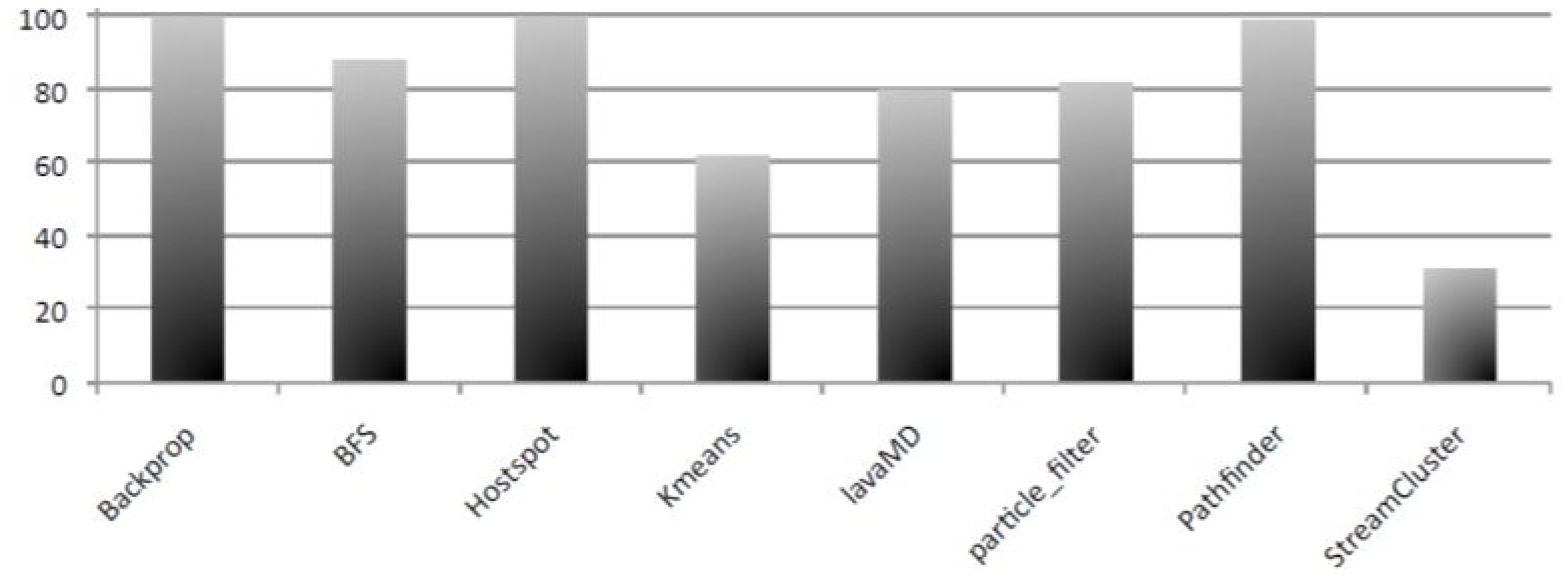
-Precise

-Uses program’s symbols

-Loop level

Results

- Analyzed **99.7%** of all loops in Polybench and **69.18%** in Rodinia



Percentage of correctly analyzed loops per benchmark of Rodinia.

Limitations of Asymptus

- Only works for polynomial programs
- Loops controlled by a data structure value
- The function needs to be executed a certain number of times

Limitations of Asymptus

- Only works for polynomial programs
- Loops controlled by a data structure value
- **The function needs to be executed a certain number of times**

We can use our input generator to help Asymptus!

Our Experiment

- We have written 10 functions having different kinds of inpus
  - Arrays
  - Matrices
  - Recursive data structures
- Modified 6 Polybench functions
  - Originally, the inputs of the functions were defined by macros (definition at compile time)
- Our input generator could help Asymptus analyze all the functions

Final Remarks

Limitations and Future Work

- The input generator doesn’t get the complete expression representing the upper bound of the array accesses

```
void foo(int *a, int size) {
  for (int i=0; i < n, i++)
    a[size+i];
  ...
}
```

Size of “a” has to be at least size*2

Limitations and Future Work

- The input generator doesn't get the complete expression representing the upper bound of the array accesses
- The forward analysis doesn't deal with statically allocated arrays

```
...
int a[128];
...
for (int i=0; i < n, i++)
  foo(&a[10], i);
...
```

Limitations and Future Work

- The input generator doesn’t get the complete expression representing the upper bound of the array accesses
- The forward analysis doesn’t deal with statically allocated arrays
- The forward analysis doesn’t track arrays over function calls

```
void f(int *a, int size) {
   ...
   foo(a, size);
   ...
}
```

Limitations and Future Work

- The input generator doesn’t get the complete expression representing the upper bound of the array accesses
- The forward analysis doesn’t deal with statically allocated arrays
- The forward analysis doesn’t track arrays over function calls
- Create an analysis to infer the size of recursive data structures

```
struct list {
  struct node *head;
  size_t size;
};

void f(struct list l) {
  struct node *n = l.head;
  for (int i=0; i < l.size; i++) {
    ...
    n = n->next;
  }
}
```

Contributions

- The design of a **forward data-flow analysis** to bind **meta information of allocated memory**
- The design of an **algorithm to generate inputs** for functions that makes **use** of **array size analyses**
- The **identification of pointers used as arrays** by looking for pointer arithmetics
- The development **Asymptus** (a tool for program complexity inference)
- The development of a **testing infrastructure for Maxtrack**

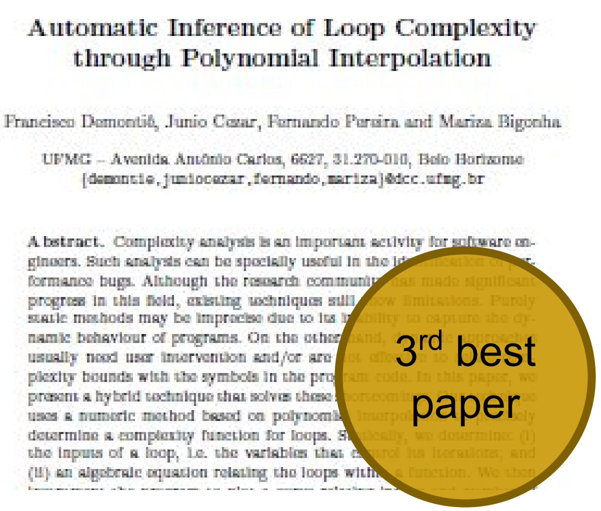
Contributions

Um Algoritmo para Emparelhamento de Chamadas de Função

Francisco Demontiê, Filipe de Lima Arcanjo e Mariza A. S. Bigonha

Universidade Federal de Minas Gerais
{demontie,filipe,mariza}@dcc.ufmg.br

Resumo A maior parte dos compiladores atuais realiza uma série de otimizações no programa fonte sem garantir a preservação da semântica desse programa. Verificar a corretude de otimizações é uma tarefa difícil, pois erros de otimização podem aparecer em pontos muito diferentes daqueles onde eles tiveram origem. A fim de facilitar essa tarefa de depuração, este artigo apresenta uma técnica para o emparelhamento de programas antes e depois de otimizações. A grande inovação deste trabalho é usar funções externas como pontos de correspondência entre programas. Essa técnica é útil por várias razões. Em particular, ela pode encontrar problemas facilmente e pode servir como âncora para análises mais poderosas. Além disso, o emparelhamento proposto é rápido e confiável, pois compiladores não podem otimizar chamadas a funções que não possuem corpo, logo, tais funções precisam ser mantidas pelo otimizador.

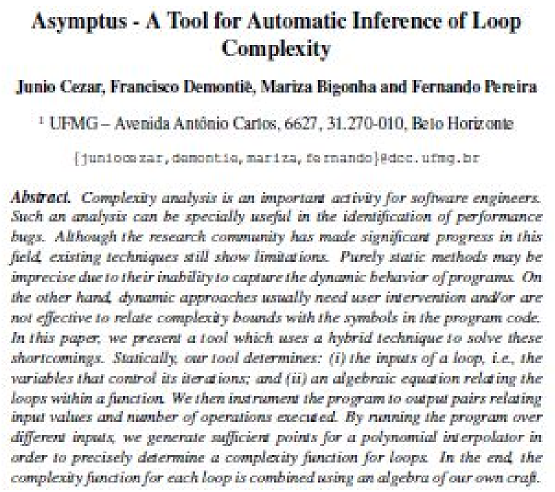


Automatic Inference of Loop Complexity through Polynomial Interpolation

Francisco Demontiê, Junio Cezar, Fernando Pereira and Mariza Bigonha

UFMG – Avenida Antônio Carlos, 6627, 31.270-010, Belo Horizonte
{demontie,juniocezar,fernando,mariza}@dcc.ufmg.br

Abstract. Complexity analysis is an important activity for software engineers. Such analysis can be specially useful in the [illegible] formance bugs. Although the research communi[illegible] has made significant progress in this field, existing techniques still [illegible]ow limitations. Purely static methods may be imprecise due to its [illegible]bility to capture the dynamic behaviour of programs. On the other [illegible] usually need user intervention and/or are [illegible] plexity bounds with the symbols in the pro[illegible]ram code. In this paper, we present a hybrid technique that solves these [illegible] uses a numeric method based on polynomi[illegible] determine a complexity function for loops. S[illegible]ically, we determine: (i) the inputs of a loop, i.e. the variables that c[illegible]trol its iterations; and (ii) an algebraic equation relating the loops withi[illegible] a function. We then



Asymptus - A Tool for Automatic Inference of Loop Complexity

Junio Cezar, Francisco Demontiê, Mariza Bigonha and Fernando Pereira

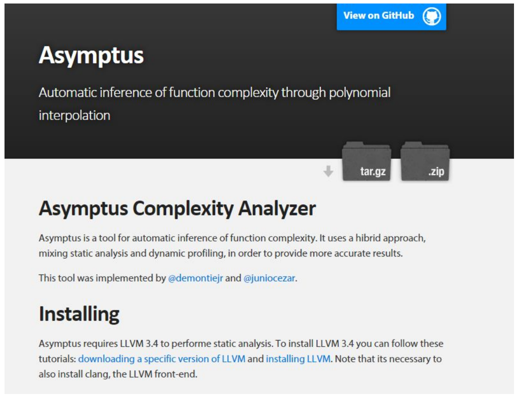
[1] UFMG – Avenida Antônio Carlos, 6627, 31.270-010, Belo Horizonte

{juniocezar,demontie,mariza,fernando}@dcc.ufmg.br

Abstract. *Complexity analysis is an important activity for software engineers. Such an analysis can be specially useful in the identification of performance bugs. Although the research community has made significant progress in this field, existing techniques still show limitations. Purely static methods may be imprecise due to their inability to capture the dynamic behavior of programs. On the other hand, dynamic approaches usually need user intervention and/or are not effective to relate complexity bounds with the symbols in the program code. In this paper, we present a tool which uses a hybrid technique to solve these shortcomings. Statically, our tool determines: (i) the inputs of a loop, i.e., the variables that control its iterations; and (ii) an algebraic equation relating the loops within a function. We then instrument the program to output pairs relating input values and number of operations executed. By running the program over different inputs, we generate sufficient points for a polynomial interpolator in order to precisely determine a complexity function for loops. In the end, the complexity function for each loop is combined using an algebra of our own craft.*

LAC
Asymptus – A Tool for Automatic Inference of Loop Complexity
UFMG
ICEx
6:48

Asymptus - A Tool for Automatic Inference of Loop Complexity



View on GitHub

Asymptus

Automatic inference of function complexity through polynomial interpolation

tar.gz .zip

Asymptus Complexity Analyzer

Asymptus is a tool for automatic inference of function complexity. It uses a hibrid approach, mixing static analysis and dynamic profiling, in order to provide more accurate results.

This tool was implemented by @demontiejr and @juniocezar.

Installing

Asymptus requires LLVM 3.4 to performe static analysis. To install LLVM 3.4 you can follow these tutorials: downloading a specific version of LLVM and installing LLVM. Note that its necessary to also install clang, the LLVM front-end.

Primeira demo infraestrutura de testes

https://youtu.be/pzfrIDfoCEc https://demontiejr.github.io/asymptus https://youtu.be/GRnpbE8f_U4

Generation of Test Cases for Languages with Pointer Arithmetics

Demontiê Junior
Advisor: Mariza Bigonha
Co-advisor: Fernando Pereira