Analysing C programs: arrays, pointers with precision and scale

IBISC Seminar

Laure Gonnord and Maroua Maalej (LIP) and Fernando Pereira (UFMG, Brasil) and ...

University of Lyon / LIP

October 13th, 2016

Plan

Motivations and big picture

Technical context: LLVM

Symbolic range analysis

Application 1: GreenArrays toolsuite
- Overview
- A bit on the other analyses
- Experimental results
- Conclusion

Application 2: Symbolic Pointer Analysis
- Overview
- Experimental results

Conclusion

Goal: Safety

Prove that (some) memory accesses are safe:

```
int main () {
    int v[10] ;
    v[0]=0;
    return v[20]
}
```

- Fight against bugs and overflow attacks.

Goal: Performance

Prove that (some) memory accesses are non-overlapping:

```
void fill_array(char* p)
    int i;
    for (i=0;i<4;i++) {
        p[i]=0;
    }
    for (i=4;i<8;i++) {
        p[i]=2*i;
    }
```

- Base analysis for automatic parallelisation, code motion, ...

Issues

Challenges:

- Thousands, millions of lines of code to analyse
- Compilers are complex programs.

- Growing need for simple **specialized** analyses that **scale**

Scaling analyses: two experiences

In this talk, I will present:

- (OOPSLA'14) The design of an efficient symbolic range analysis.
- (OOPSLA'14) Its application to the array out-of-bound problem.
- (CG0'16) Its use in pointer disambiguisation.

Plan

Motivations and big picture

Technical context: LLVM

Symbolic range analysis

Application 1: GreenArrays toolsuite
- Overview
- A bit on the other analyses
- Experimental results
- Conclusion

Application 2: Symbolic Pointer Analysis
- Overview
- Experimental results

Conclusion

A bit on LLVM

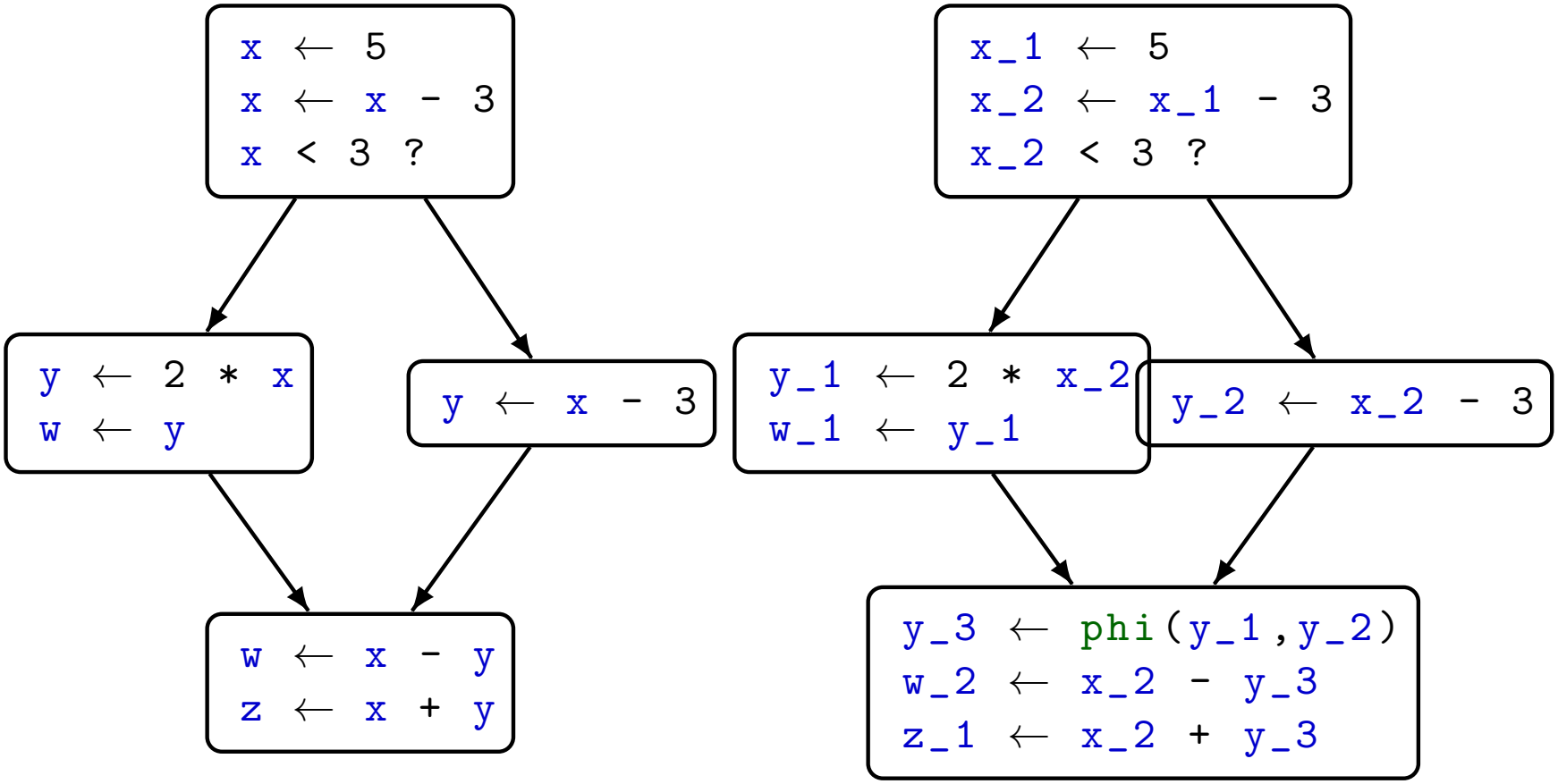
LLVM is a **compiler infrastructure**:

- Open source
- Various frontends (C, C++, Fortran)
- Various code generators (x86, ...)

Writing optimisations is easier :

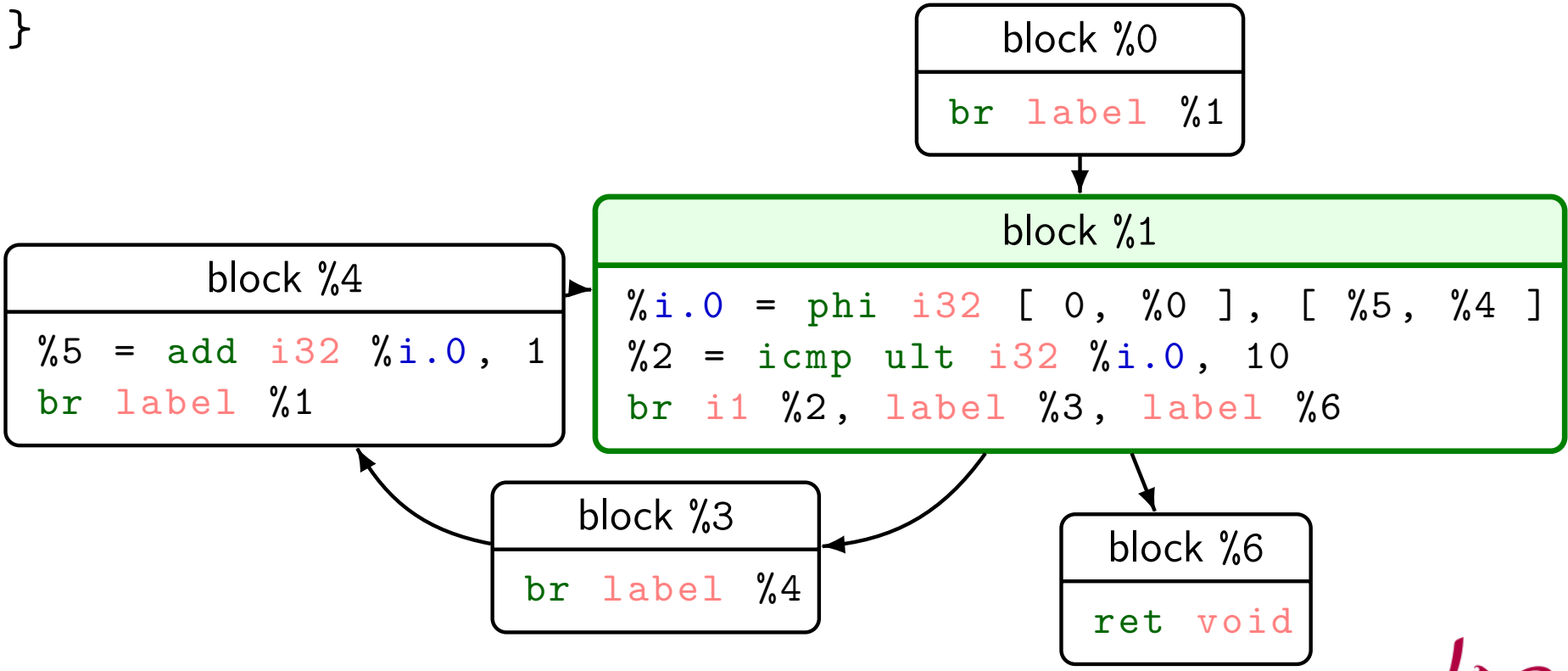
- A unique IR (**intermediate representation**)
- C++ iterators (functions, blocks, ...)

LLVM representation : SSA form



LLVM representation: a toy example

```
void simple_loop_constant() {
  for(unsigned i=0; i<10; i++) {
    // Do nothing
  }
}
```



Credits G. Radanne

Plan

Motivations and big picture

Technical context: LLVM

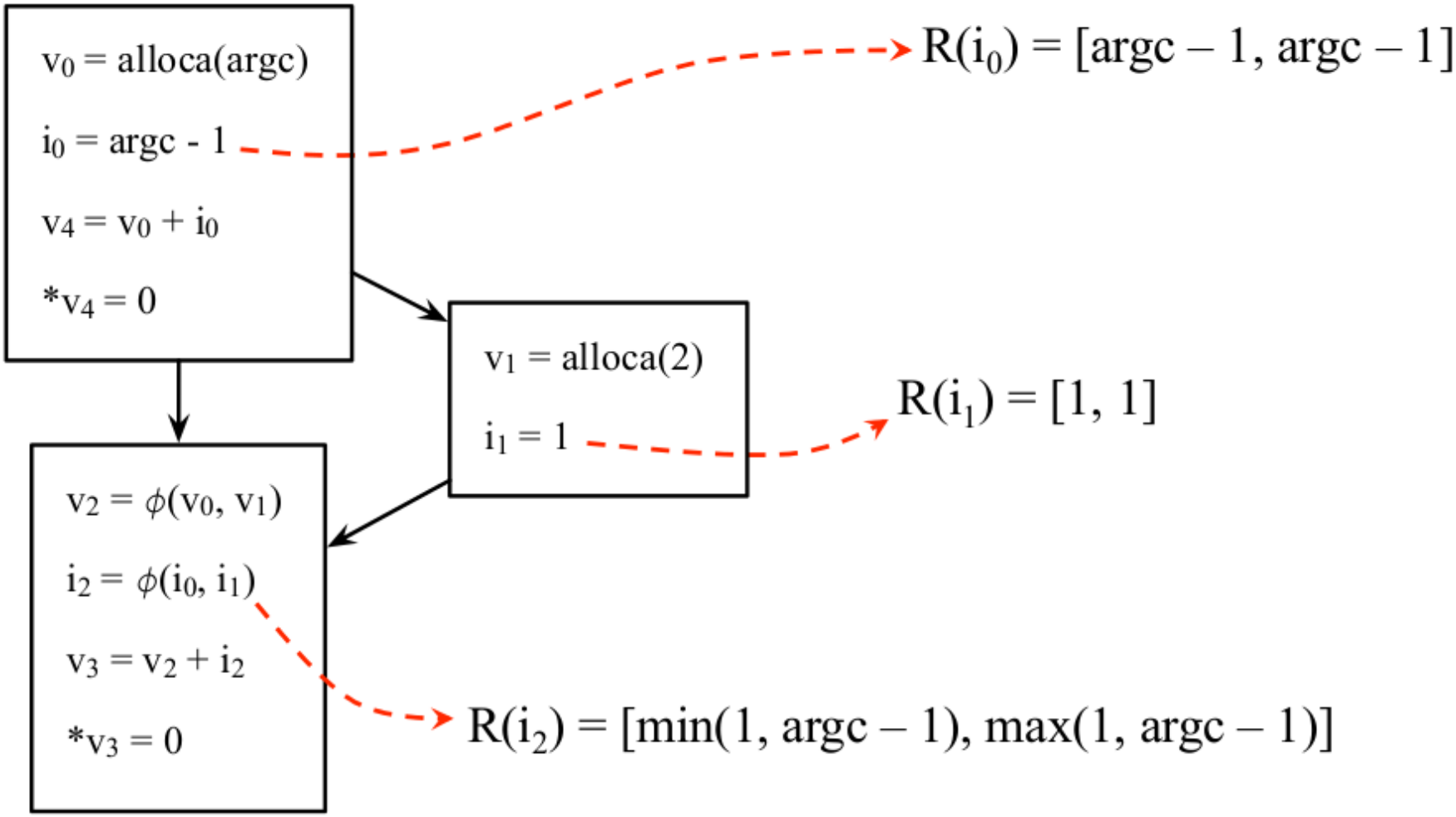
Symbolic range analysis

Application 1: GreenArrays toolsuite
- Overview
- A bit on the other analyses
- Experimental results
- Conclusion

Application 2: Symbolic Pointer Analysis
- Overview
- Experimental results

Conclusion

Symbolic Ranges (SRA): Goal



SRA on SSA form: a sparse analysis

- An abtract interpretation-based technique.
- Very similar to classic range analysis.
- One abstract value (R) **per variable**: sparsity.

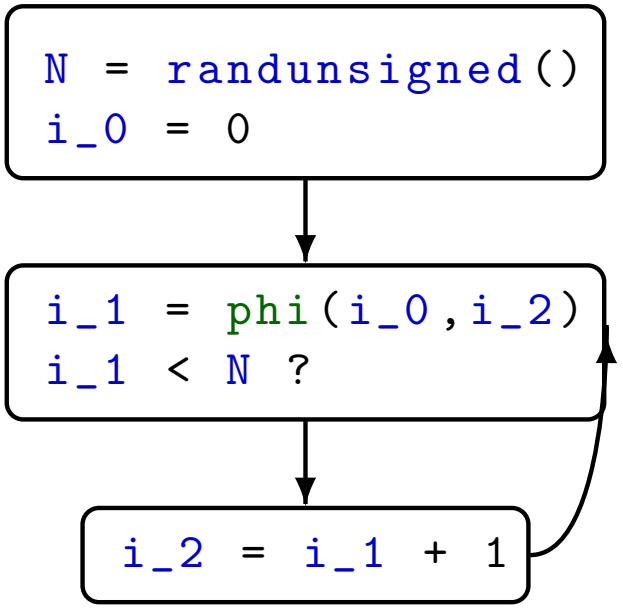
- Easy to implement (simple algorithm, simple data structure).

SRA on SSA form: constraint system

$$
\begin{aligned}
v = \bullet \quad &\Rightarrow \quad R(v) = [v, v] \\
v = o \quad &\Rightarrow \quad R(v) = R(o) \\
v = v_1 \oplus v_2 \quad &\Rightarrow \quad R(v) = R(v_1) \oplus^I R(v_2) \\
v = \phi(v_1, v_2) \quad &\Rightarrow \quad R(v) = R(v_1) \sqcup R(v_2) \\
\text{other instructions} \quad &\Rightarrow \quad \emptyset
\end{aligned}
$$

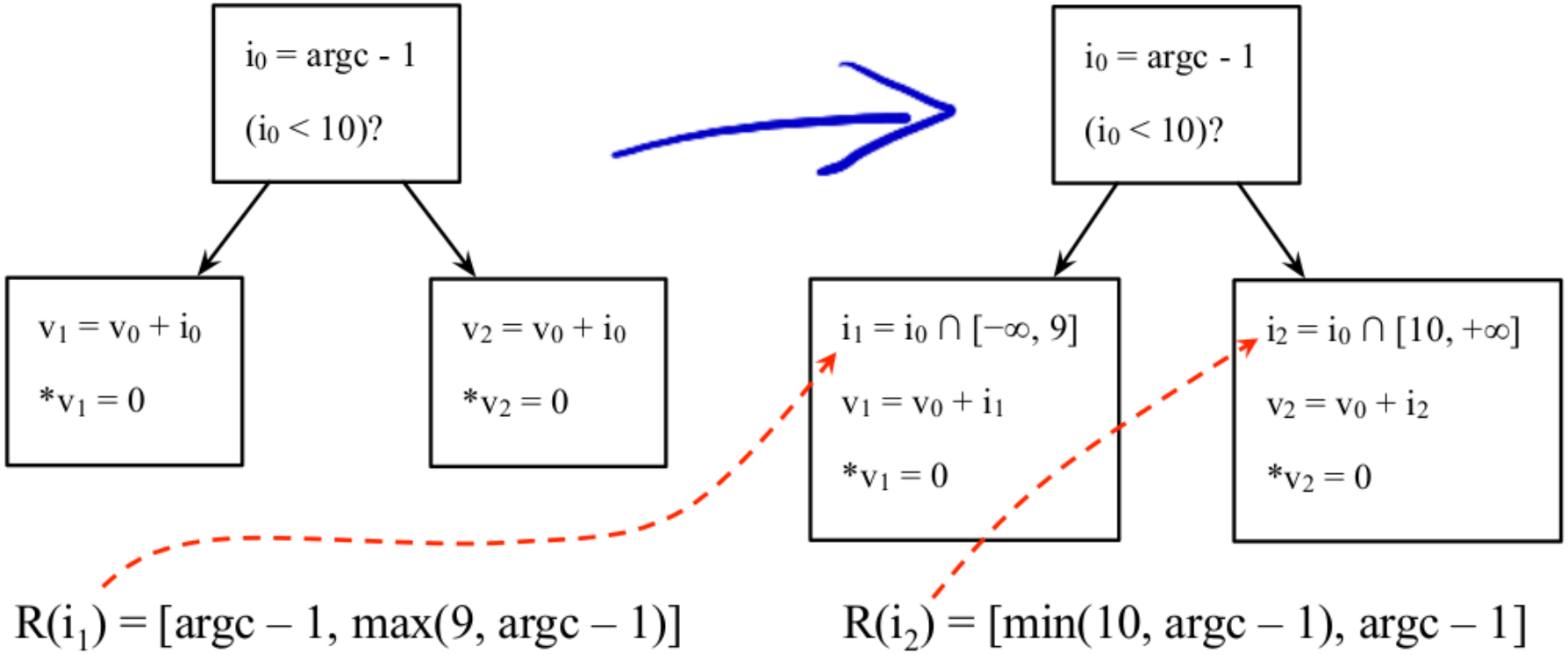
$\oplus^I$: abstract effect of the operation $\oplus$ on two intervals.
$\sqcup$: convex hull of two intervals. ▶ All these operation are performed symbolically thanks to **GiNaC**

SRA on SSA form: an example



- $R(i_0) = [0, 0]$
- $R(i_1) = [0, +\infty]$
- $R(i_2) = [1, +\infty]$

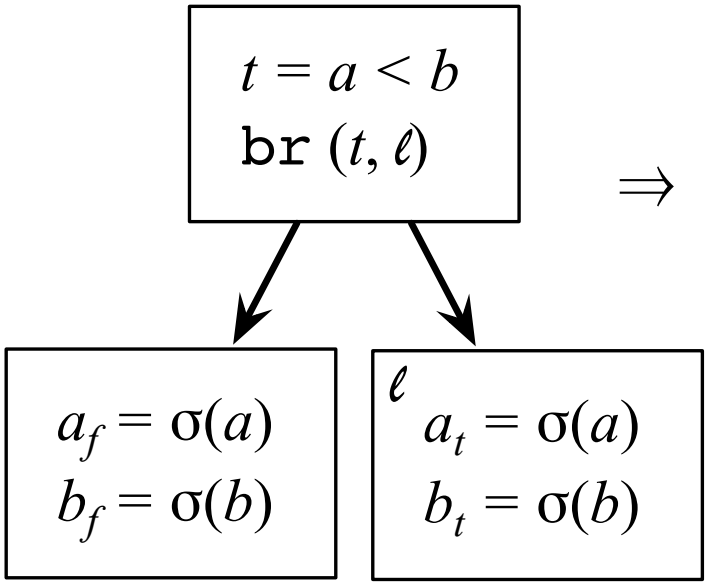
Improving precision of SRA : live-range splitting 1/2



- e-SSA form.

Improving precision of SRA : live-range splitting 2/2

Rule for live-range splitting :

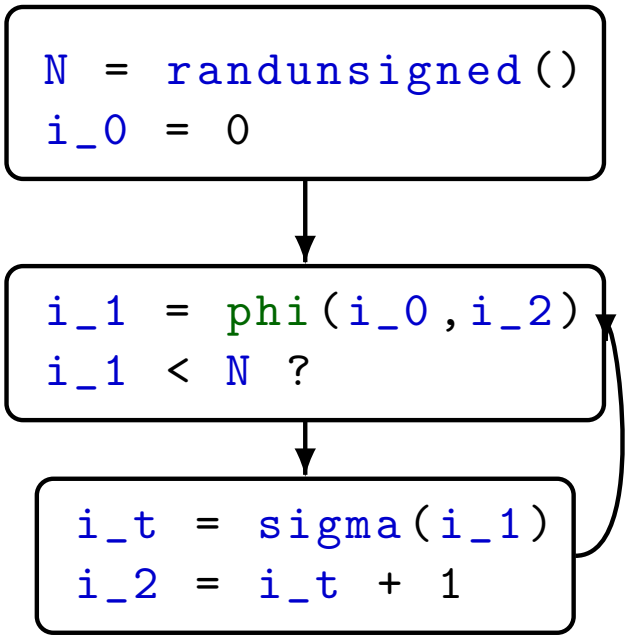


$\Rightarrow$

$$R(a_t) = [R(a)_\downarrow, \min(R(b)_\uparrow - 1, R(a)_\uparrow)]$$
$$R(b_t) = [\max(R(a)_\downarrow + 1, R(a)_\downarrow), R(b)_\uparrow]$$
$$R(a_f) = [\max(R(a)_\downarrow, R(a)_\uparrow), R(a)_\uparrow]$$
$$R(b_t) = [R(b)_\downarrow, \min(R(a)_\uparrow, R(b)_\uparrow)]$$

- All simplications are done by GiNaC.

SRA + live-range on an example



$$R(i_t) = [R(i_1)\downarrow, min(N-1, R(i_1)\uparrow)]$$

- $R(i_0) = [0, 0]$
- $R(i_1) = [0, N]$

Plan

Motivations and big picture

Technical context: LLVM

Symbolic range analysis

Application 1: GreenArrays toolsuite
- Overview
- A bit on the other analyses
- Experimental results
- Conclusion

Application 2: Symbolic Pointer Analysis
- Overview
- Experimental results

Conclusion

Contributions (OOPSLA'14)

- A technique to prove that (some) memory accesses are safe :
  - Less need for additional guards.
  - Based on abstract interpretation.
  - Precision and cost compromise.
- Implemented in LLVM-compiler infrastructure :
  - Eliminate 50% of the guards inserted by AddressSanitizer
  - SPEC CPU 2006 17% faster

Our key insight : Symbolic (parametric) Analyses

```
int main(int argc, char** a) {
  char* p = malloc(argc);
  int i = 0;
  while (i < argc) {
    p[i] = 0;
    i++;
  }
  return 0;
}
```

$W(p) = [0, \texttt{argc} - 1]$.

$R(i) = [0, \texttt{argc} - 1]$

- $R(i) \subseteq W(p)$ thus $p[i]$ is **safe**.

A bit on sanitizing memory accesses

Different techniques : but all have an overhead.

Ex : Address Sanitizer

- Shadow every memory allocated : 1 byte $\rightarrow$ 1 bit (allocated or not).
- Guard every array access : check if its shadow bit is valid.
  - slows down SPEC CPU 2006 by 25%

- We want to **remove these guards**.

Green Arrays : overview 1/2

```
1.  int main(int argc, char** argv) {
2.    int size = argc + 1;
3.    char* buf = malloc(size);
4.    unsigned index = 0;
5.    scanf("%u", &index);
6.    if (index < argc) {
7.      buf[index] = 0;
8.    }
9.    return index;
10. }
```

Any address from buf + 0 to buf + argc is safe!

Inside the branch index is at least 0 and at most argc-1

We know that "argc - 1" is less than argc

As long as we do not have integer overflows!

Green Arrays : overview 2/2

Symbolic Range Analysis: finds the lower and upper values that variables can assume

Symbolic Region Analysis: finds the lower and upper values that a pointer can address

Integer Overflow Analysis: Which arithmetic operations can overflow?

Any address from buf + 0 to buf + argc is safe!

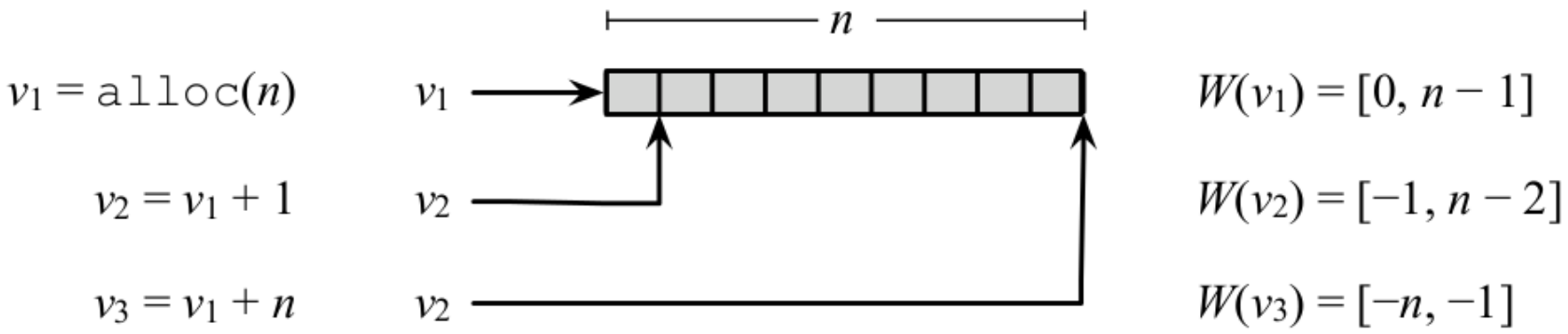
Inside the branch index is at least 0 and at most argc-1

We know that "argc - 1" is less than argc

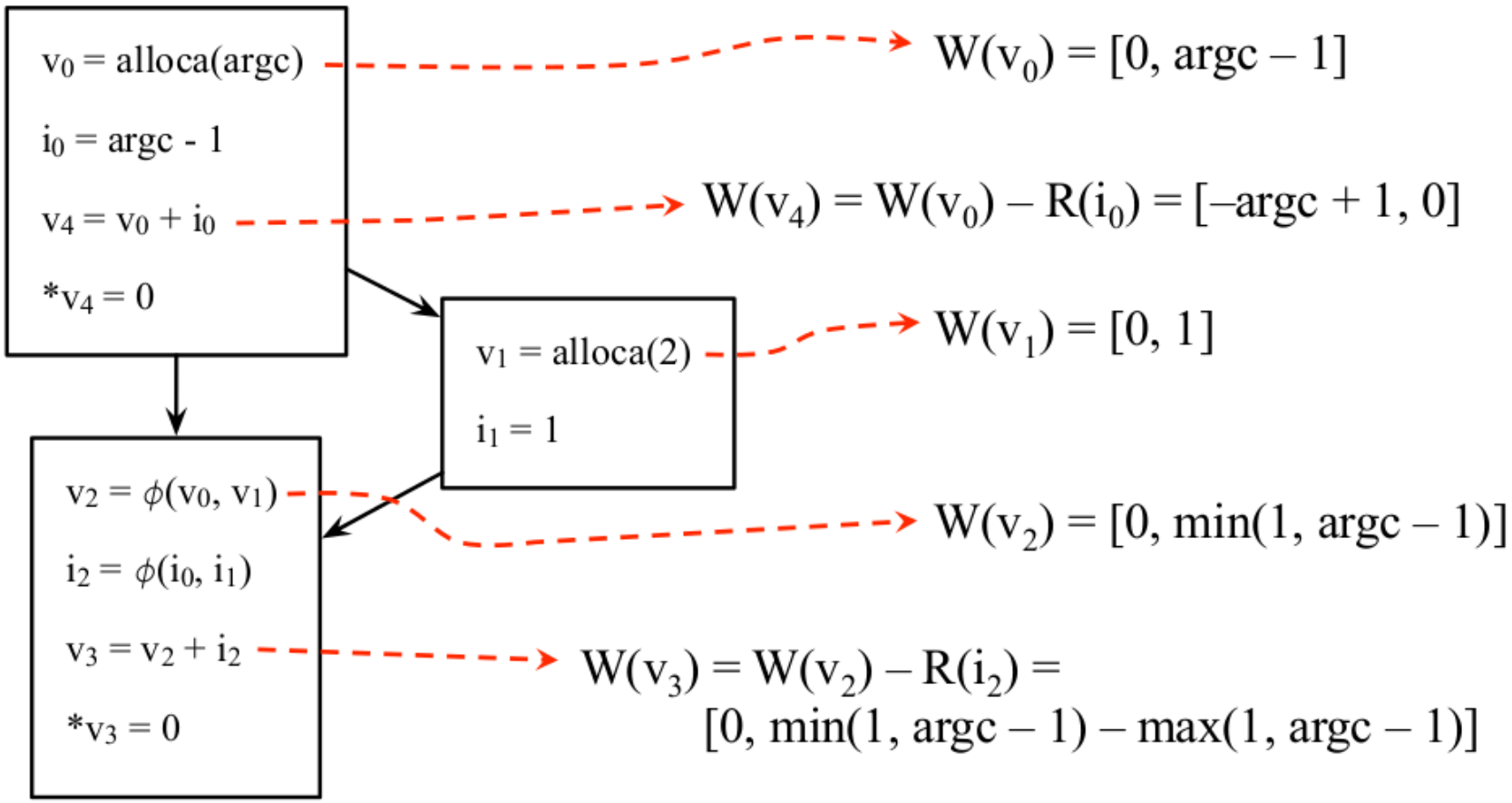
As long as we do not have integer overflows!

Symbolic regions 1/2: Goal

Compute (an underapproximation of) the range of **valid accesses** from base pointers:

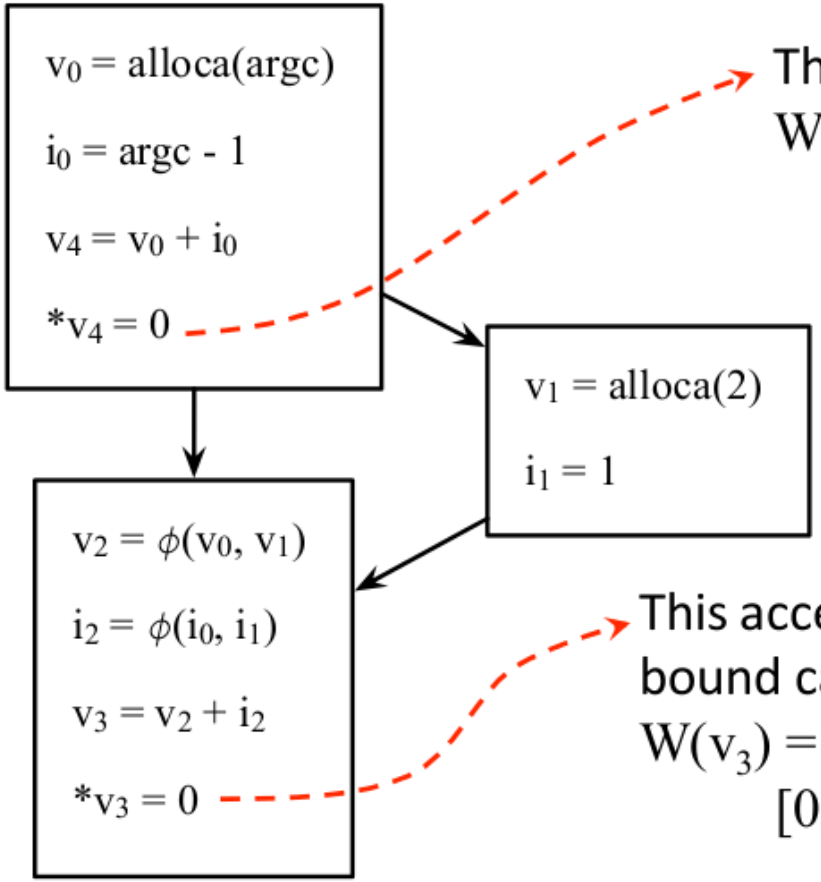


Symbolic regions 2/2: An example



Safety: result

If $0 \in W(p)$, then $*p$ is **safe**, else **DK**



This access is **not** safe (if argc == 0):
$W(v_4) = W(v_0) - R(i_0) = [-argc + 1, 0]$

This access is **not** safe, because its upper bound can be negative:
$W(v_3) = W(v_2) - R(i_2) =$
$[0, \min(1, argc - 1) - \max(1, argc - 1)]$

Overflows 1/2

```
int main(int argc, char** argv) {
  int index = argc + 1;
  int size = index * index;
  char* buf = malloc(size);
  return buf[index];
}
```

index * index may wrap around.

Do you know what malloc will return?

Because we manipulate symbols, "argc + 1 < (argc + 1) * (argc + 1)" only in the absence of integer overflows

Overflows 2/2

- We find every arithmetic operation that may influence memory **allocation** or memory **indexing**.

```
int main(int argc, char** argv) {
  int index = argc + 1;
  int size = index * index;
  char* buf = malloc(size);
  return buf[index];
}
```

We find them via program slicing

- We **instrument the code** to detect overflows.

Experimental setup

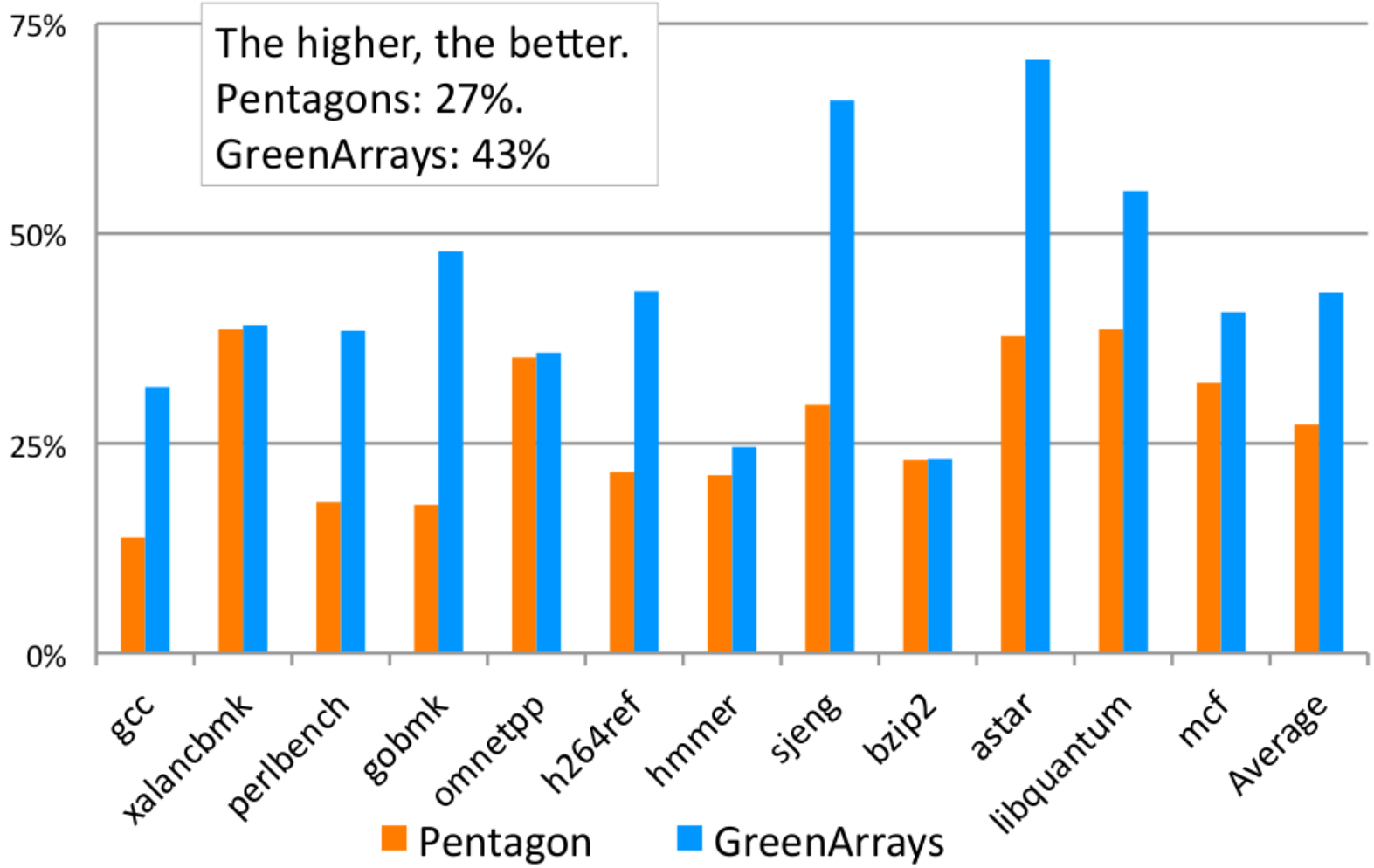
- **Implementation**: LLVM + AddressSanitizer
- **Benchmarks**: SPEC CPU 2006 + LLVM test suite
- **Machine**: Intel(R) Xeon(R) 2.00GHz,with 15,360KB of cache and 16GB or RAM
- **Baseline**: Pentagons
  - Abstract interpretation that combines "less-than" and "integer ranges".†

```
int i = 0;
unsigned j = read();
if (...)
    i = 9;
if (j < i)
  ...
```

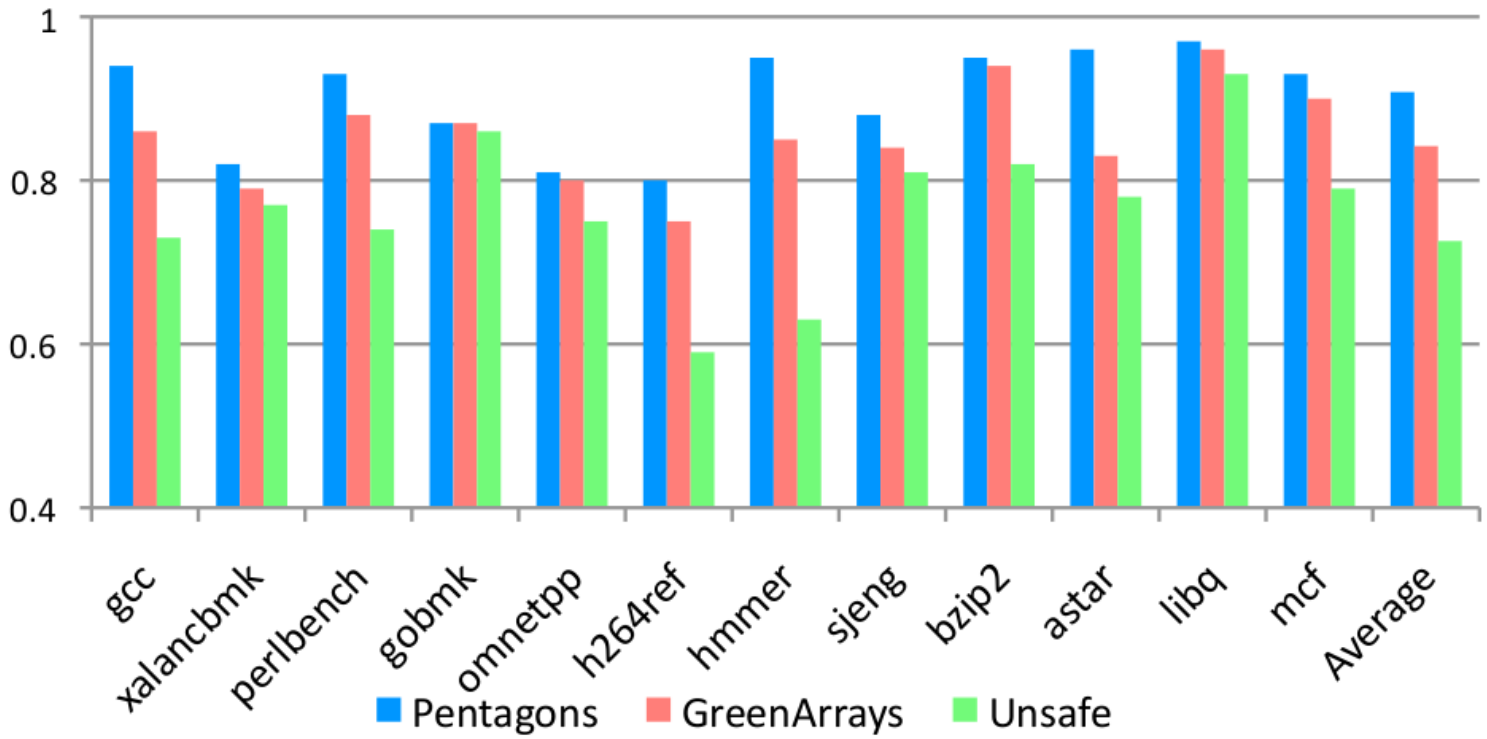
P(j) = (less than {i}, [0, 8])

†: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses, 2010, Science of Computer Programming

Percentage of bound checks removed



Runtime improvement



The lower the bar, the faster. Time is normalized to AddressSanitizer without bound-check elimination. Average speedup: Pentagons = 9%. GreenArrays = 16%.

In the paper (OOPSLA'14)

A complete formalisation of all the analyses :

- Concrete and abstract semantics.
- Safety is proved.
- Interprocedural analysis.

- `https://code.google.com/p/ecosoc/`

Remaining question : improving precision of the symbolic range analysis ?

Plan

Motivations and big picture

Technical context: LLVM

Symbolic range analysis

Application 1: GreenArrays toolsuite
- Overview
- A bit on the other analyses
- Experimental results
- Conclusion

Application 2: Symbolic Pointer Analysis
- Overview
- Experimental results

Conclusion

Goal + Contribution (CGO '16)

Goal:

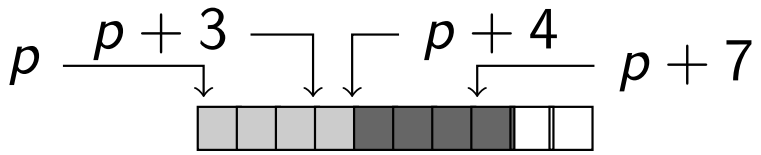
- Optimizing languages with pointers;
- Solving pointer arithmetic, disambiguating pointers;
- Low cost analysis.

Contribution:

- Combine alias analysis with range analysis;
- Speed up;

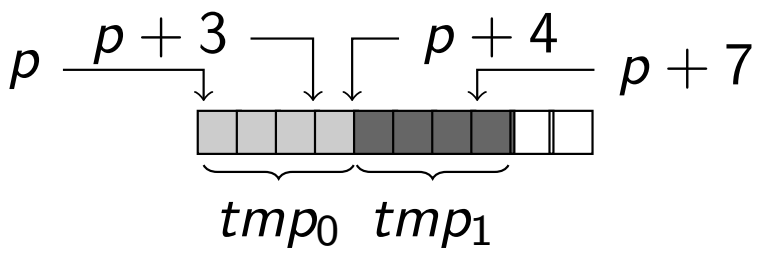
Motivating example

```
void fill_array (char* p) {
    int i ;
    for (i = 0; i < 4; i++)
        p[i] = 0 ;
    for (i = 4; i < 8; i++)
        p[i] = 2 × i ;
}
```

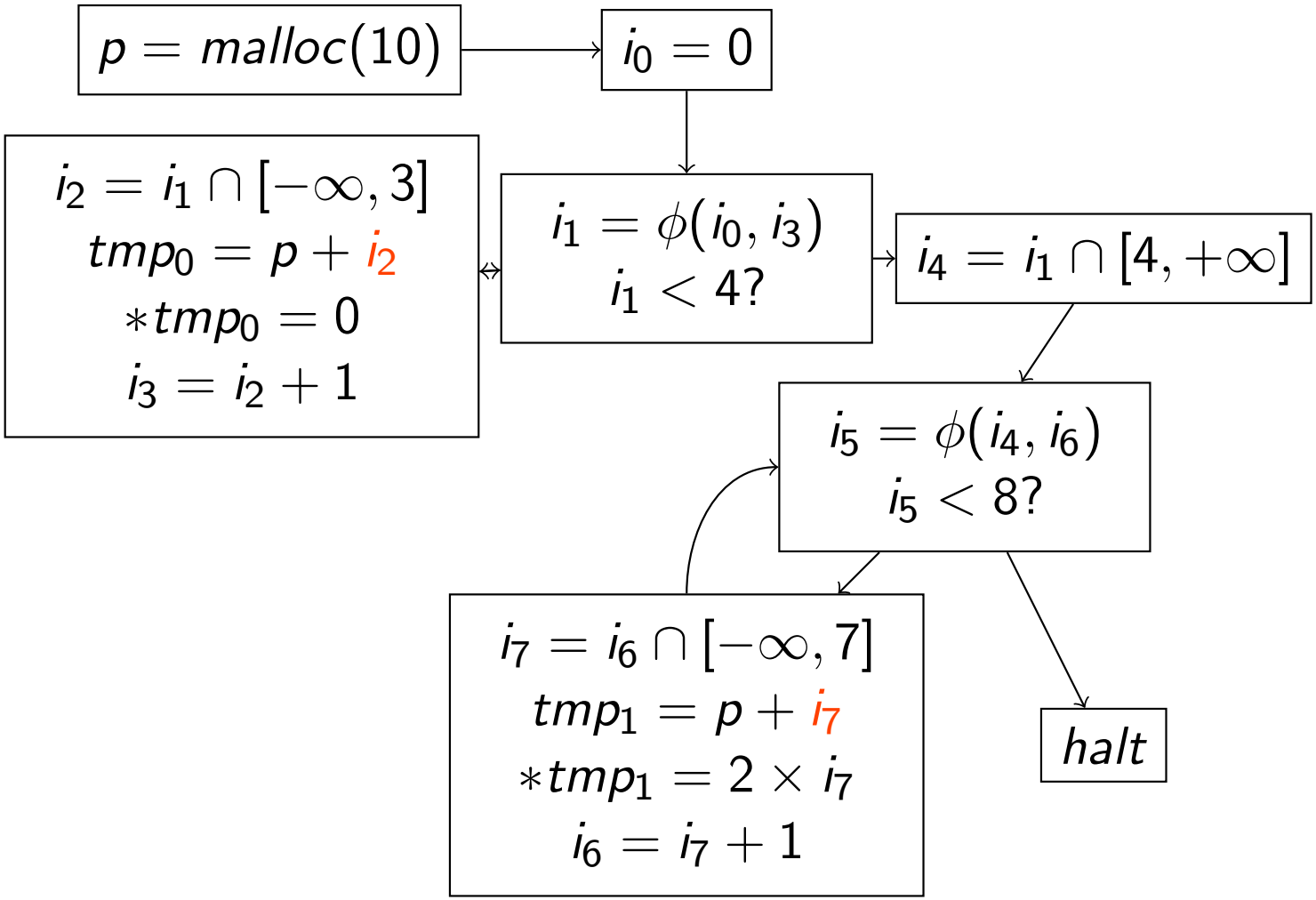


Motivating example

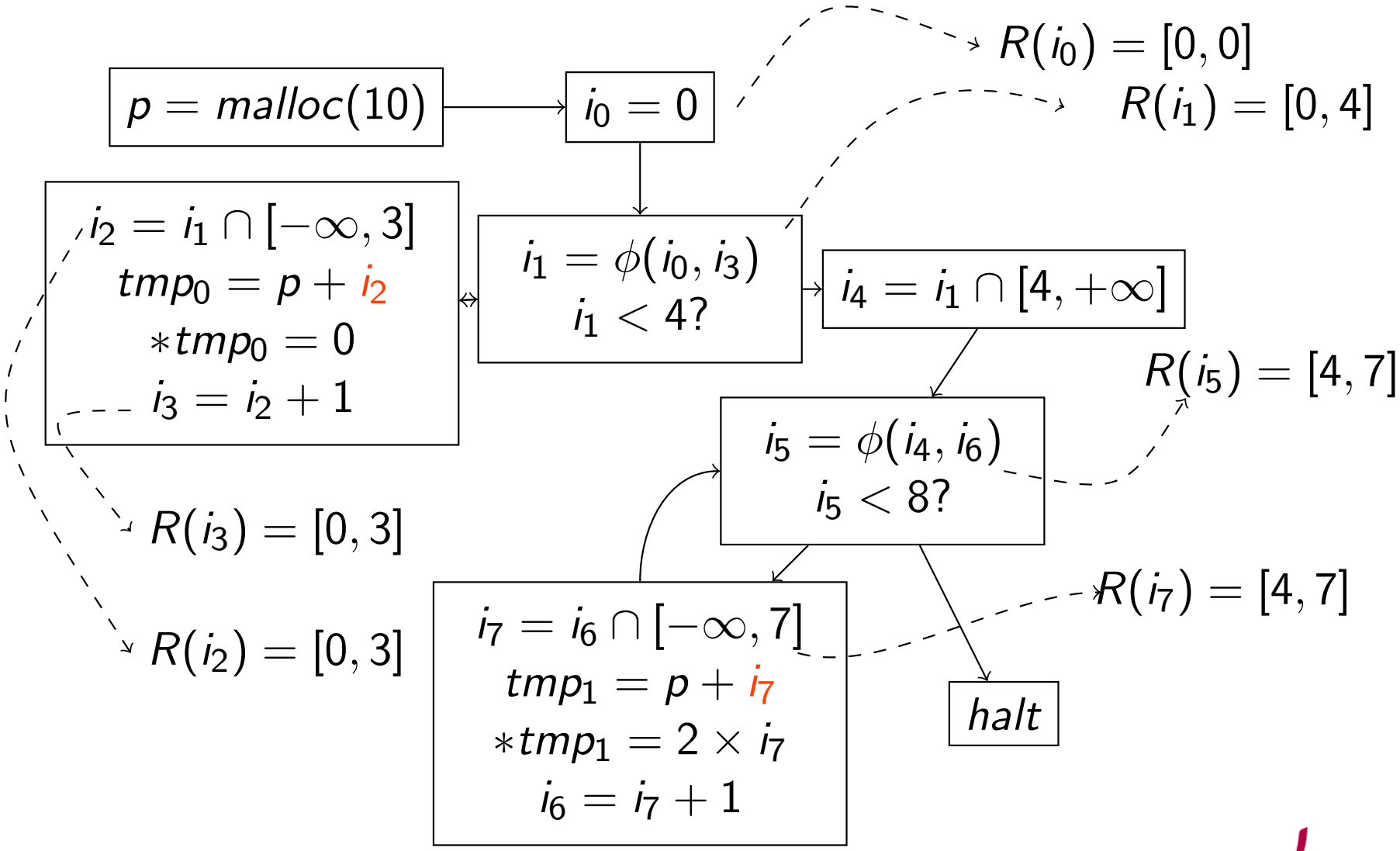
```
void fill_array (char* p) {
    int i ;
    for (i = 0; i < 4; i++)
        p[i] = 0 ; // char* tmp0 = p + i ; *tmp0 = 0
    for (i = 4; i < 8; i++)
        p[i] = 2 × i ;// char* tmp1 = p + i ; *tmp0 = 2 × i
}
```



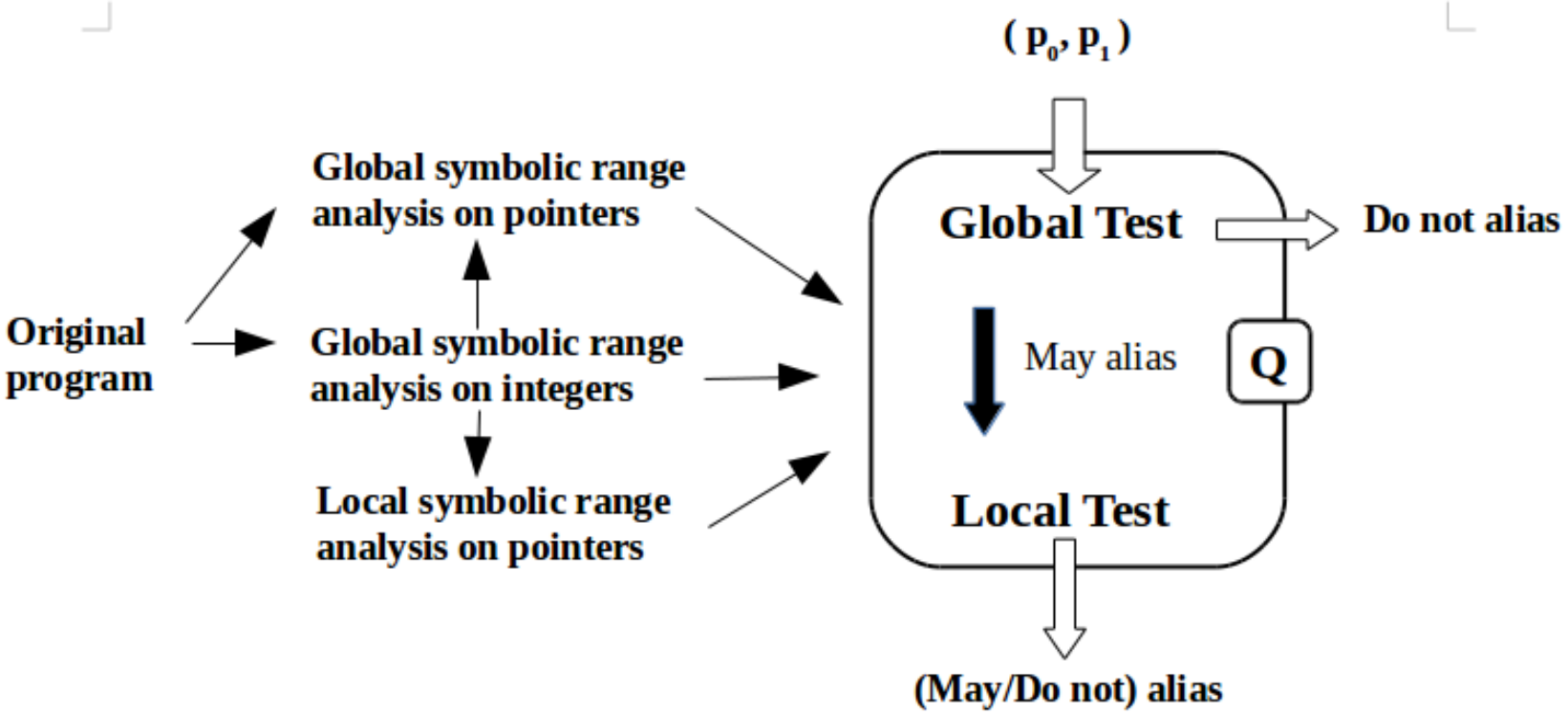
First step: Range Analysis on e-SSA



First step: Range Analysis on e-SSA



Pointer Range Analysis: Overview



Abstract Analysis

$$p_i \rightsquigarrow loc_i + [l_i, u_i] \qquad \text{i} = \text{j or i} \neq \text{ i}$$
$$p_j \rightsquigarrow loc_j + [l_j, u_j]$$

Property (overapprox):
If $(loc_i = loc_j$ and $[l_i, u_i] \sqcap [l_j, u_j] = \emptyset)$
Then p_i and p_j **do not alias** Else **may alias** .

Global Pointer Range Analysis

Let n be the number of program sites where memory is allocated. We associate pointers with tuples of size n: (SymbRanges $\uplus \perp)^n$: GR(p) $= (p_0, \ldots, p_{n-1})$.

Notation

- GR(p) $= \{loc_i + p_i, loc_j + p_j, \ldots\}$

Constraint System:

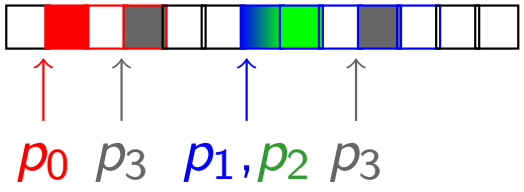
$$j : p = \text{ malloc } (v) \quad \Rightarrow \quad GR(p) = (\perp, \ldots, [0,0]_j, \ldots)$$

$$v = v_1 \quad \Rightarrow \quad GR(v) = GR(v_1)$$

$$\begin{array}{r} q = p + c \\ \text{with } c \text{ scalar} \end{array} \quad \Rightarrow \quad q_i = \begin{cases} \perp \text{ if } p_i = \perp \\ p_i + R(c) \text{ else} \end{cases}$$

$$q = \phi(p^1, p^2) \quad \Rightarrow \quad \text{GR}(q) = \text{GR}(p^1) \sqcup \text{GR}(p^2)$$

Global Pointer Range Analysis



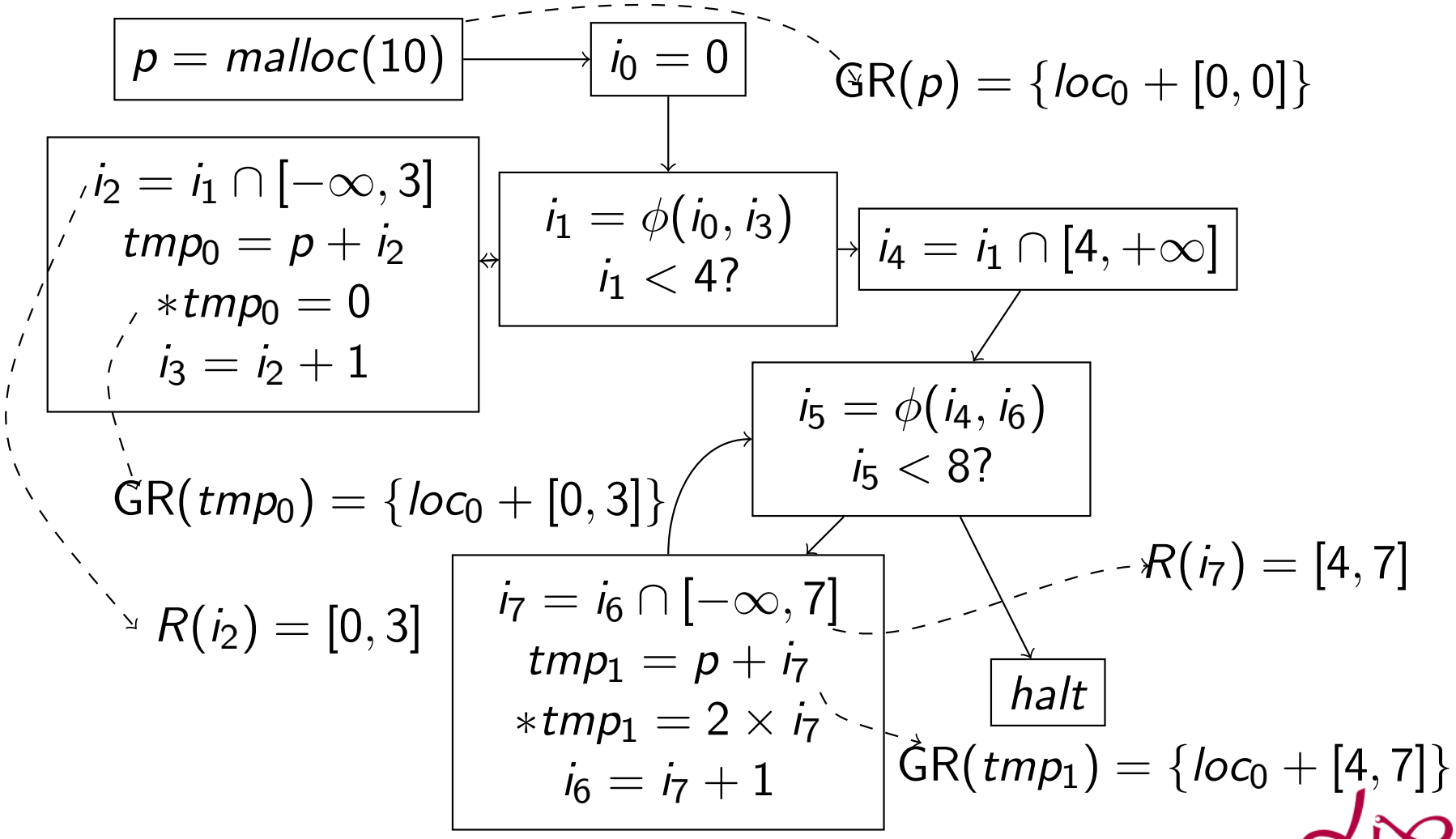
$GR\ (p_0) = loc_0 + [0,\ 0]$
$GR\ (p_1) = loc_1 + [0,\ 0]$
$GR\ (p_2) = loc_1 + [0,\ 1]$
$GR\ (p_3) = \{loc_0 + [2,\ 2],\ loc_1 + [3,\ 3]\}$

$p_0 = malloc(3)$;
$p_1 = malloc(5)$;

if (...) $p_2 = p_1$;
else $p_2 = p_1 + 1$;
if (...) $p_3 = p_0 + 2$;
else $p_3 = p_1 + 3$

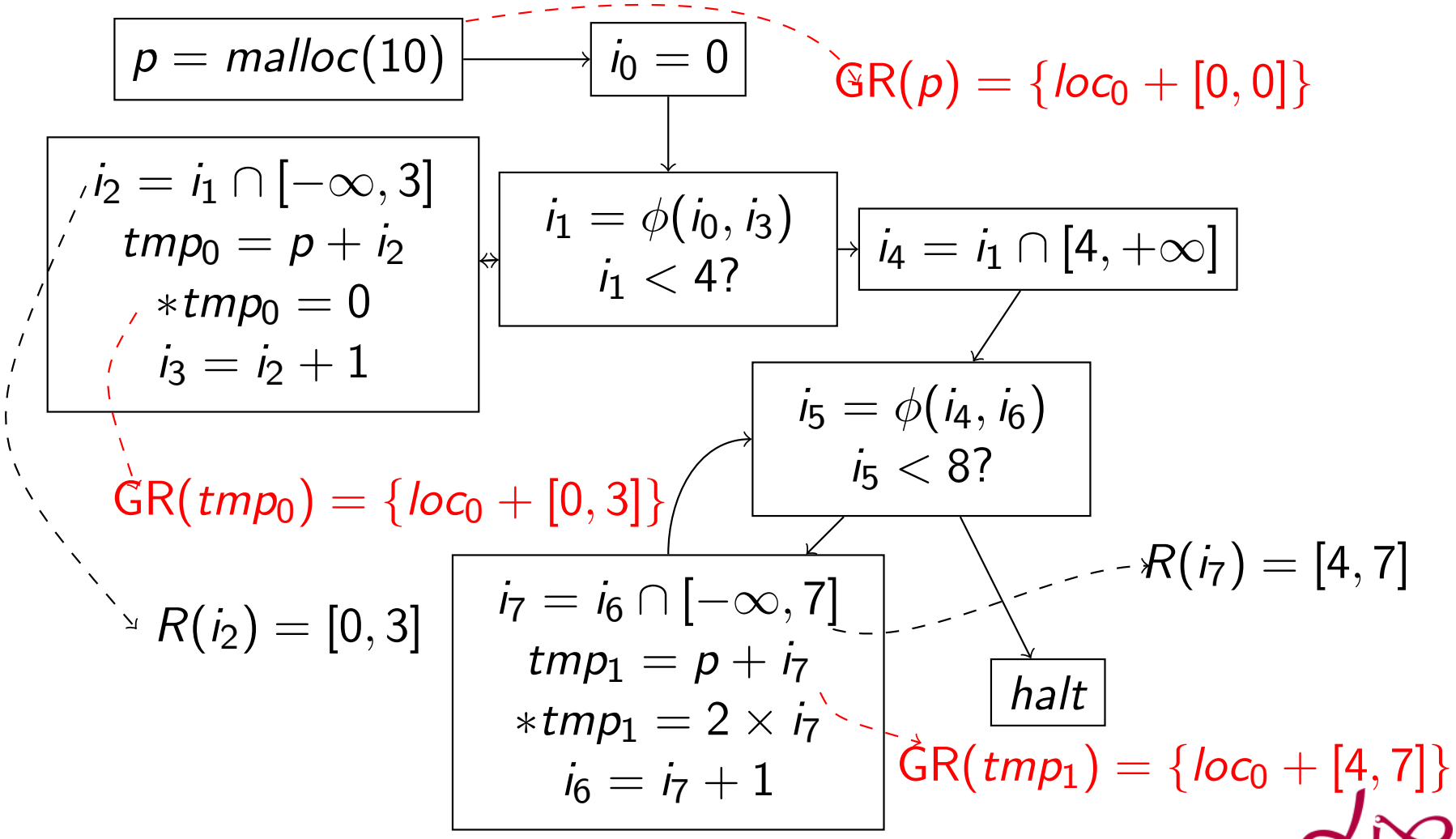
Global Pointer Range Analysis

Example:



Global Pointer Range Analysis

Example:



Local Pointer Range Analysis

Motivation:

$a_0 = \text{malloc}\ (N)\ ; \rightsquigarrow\ \ GR\ (a_0) = \{loc_0 + [0,\ 0]\}$

if (...)

$\quad a_1 = a_0 + 1\ ; \rightsquigarrow\ \ GR\ (a_1) = \{loc_0 + [1,\ 1]\}$

else $a_2 = a_0\ ; \rightsquigarrow\ \ GR\ (a_2) = \{loc_0 + [0,\ 0]\}$

$a_3 = \phi(a_1, a_2); \rightsquigarrow\ \ GR\ (a_3) = \{loc_0 + [0,\ 1]\}$

$a_4 = a_3 + 1\ ; \rightsquigarrow\ \ GR\ (a_4) = \{loc_0 + [1,\ 2]\}$

$a_5 = a_3 + 2\ ; \rightsquigarrow\ \ GR\ (a_5) = \{loc_0 + [2,\ 3]\}$

$[1,\ 2] \cap [2,\ 3] \neq \emptyset$

Local Pointer Range Analysis

Motivation:

$a_0 = \text{malloc}\ (N)\ ; \rightsquigarrow\ GR\ (a_0) = \{loc_0 + [0,\ 0]\}$

if (...)

$a_1 = a_0 + 1\ ; \rightsquigarrow\ GR\ (a_1) = \{loc_0 + [1,\ 1]\}$

else $a_2 = a_0\ ; \rightsquigarrow\ GR\ (a_2) = \{loc_0 + [0,\ 0]\}$

$a_3 = \phi(a_1, a_2); \rightsquigarrow\ GR\ (a_3) = \{loc_0 + [0,\ 1]\}\ \rightsquigarrow\ LR\ (a_3) = \{loc_1 + [0,0]\}$

$a_4 = a_3 + 1\ ; \rightsquigarrow\ GR\ (a_4) = \{loc_0 + [1,\ 2]\}\ \rightsquigarrow\ LR\ (a_4) = \{loc_1 + [1,\ 1]\}$

$a_5 = a_3 + 2\ ; \rightsquigarrow\ GR\ (a_5) = \{loc_0 + [2,\ 3]\}\ \rightsquigarrow\ LR\ (a_5) = \{loc_1 + [2,\ 2]\}$

$$[1,\ 2] \cap [2,\ 3] \neq \emptyset$$

Local Pointer Range Analysis

Constraint System:

$$\begin{array}{r} p = \text{ malloc } (v) \\ \text{with } v \text{ scalar} \end{array} \Rightarrow \quad \mathrm{LR}(p) = \mathcal{N}ewLocs() + [0,0]$$

$$\begin{array}{r} j : q = \phi(p_1, p_2) \\ \text{with } loc_j = \mathcal{N}ewLocs() \end{array} \Rightarrow \quad \mathrm{LR}(q) = loc_j + [0,0]$$

Experimental setup

- Implementation : LLVM 3.5
- Benchmarks : LLVM test suite + Micro benchmarks + PtrDist + Prolangs + MallocBench
- Machine : Intel i7-4770K, 8GB of memory Ubuntu 14.04.2

Experimental results

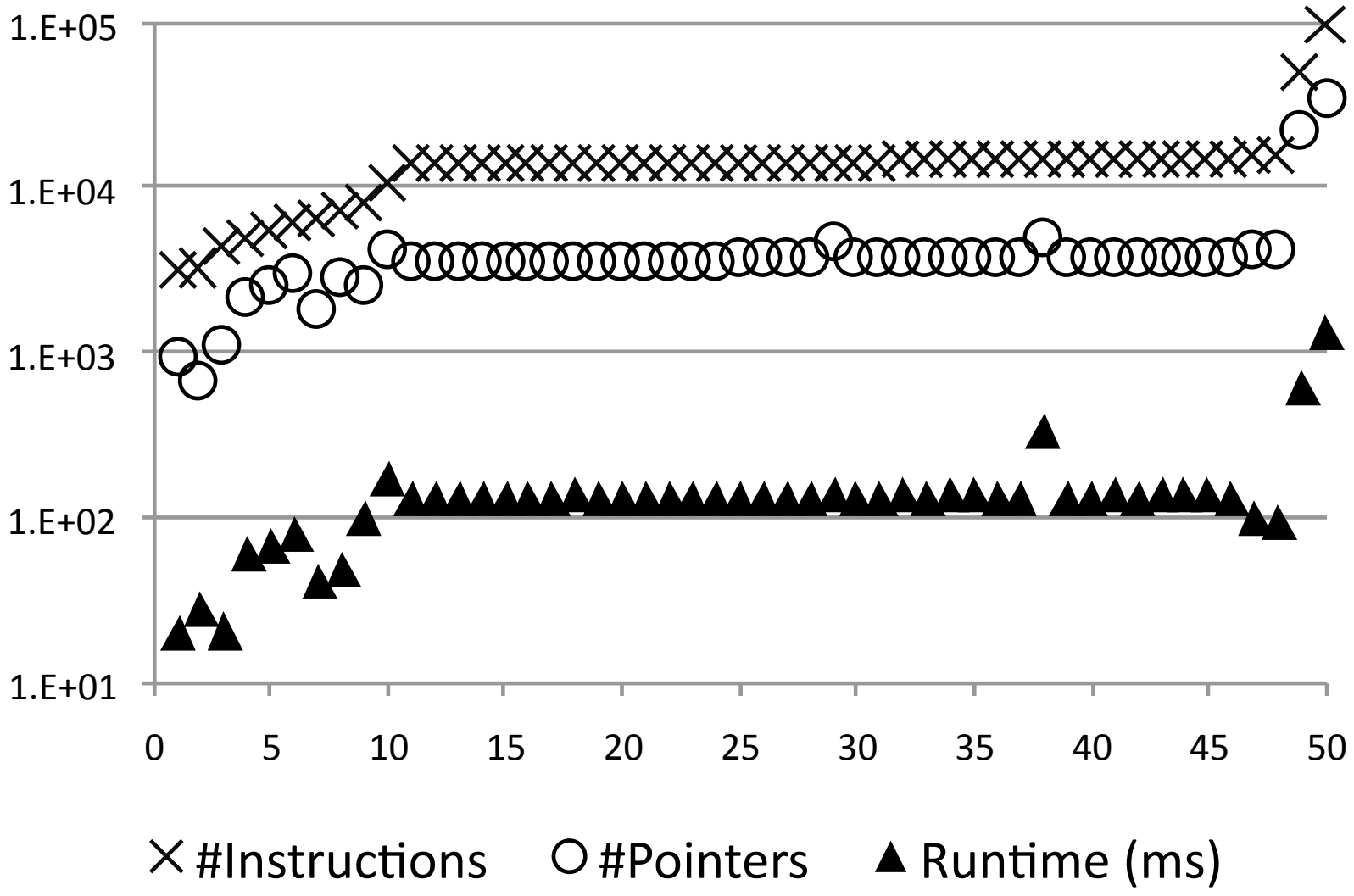
	#Queries	scev	basic	rbaa	rbaa + basic
Total %	7,243,418	2.8	17.8	16.9	22.1

- #Queries : number of pair of pointers.
- scev : scalar evolution based alias-analysis.
- basic : -O3 LLVM analysis (global + local pointers).
- rbaa : range based alias analysis.

Answering queries: number of pairs that do not alias.

- The analysis is precise but still a room for improvement.

Experimental results



- The analysis scales well!

Work in progress (with M. Maalej)

- Submitted to CGO'17: an adaptation of Pentagons for pointer disambiguisation. (less-than constraints).
- A combination of 3 different analyses: journal paper in preparation.

Plan

Motivations and big picture

Technical context: LLVM

Symbolic range analysis

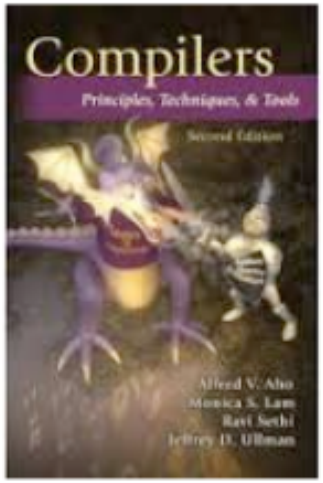
Application 1: GreenArrays toolsuite
- Overview
- A bit on the other analyses
- Experimental results
- Conclusion

Application 2: Symbolic Pointer Analysis
- Overview
- Experimental results

Conclusion

A Bit of Perspective

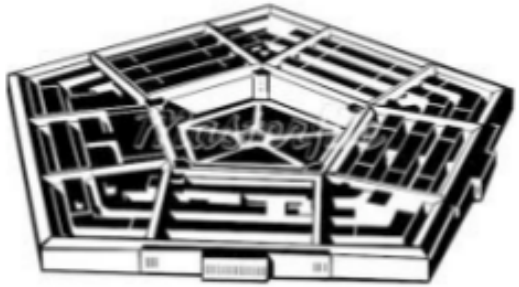
Chapter 9



Precision

- Non-Relational Analyses:
  - They associate variables with information that do not depend on other variables. **Examples**: the usual data flow analyses.
- Semi-Relational analyses:
  - They associate variables with information that may contain other variable names. **Examples**: pentagons, **symbolic range analysis, symbolic region analysis**.
- Relational analyses:
  - They associate sets of variables with information. **Examples**: octagons and polyhedrons.

Speed



Take home message

- Code optimisation are good applications for static analyses!
- They have to be thought in terms of scaling as well as precision.

- Sparse analyses are the key but they still have to be invented/redesigned.