Abstract interpretation for compilers

A journey from theory to practice back to . . .

Laure Gonnord,
`http://laure.gonnord.org/pro`

PLISS 2022

Intro

Recall memories

Question

How to design *static analyses* that are **correct by construction** ?

- From dataflow analyses to abstract interpretation (course 1)

Question

How to design *static analyses* that **scale enough** to be embedded inside compilers ?

- From abstract interpretation to sparse abstract interpretation (course 2)

Questions here

Question

What are sources of (eventual) complexities ?

- complexity of abstract domains, complexity of fixpoint computation

Question

How to scale ?

- specialised tailored abstract domains/intermediate representation.

A tour of some relational abstract domains

When intervals are not sufficient

```
assume(x >= 0 && x <= 1);
y = x;
z = x-y;
```

- The human (smart) sees $z = 0$ thus interval $[0, 0]$, taking into account $y = x$.
- Interval arithmetic does not see $z = 0$ because it does not take $y = x$ into account.

How to track relations

Using **relational domains**.

E.g. : Difference bound matrices

- for each variable an interval
- for each pair of variables (x, y) an information $x - y \leq C$.
- (One obtains $x = y$ by $x - y \leq 0$ and $y - x \leq 0$.)

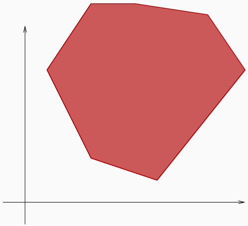
How to **compute** on that ? with difference bound matrices.

Can we do better (more expressive) ?

How about tracking relations such as $2x + 3y \leq 6$?

At a given program point, a set of **linear inequalities**.

In other words, a **convex polyhedron** (Linear Relation Analysis).



(also needs widening).

Complexity of individual operations

(In general) The more precise we are, the higher the costs.

- Intervals : algorithms $O(n)$, n number of variables.
- Differences $x - y \leq C$: algorithms $O(n^3)$
- Octagons $\pm x \pm y \leq C$ (Miné) : algorithms $O(n^3)$
- Polyhedra (Cousot / Halbwachs) : algorithms often $O(2^n)$.

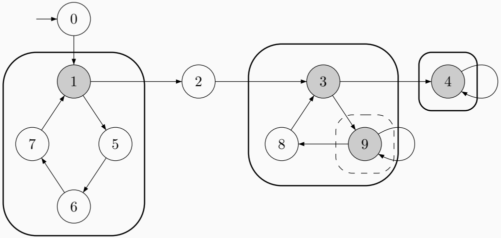
In compilers ?

Usually stricly less than n^2 algorithms.

Implementing chaotic iterations

“Concrete complexity” of the chaotic iterations can be improved :

- by using worklists
- by using “clever” iterations strategies (SCCs, for instance)
- by working on **abstract domain themselves**
- by working on **intermediate representations**



Sparse dataflow

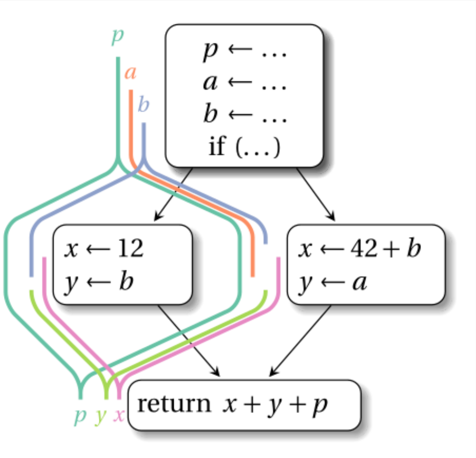
Liveness rephrased

Liveness is essential for many optimization, notably register allocation.

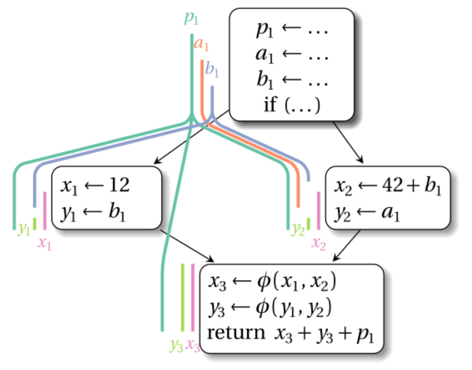
Alive Variable

In a given program point, a variable is said to be *alive* if the value it contains may be used in the rest of the execution.

Liveness : SSA to the rescue



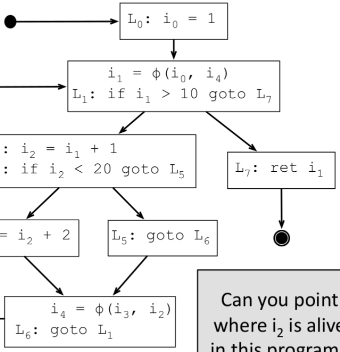
Live range on a CFG



Live range with SSA

Liveness on SSA

- The problem of determining the program points along which a variable is alive has a simple solution for SSA form programs.



```
L0: i0 = 1

    i1 = φ(i0, i4)
L1: if i1 > 10 goto L7

L2: i2 = i1 + 1
 3: if i2 < 20 goto L5

L4: i3 = i2 + 2

L5: goto L6

    i4 = φ(i3, i2)
L6: goto L1

L7: ret i1
```

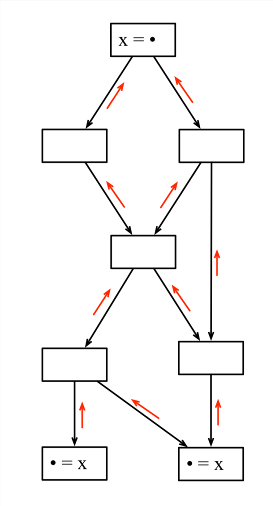
Can you point where i_2 is alive in this program?

```
For each statement S in the program:
    IN[S] = OUT[S] = {}

For each variable v in the program:
    For each statement S that uses v:
        live(S, v)

live(S, v):
    IN[S] = IN[S] ∪ {v}
    For each P in pred(S):
        OUT[P] = OUT[P] ∪ {v}
        if P does not define v
            live(P, v)
```

Liveness on SSA - 2



Correctness

Our algorithm works due to the key property of SSA form programs : every use of a variable v is **dominated** by the definition on of v. Thus, we can traverse the CFG of the program, start from the uses of a variable, until we stop at its definition.

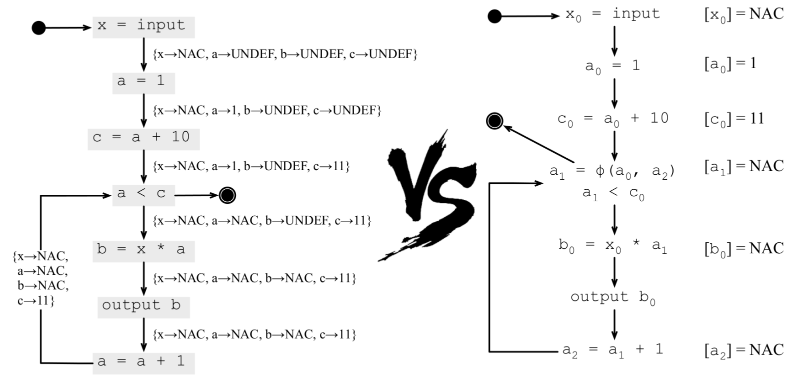
Other dataflow analyses fit well with SSA

- Constant propagation
- Tainted flow
- and much more . . .

▶ Let us see why it helps improving **actual complexity**

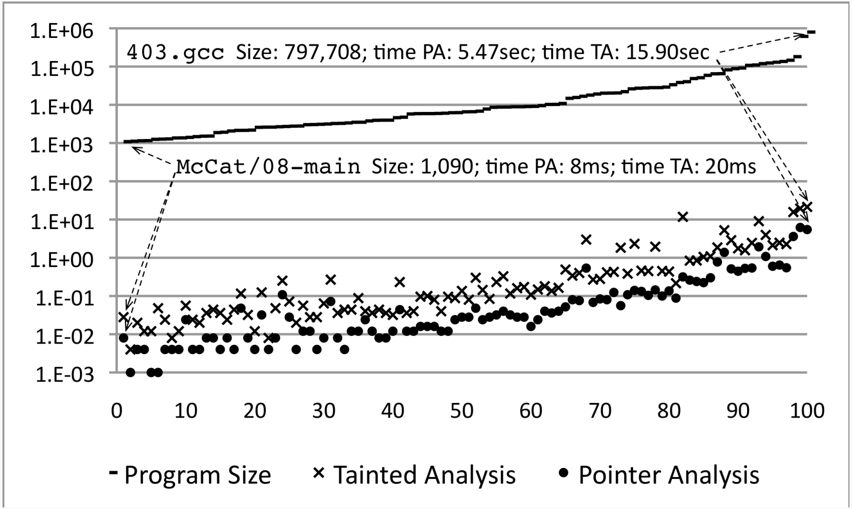
Another example : Constant propagation

Sparse/Dense



- This helps to be **quasi-linear**.

Concrete complexity



Experiments by F. Pereira

Sparse abstract interpretation

Designing a scalable static analysis : an example

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

A bit on sanitizing memory accesses

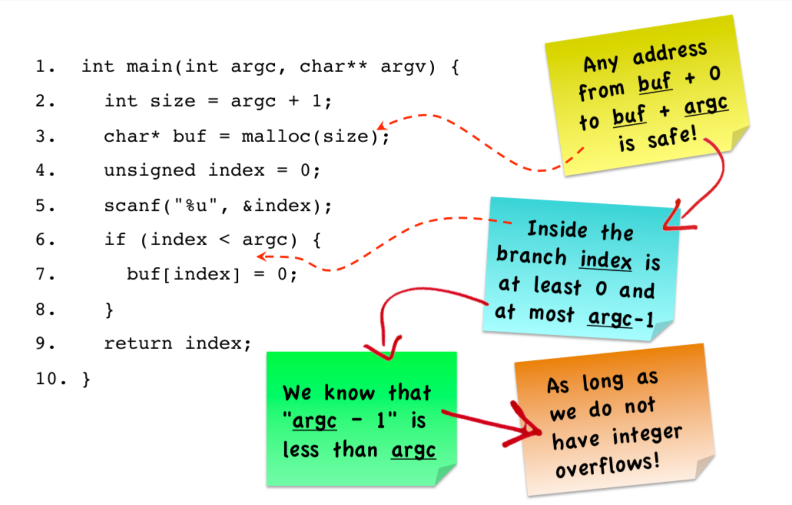
Different techniques : but all have an overhead.

Ex : Address Sanitizer

- Shadow every memory allocated : 1 byte → 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



```
int main(int argc, char** argv) {
  int size = argc + 1;
  char* buf = malloc(size);
  unsigned index = 0;
  scanf("%u", &index);
  if (index < argc) {
    buf[index] = 0;
  }
  return index;
}
```

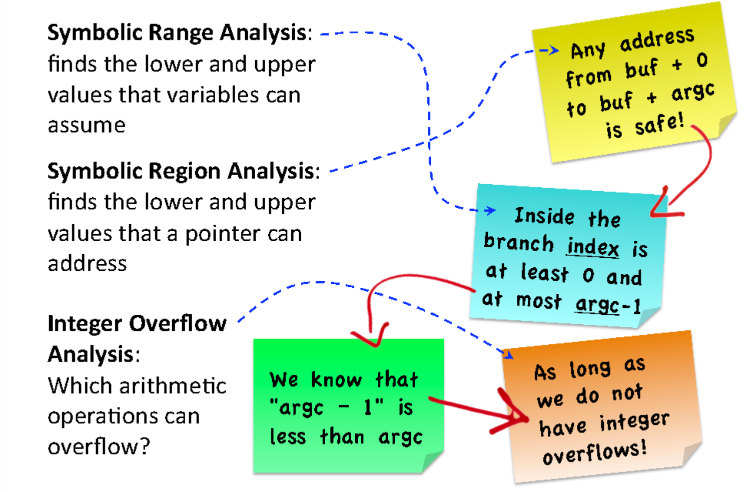
Any address from buf + 0 to buf + argc is safe!

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 <u>index</u> is at least 0 and at most <u>argc</u>-1

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

As long as we do not have integer overflows!

Symbolic ranges : How to ensure scalability ?

The idea is to work on the intermediate representation to ensure the following key property :

SSI Property

All abstract values are **stable** on their live ranges.

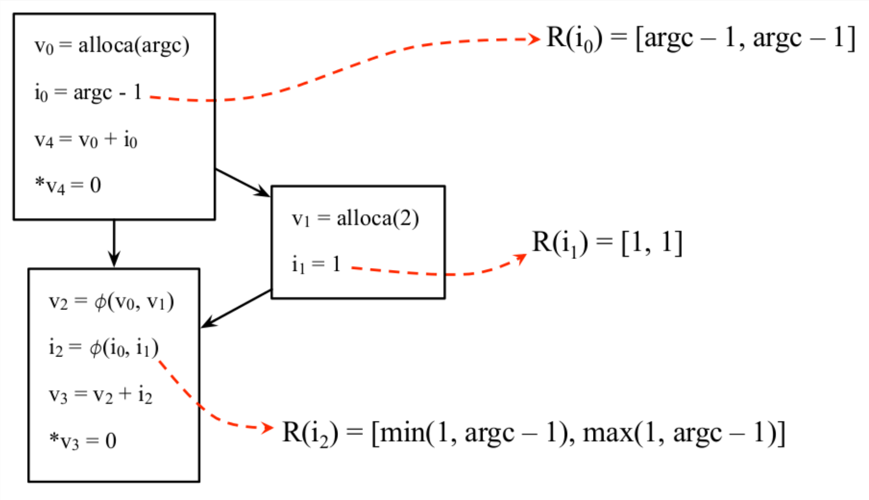
How ? Splitting variables (v, i in the last example).

Symbolic Ranges (SRA) : Running example

```
int main(int argc){
  int* v = malloc(sizeof(int)*argc);
  int i = argc -1;
  v[i] = 0;
  if (?) {v = realloc(sizeof(int)*2); i=1 ;}
  v[i] = 0;
}
```

- Are all accesses to v **safe** ?

Symbolic Ranges (SRA) : On the SSA form



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).

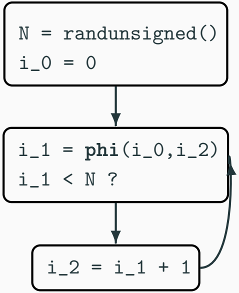
SRA on SSA form : constraint system

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

$\oplus^I$: 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



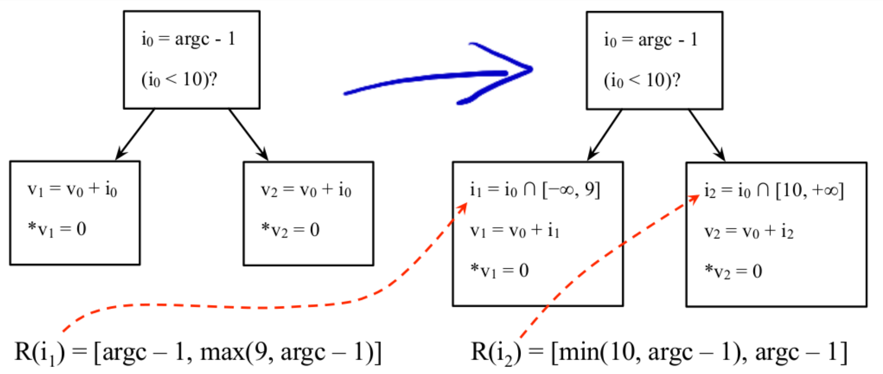
```
N = randunsigned()
i_0 = 0
```

```
i_1 = phi(i_0,i_2)
i_1 < N ?
```

```
i_2 = i_1 + 1
```

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

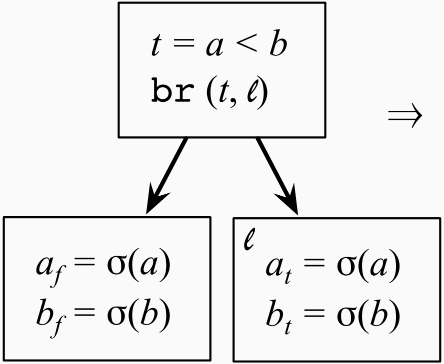
Improving precision of SRA : live-range splitting 1/2



- e-SSA form.

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

Rule for live-range splitting :



$t = a < b$

$\texttt{br}\ (t, \ell)$

$a_f = \sigma(a)$

$b_f = \sigma(b)$

ℓ

$a_t = \sigma(a)$

$b_t = \sigma(b)$

$$\Rightarrow$$

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

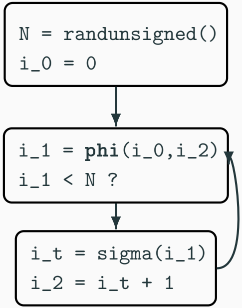
$$R(b_t) = [\max(R(a)_\downarrow + 1, R(a)_\downarrow), R(b)_\uparrow]$$

$$R(a_f) = [\max(R(a)_\downarrow, R(a)_\uparrow), R(a)_\uparrow]$$

$$R(b_t) = [R(b)_\downarrow, \min(R(a)_\uparrow, R(b)_\uparrow)]$$

- All simplications are done by GiNaC.

SRA + live-range on an example



```
N = randunsigned()
i_0 = 0
```

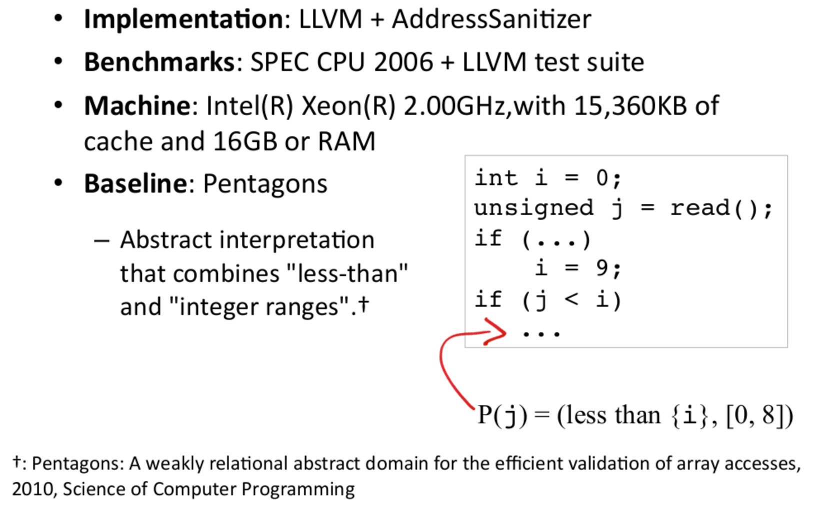
```
i_1 = phi(i_0,i_2)
i_1 < N ?
```

```
i_t = sigma(i_1)
i_2 = i_t + 1
```

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

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

Experimental setup



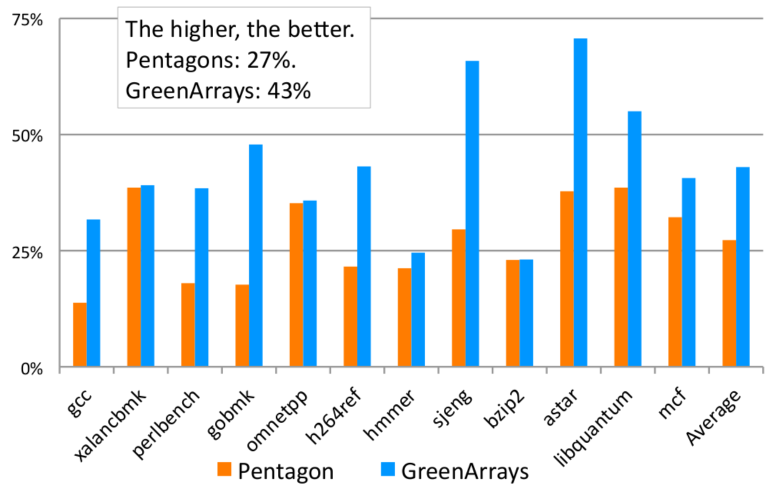
- **Implementation**: LLVM + AddressSanitizer
- **Benchmarks**: SPEC CPU 2006 + LLVM test suite
- **Machine**: Intel(R) Xeon(R) 2.00GHz,with 15,360KB of cache and 16GB or RAM
- **Baseline**: Pentagons
  - Abstract interpretation that combines "less-than" and "integer ranges".†

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

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

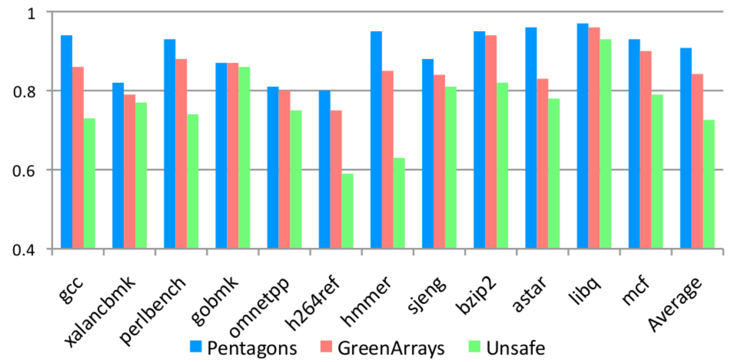
†: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses, 2010, Science of Computer Programming

Percentage of bound checks removed



The higher, the better.
Pentagons: 27%.
GreenArrays: 43%

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.

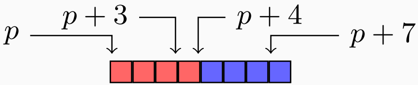
▶ `https://code.google.com/p/ecosoc/`

Remaining question : improving precision of the symbolic range analysis ?

Another example : pointer analysis

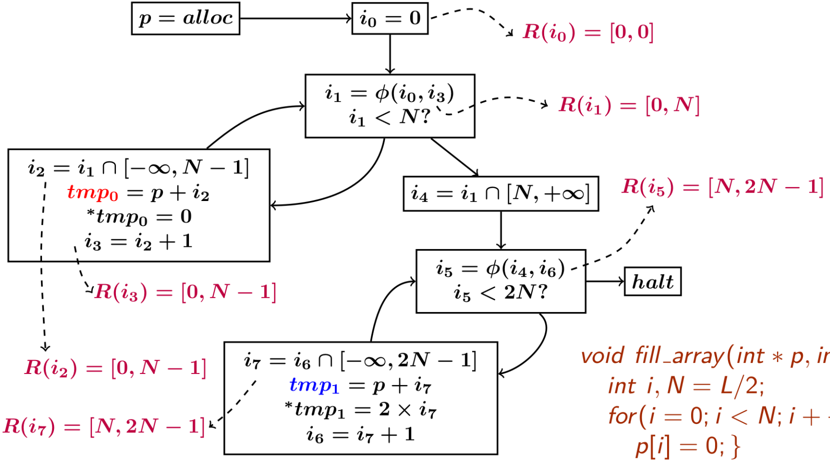
```
void fill_array (char *p){
  unsigned int i;
  for (i=0; i<4; i++)
      *(p + i) = 0 ;
  for (i=4; i<8; i++)
      *(p + i) = 2*i ;
}
```

Parallel loops



slides courtesy of Maroua Maalej - hence strange colors

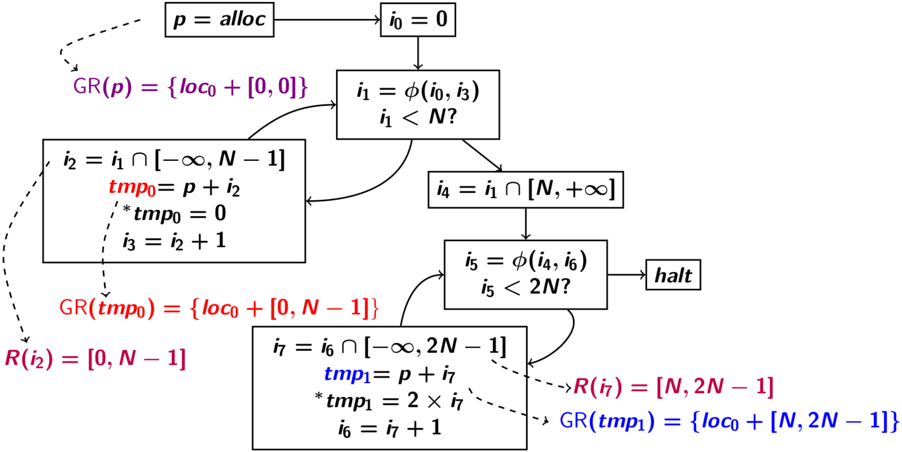
Pre-analysis



```
void fill_array(int * p, intL){
   int i, N = L/2;
   for(i = 0; i < N; i + +){
      p[i] = 0; }
   for(; i < 2 × N; i + +){
      p[i] = 2 × i; }
}
```

Final result after propagation

We propagate range+offset $\rightsquigarrow$ not alias information.



$p = alloc$

$i_0 = 0$

$i_1 = \phi(i_0, i_3)$
$i_1 < N?$

$i_2 = i_1 \cap [-\infty, N-1]$
$tmp_0 = p + i_2$
$^*tmp_0 = 0$
$i_3 = i_2 + 1$

$i_4 = i_1 \cap [N, +\infty]$

$i_5 = \phi(i_4, i_6)$
$i_5 < 2N?$

$halt$

$i_7 = i_6 \cap [-\infty, 2N-1]$
$tmp_1 = p + i_7$
$^*tmp_1 = 2 \times i_7$
$i_6 = i_7 + 1$

$\mathrm{GR}(p) = \{loc_0 + [0, 0]\}$

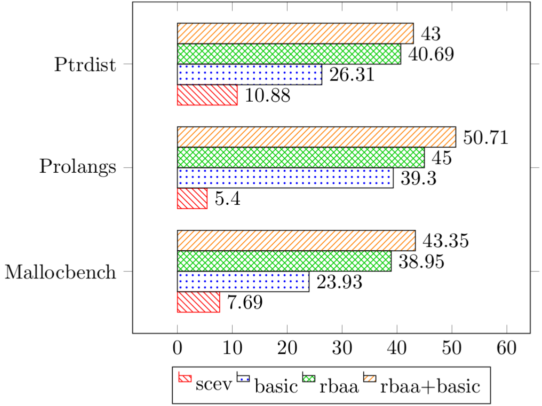
$\mathrm{GR}(tmp_0) = \{loc_0 + [0, N-1]\}$

$R(i_2) = [0, N-1]$

$R(i_7) = [N, 2N-1]$

$\mathrm{GR}(tmp_1) = \{loc_0 + [N, 2N-1]\}$

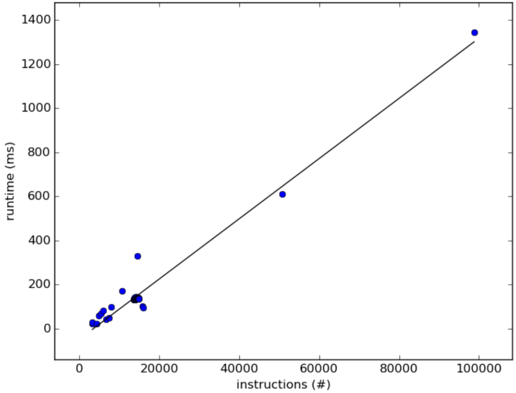
Experimental results 1/2



- basic: -O3 LLVM alias analysis (global + local pointers).
- scev: scalar evolution based alias-analysis.
- rbaa: range based alias analysis.
- running rbaa with basicaa

Percentage of queries that answered “no-alias” in LLVM’s scev-aa, LLVM’s basicaa and our analysis named rbaa.

Experimental results 2/2



$r = 0.98$

- Runtime on the 50 largest benchmarks in llvm test-suite (ordered)
- Complexity: linear analysis time
- Speed: less than 10 seconds to analyse 100,000 instructions

Conclusion and Open Research questions

We have a framework to design “quasi linear” analyses in compilers.

- How to mechanize the proofs ? (for any sparse analysis)
- How to adapt a relational abstract domain into a sparse version ? (in general)
- No clean framework for this kind of analysis (even on paper) : live range splitting is only an attempt (in my opinion).

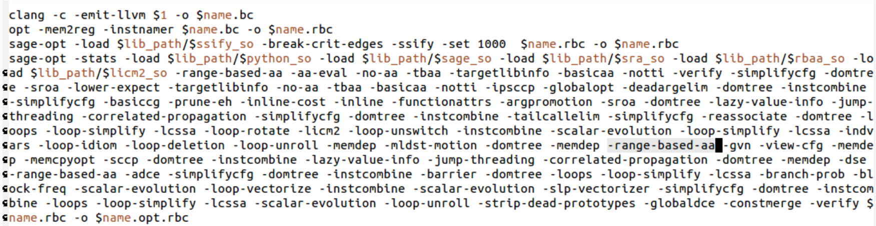
side ? question

How to make an impact on optimisations ?

Evaluating analyses in LLVM ?

LLVM compiler :

- comes with a test infrastructure and benchmarks.
- analysis and optimisation passes log information.
- you can add your own pass, but **where ?**



```
clang -c -emit-llvm $1 -o $name.bc
opt -mem2reg -instnamer $name.bc -o $name.rbc
sage-opt -load $lib_path/$ssify_so -break-crit-edges -ssify -set 1000  $name.rbc -o $name.rbc
sage-opt -stats -load $lib_path/$python_so -load $lib_path/$sage_so -load $lib_path/$sra_so -load $lib_path/$rbaa_so -lo
ad $lib_path/$licm2_so -range-based-aa -aa-eval -no-aa -tbaa -targetlibinfo -basicaa -notti -verify -simplifycfg -domtre
e -sroa -lower-expect -targetlibinfo -no-aa -tbaa -basicaa -notti -ipsccp -globalopt -deadargelim -domtree -instcombine
-simplifycfg -basiccg -prune-eh -inline-cost -inline -functionattrs -argpromotion -sroa -domtree -lazy-value-info -jump-
threading -correlated-propagation -simplifycfg -domtree -instcombine -tailcallelim -simplifycfg -reassociate -domtree -l
oops -loop-simplify -lcssa -loop-rotate -licm2 -loop-unswitch -instcombine -scalar-evolution -loop-simplify -lcssa -indv
ars -loop-idiom -loop-deletion -loop-unroll -memdep -mldst-motion -domtree -memdep -range-based-aa -gvn -view-cfg -memde
p -memcpyopt -sccp -domtree -instcombine -lazy-value-info -jump-threading -correlated-propagation -domtree -memdep -dse
-range-based-aa -adce -simplifycfg -domtree -instcombine -barrier -domtree -loops -loop-simplify -lcssa -branch-prob -bl
ock-freq -scalar-evolution -loop-vectorize -instcombine -scalar-evolution -slp-vectorizer -simplifycfg -domtree -instcom
bine -loops -loop-simplify -lcssa -scalar-evolution -loop-unroll -strip-dead-prototypes -globaldce -constmerge -verify $
name.rbc -o $name.opt.rbc
```

▶ Evaluating the impact of a given analysis is a **nightmare** !

Impact of our analyses - negative results ?

Program	**#Inst**	**#moved**	
		O3	**O3+our analysis (CGO16)**
fixoutput	369	1	**5**
compiler	3515	0	0
bison	15645	165	**179**
archie-client	5939	0	0
TimberWolfMC	98792	1287	**1447**
allroots	574	0	0
unix-smail	5435	3	3
plot2fig	3217	3	3
bc	10632	18	**19**
yacr2	6583	144	**190**
ks	1368	8	**11**
cfrac	7353	5	**6**
espresso	50751	301	**398**
gs	55281	20	X

More in Maroua Maalej’s thesis