Experiences in designing scalable static analyses
FAC days, Toulouse

Laure Gonnord, University Lyon 1
& Laboratoire d’Informatique du Parallélisme
http://laure.gonnord.org/pro

April 5th, 2018
Compilation and Analysis for Software and Hardware - Location

LIP:
Laboratoire de l'Informatique du Parallélisme
Optimized (software/hardware) compilation for HPC software with data-intensive computations.

⇒ Means: dataflow IR, static analyses, optimisations, simulation.

Christophe Alias, Laure Gonnord, Matthieu Moy
http://www.ens-lyon.fr/LIP/CASH/
Outline

Motivations
  Static analyses, examples
  Static analysis of software, how?

Abstract Interpretation 101

Abstract Interpretation for optimising compilers
  Example 1: a scalable analysis for pointers
  Example 2: array bound check elimination
  Impact on compiler optimisation pathes

Conclusion
Software needs safety and performance

- For safety-critical systems . . .
- **and** general purpose systems!
Software needs safety and performance

- For safety-critical systems . . .
- and general purpose systems!

▶ Programs crash because of array out-of-bounds accesses, complex pointer behaviour, . . .
Software guarantees, how?

• Development processes: coding rules, . . .
• Testing: do not cover all cases.
• Proof assistants: expensive.

▶ Static analysis of programs.
Outline

Motivations

Static analyses, examples
Static analysis of software, how?

Abstract Interpretation 101

Abstract Interpretation for optimising compilers
Example 1: a scalable analysis for pointers
Example 2: array bound check elimination
Impact on compiler optimisation pathes

Conclusion
Goal: safety 1/2

Prove that (some) memory accesses are safe:

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

▶ This program has an illegal array access.
Goal: safety 2/2

Prove program correctness/absence of functional bug:

```c
void find_mini (int a[N], int l, int u){
    unsigned int i=l;
    int b=a[l]
    while (i <= u){
        if(a[i]<b) b=a[i] ;
        i++ ;
    }
    // here b = min (a[l..u])
}
```

▶ This program finds the minimum of the sub-array.
Enable loop parallelism:

```c
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 ;
}
```

The two regions do not overlap.
Enable code motion:

```c
void code_motion(int* p1, int *p2, int *p){
    // ...
    while(p2 > p1){
        a = *p;
        *p2 = 4;
        p2 --;
    }
}
```

- If $p$ and $p_2$ do not alias, then $a = *p$ is invariant.
- Hoisting this instruction saves one \texttt{load} per loop.
Outline

Motivations
  Static analyses, examples
  Static analysis of software, how?

Abstract Interpretation 101

Abstract Interpretation for optimising compilers
  Example 1: a scalable analysis for pointers
  Example 2: array bound check elimination
  Impact on compiler optimisation pathes

Conclusion
Proving non trivial properties of software

- Basic idea: software has **mathematically defined behaviour**.
- **Automatically** prove properties.

Acceptable Behaviours

Program

(verify) No crash

(compile) Optimisable
There is no free lunch

i.e. no magical static analyser. It is impossible to prove interesting properties:

• automatically
• exactly
• on unbounded programs
There is no free lunch

i.e. no magical static analyser. It is impossible to prove interesting properties:

- automatically
- exactly with false positives!
- on unbounded programs

► Abstract Interpretation = conservative approximations.
Outline

Motivations
Static analyses, examples
Static analysis of software, how?

Abstract Interpretation 101

Abstract Interpretation for optimising compilers
Example 1: a scalable analysis for pointers
Example 2: array bound check elimination
Impact on compiler optimisation pathes

Conclusion
Computing (inductive) invariants

\[ \{ x \in \mathbb{N}, 0 \leq x \leq 100 \} \] is the most precise invariant in control point loop.
Problems and solution

We want to:

- Compute infinite sets.
- In finite time.

▶ How?

- Approximate sets (abstract domains), compute in this abstract world.
- Extrapolate (widening).
Main ingredient: abstract values

Idea: represent values of variables:

\[ R_{pc} \in \mathcal{P}(\mathbb{N}^d) \]

by a **finite computable superset** \( R^\#_{pc} \):

▶ And compute such **abstract values** for each control point.
▶ How? mimic the program operations

\[ \mathbb{N}^d \times pcs \rightarrow \mathbb{N}^d \times pcs \]

by their abstract versions.

There is also this magical widening stuff, let’s forget it in this talk
Example (Pagai, Verimag)

```c
int main(int argc, char** argv){
    int x, y;
    x = 1;
    y = 2;
    /* reachable */
    /* invariant: */
    3-2*y+x = 0
    5-y >= 0
    -2+y >= 0
    /*
    while (x<8){
        x = x+2;
        y = y+1;
    }
    /* reachable */
    return 0;
```
Other famous AI tools

- Frama-C: “Evolved Value Analysis”.
- Astree: originally designed for safety critical C Compiled from Scade.
- Polyspace (Mathworks).
Complexity in Abstract Interpretation

**Classical** abstract interpretation analyses:

- Information attached to \((\text{block}, \text{variables})\).
- A new information is computed after each statement.
- Abstract operations are sometimes costly.

For the polyhedral abstract domain, the complexity is \(3\text{EXP}\).
Challenges in Abstract Interpretation

- Precision of the abstract domain.
- Thousands, millions of lines of code to analyze.
- Static analyzers and compilers are complex programs (that also have bugs).

Growing need for simple specialized analyses that scale
Credo: future of Abstract Interpretation

- Focus more on applicability, and less on expressivity.
- Scale and demonstrate that it scales.
- For general-purpose programs.

and use techniques from other communities (optimization, model-checking, logic, rewriting)
Motivations
Static analyses, examples
Static analysis of software, how?

Abstract Interpretation 101

Abstract Interpretation for optimising compilers
   Example 1: a scalable analysis for pointers
   Example 2: array bound check elimination
   Impact on compiler optimisation pathes

Conclusion
Safe compilation?

- **Correct-by-construction** non-optimising compilers: Lustre, Scade.
- Translation validation: specialized proof of the generated code.
- Compcert.

▶ An evolution toward more trustable compilers. But what about code optimisation?
Motivation

Classical analyses (and optimisation) inside (production) compilers:

- Apart from classical dataflow algorithm, often syntactic.
- Usual abstract-interpretation based algorithms are too costly.
- Expressive algorithms: rely on “high level information”.

Need for safe and precise quasi linear-time algorithms at low-level. Illustrations in the rest of the talk.
Motivation

Classical analyses (and optimisation) inside (production) compilers:

- Apart from classical dataflow algorithm, often syntactic.
- Usual abstract-interpretation based algorithms are too costly.
- Expressive algorithms: rely on “high level information”.
  - Need for safe and precise quasi linear-time algorithms at low-level.
- Illustrations in the rest of the talk.
Some contributions

- Abstract domains/iteration strategies for numerical invariants [SAS11], [OOPSLA14].
- Applications to memory analysis [OOPSLA14], just in time compilers [WST14].
- Pointer analysis with “sparse” abstract interpretation [CGO16] [CGO17] [SCP17].

Collaborations with M. Maalej, F. Pereira and his team at UFMG, Brasil
Outline

Motivations
Static analyses, examples
Static analysis of software, how?

Abstract Interpretation 101

Abstract Interpretation for optimising compilers
Example 1: a scalable analysis for pointers
Example 2: array bound check elimination
Impact on compiler optimisation pathes

Conclusion
void partition(int *v, int N) {
    int i, j, p, tmp;
    p = v[N/2];
    for (i = 0, j = N - 1;; i++, j--) {
        while (v[i] < p) i++;
        while (p < v[j]) j--;
        if (i >= j)
            break;
        tmp = v[i];
        v[i] = v[j];
        v[j] = tmp;
    }
}

v[i] = *(v+i)

v+i

v+j
Less than information for pointers [CGO17,SCP17]

```c
void partition(int *v, int N) {
    int i, j, p, tmp;
    p = v[N/2];
    for (i = 0, j = N - 1;; i++ , j--) {
        while (v[i] < p) i++;
        while (p < v[j]) j--;
        if (i >= j)
            break;
        tmp = v[i];
        v[i] = v[j];
        v[j] = tmp;
    }
}
```

- Range information is not sufficient to disambiguate `v[i]` and `v[j].`
- We need to propagate **relational information.**
Our setting for scaling analyses

**Classical** abstract interpretation analyses:
- Information attached to \((block, variable)\).
- A new information is computed after each statement.

Sparse analyses \(\Rightarrow\) **Static Single Information (SSI)**

**Property [Ana99]:**
- Attach information to variables.
- The information must be invariant throughout the live range of the variable.

▶ A simple assignment breaks SSI!
▶ Work on suitable intermediate representations
Scaling analyses: program representation 1/2

Static Single Assignment (SSA) form: each variable is defined/assigned once.

```c
void partition(int *v, int N) {
    int i, j, p, tmp;
    p = v[N/2];
    for (i = 0, j = N - 1;; i++, j--) {
        while (v[i] < p) i++;
        ...
    }
}
```
Static Single Assignment (SSA) form: each variable is defined/assigned once.

```c
void partition(int *v, int N) {
    int i, j, p, tmp;
    p = v[N/2];
    for (i = 0, j = N - 1;; i++, j--) {
        while (v[i] < p) i++;
        ...
    }
}
```

- Sparse storage of value information (one value range per variable name).
Within **SSA** form, tests information cannot be propagated!

```c
void partition(int *v, int N) {
    ...
    if (i >= j)
        break;
    tmp = v[i];
    v[i] = v[j];
}
```

- **i ≥ j** is invariant nowhere.
- The $\sigma$ renaming (**e-SSA**) enables to propagate “$i_F < j_F$”.
Recall the SSI setting:

- Information must be invariant throughout the live range of the variable.
- Attach information to variables (and not blocks).

Work on semi-relational domains, for instance:

- Parametric ranges [OOPSLA14] $x \mapsto [0, N + 1]$
- Pentagons [LF10]: $x \mapsto \{u, t\}$ means $u, t \leq x$. 
Contributions on static analyses for pointers

(with Maroua Maalej) [CGO16, CGO17, SCP17]

• A new sequence of static analyses for pointers.
• Based on semi-relational sparse abstract domains:
  • In CGO’16: \( p \mapsto loc + [a, b] \).
  • In CGO’17: adaptation of Pentagons.
• Implemented in LLVM.
• Used as oracles for a common pass called AliasAnalysis.
• Experimental evaluation on classical benchmarks.
Experimental results [SCP17]

- Comparison with LLVM basic alias analysis.
- Our sraa outperforms basicaa in the majority of the tests.
- The combination outperforms each of these analyses separately in every one of the 100 programs.
Motivations
Static analyses, examples
Static analysis of software, how?

Abstract Interpretation 101

Abstract Interpretation for optimising compilers
Example 1: a scalable analysis for pointers
Example 2: array bound check elimination
Impact on compiler optimisation pathes

Conclusion
Contribution [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: a set of sparse analyses 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

As long as we do not have integer overflows!

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

- We know that “argc - 1” is less than argc
- As long as we do not have integer overflows!
- Any address from buf + 0 to buf + argc is safe!
- Inside the branch index is at least 0 and at most argc-1
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)  
  ...
```

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

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%.
Outline

Motivations
  Static analyses, examples
  Static analysis of software, how?

Abstract Interpretation 101

Abstract Interpretation for optimising compilers
  Example 1: a scalable analysis for pointers
  Example 2: array bound check elimination
  Impact on compiler optimisation pathes

Conclusion
Some comments on the methodology

LLVM compiler:

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

► Evaluating the impact of a given analysis is a **nightmare!**
Loop invariant code motion (LICM):

```c
void code_motion(int* p1, int *p2, int *p){
    // ...
    while(p2>p1){
        a = *p;
        *p2 = 4;
        p2 --;
    }
}
```

- If p and p2 do not alias, then a=*p is invariant.
Impact of our analyses (excerpt) 2/2

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

More in Maroua Maalej's thesis.
Outline

Motivations
  Static analyses, examples
  Static analysis of software, how?

Abstract Interpretation 101

Abstract Interpretation for optimising compilers
  Example 1: a scalable analysis for pointers
  Example 2: array bound check elimination
  Impact on compiler optimisation pathes

Conclusion
Summary

Static analyses for compilers:

- Application domain: code optimisation.
- Adaptation of abstract interpretation algorithms inside this particular context.
- Algorithmic and compilation techniques to scale.
- Future work: more relational domains (and data structures).

▶ I have a Phd funding!
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.
References I

Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs.

Rank: a tool to check program termination and computational complexity.
In Workshop on Constraints in Software Testing Verification and Analysis (CSTVA’13), Luxembourg, March 2013.

[Ana99] Scott Ananian.
The static single information form.

[Fea92] Paul Feautrier.
Some efficient solutions to the affine scheduling problem. part ii. multidimensional time.

Synthesis of ranking functions using extremal counterexamples.

Pentagons: A weakly relational abstract domain for the efficient validation of array accesses.

Combining Range and Inequality Information for Pointer Disambiguation.
Accepted in Oct 2017, final published version of https://hal.inria.fr/hal-01429777v2.
References II

