Interprocedural Parallel Invariants

Helmut Seidl
Varmo Vene
Markus Müller-Olm
0. Background

EU Project Daedalus:

... use abstract interpretation to improve Avionic software:

- to obtain high quality WCET prediction;
- to detect errors in
  - ... floating-point operations;
  - ... string manipulation and heap accesses;
  - ... multi-threaded code.
0. Background

EU Project Daedalus:

... use abstract interpretation to improve Avionic software:

- to obtain high quality WCET prediction;
- to detect errors in
  - ... floating-point operations;
  - ... string manipulation and heap accesses;
  - ... multi-threaded code.
1. Achievements

Design and implementation of an analyzer for C programs which ...

+ deals with (arrays of) function pointers;
+ deals with heap-allocated data;
+ tracks flow- and context-sensitive information;
+ deals safely with multiple pthreads ...
detects global invariants ...
  → data races
  → (some simple forms) deadlock freeness
+ is sufficiently efficient :-))

Idea:

• track values of local variables using the functional approach to interprocedural analysis and
• demand-driven fixpoint computation;
• approximate globals and heap by safe invariants ...
Functional Approach:

Describe functions by functions:

\[ \mathbb{D}_1 \rightarrow (\text{state before call}) \]

\[ \mathbb{D}_1 \rightarrow (\text{state after call}) \]

Just tabulate return values of occurring calls:

\[ Calls = \text{Funs} \times \mathbb{D}_1 \quad (\text{name, state before call}) \]
Transfer functions for edges:

$$\mathcal{D}_1 \times (\text{Calls} \rightarrow \mathcal{D}_1) \rightarrow \mathcal{D}_1$$

(old local state)  (returns for calls)  (new local state)

... for demand-driven evaluation:
main()

int f (int y)
return y+1;

x = f(x);
x = 7;
main()

int f (int y)

x = 7;

x = f(x);

return y+1;
main()

int f (int y)

x = 7;

return y+1;

x = f(x);
main()

int f (int y)

\[
x = f(x);
\]

\[
x = 7;
\]

\[
\text{return } y+1;
\]
main()

int f (int y)

```c
int f (int y)
{ return y+1; }
```

```c
main()
{ int x; x = f(x); x = 7; return x; }
```
main()

```
x = f(x);
x = 7;
```

int f (int y)

```
return y+1;
```
main()

\[
\begin{align*}
\text{x} &= 7; \\
\text{x} &= \text{f(x)}; \\
\text{x} &= 8
\end{align*}
\]

\[
\begin{align*}
\text{y} &= 7; \\
\text{y} &= \text{return } y+1;
\end{align*}
\]

int \text{f} (\text{int} \ y)
Global Invariants:

- Different threads may communicate through globals;
- Maintain one single description for all possible values of globals;
- Local computations of threads should be invariant under the description of globals.
\[ z = x + y; \]
\[ z = x + y; \]
\[ z = x + y; \]
The Problem:

- Fixpoint engine solves general constraints.
- It explores variable dependencies demand-driven:
  - only occurring calls are analyzed
  - global variables formally also depend on un-realizable function calls

\[ \rightarrow \text{solving with side effects} \]
More general transfer function:

\[
\mathbb{D}_1 \times (\text{Calls} \to \mathbb{D}_1) \times (\text{Globals} \to \mathbb{D}) \rightarrow \mathbb{D}_1 \times \text{Calls list} \times (\text{Globals} \times \mathbb{D}) \text{ list}
\]

(old local state) (returns for calls) (values for globals) (new local state) (spawned calls) (side effects to globals)
Our System:

Program

Frontend

Base Analysis

Fixpoint Engine

Answer
Our System:

Program

Frontend

Base Analysis

Analysis

Fixpoint Engine

Property

Answer
Our System:

Program

Frontend

Base Analysis

Analysis

Fixpoint Engine with Side Effects

Property

Answer
2. Some Details:  

**Base Analysis**

**Demand:**

- tracking of storage layout;
- tracking of addresses;
- tracking of `int` values;
- tracking of mutex locks;
- tracking of strings.
2.1. Storage Layout and Addresses

Example:

```c
int f (int x) {return x+1;}
int g (int x) {return x*x;}

typedef int fun (int x);
fun *A [] = {f, g, f, g};
...
x = A[i](y);
...
```
Idea:

- All entries of arrays are merged into one;
- Multiple addresses are merged into a set.
2.2. int Constants

Example:

```c
if (x == 7) pthread_mutex_lock (&me);
...
if (x == 7) pthread_mutex_unlock (&me);
```
The Classical Analysis:

... does not track negative information :-(

![Diagram](image-url)
lock(...);

unlock(...);

x == 7

x == 7

lock(...);

unlock(...);
lock(...);

x == 7

x == 7

lock(...);

unlock(...);
lock(...);

x == 7

x == 7

lock(...);

unlock(...);
lock(...);

x == 7

unlock(...);

x == 7

lock(...);

x == 7
Idea 1:

```
\begin{center}
\begin{tikzpicture}
  \node (root) {?};
  \node (zero) [below left of=root] {0};
  \node (one) [below right of=root] {+1};
  \node (zero-one) [below left of=one] {+1,+2};
  \node (one-one) [below right of=one] {+1,+3};
  \node (two) [below left of=zero-one] {-2};
  \node (three) [below right of=zero-one] {-1};
  \node (four) [below left of=one-one] {0};
  \node (five) [below right of=one-one] {+1};
  \node (six) [below right of=five] {+2};

  \draw (root) -- (zero);
  \draw (root) -- (one);
  \draw (zero) -- (zero-one);
  \draw (one) -- (one-one);
  \draw (zero-one) -- (two);
  \draw (zero-one) -- (three);
  \draw (one-one) -- (four);
  \draw (one-one) -- (five);
  \draw (five) -- (six);
\end{tikzpicture}
\end{center}
```
lock(...);

x == 7

unlock(...);
Anti-constants are not quite sufficient  :-(

Idea 2:

- Track each possibly hold set of locks separately;
- Join variable assignments only relative to a possible set of locks  :-)
lock(...);

x == 7

x

x == 7

lock(...);

unlock(...);
lock(...);

x == 7

x

∅ x 7

∅ x ?

lock(...);

x == 7

unlock(...);
lock(...);

unlock(...);

x == 7

me x 7

lock(...);

x == 7

unlock(...);
lock(...);
unlock(...);
x == 7
lock(...);
x == 7
unlock(...);
lock(...);

me x 7

x == 7

lock(...);

x == 7

unlock(...);

∅ x ?

∅ x 7

∅ x 7

me x 7

me x 7
2.3. Tracking Mutex Locks

Sadly enough,

- ... pthread mutex locking may fail;
- ... some code checks whether locking has succeeded.

We cannot assume that a lock has been acquired before the check point;

We must track variables holding return values of `pthread_mutex_lock()`.
Example:

```c
ret = pthread_mutex_lock (Mutex);
while (ret == 4)
    ret = pthread_mutex_lock (Mutex);
if (ret != 0) RetCond = FALSE;
else while (RetCond) {...}
```
2.4. Tracking Shared Memory

In our benchmark,

- ... various **shared memory regions** are allocated
- ... which are distinguished by **strings**
- ... which are manipulated with C string library functions
  `strcpy()`, `strncpy()`, `strcat()`, `strncat()`,
  `strstr()`, `strcmp()`, `strncmp()`
- ... whose mappings to shared memory regions eventually
  is read from file   :-(

What to do here???
Application Specific Observation:
String manipulation of name and association with the shared memory is concentrated inside a single application function:

```c
int get_shared_memory (char * name,
                       void ** shmem_ptr,
                       int flags);
```

Our Solution: Application specific abstraction

- Provide abstract function `get_shared_memory()``;
- Perform simple propagation of strings through `strcpy()`, `strcmp()` only ...
- Give up whenever a pointer occurs inside a string.
3. Current Implementation:

- ... in SML using smlnj-110.0.7
- ... running under Suse Linux
- ... on a 800 MH Atlon with 1 GB main memory
- ... tested with parts of EADS Airbus application systems of sizes \( \approx 40,000 \) LOC (excluding header files)
Implemented Analyses:

- Thread analysis;
- Various data race detection analyses:
  - ... assuming always succeeding locks;
  - ... assuming failing locks;
  - ... definite errors;
  - ... potential errors;
  - ...
Runtimes:

2 min. for simple to
15 min. for elaborated analysis

... solving upto 900,000 constraints :-)
4. Conclusions

- We implemented an analyzer generator for C with pthreads which is able to handle the EADS Airbus benchmarks.
- In order to obtain sanity checks or certificates for absence of (certain kinds of) errors, we need:
  1. explicit general assumptions about the system;
  2. strong analysis techniques;
  3. adaption to the software to be validated and
  4. disciplined programming :-)