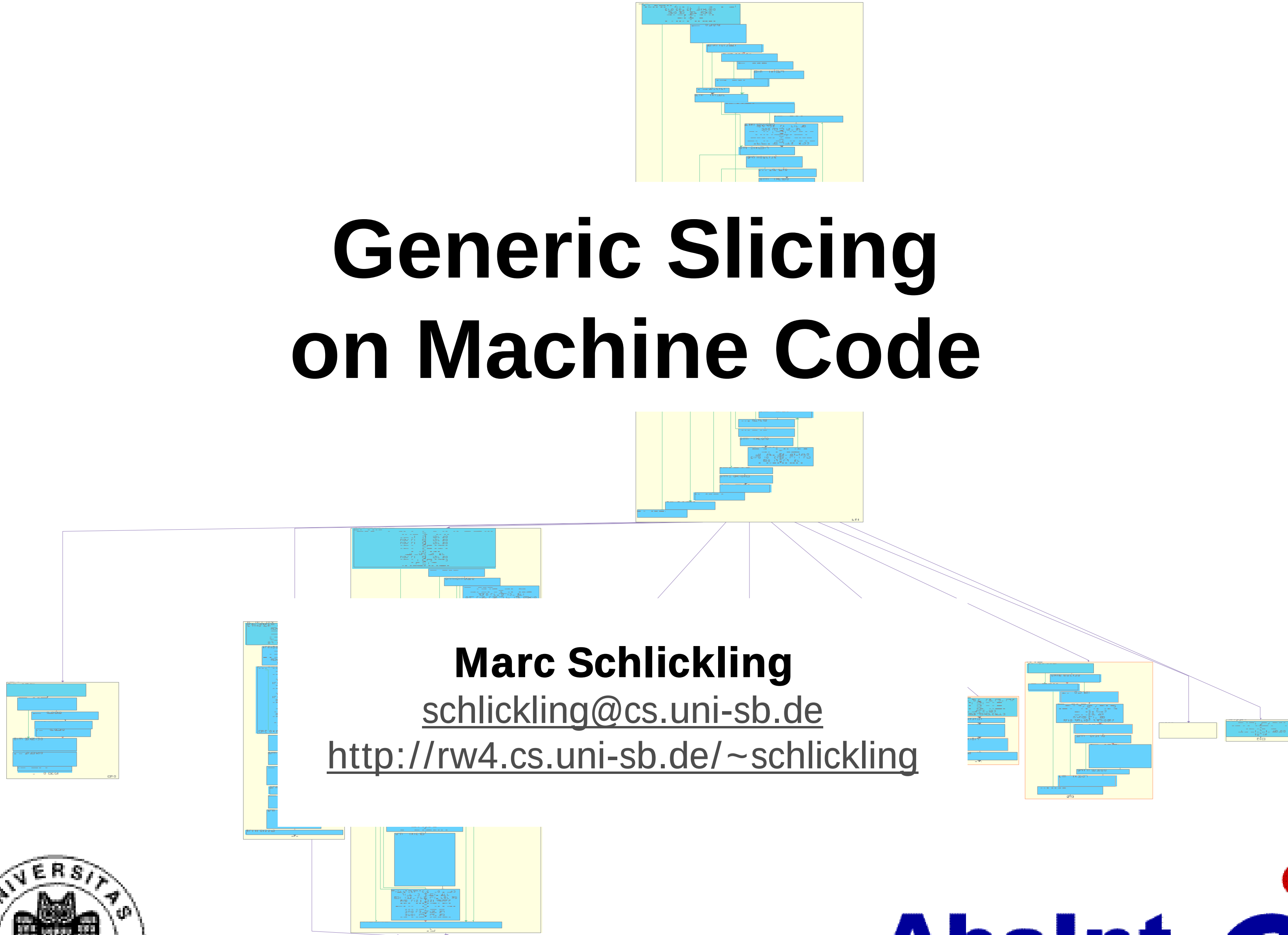
Generic Slicing on Machine Code



Marc Schlickling
schlickling@cs.uni-sb.de
http://rw4.cs.uni-sb.de/~schlickling

Embedded Systems (1)

- Embedded Systems (ES) are widely used
  - Many systems of daily use: handy, handheld, pda, ...
  - Safety critical systems: airbag control, flight control system, ...
- Rapidly growing complexity of software in ES
- But what happens, if such a system crashes?

Embedded Systems (1)

- Embedded Systems (ES) are widely used
  - Many systems of daily use: handy, handheld, pda, ...
  - Safety critical systems: airbag control, flight control system, ...
- Rapidly growing complexity of software in ES
- But what happens, if such a system crashes?

Embedded Systems (1)

- Embedded Systems (ES) are widely used
  - Many systems of daily use: handy, handheld, pda, ...
  - Safety critical systems: airbag control, flight control system, ...
- Rapidly growing complexity of software in ES
- But what happens, if such a system crashes?

Embedded Systems (1)

- Embedded Systems (ES) are widely used
  - Many systems of daily use: handy, handheld, pda, ...
  - Safety critical systems: airbag control, flight control system, ...
- Rapidly growing complexity of software in ES
- But what happens, if such a system crashes?

Embedded Systems (2)

- ES subject to strict safety constraints
  - Strict timing constraints
    - Execution has to be always fast enough
  - Strict resource constraints
    - No stack overflow during run time
  - Strict spacial partitioning
    - Different tasks run within their own memory spaces

➢ Urgent need to guarantee these characteristics

Guaranteeing safety constraints (1)

- Not only possible at source code level
  - Caused by compilers
    - Optimizations
    - Transformations
- Complex due to the use of integrated modular platforms
  - E.g. IMA (integrated modular avionics)
    - Many components sharing the same resource
    - Used in the new Airbus A380

Guaranteeing safety constraints (2)

- Programs checking such constraints
  - aiT
    - Worst case execution time (WCET) prediction
  - StackAnalyzer
    - Worst case stack usage determination
- Analyses are done on real executable
  - Due to the complexity of analyzed software, some help of the developers required

Appliance of slicing

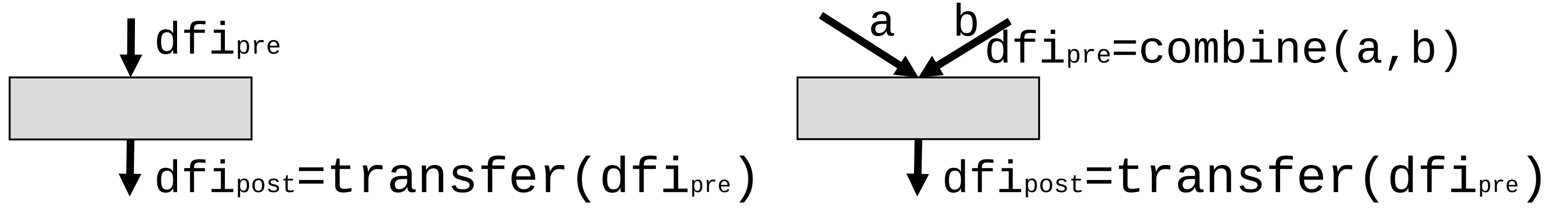
- Program understanding
  - Hard at machine code level
- Traceability
  - Is a branch in the control flow taken or not?
  - Ensure, that a specific variable in the program is not changed by other parts
    - E.g. co-driver airbag module

In this talk

- Interprocedural backward slicing on machine code
- Based on data flow
  - Data dependencies
    - Only flow dependencies required
  - Control dependencies
- Setup
  - Textual representation of a control flow graph + some additional attributes
  - Results of a value analysis
    - Bounds of memory accesses

Data flow analysis/Abstract interpretation

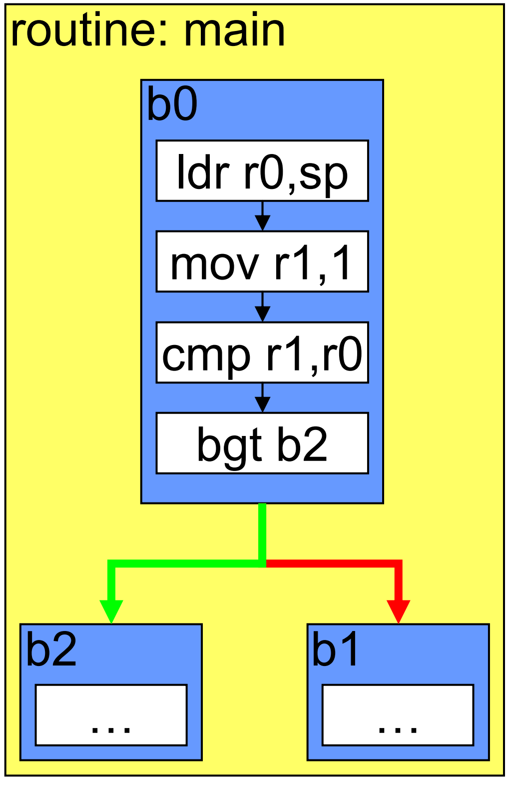
- A technique for collecting run-time information about data in programs without actually executing them
- Based on a control flow graph



- Abstract interpretation: computation on abstract values

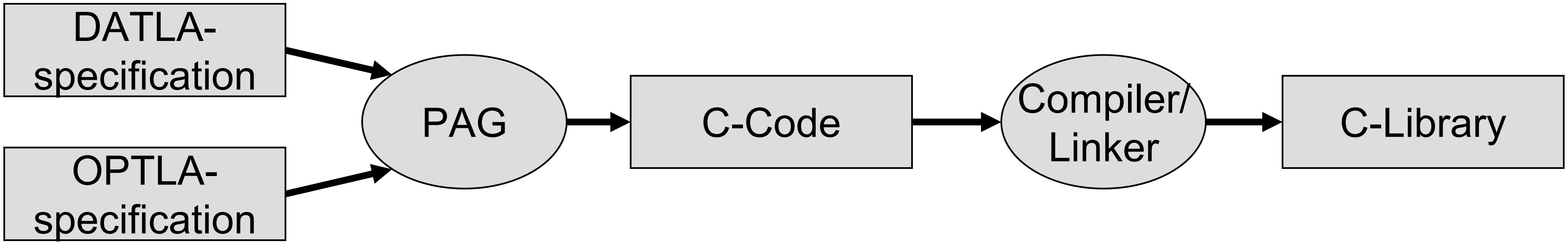
Control Flow Graph & CRL

```
start with main;
routine main: name="main"
{ entry b0: id="b0"
 { edges to b1/f , b2/t;
 contains {
  0x00 "ldr r0,sp,0": src1="al",dst2="
  src3="sp", src4="Mem", guard="always";
  0x04 "mov r1,1": src1="al",dst2="
  src3="1", guard="always";
  0x08 "cmp r1,r0": src1="al",src2="
  src3="r0",dst4="CPSR",
  guard="always";
  0x0C "bgt b2": src1="gt",src2="0x14",
  dst3="PC",guard="conditional";
 }
}
```



Program Analyzer Generator (PAG)

- Analogous to flex and bison
- For generating efficient program analyzers
- CRL frontend available to translate CRL files into data structures describing the CFG
- Interface to read results from other analyses and using them in the current one



Reconstruct data dependencies

- Simple reaching definition analysis
- Hardware-specific problems
  - Handling of composed registers / register aliasing
    - Architectures provide registers or register parts under different names
  - Guarded execution
    - Execution of an instruction depends on a condition
      - Instruction is definitively executed: must-update
      - Instruction may be executed: may-update

Reconstruct control dependencies

- Computed by two consecutive analyses
  - Control point analysis (cp)
    - Forward directed
    - Collecting all nodes with more than one successor
  - Post dominator analysis (pdom)
    - Backward directed
- Combination of the results
  - For a node n, all nodes, n is dependent on, get computed by: $\{m \mid m \in dfi_{cp}(n) : n \notin dfi_{pdom}(m)\}$

Model memory accesses

- Consider the following example with slicing criterion C=(5,R1)

```
M[0] = 1;
M[1] = 2;
R1 = M[0];
R2 = M[1];
R2 = R1 + R2;
M[2] = R2;
```

input program

```

M[1] = 2;
R1 = M[0];

R2 = R1 + R2;
```

wrong slice

```
M[0] = 1;
M[1] = 2;
R1 = M[0];

R2 = R1 + R2;
```

conservative slice

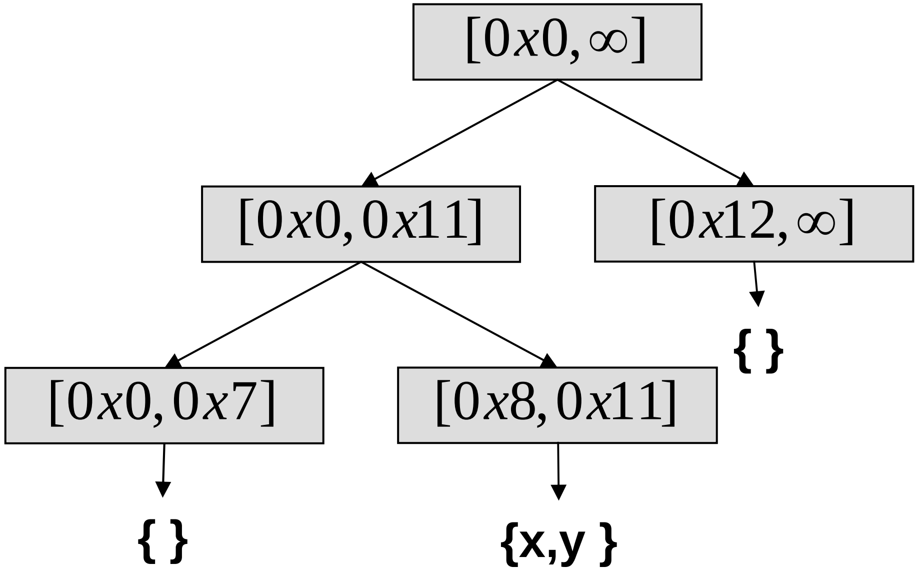
```
M[0] = 1;

R1 = M[0];

R2 = R1 + R2;
```

minimal slice

- Memory consists of many cells
- Partitioning into fixed cells is not possible
- ➢ Dynamic model of memory necessary

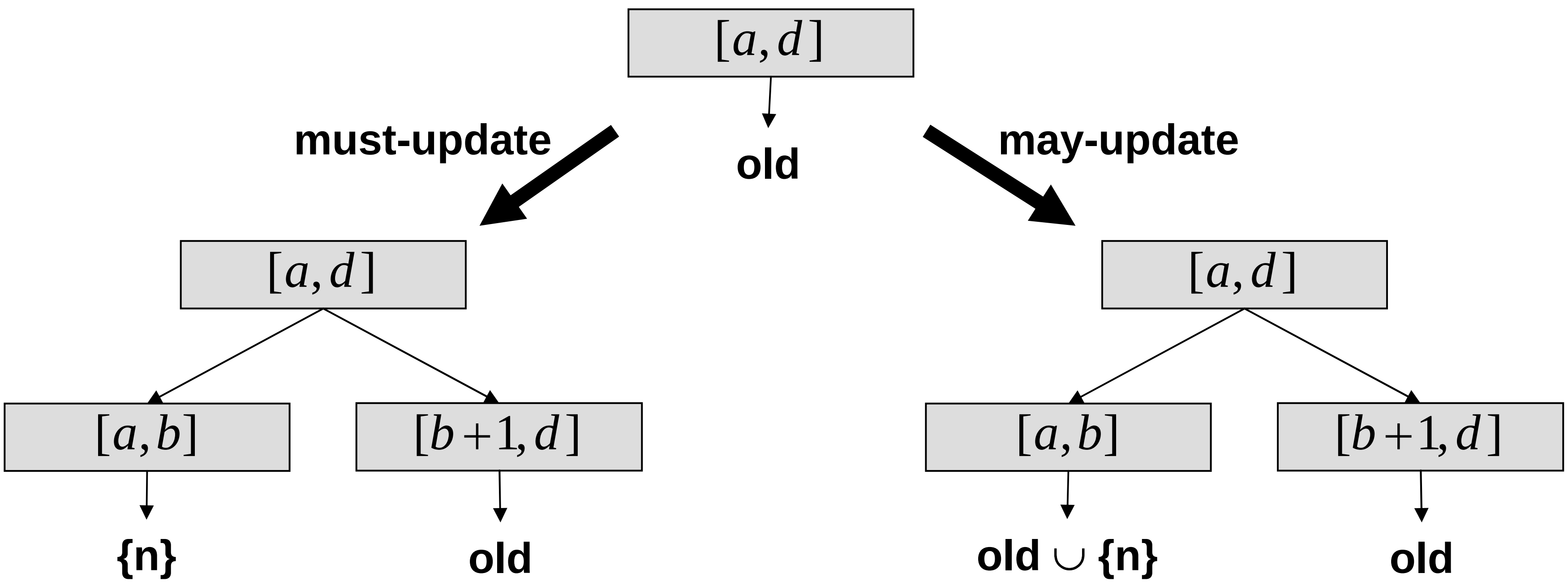
Dynamic memory tree

- Binary tree
- Each inner node consists of an interval and two successors
- Each leaf consists of an interval and a set of program points
- Interval corresponds to bounds of different memory cells
- Start a forward analysis with $([0x0,\infty],\{\})$



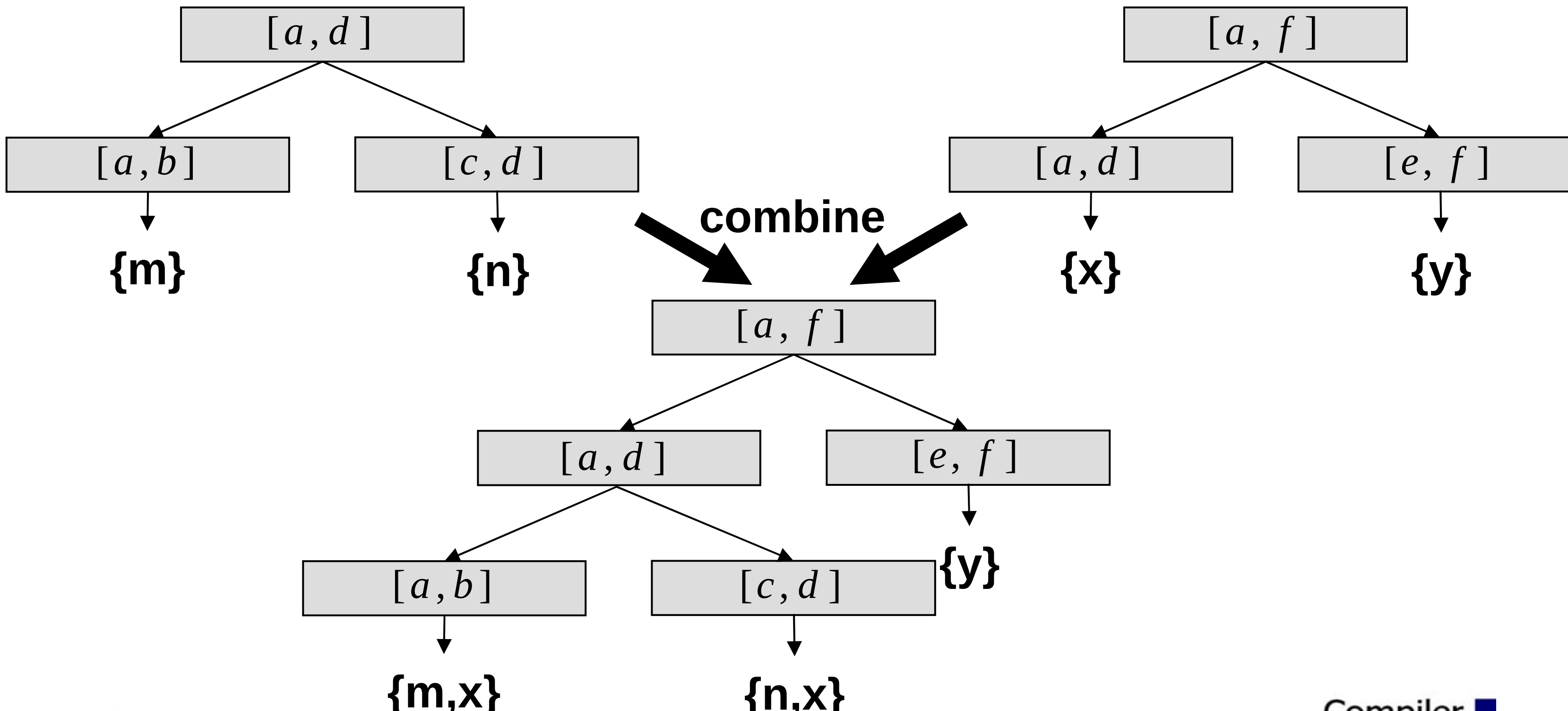
Transfer function

- For instruction n writing to the memory cells [a,b]



Combine function

- Take the leafs of one tree and add it to the second one by doing a may-update



Slicing the program (1)

- Already seen:
  - Reconstruction of data dependencies
  - Reconstruction of control dependencies
  - Dynamic model of the memory system
- Follows next:
  - Using these information to compute approximations to statement-minimal slices

Slicing the program (2)

- Precomputation of data, control and memory dependencies
- while no abort
  - wait for slicing criterion (n,V)
  - workset = Data(n,V) ∪ Mem(n) ∪ Data(Ctrl(n)) ∪ Mem(Ctrl(n))
  - visited = {(n,V)} ∪ Ctrl(n)
  - while workset ≠ { }
    - (x,W) ← workset
    - visited = visited ∪ {(x,W)} ∪ Ctrl(x)
    - workset = workset \ {(x,W)} ∪ { Data(x,W) ∪ Mem(x,W) ∪ Data(Ctrl(x)) ∪ Mem(Ctrl(x)) \ visited }
  - slice = { n | (n,_)∈visited }

Experimental results (1)

- Usability depends on
  - Time for analyzing the executable
    - Depends on the analyzed executable
      - #instructions, #calls, #memory accesses
  - Time for computing a slice
    - Depends on the chosen criterion
  - Preciseness of calculated slices
    - Also depends on the chosen slicing criterion
    - Hard to verify

Experimental results (2)

- Characteristics of some test files

	minmax	fac	prime	dry2_1	st30
routines	4	2	4	17	163
calls	5	2	4	32	309
instructions	114	24	119	773	3820
loops	0	1	2	9	50
loads	4	2	20	296	984
stores	4	2	10	140	877

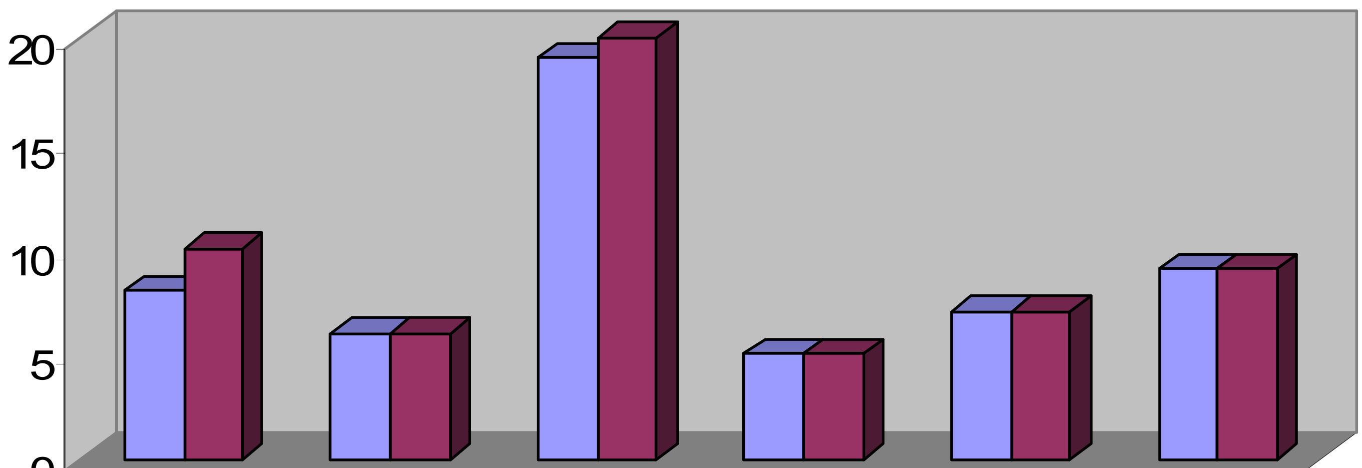
Experimental results (3)

- Measured precomputation times (in seconds)

	minmax	fac	prime	dry2_1	st30
cfg	0.03	0.02	0.03	0.17	0.5
data	0.02	0.03	0.04	0.25	173.2
cp	0.01	0.02	0.02	0.1	0.4
pdom	0.01	0.01	0.01	0.07	0.4
mem	0.01	0.02	0.02	0.32	1.2
$\sum$	0.07	0.1	0.13	0.82	177.5

Experimental results (4)

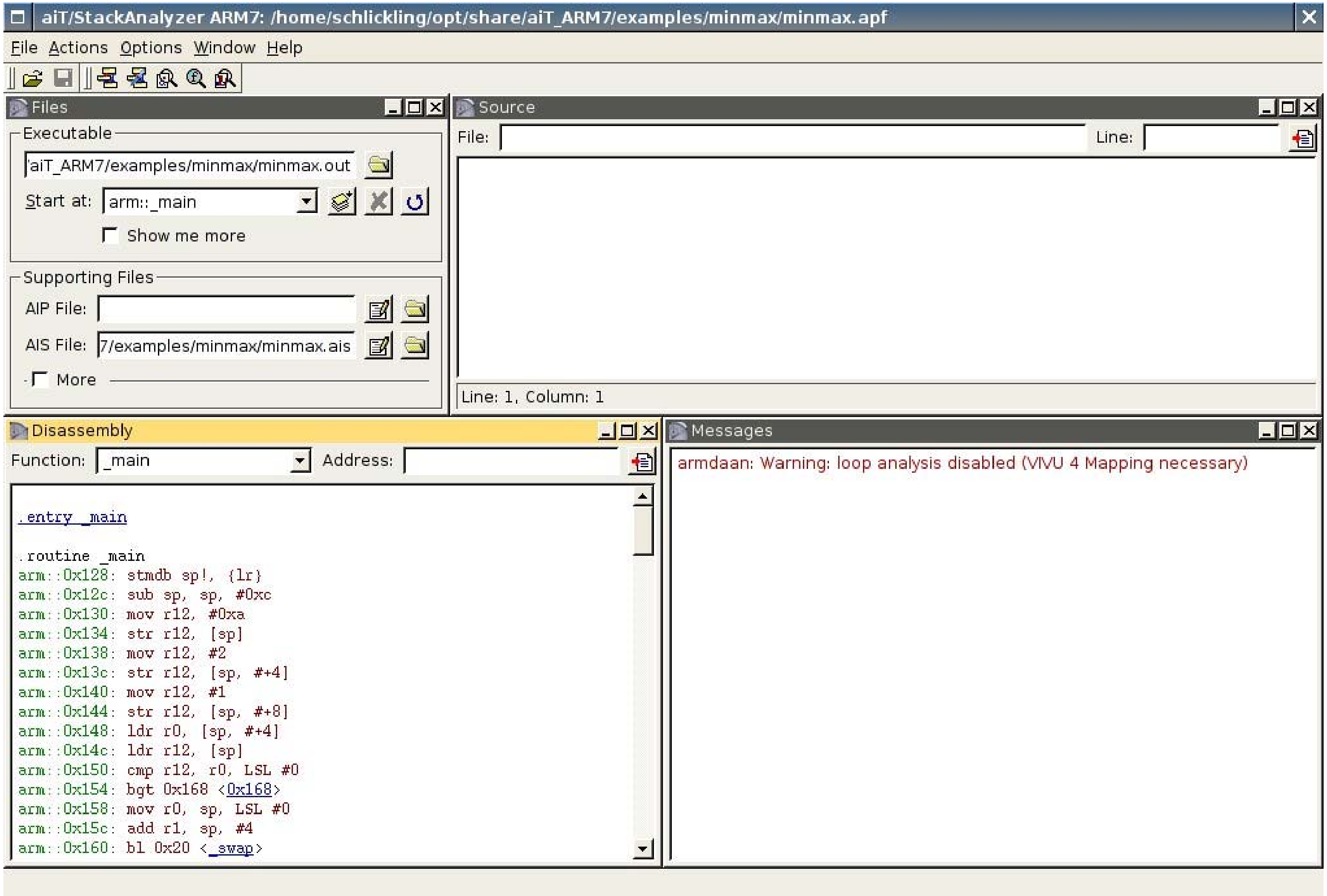
Variation from statement-minimal slices

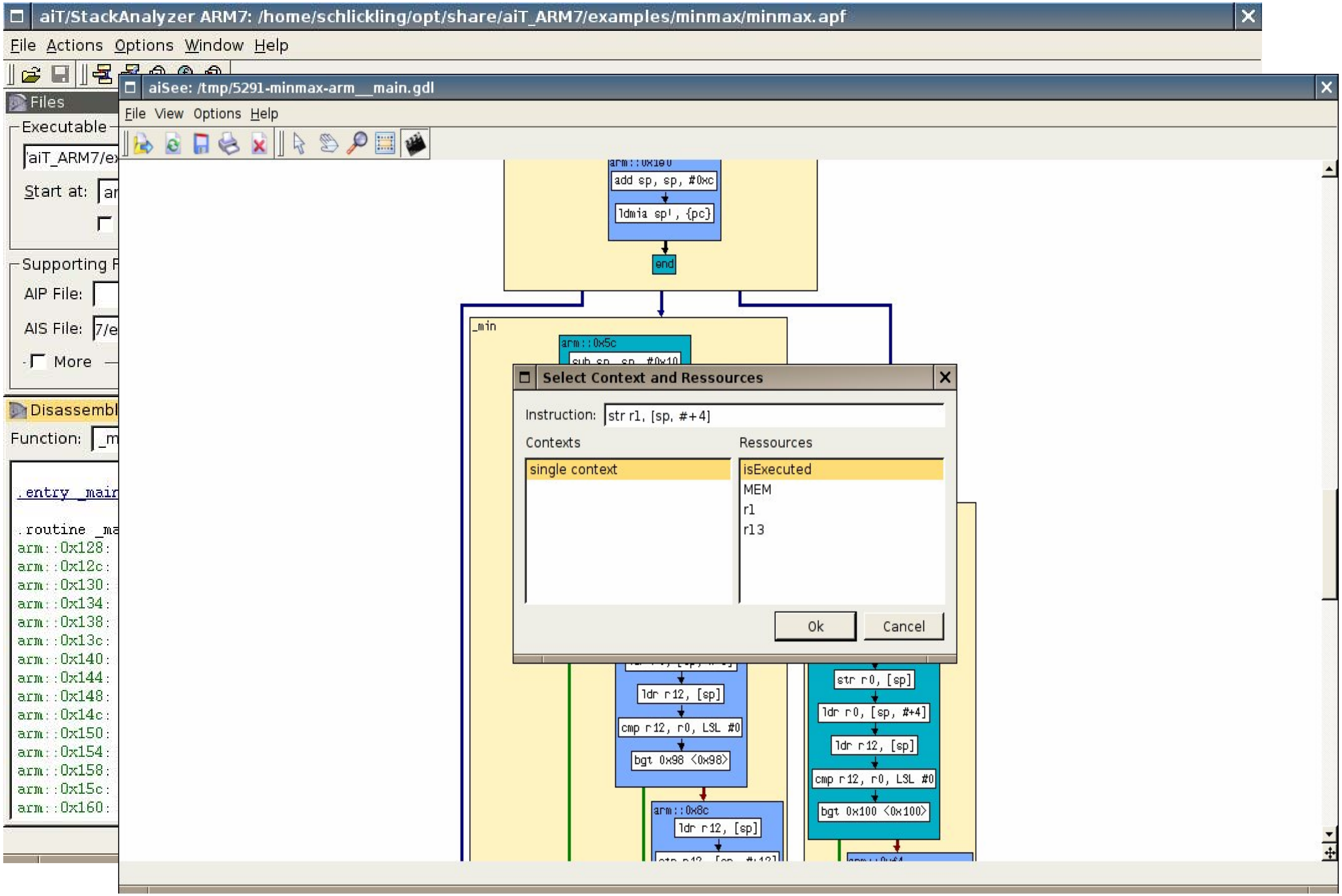


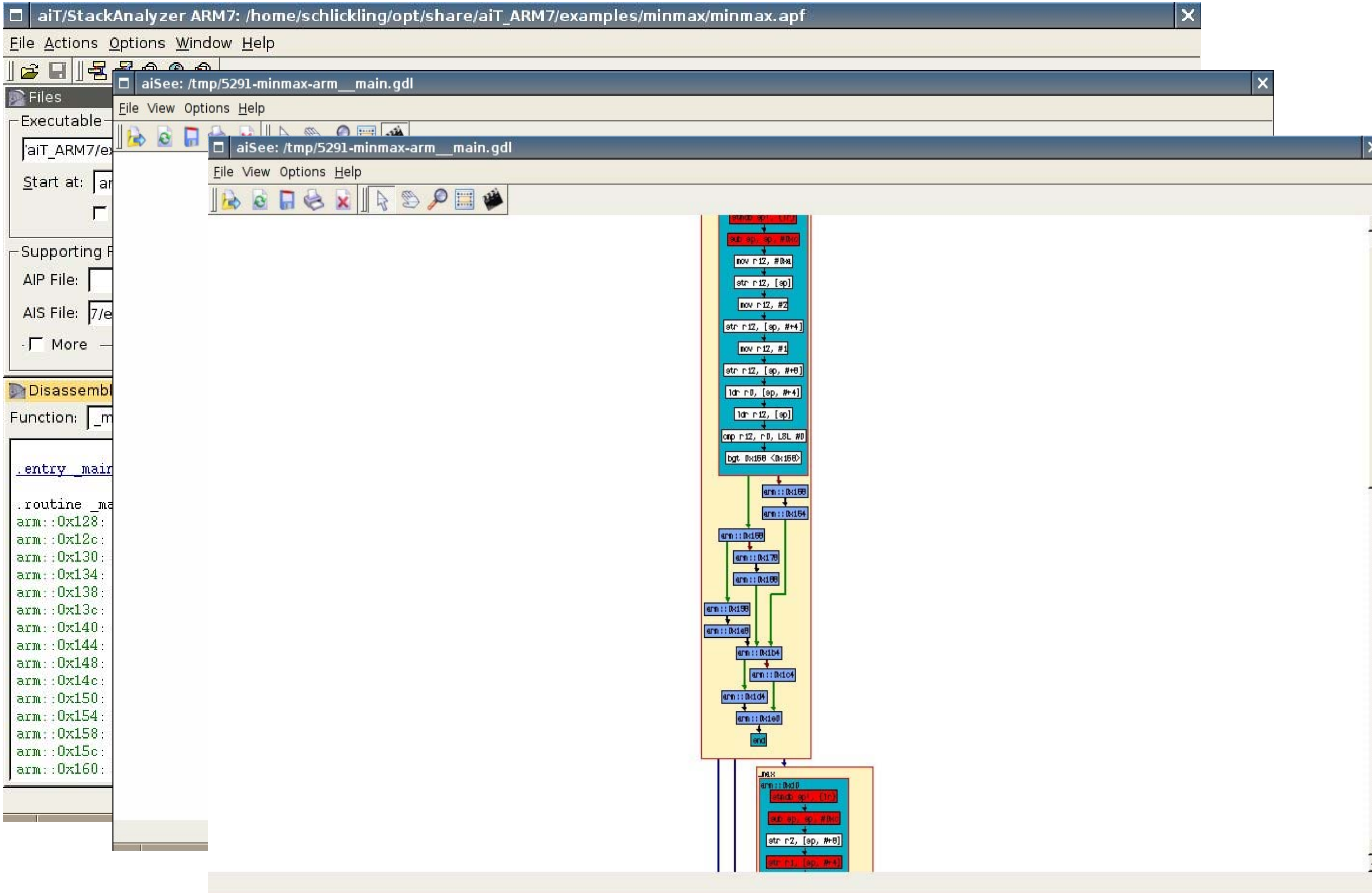
	minmax criterion 1	minmax criterion 2	minmax criterion 3	fac criterion 1	fac criterion 2	fac criterion 3
minmal	8	6	19	5	7	9
computed	10	6	20	5	7	9

Conclusions

- Generic solution for the slicing of executable programs
  - Only a few architecture specific parameters required
    - Registers
    - Register aliases
- Dynamic model of the memory
- Currently support for ARM, PPC5xx and PPC755 Architectures
- Successfully integrated in the aiT framework
  - Interactive visualization with aiSee3


aiT/StackAnalyzer ARM7: /home/schlickling/opt/share/aiT_ARM7/examples/minmax/minmax.apf
File Actions Options Window Help
Files
Executable
'aiT_ARM7/examples/minmax/minmax.out
Start at: arm::_main
Show me more
Supporting Files
AIP File:
AIS File: 7/examples/minmax/minmax.ais
More
Source
File:
Line:
Line: 1, Column: 1
Disassembly
Function: _main
Address:
.entry _main
.routine _main
arm::0x128: stmdb sp!, {lr}
arm::0x12c: sub sp, sp, #0xc
arm::0x130: mov r12, #0xa
arm::0x134: str r12, [sp]
arm::0x138: mov r12, #2
arm::0x13c: str r12, [sp, #+4]
arm::0x140: mov r12, #1
arm::0x144: str r12, [sp, #+8]
arm::0x148: ldr r0, [sp, #+4]
arm::0x14c: ldr r12, [sp]
arm::0x150: cmp r12, r0, LSL #0
arm::0x154: bgt 0x168 <0x168>
arm::0x158: mov r0, sp, LSL #0
arm::0x15c: add r1, sp, #4
arm::0x160: bl 0x20 <_swap>
Messages
armdaan: Warning: loop analysis disabled (VIVU 4 Mapping necessary)


aiT/StackAnalyzer ARM7: /home/schlickling/opt/share/aiT_ARM7/examples/minmax/minmax.apf
File Actions Options Window Help
Files
Executable
Start at:
Supporting F
AIP File:
AIS File:
More
Disassembl
Function:
.entry _main
.routine _ma
arm::0x128:
arm::0x12c:
arm::0x130:
arm::0x134:
arm::0x138:
arm::0x13c:
arm::0x140:
arm::0x144:
arm::0x148:
arm::0x14c:
arm::0x150:
arm::0x154:
arm::0x158:
arm::0x15c:
arm::0x160:
aiSee: /tmp/5291-minmax-arm__main.gdl
File View Options Help
add sp, sp, #0xc
ldmia sp!, {pc}
end
_min
arm::0x5c
Select Context and Ressources
Instruction: str r1, [sp, #+4]
Contexts
Ressources
single context
isExecuted
MEM
r1
r13
Ok
Cancel
ldr r12, [sp]
cmp r12, r0, LSL #0
bgt 0x98 <0x98>
arm::0x8c
ldr r12, [sp]
str r0, [sp]
ldr r0, [sp, #+4]
ldr r12, [sp]
cmp r12, r0, LSL #0
bgt 0x100 <0x100>


aiT/StackAnalyzer ARM7: /home/schlickling/opt/share/aiT_ARM7/examples/minmax/minmax.apf
File Actions Options Window Help
Files
Executable
Start at:
Supporting F
AIP File:
AIS File:
More
Disassembl
Function:
.entry _main
.routine _ma
arm::0x128:
arm::0x12c:
arm::0x130:
arm::0x134:
arm::0x138:
arm::0x13c:
arm::0x140:
arm::0x144:
arm::0x148:
arm::0x14c:
arm::0x150:
arm::0x154:
arm::0x158:
arm::0x15c:
arm::0x160:
aiSee: /tmp/5291-minmax-arm__main.gdl
File View Options Help
aiSee: /tmp/5291-minmax-arm__main.gdl
File View Options Help