stacksize

- What is it all about ?
- Call Graph
- At a glance
- How to use ?
- What can I do ?
- Output
- Problems
- Internals

DU GPF Berlin

Frank Reglin

stacksize: What is it all about?

Motivation

- stack size too big → waste of RAM
- stack size too small → crash

stacksize:

simple call graph analyzer

"Who calls whom ?"

calculates stack consumption

maximum for all call pathes

detects

- entry functions ("start nodes")
- external functions
- recursive functions

stacksize: Call Graph

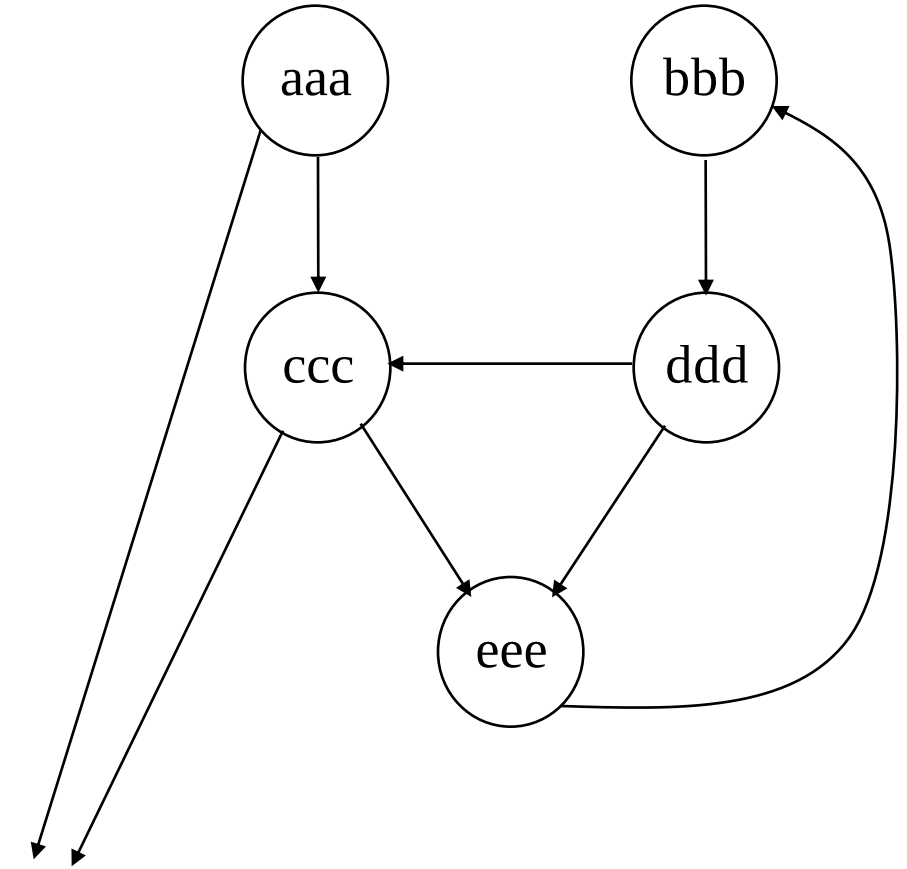
```
aaa()
{
    ccc();
    fff();
}

bbb()
{
    ddd();
}

ccc()
{
    eee();
    fff();
}

ddd()
{
    ccc();
    eee();
}

eee()
{
    bbb();
}
```

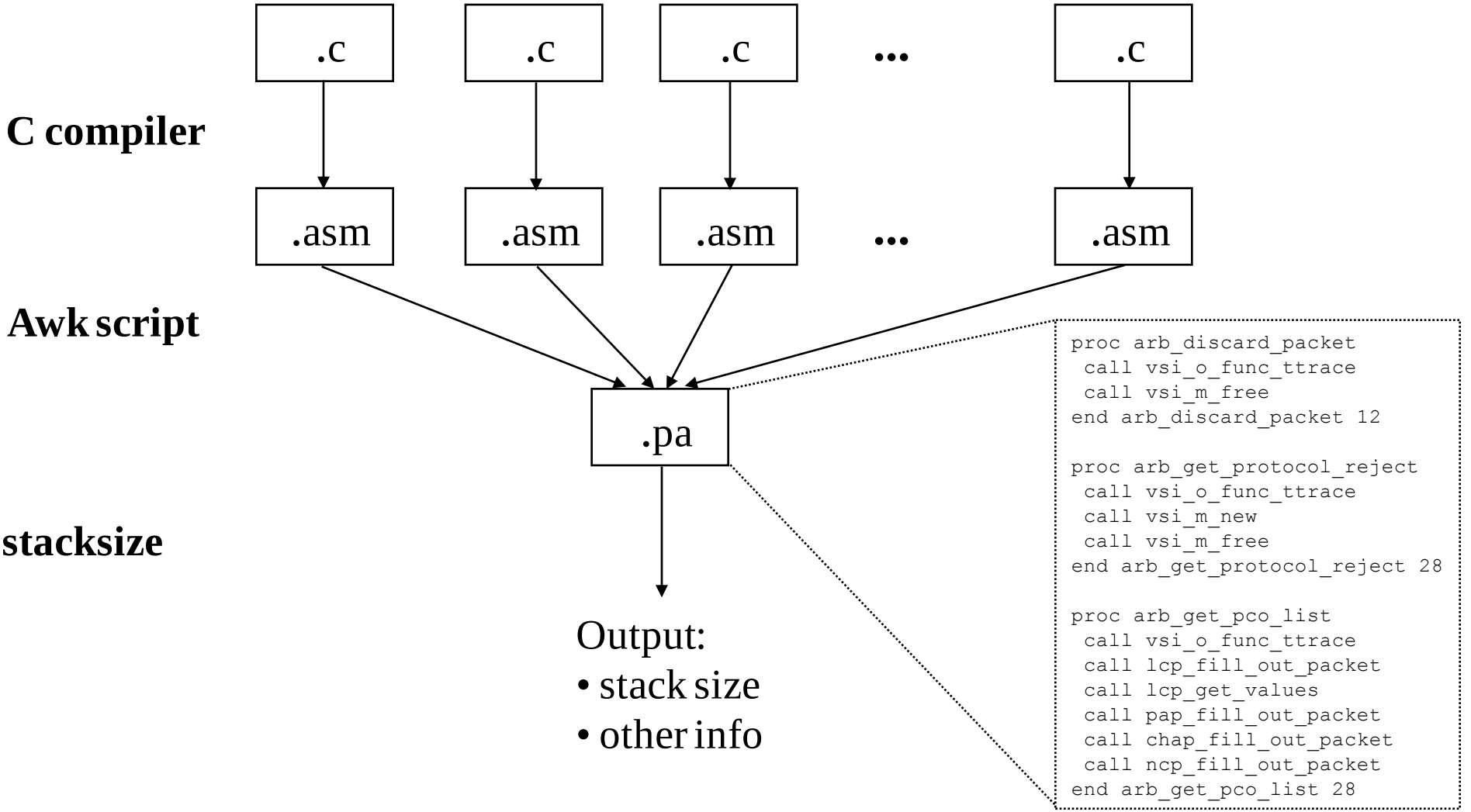


start nodes

recursion

external

stacksize at a glance



stacksize: How to use ?

1. Build the .pa file

1.1 Generate .asm files

edit \g23m\condat\condat_lib_ip.mak :

change `COPT:=  -me -mt -o -pw2 -x  -mw  ...`

into `COPT:= -n -me -mt -o -pw2 -x0 -mw -ea.asm ...`

Assembly output — **No inline functions** — **File extension**

`cd \g23m` prepare g23.inf as usual

`perl g23.pl`

You will get link errors (because no .obj files generated).

The .asm files are in the source directories.

1.2 Generate .pa file

```
cd condat\ms\src\abcde
gawk -f \gpf\bin\cg_na7.awk *.asm > abcde.pa
```

2. Analyze

```
stacksize [Options] abcde.pa
```

stacksize: What can I do ?

Information provided by stacksize examples ! *options can be combined*

stack consumption

- maximum `stacksize [-m] abcde.pa`
- function `stacksize [-m] -f aaa abcde.pa`

functions

- all `stacksize -a abcde.pa`
- start nodes `stacksize -s abcde.pa`
- end notes `stacksize -e abcde.pa`
- external `stacksize -x abcde.pa`
- recursive `stacksize -r abcde.pa`

call graph

- complete `stacksize -g [-x] [-l] abcde.pa`
- max stack `stacksize -p abcde.pa`
- function `stacksize -{g|p} -f aaa abcde.pa`
- recursion `stacksize -g abcde.pa | grep recursive`

stacksize: Output

stack consumption

```
>stacksize -m abcde.pa
104
```

max total stack consumption (bytes)

functions

```
>stacksize -a abcde.pa
                sum   this   refs   calls
start           100     44      0       2  aaa
start           104      8      0       1  bbb
                 56     48      2       2  ccc
                 96     40      1       2  ddd
       end        8      8      2       0  eee
external          0      0      2       0  fff
```

classification

sum = total stack consumption for this function (bytes)

this = local stack size for this function (bytes)

refs = number of references to this function

calls = number of functions called by this function

function name

call graph

```
>stacksize -g abcde.pa
call graph:
.  100 aaa    44 ==>
       .    56 ccc    48 ==>
              .     8 eee     8 end
.  104 bbb     8 ==>
       .    96 ddd    40 ==>
              .    56 ccc    48 ==>
                     ... (see above)
              .     8 eee     8 end
```

. ***sum*** function name ***this*** classification

. . ***called function***

stacksize: Problems

Recursion

- stacksize calculates 1 loop (without closing)

Variable number of arguments

stack consumption for calls with a variable number of arguments (printf ...) cannot be calculated

- add an offset

Interrupts

interrupt service routines save registers on the stack of the current task

- add an offset

Function pointers

- calls via pointer are not detected
- called functions are normally classified as ‘start nodes’
- you may insert calls in the .pa file

Doubly defined names

same function name in different source files:

- stacksize uses the most stack consuming function (+ issues a warning)

stacksize Internals

Data structures

- generic binary tree ⇨ sbt.h
- call graph node PROCDATA

Execution steps

1. **parse command line**
2. **read .pa file**
3. **initialize binary tree and call graph** dummy function 'm$'
4. **process .pa file**
   - proc initialize global call table
   - call store call in global call table
   - end eliminate double entries in call table
     build PROCDATA
5. **make externals** name only → PROCDATA
6. **analyze** classification, stack size
   - recursive through binary tree
   - recursive through call graph
7. **get global max** let m$ 'call' all start nodes
8. **output**

stacksize Internals 2

Recursion detection

Problem: Which functions are recursive ?

Example: p → q → r → s → q

⇗

Recursion is detected for q.
But r and s are recursive, too.

Principle:

- mark path of recursive descent (BEING_PROCESSED) to detect recursion
- mark recursive functions for later processing (PROREC)
- propagate PROREC mark back on return from the descent and replace it by RECURSIVE

```
void analyze( NODE * node )  -- extract --
{
   procdata = node->data;
   if( procdata->flags & BEING_PROCESSED )
   {
      procdata->flags &= ~BEING_PROCESSED;
      procdata->flags |= PROREC | ANALYZED;
      return;
   }
   procdata->flags |= BEING_PROCESSED;
   for( i = 0; i < procdata->ncalls; i++ )
   {
      analyze( procdata->call[i] );

      subprocdata = procdata->call[i]->data;
      if( subprocdata->flags & PROREC )
      {
         subprocdata->flags &= ~PROREC;
         subprocdata->flags |= RECURSIVE;
         if( !(procdata->flags & RECURSIVE) )
            procdata->flags |= PROREC;
      }
   }
   procdata->flags &= ~BEING_PROCESSED;
}
```