Troubleshooting a Large Erlang System

Mats Cronqvist
Ericsson Hungary

Analyzing the title

- **Erlang**
  A programming language
- **System**
  AXD 301
- **Large**
  ~2.1 million lines of Erlang
  ~300 coders (cumulative)
- **Troubleshooting**
  What kind of errors
  When do we find them
  How do we find them

Erlang, the vision

(This bit stolen from Mike Williams, EUC 2003)

- **Concurrent/Distributed**
  - Thousands of simultaneous transactions
  - Many computers
  - Many OS's
- **No Down Time (99.999% availability)**
  - Recovery from hardware and software errors
  - Enable adding/removing hardware at run time
  - Update code in running systems
- **"Ease of Programming"**
  - Highly "expressive" programming language
  - Large scale development (100's of programmers)
  - Debugging and tracing - even at customer sites
  - Easy to fix bugs (patch) and upgrade at all phases

Erlang, the reality

(I.e. the AXD 301)

- **Concurrent/Distributed**
  - Tens of thousands of calls, few thousand Erlang processes
  - 2-20 CPU's (running Erlang)
  - One OS (solaris)
- **No Down Time (99.999% availability)**
  - Resilient against hardware failure
  - Replacing failed hardware at run time is routine
  - Updating code in running systems is routine
- **"Ease of Programming"**
  - The language really is highly productive
  - Cumulative 300 programmers, virtually all complete beginners
  - Tracing at live sites is (luckily) not routine, but happens often enough
  - Ability to patch lab systems without having to restart is priceless

The System

AXD 301 Description

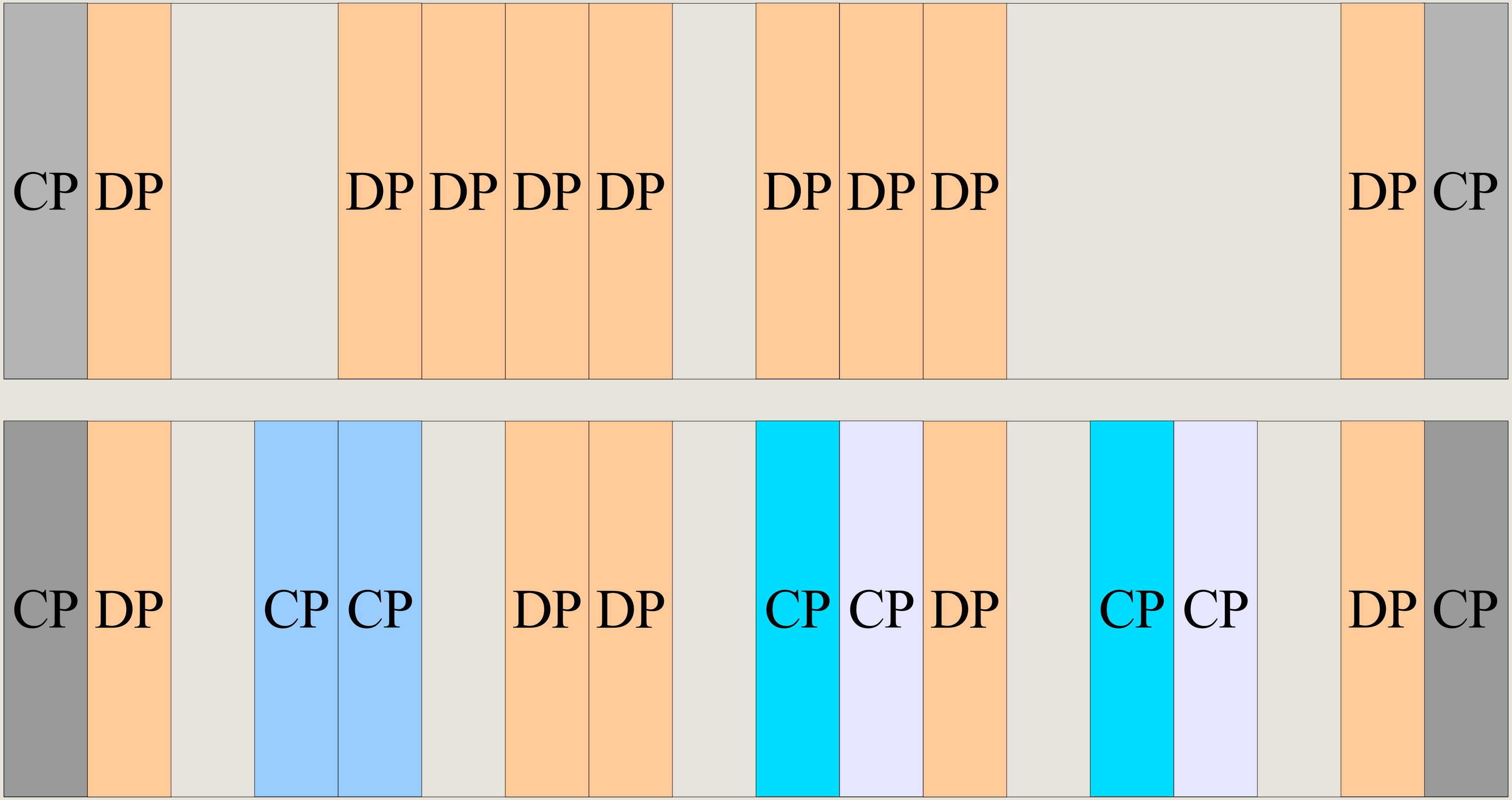
- subracks (typically 1-4)
  - Central Processors (typically 2-4)
    - traffic control, configuration and administration
  - Device Processors (typically ~10)
    - handles the physical interfaces(Ethernet, SONET...)
    -

CP's are paired with active and a standby roles

Applications on the active CP can run with hot or warm standby.

AXD 301

highly schematic



CP DP DP DP DP DP DP DP DP DP CP

CP DP CP CP DP DP CP CP DP CP CP DP CP

Development Process

- Block test
  - On workstation, other blocks stubbed

From this point on bugs should be logged in Trouble Reports

- Function test / System test
  - Real AXD 301 hardware in the lab
- Network integration
  - Joint testing with other products in the lab
- Deployment
  - At customer premises

Development Process

Errors found

- Block test
  - Calls to non-existing functions, typos, malformed pattern matches...
- Function test / System test
  - API bugs, race conditions, wrong context, typos...
- Network integration
  - Timing problems, scalability problems, interworking problems...
- Deployment
  - Handling errors, hardware problems, sourced C code...

Block Test Errors

Block testing can be considered part of the design stage.

Objective is to verify basic functionality.

Ideally, design is still flexible.

Typical errors found;

- Calls to non-existing functions
- Typos
- Malformed pattern matches

Many of these could have been found by a type-checking compiler

However, the ability to run the code "before it's ready" is valuable

- Morale
- Design flexibility

Function Test / System Test Errors

TR statistics, ~150 studied (work in progress)

- API bugs
- race conditions
- wrong context
- typos

Surprisingly, almost no "typing" errors

Problems are typically

- misunderstandings (of the API or the functionality)
- concurrency related (race conditions, context related)
- typos

Network Integration Errors

TR's not studied yet.

Experience shows that the major problems are

- Timing problems
- Scalability problems
- Interworking problems

None of these are Erlang specific

My personal experience shows that problems are often identified in the AXD 301 because of the superior tracing

Deployed System Errors

Interviews with 3rd line support suggests major areas are

- Handling errors
- Hardware problems
- Sourced C code

Erlang bugs are fairly rare (further investigation needed)

Errors
how do we find them

- xref (axdref)
  - finds unresolved function calls
- runtime logging

```
** exited: {undef,[{ets,insert,[1]},
                   {example,undef,0},
                   ...]} **
```

- performance meter (eperf)
  - overall system status
- top (dtop)
  - Erlang machine status
- the trace BIF (pan, dbg)
  - debugging, profiling

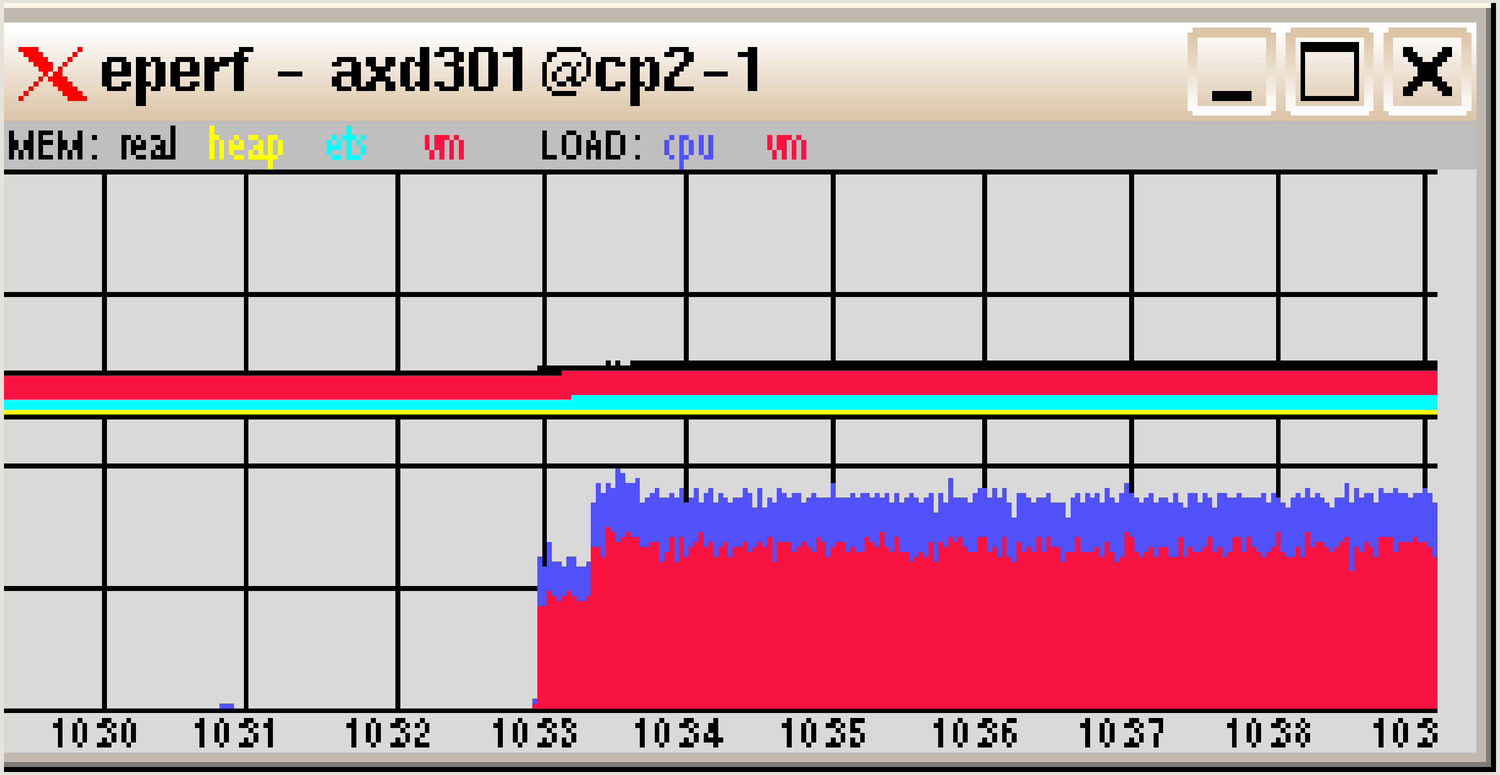
Performance

- Profiling
  - The system is potentially adequately fast
  - It can easily be made very slow
  - It has good support for profiling

eperf

overall system status (CPU load and memory)

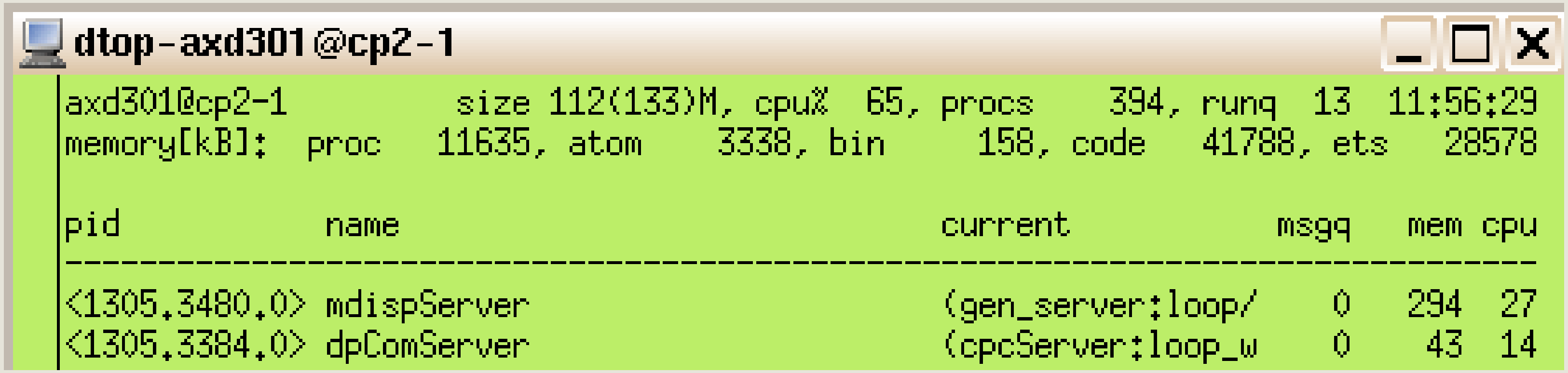
very cheap (< 1 % extra load)



dtop

Uses erlang:process_info and erlang:system_info

- what's going on?
- what process is doing it?



```
axd301@cp2-1         size 112(133)M, cpu%  65,  procs    394, runq  13  11:56:29
memory[kB]:  proc   11635, atom    3338, bin     158, code   41788, ets    28578

pid           name                                  current           msgq   mem cpu
--------------------------------------------------------------------------------
<1305.3480.0> mdispServer                           (gen_server:loop/    0   294  27
<1305.3384.0> dpComServer                           (cpcServer:loop_w    0    43  14
```

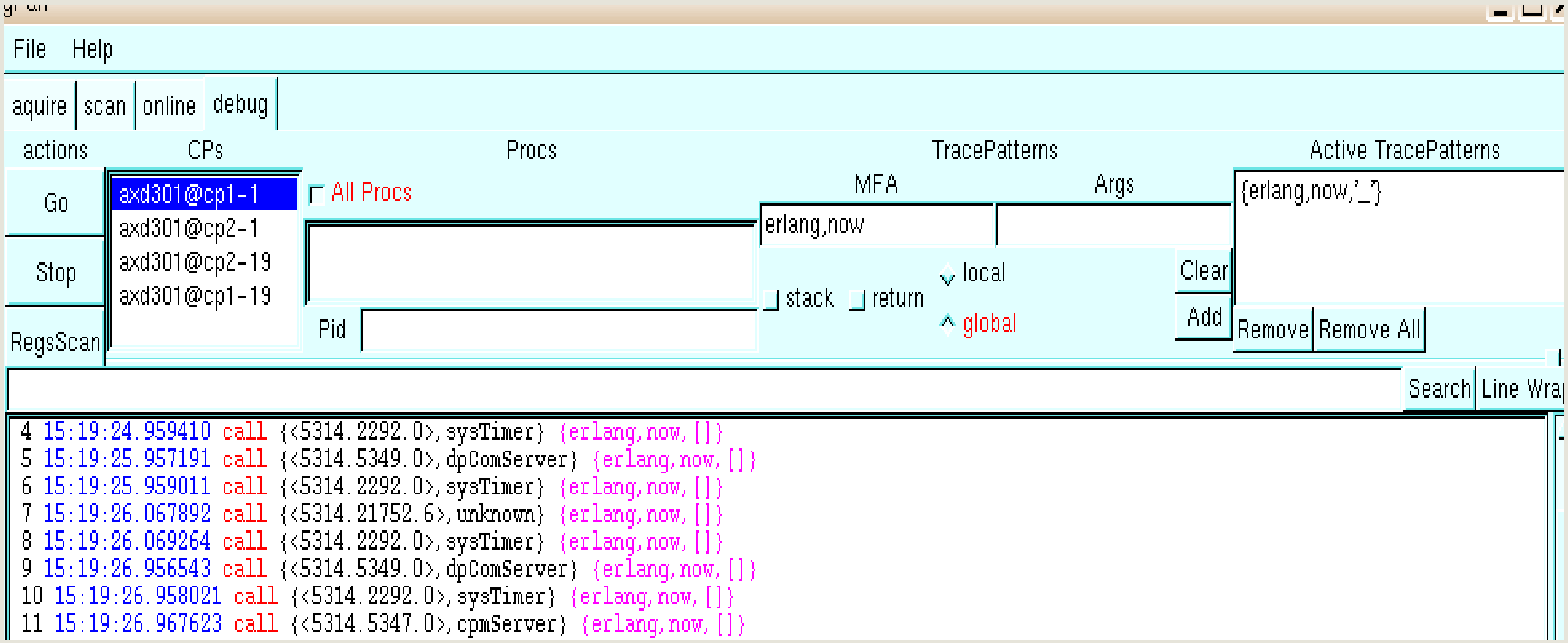
pan

Interface to the erlang:trace BIF

- debugging
  - similar to dbg
- profiling
  - process level
  - function level

pan debugger

Interface to the erlang:trace BIF



MFA: erlang,now

Active TracePatterns: {erlang,now,'_'}

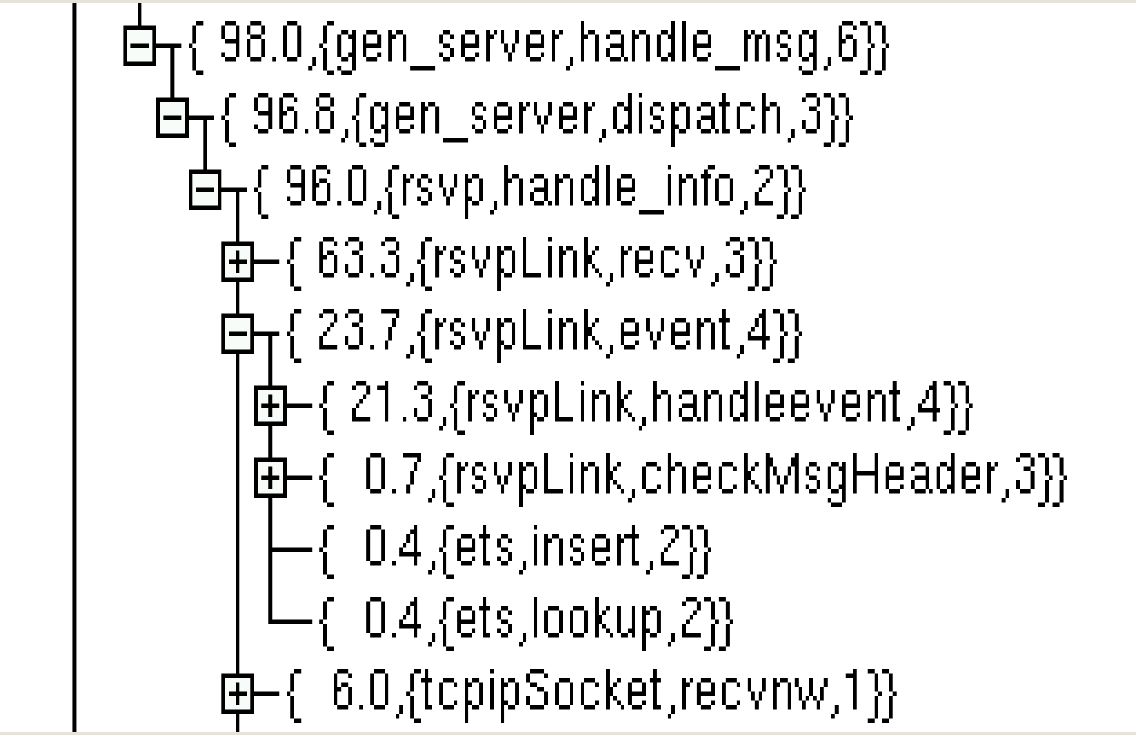
```
4 15:19:24.959410 call {<5314.2292.0>,sysTimer} {erlang,now,[]}
5 15:19:25.957191 call {<5314.5349.0>,dpComServer} {erlang,now,[]}
6 15:19:25.959011 call {<5314.2292.0>,sysTimer} {erlang,now,[]}
7 15:19:26.067892 call {<5314.21752.6>,unknown} {erlang,now,[]}
8 15:19:26.069264 call {<5314.2292.0>,sysTimer} {erlang,now,[]}
9 15:19:26.956543 call {<5314.5349.0>,dpComServer} {erlang,now,[]}
10 15:19:26.958021 call {<5314.2292.0>,sysTimer} {erlang,now,[]}
11 15:19:26.967623 call {<5314.5347.0>,cpmServer} {erlang,now,[]}
```

pan perf

```
pid           id                gc     in       cpu
================================================
<122.5440.0>  dpComServer     73    849    548900
<122.5574.0>  plcMemory        0    170    192779
<122.5435.0>  cpmServer        7     58    119229
<122.5569.0>  sbm              3     69     68211
38            {sysTimer,do_    0     75     13055
5             {inet_tcp_dis   10     39     11257
2             {pthTcpNetHan    8     36      8757
<122.6819.0>  pthTcpCh2       16     28      8296
<122.6229.0>  {jive_broker,    5     15      7948
<122.6818.0>  pthTcpOm1       16     27      7494
<122.6778.0>  pthOm            6     31      7431
2             {sysTimer,do_    4     16      4083
<122.6781.0>  pthOmTimer       1     22      2784
<122.17.0>    net_kernel       0      7      1697
<122.10.0>    rex              1      6      1037
```

pan prof

Interface to the erlang:trace BIF



```
{ 98.0,{gen_server,handle_msg,6}}
 { 96.8,{gen_server,dispatch,3}}
  { 96.0,{rsvp,handle_info,2}}
   { 63.3,{rsvpLink,recv,3}}
   { 23.7,{rsvpLink,event,4}}
    { 21.3,{rsvpLink,handleevent,4}}
    {  0.7,{rsvpLink,checkMsgHeader,3}}
    {  0.4,{ets,insert,2}}
    {  0.4,{ets,lookup,2}}
   {  6.0,{tcpipSocket,recvnw,1}}
```

Summary

- it works
- it's not inherently slow
- dynamic typing is not unsafe
- the support for profiling and debugging is excellent
- the short debug cycle is good for morale
- informational crashes means we find the rare bugs

Eric S. Raymond on Python

"[Accepting] the debugging overhead of buffer overruns, pointer-aliasing problems, malloc/free memory leaks and all the other associated ills is just crazy on today's machines. Far better to trade a few cycles and a few kilobytes of memory for the overhead of a scripting language's memory manager and economize on far more valuable human time.“

http://www.linuxjournal.com/article.php?sid=3882