What run-time services could help scientific programming?

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge
Contrariwise...
Some difficulties of software

- performance!
- correctness!
Some difficulties of software

- performance! → optimise! parallelise!
- correctness! → test! verify!
Some difficulties of software

- performance! $\rightarrow$ optimise! parallelise!
- correctness! $\rightarrow$ test! verify!

... but doing *anything* with software is hard!

- “my code crashes in a way I don’t understand”
- “my code has a bug but I can’t see what’s going on”
- “I’d like to use your code but . . .”
- “my code behaves differently today / for you / . . .”
What to expect from this talk

Mostly show-and-tell…

- some things I’m working on
- some other things I think are interesting
- … in the rough space of run-time systems

Not-so-hidden messages

- these are not really hot issues in CS research
- … do they resonate with scientific programmers?

Let’s keep it informal…
$ cc -o myprog myprog.c
$ ./myprog
Segmentation fault
$ cc -o myprog myprog.c
$ ./myprog
Segmentation fault

... versus

$ crunchcc -o myprog myprog.c
$ LD_PRELOAD=libcrunch.so ./myprog
myprog: Failed check __is_a(0x5a1220, <int$32>)
at 0x40dade (myprog.c:400);
allocation was a heap block of <__PTR_int$32>
originating at 0x40daa1 (myprog.c:56)
Segmentation fault
void *buf = malloc(n * sizeof(int *));

fill_buf(buf);  // thinks buf is an array of int*
use_buf(buf);   // thinks buf is an array of int

(gdb) ptype *buf       # compiler-supplied approximation
    type = void

(gdb) ptttype *buf     # my whizzy new capability
    rtttype = int*

Adding *run-time observability* is no easy feat

- not just C!
- ... the cycle continues
Why care?

- C is (still) not going away
- C and Unix are the cornerstone of the universe

Things I am currently caring about

- cleaner error reporting
- mixing \{Python, JavaScript, \ldots\} \leftrightarrow \{C, Fortran, C++\}
- type information / “descriptive plane”
- making compilers work for, not against…
To: stephen.kell@cl.cam.ac.uk
Subject: Debugging with Cling

Hi Stephen. I came across your recent blog post about debugging...

For numerical algorithm development a fully interactive environment is crucial.

Most of the engineering world uses Matlab.... However, Matlab has limitations.

I recently came across Cling from CERN, an interactive C++ environment...

However, it does not currently support breakpoints(!).

I am looking into adding this functionality...

and it occurred to me that you would great insight into the problem.
$ ./cling

-------------------- CLING -----------------------
* Type C++ code and press enter to run it *
* Type .q to exit  *
--------------------

[cling]$ #include <stdio.h>
[cling]$ printf("Hello, world!\n");
Hello, world!

Advert: I am working on a very similar system where

- the scripting language is JavaScript (or, er, Python)
- the scripted objects can be C / C++ / Fortran / …
“Cling is

- an interactive C++ interpreter...
- built on the top of LLVM and Clang libraries...

“Once we start Cling it automatically

- includes several header files ...
- ... and its own runtime universe.
- Thus creates a minimal environment for user to start.”

Start over! New universe!
Travel between universes is expensive.

Occam: can we avoid creating them?

This means

- have interpreter mechanism
- *sitting alongside*…
- … compiled code
- … existing debugger
Stephen’s great insight (2)

We can do it! But we have to

- imagine (and build!) new infrastructure
- OS run-time and debug mechanisms are inadequate!

Want to evolve OS-level services to

- have a richer run-time model of code and data
- support more dynamic operations
- *within*, not *on top of*, the C–Unix cornerstone

(should sound familiar!)
“my code behaves differently today / for you / …”

Computation is fundamentally about dependency

- the answer depends on the input

  me@here$ ./meaning <universe
  42

  It can depend on a lot of things…

  you@there$ ./meaning <universe
  41
  Segmentation fault
Controlled variation

The behaviour of a program depends on lots of variables

- input, environment, scheduling, …

Why can’t computers help us *control that variation*?

- keep track of that input *over an execution*
- e.g. record a file that describes *all input that went in*
- how feasible is this?
- bonus points: human-readable!
  - *see* and *edit* that input for later runs
First step: replay debugging

you@there$ ./meaning <universe # "live execution"
41
Segmentation fault

Given a suitable *record* tool, generate & send a *replay log*...

me@here$ replay-it <log # "replayed exec."
41
Segmentation fault

Many uses!

- during development: bug reproduction
- “science”: distributing reproducible behaviour
State of the art: \texttt{rr}

\begin{verbatim}
$ rr --record ./meaning <universe # record input
$ rr --replay # replay from recording
\end{verbatim}

Problem solved? Not quite:

- opaque to [some] tools
- can’t record all processes/executions
- logs are fairly large, albeit redundant
- \( \rightarrow \) definitely not “on by default”

Pushing the service into the OS could help a lot

- ask me!
- on my “projects I would like to work on” pile
Conclusion: quality software takes effort, but…

… better infrastructure can reduce that effort.

- helping test / verify / optimise is one kind
- helping the poor human: observe, understand, repeat

Some ideas under exploration:

- richer descriptive plane at run time
- debugging across the interpreted–compiled continuum
- effective mixed-language programming
- executions as a first-class object

Thanks for listening! Any remaining questions?
Record all “input”, a.k.a. sources of nondeterminism

- input data ($\approx$ outputs of system calls)
  - ptrace() system calls

- thread scheduling & descheduling points
  - more ptrace()
  - clever sampling of perf counters

- memory interleaving [, weak memory effects...]
  - serialize the program!
  - i.e. no solution (needs hw changes)
Cling’s problems

- run-time mechanisms don’t embrace interpretation
- ... debugger has to (quirkily) reimplement source lang
- → Cling can’t “plug in” to existing process state
- ... nor debugged by a normal debugger

Ideally want

- “get a REPL on” any process, any time!
- process can be “empty” or mid-execution

Observability has to come by default, not as an afterthought

- make sure the universe’s mechanisms are adequate
- embrace the spectrum of interpreted ↔ compiled
- I’m (side-project-)working on it! ask me
<table>
<thead>
<tr>
<th>benchmark</th>
<th>normal/s</th>
<th>crunch</th>
<th>nopreload</th>
<th>just allocs</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>5.05</td>
<td>+0 %</td>
<td>+0%</td>
<td>+0%</td>
</tr>
<tr>
<td>mcf</td>
<td>2.49</td>
<td>+6.8%</td>
<td>−1%</td>
<td>+0%</td>
</tr>
<tr>
<td>milc</td>
<td>8.75</td>
<td>+38 %</td>
<td>+2%</td>
<td>−1%</td>
</tr>
<tr>
<td>gobmk</td>
<td>14.5</td>
<td>+13 %</td>
<td>+1%</td>
<td>+1%</td>
</tr>
<tr>
<td>hammmer</td>
<td>2.13</td>
<td>+8.5%</td>
<td>+8%</td>
<td>+0%</td>
</tr>
<tr>
<td>sjeng</td>
<td>3.25</td>
<td>−2.2%</td>
<td>−2%</td>
<td>+0%</td>
</tr>
<tr>
<td>h264ref</td>
<td>10.0</td>
<td>+5 %</td>
<td>+5%</td>
<td>+1%</td>
</tr>
<tr>
<td>lbm</td>
<td>3.43</td>
<td>+24 %</td>
<td>+0%</td>
<td>+0%</td>
</tr>
<tr>
<td>sphinx3</td>
<td>1.58</td>
<td>+15 %</td>
<td>+2%</td>
<td>+4%</td>
</tr>
<tr>
<td>gcc</td>
<td>0.989</td>
<td>+289 %</td>
<td>−</td>
<td>+4%</td>
</tr>
<tr>
<td>perlbench</td>
<td>1.48</td>
<td>+≈300 %</td>
<td>−</td>
<td>+3%</td>
</tr>
</tbody>
</table>