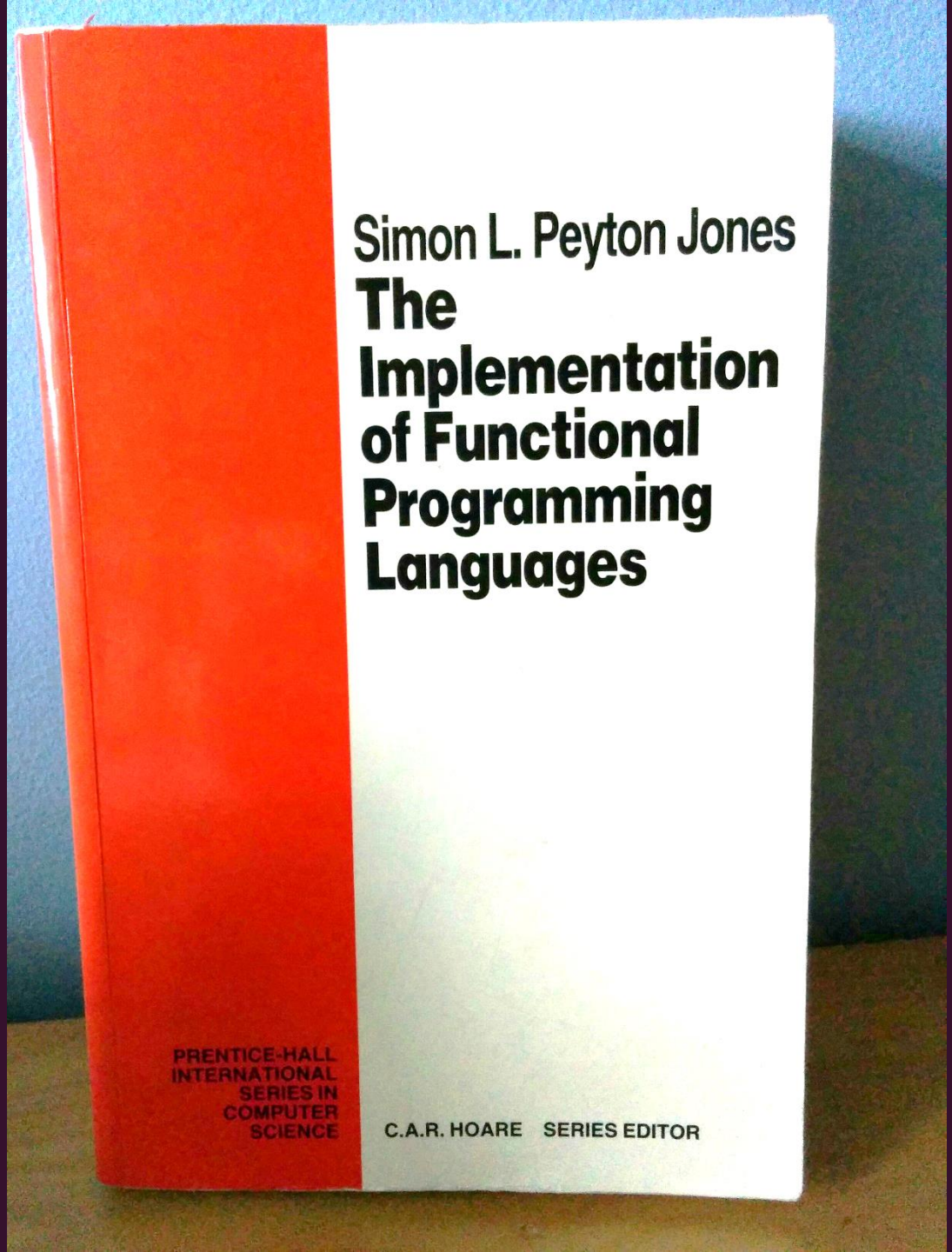
FP Implementation

Simon Marlow

PLMW'15


Simon L. Peyton Jones
The Implementation of Functional Programming Languages
PRENTICE-HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE
C.A.R. HOARE SERIES EDITOR


File Tools Selection Patterns Font Style
PAINT
B:\RAINFALL.IMG
ABC
NONE
JAN FEB MAR APR MAY JUN JLY
AMSTRAD
PC-CM
AMSTRAD System Unit
PC 512 DD
PC

```
I x = x
K x y = x
S x y z = x z (y z)
```

SKI in GNU make

```
I  = $1

S  = (PAP_S2,$1)
S2 = (PAP_S3,$1,$2)
S3 = $(call ap,$(call ap,$3,$1),$(call ap,$2,$1))

K  = (PAP_K2,$1)
K2 = $2

define ap
$(strip \
 $(if $(findstring $(dollar),$1$2),\
   $(dollar)$(lpar)call ap$(comma)$1$(comma)$2$(rpar),\
   $(if $(findstring >>$(lpar)PAP,>>$1),\
     $(subst $(space)<-TMPSPACE,$(comma),\
      $(patsubst >>$(lpar)PAP_%,$(dollar)$(lpar)call %$(comma)$2,\
         >>$(subst $(comma),$(space)<-TMPSPACE,$1))),\
     $(dollar)$(lpar)call $1$(comma)$2$(rpar))))
endef
```

```
include ski.mk

# The example from Simon Peyton Jones' "The Implementation of
# Functional Languages", pp 263.
#
#  (\x. (+) x x) 5
#
# translates to
#
#   S (S (K +) I) I 5
#
test=$(call ap,$(call ap,$(call ap,S,$(call ap,$(call ap,S,$(call
ap,K,plus)),I)),I),5)

$(info before: $(test))
$(eval $(call reduce,$(test)))
$(info result: $(result))

all :
```

```
$ make -f test1.mk
before: $(call ap,$(call ap,$(call ap,S,$(call ap,$(call
ap,S,$(call K,plus)),I)),I),5)
reduce: $(call ap,$(call ap,$(call ap,S,$(call ap,$(call
S,(PAP_K2,plus)),I)),I),5)
reduce: $(call ap,$(call ap,$(call ap,S,$(call
S2,I,(PAP_K2,plus))),I),5)
reduce: $(call ap,$(call ap,$(call
S,(PAP_S3,I,(PAP_K2,plus))),I),5)
reduce: $(call ap,$(call S2,I,(PAP_S3,I,(PAP_K2,plus))),5)
reduce: $(call S3,5,I,(PAP_S3,I,(PAP_K2,plus)))
reduce: $(call ap,$(call S3,5,I,(PAP_K2,plus)),$(call I,5))
reduce: $(call ap,$(call ap,$(call K2,5,plus),$(call I,5)),5)
reduce: $(call ap,$(call plus,5),5)
reduce: $(call plus2,5,5)
reduce: 10
result: 10
make: Nothing to be done for `all'.
```

But seriously...

- SKI is pure magic and a lot of fun
- But really we want to compile functional languages to machine code

Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine *
Version 2.5

Simon L Peyton Jones
Department of Computing Science, University of Glasgow G12 8QQ
simonpj@dcs.glasgow.ac.uk

July 9, 1992

Abstract

The Spineless Tagless G-machine is an abstract machine designed to support non-strict higher-order functional languages. This presentation of the machine falls into three parts. Firstly, we give a general discussion of the design issues involved in implementing non-strict functional languages.

Next, we present the *STG language*, an austere but recognisably-functional language, which as well as a *denotational* meaning has a well-defined *operational* semantics. The STG language is the "abstract machine code" for the Spineless Tagless G-machine.

Lastly, we discuss the mapping of the STG language onto stock hardware. The success of an abstract machine model depends largely on how efficient this mapping can be made, though this topic is often relegated to a short section. Instead, we give a detailed discussion of the design issues and the choices we have made. Our principal target is the C language,

Evolution of GHC's Backend

- We knew we wanted to plug into an existing retargettable code generator
- But gcc was too hard to pull apart
- So... generate C code
  - With a multitude of hacks
  - Including running a 5000-line Perl script on the generated assembly code ("the dreaded mangler")
- Later:
  - Write a native backend
- Even later:
  - LLVM

What about the evaluation model?

- STG did not survive in its original form

Making a Fast Curry: Push/Enter vs. Eval/Apply for Higher-order Languages

Simon Marlow and Simon Peyton Jones
Microsoft Research, Cambridge

Abstract

Higher-order languages that encourage currying are implemented using one of two basic evaluation models: push/enter or eval/apply. Implementors use their intuition and qualitative judgements to choose one model or the other.

Our goal in this paper is to provide, for the first time, a more substantial basis for this choice, based on our qualitative and quantitative experience of implementing both models in a state-of-the-art compiler for Haskell.

Our conclusion is simple, and contradicts our initial intuition: compiled implementations should use eval/apply.

And the tags were added back…

Faster Laziness Using Dynamic Pointer Tagging

Simon Marlow
Microsoft Research
simonmar@microsoft.com

Alexey Rodriguez Yakushev
University of Utrecht, The Netherlands
alexey@cs.uu.nl

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Abstract

In the light of evidence that Haskell programs compiled by GHC exhibit large numbers of mispredicted branches on modern processors, we re-examine the "tagless" aspect of the STG-machine that GHC uses as its evaluation model.

We propose two tagging strategies: a simple strategy called semi-tagging that seeks to avoid one common source of unpredictable indirect jumps, and a more complex strategy called dynamic pointer-tagging that uses the spare low bits in a pointer to encode information about the pointed-to object. Both of these strategies have been implemented and exhaustively measured in the context of a production compiler, GHC, and the paper contains detailed descriptions of the implementations. Our measurements demonstrate significant performance improvements (14% for dynamic pointer-tagging with only a 2% increase in code size), and we further demonstrate that much of the improvement can be attributed to the elimination of mispredicted branch instructions.

As part of our investigations we also discovered that one optimisation in the STG-machine, vectored-returns, is no longer worthwhile and we explain why.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; D.3.4 [*Programming Languages*]: Processors—Code Gen-

```
f x y = case x of (a,b) -> a+y
```

In a lazy language, before taking `x` apart the compiler must ensure that it is evaluated. So it generates code to push onto the stack a continuation to compute `a+y`, and jumps to the *entry code* for `x`. The first field of every heap closure is its entry code, and jumping to this code is called *entering* the closure. The entry code for an unevaluated closure will evaluate the closure and return its value to the continuation; an already-evaluated closure will return immediately.

In contrast, in a tag-ful approach, the closure to evaluate is entered only if it is found to be not yet evaluated. Here the generated code performs an extra test on the closure type – the tag – to determine its evaluatedness and to avoid entering it unnecessarily.

The tagless scheme is attractive because the code to evaluate a closure is simple and uniform: any closure can be evaluated simply by entering it. But this uniformity comes at the expense of performing indirect jumps, one to enter the closure and another to return to the evaluation site. These indirect jumps are particularly expensive on a modern processor architecture, because they fox the branch-prediction hardware, leading to a stall of 10 or more cycles depending on the length of the pipeline.

If the closure is unevaluated, then we really do have to take an indirect jump to its entry code. However, if it happens to be evaluated

Stepping back a little bit…

Build an artefact that people want to use... and give it away.

if your research is relevant,
people will want to use your thing

Good things happen…

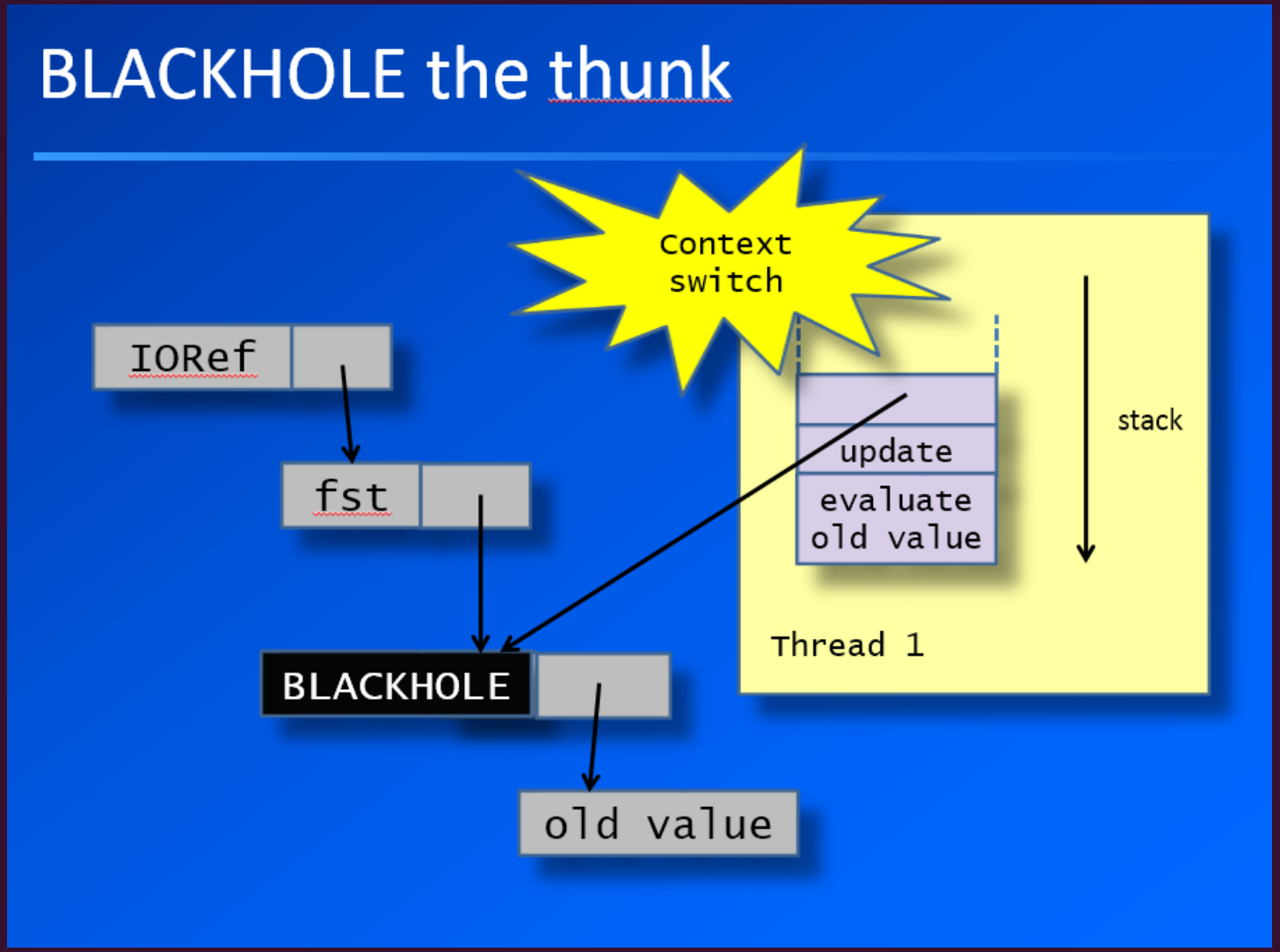
- Easy for others to
  - reproduce your results
  - improve on them
- Users will
  - report bugs
  - suggest improvements
  - *actually help*
  - worth devoting time to fostering a community

User feedback leads to further research opportunities

An innocuous email…

```
From: Johan Tibell <johan.tibell@gmail.com>
Date: Sat, 23 Jan 2010 20:00:59 +0100
Subject: Battling interesting performance bug

If you run ./benchmarks/thread-delay -n 20000 it
sometimes takes orders of magnitude longer to
finish. I haven't quite yet figured out what's
going on. I can't reproduce the problem with
profiling on ...
```

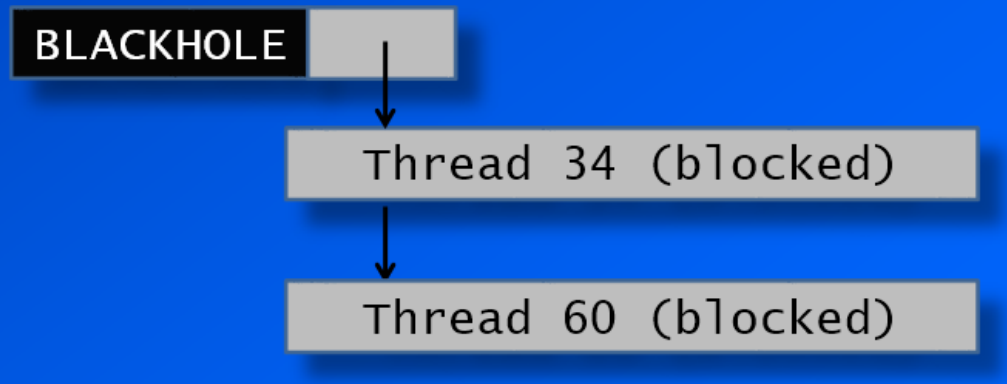
Leads to interesting problems



And interesting solutions

How does blocking work?

- It would be great if we could attach the blocked threads to the BLACKHOLE:



- because then we could easily find all the blocked threads to wake them up

- People using your system in anger will discover things that you didn’t
- Fixing the problems improves things for everyone

Nurture your users

- For they are a source of
  - ideas
  - bug reports
  - feedback
  - patches
  - research collaborations
- Not all users are appreciative

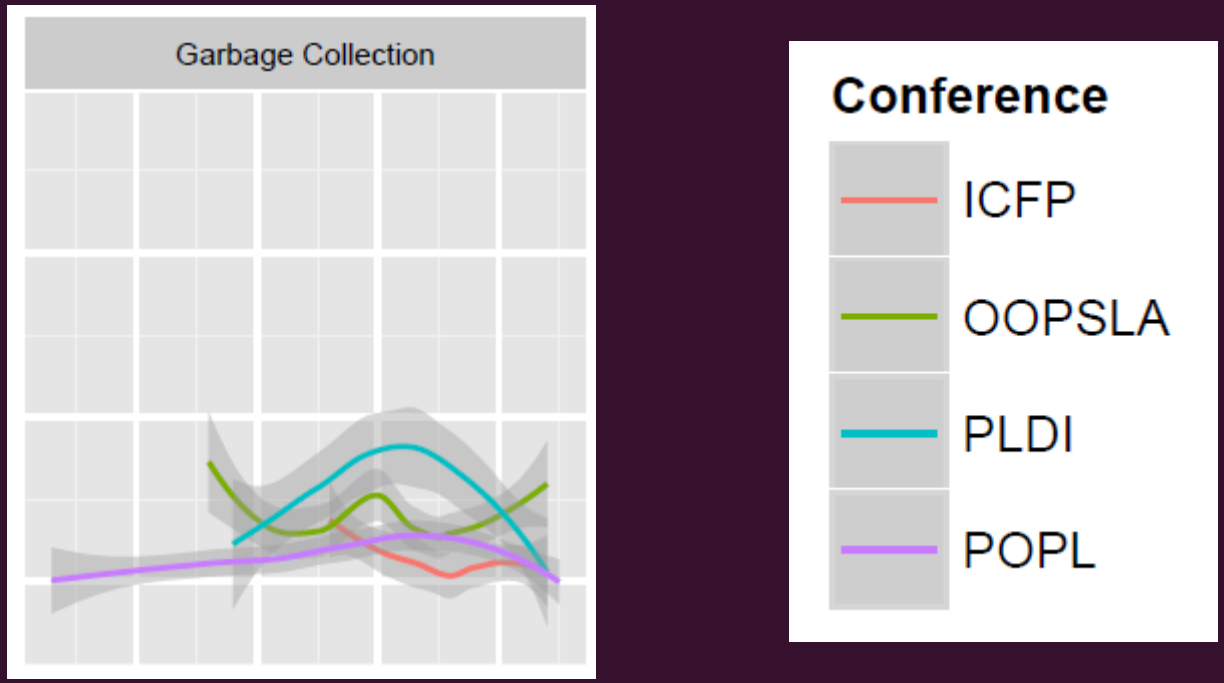
```
Date: Fri, 10 Mar 2006 08:54:33 +0100
To: glasgow-haskell-bugs@haskell.org

Are you fully nerd???? How can ghc expect an installed ghc
for the first build stage????? What is that???? ...
```

Getting back to specific areas

Memory management

- i.e. garbage collection
- Anecdotally, GC is the biggest perf issue in deployed managed-language systems

GC is not a solved problem



- From *Tracking the Flow of Ideas through the Programming Languages Literature* (Michael Greenberg, Kathleen Fisher, and David Walker)

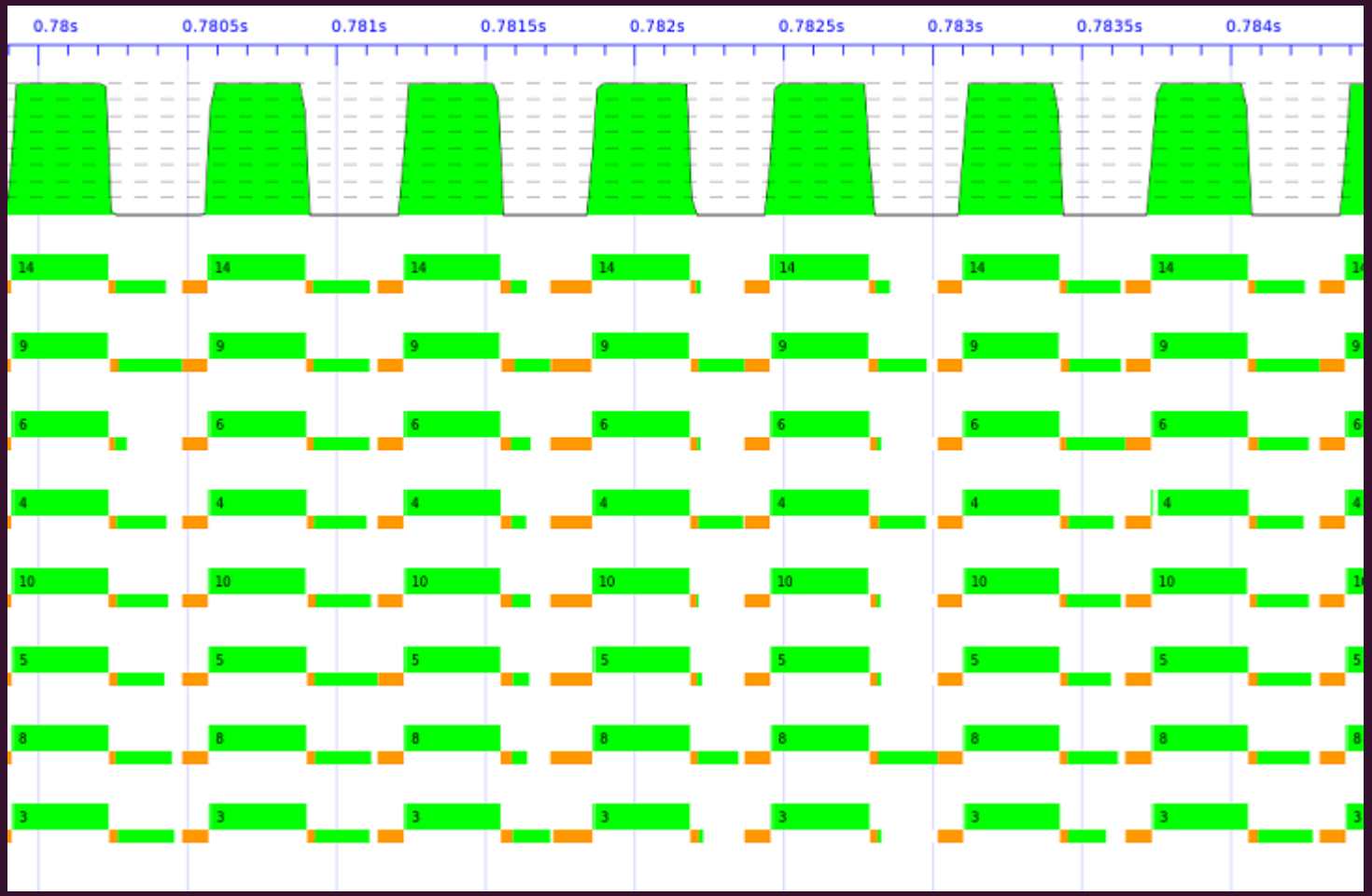
GC

- broad and complex design space
- many tradeoffs along different axes:
  - memory vs. time
  - latency guarantee vs. throughput
- language design affects GC design
  - mutation frequency affects barrier design
  - JVM not optimised for functional languages
- Parallelism adds a whole new dimension
- But hard to publish in major venues

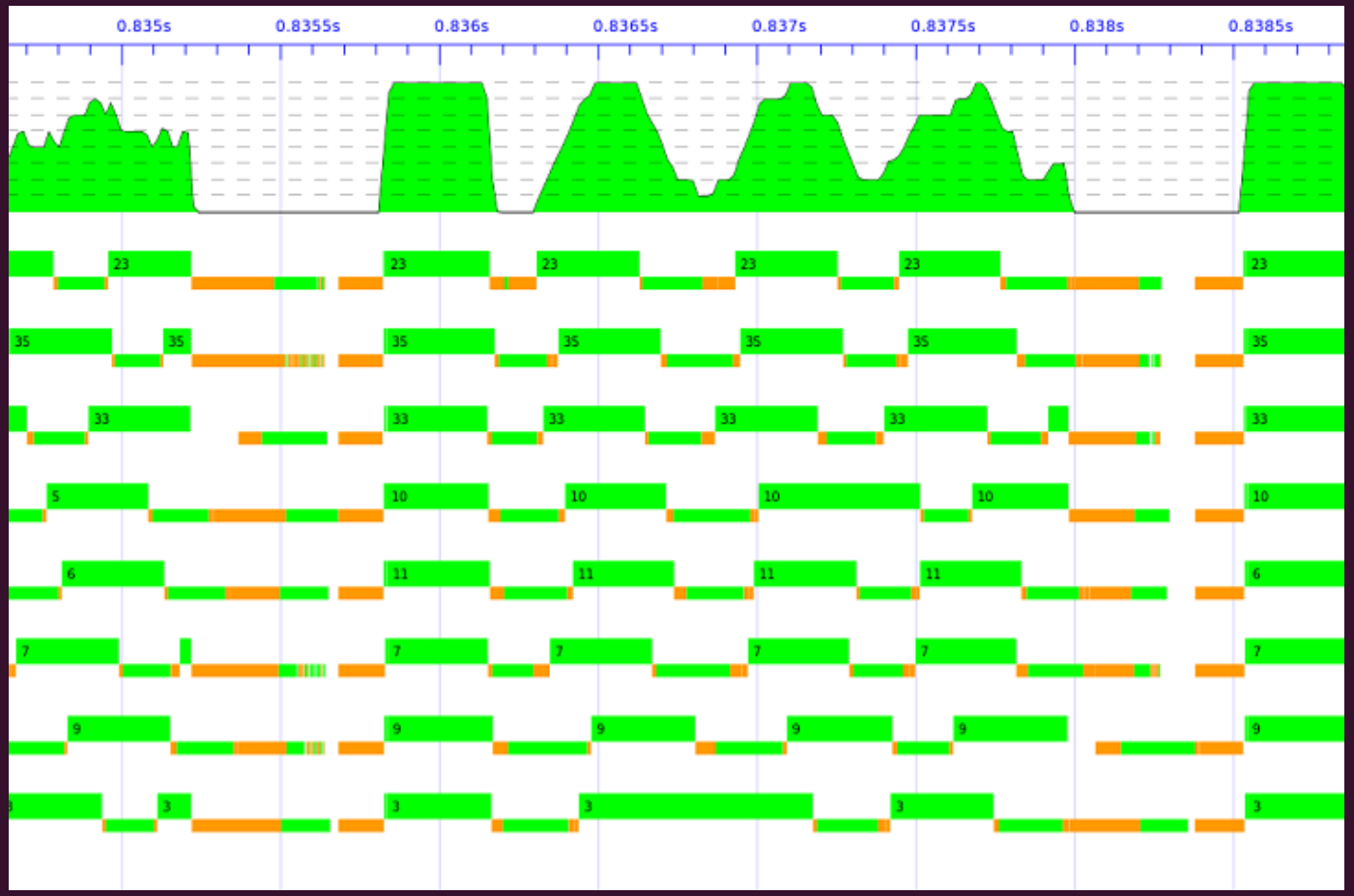
GHC's garbage collector

- GHC has always used a custom GC
- Began with standard copying + mark/sweep
- Generational
  - thunks = mutation!
- Rewrote with a block layer
  - GC'd memory is chains of blocks rather than contiguous
  - much more flexible: heap can resize
  - managing large objects is easy
  - arbitrary number of generations, aging

2008: parallel GC

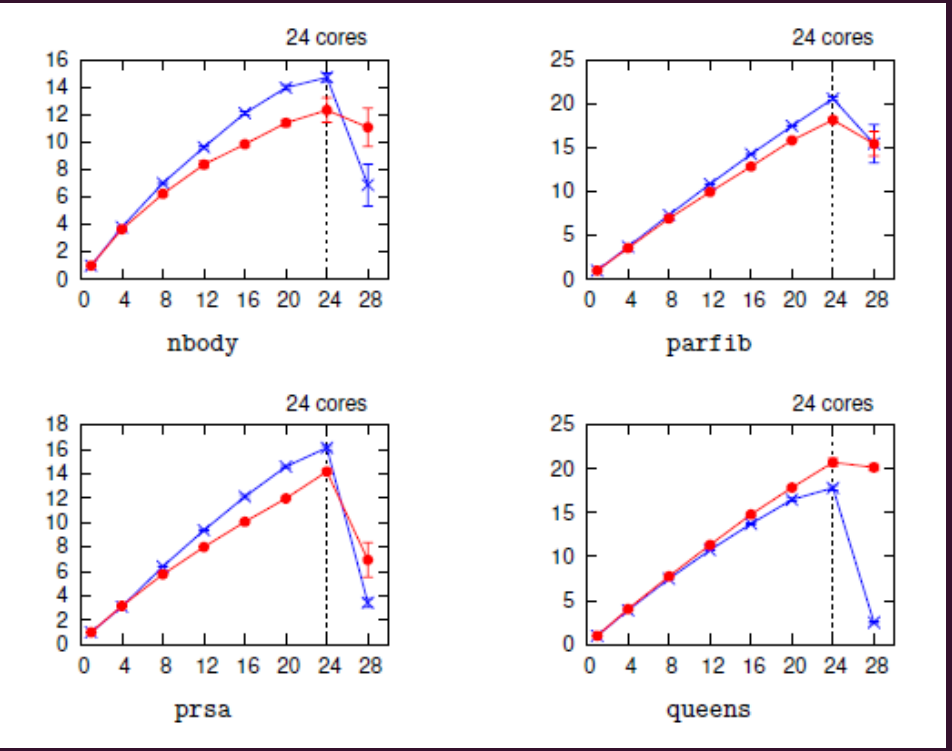


2010: local GC



Local GC, retrospective

- Results were mixed
- Hugely complicated
- Not ready for production
- One of the rare times we published something but it didn’t go into GHC
- Still an important area for research



Benchmarking

- Proper benchmarking and measurement is essential for evaluating compiler and runtime techniques
- Systems today are incredibly complex and non-deterministic
  - e.g. a different version of the linker might arrange code differently in memory, causing differences in cache behaviour
  - Need rigour and good statistical methods
- Computer scientists are (often) bad at statistics

Quantifying Performance Changes with Effect Size Confidence Intervals

TOMAS KALIBERA, RICHARD JONES, University of Kent

Measuring performance and quantifying a performance change are core evaluation techniques in programming language and systems research. Out of 122 recent scientific papers published at PLDI, ASPLOS, ISMM, TOPLAS, and TACO, as many as 65 included experimental evaluation that quantified a performance change using a ratio of execution times. Unfortunately, few of these papers evaluated their results with the level of rigour that has come to be expected in other experimental sciences. The uncertainty of measured results was largely ignored. Scarcely any of the papers mentioned uncertainty in the ratio of the mean execution times, and most did not even mention uncertainty in the two means themselves. Furthermore, most of the papers failed to address the non-deterministic execution of computer programs (caused by factors such as memory placement, for example), and none addressed non-deterministic compilation (when a compiler creates different binaries from the same sources, which differ in performance, for example again because of impact on memory placement). It turns out that the statistical methods presented in the computer systems performance evaluation literature for the design and summary of experiments do not readily allow this either. This poses a hazard to the repeatability, reproducibility and even validity of quantitative results.

Inspired by statistical methods used in other fields of science, and building on results in statistics that did not make it to introductory textbooks, we present a statistical model that allows us both to quantify uncertainty in the ratio of (execution time) means and to design experiments with a rigorous treatment of those multiple sources of non-determinism that might impact measured performance. Better still, under our framework summaries can be as simple as "system A is faster than system B by 5.5% ± 2.5%, with 95% confidence", a more natural statement than those derived from typical current practice, which are often misinterpreted.

Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*

General Terms: Experimentation, Measurement, Performance

Additional Key Words and Phrases: statistical methods, random effects, effect size

Your benchmarks can deceive you, don’t trust them

Benchmarks are just programs

- You have no expectation that your set of benchmarks is representative
- Most programs have an “inner loop”
- Can get disproportionate wins by optimising the inner loops

Understand what's going on

- Results need analysis
- Understand where the differences come from
- Do more experiments to find out
- Don't leave it until just before the deadline to collect your results

What else?

- Stack traces & debugging
  - GHC has 3 ways to get a stack trace, all with drawbacks
- Intermediate languages
  - Strict Core
- Optimisation
  - Supercompilation
- Low-level code generation
  - Hoopl
- Parallel performance
  - Mio

- Is front-end important any more?


File Tools Selection Patterns Font Style
PAINT
B:\RAINFALL.IMG
ABC
NONE
JAN FEB MAR APR MAY JUN JLY
AMSTRAD
PC-CM
AMSTRAD System Unit
PC 512 DD
PC


OPEN

Questions?