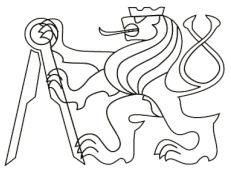
Byte Code Compiler
Recent Work on R Runtime

Tomas Kalibera

with Luke Tierney
Jan Vitek



CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

“Math” related improvements

- Matrix products
  - Check NaN/Inf inputs more consistently
  - Faster NaN/Inf checks (pqR, SIMD)
  - Faster BLAS matrix * vector using DGEMV
  - Multiple implementations
    - Default, internal (long double), blas, default.simd
- BLAS/LAPACK library path detection
- Use of ctanh/ctan workarounds

Runtime improvements

- applyClosure/execClosure
- Stack detection
- Timeout support for system, system2
- Customizable maximum number of DLLs
- Bug fixes:
  - Error/warning expressions, pairlist subsetting, protect fixes, #! line in Rscript, sprintf coerction, summaryRprof, Windows file timestamps, installChar->installTrChar, Windows Ctrl+C in cfe.exe, ...

Package checking

- PROTECT errors – static analysis tool
  - A full check (CRAN, BIOC) reported to maintainers
  - Automated checks using rchk, integrated into CRAN results
- Constants corruption
  - Repeated manual checks, reported to maintainers
- ‘If’ statement with non-scalar condition
  - A full check

JIT, byte code compiler/interpreter

- Srcref and expression tracking support
- Improvements and bug fixes:
  - Error messages, interaction of serialization with JIT, invocation of bcEval in loops without context, triggering JIT compilation of loops, C stack use in bcEval with gcc6, compilation with source references, cleaner cmpfun invocation
- Debugging of packages and reaching to maintainers

“debugging correctness”

Performance improvements with BC

```
convolveSlow <- function(x,y) {
    nx <- length(x)
    ny <- length(y)
    z <- numeric(nx + ny - 1)
    for(i in seq(length = nx)) {
      xi <- x[[i]]
      for(j in seq(length = ny)) {
        ij <- i + j - 1
        z[[ij]] <- z[[ij]] + xi * y[[j]]
      }
    }
    z
  }
```

BC is 9x faster than AST
(even including compilation)

x = y = as.double(1:8000)

Example from J. Chambers: Extending R

Performance improvements with BC

CRAN package PLIS
examples for em.hmm, EM algorithm for HMM to estimate LIS statistic,
Excerpt from function PLIS:::bwfw.hmm

```
dgamma <- array(rep(0, (NUM - 1) * 4), c(2, 2, (NUM - 1)))
for (k in 1:(NUM - 1)) {
  denom <- 0
  for (i in 0:1) {
    for (j in 0:1) {
      fx <- (1 - j) * f0x[k + 1] + j * f1x[k + 1]
      denom <- denom + alpha[k, i + 1] * A[i + 1, j + 1]
                                * fx * beta[k + 1, j + 1]
    }
  }
  for (i in 0:1) {
    gamma[k, i + 1] <- 0
    for (j in 0:1) {
      fx <- (1 - j) * f0x[k + 1] + j * f1x[k + 1]
      dgamma[i + 1, j + 1, k] <- alpha[k, i + 1] *
          A[i + 1, j + 1] * fx * beta[k + 1, j + 1]/denom
      gamma[k, i + 1] <- gamma[k, i + 1] +
                         dgamma[i + 1, j + 1, k]
    }
  }
}
```

BC is 4x faster than AST on examples for em.hmm

Performance improvements with BC

CRAN package mistat
examples for shroArlPfaCed, ARL, PFA and CED of Shiryayev-Roberts procedure
Excerpt from function mistat:::.runLengthShroNorm

```
while (m < limit && wm < ubd) {
  s1 = 0
  wm = 0
  for (i in 1:(m - 1)) {
    s1 = s1 + x[m - i + 1] - mean
    wm = wm + exp(-i * n * (delta^2)/(2 * sigma^2) +
         n * delta * s1/sigma^2)
  }
  wmv[m] <- wm
  if (wm > ubd || (m + 1) == limit) {
    res <- vector("list", 0)
    if (wm > ubd) {
      res$rl <- m
      res$w <- wmv[1:m]
    }
    else {
      res$rl <- Inf
      res$w <- wmv
    }
  }
m = m + 1
```

BC is 5x faster than AST on examples for shroArlPfaCed

Not all programs benefit from BC

```
convolveSlow <- function(x,y) {
    nx <- length(x)
    ny <- length(y)
    z <- numeric(nx + ny - 1)
    for(i in seq(length = nx)) {
      xi <- x[[i]]
      for(j in seq(length = ny)) {
        ij <- i + j - 1
        z[[ij]] <- z[[ij]] + xi * y[[j]]
      }
    }
    z
  }
```

BC is 9x faster than AST
(even including compilation)

```
convolveV <- function(x, y) {
  nx <- length(x)
  ny <- length(y)
  xy <- rbind(outer(x,y),
              matrix(0, nx, ny))
  nxy <- nx + ny - 1
  length(xy) <- nxy * ny
  dim(xy) <- c(nxy, ny)
  rowSums(xy)
}
```

BC is as fast as AST

ConvolveV is 4x faster than convolveSlow with BC, but uses a lot more memory

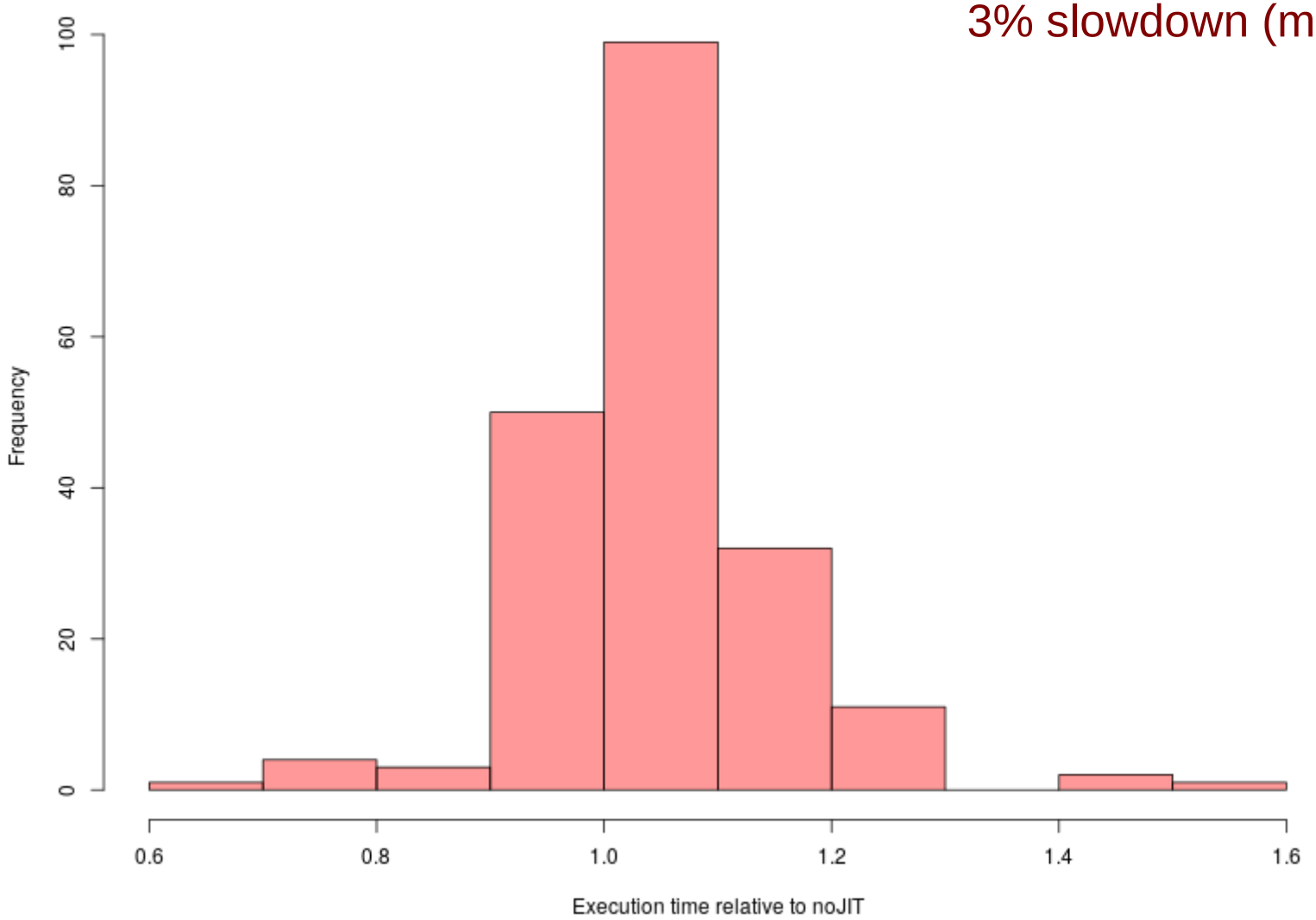
x = y = as.double(1:8000)

Examples from J. Chambers: Extending R

Summary performance: R examples

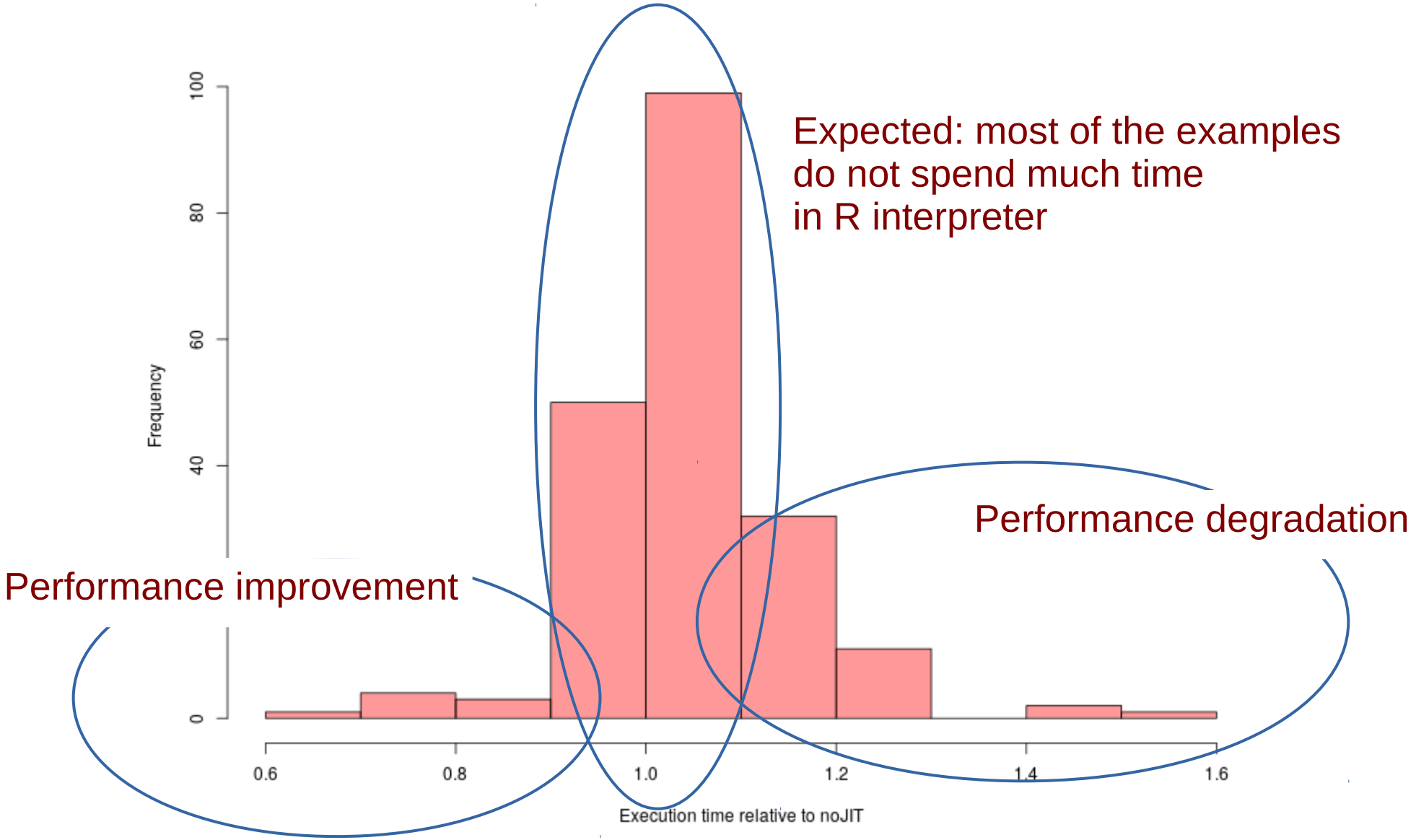
207 examples extracted from CRAN packages (runtime >5s, no downloading, set.seed)

3% slowdown (median)



Summary performance: R examples

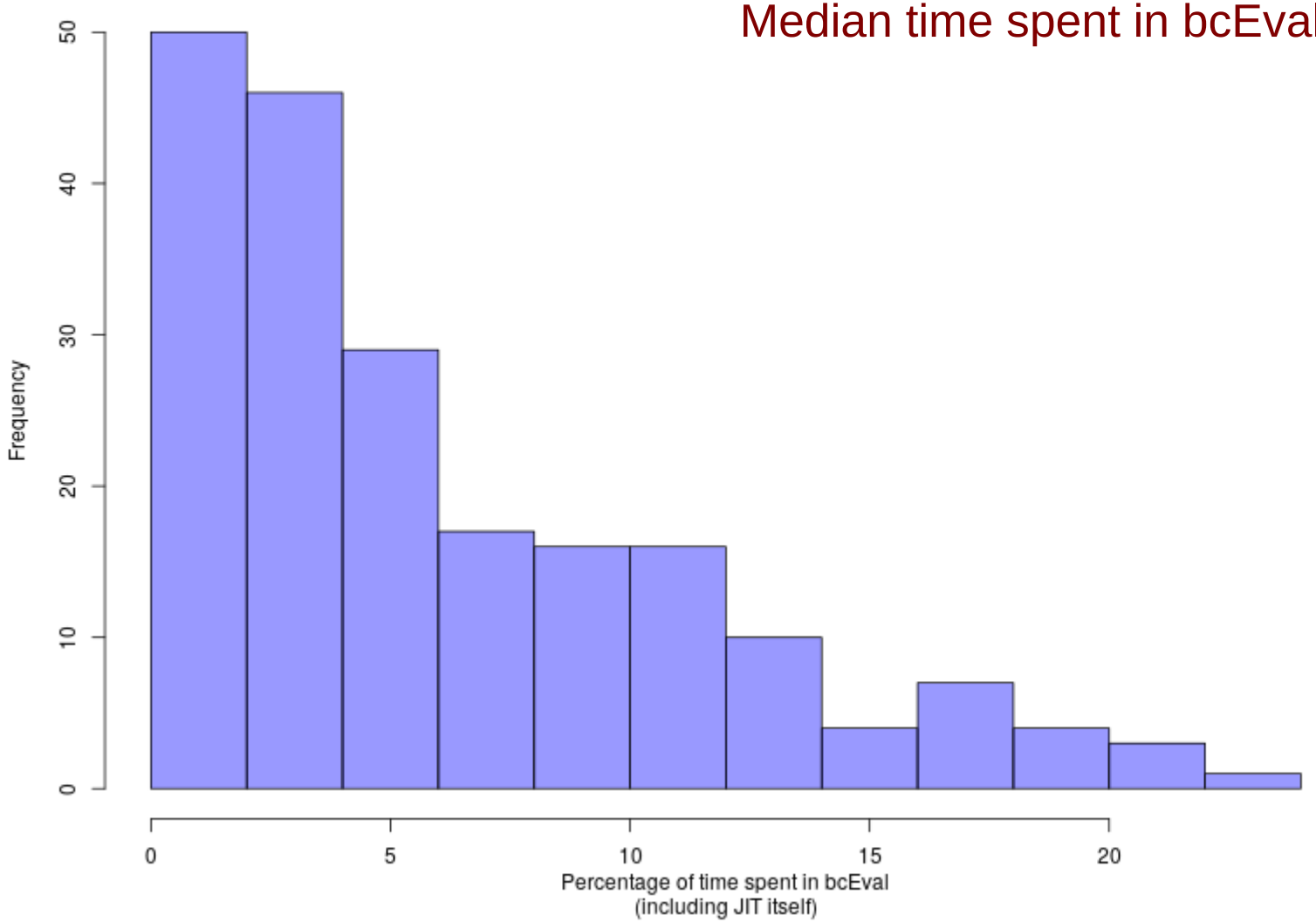
207 examples extracted from CRAN packages (runtime >5s, no downloading, set.seed)



Only small amount of time is spent in byte-code interpreter

207 examples extracted from CRAN packages (runtime >5s, no downloading, set.seed)

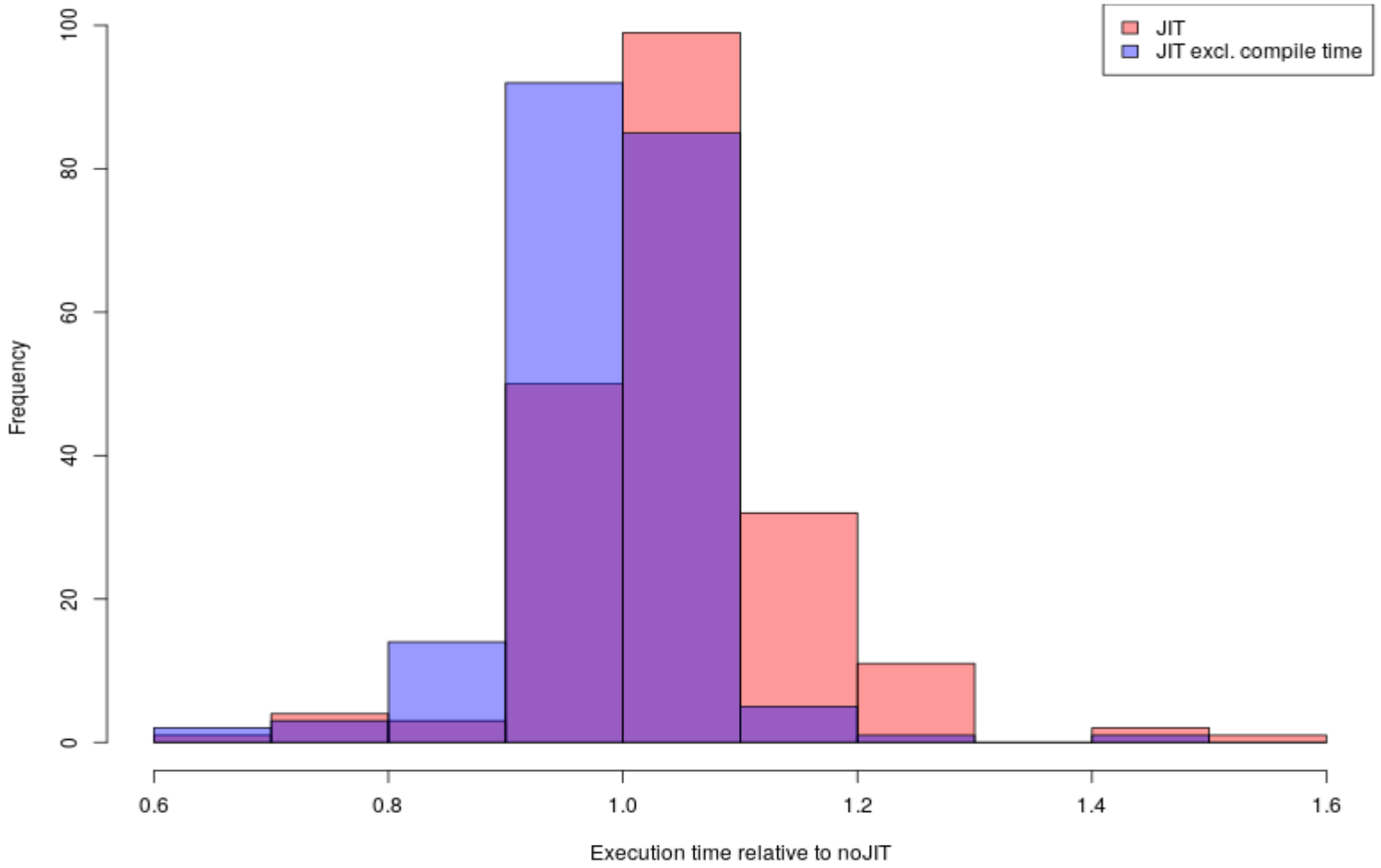
Median time spent in bcEval is 4%.



Most of the slowdown is due to extra time it takes to compile

207 examples extracted from CRAN packages (runtime >5s, no downloading, set.seed)

With compilation time excluded, median performance change is 0.



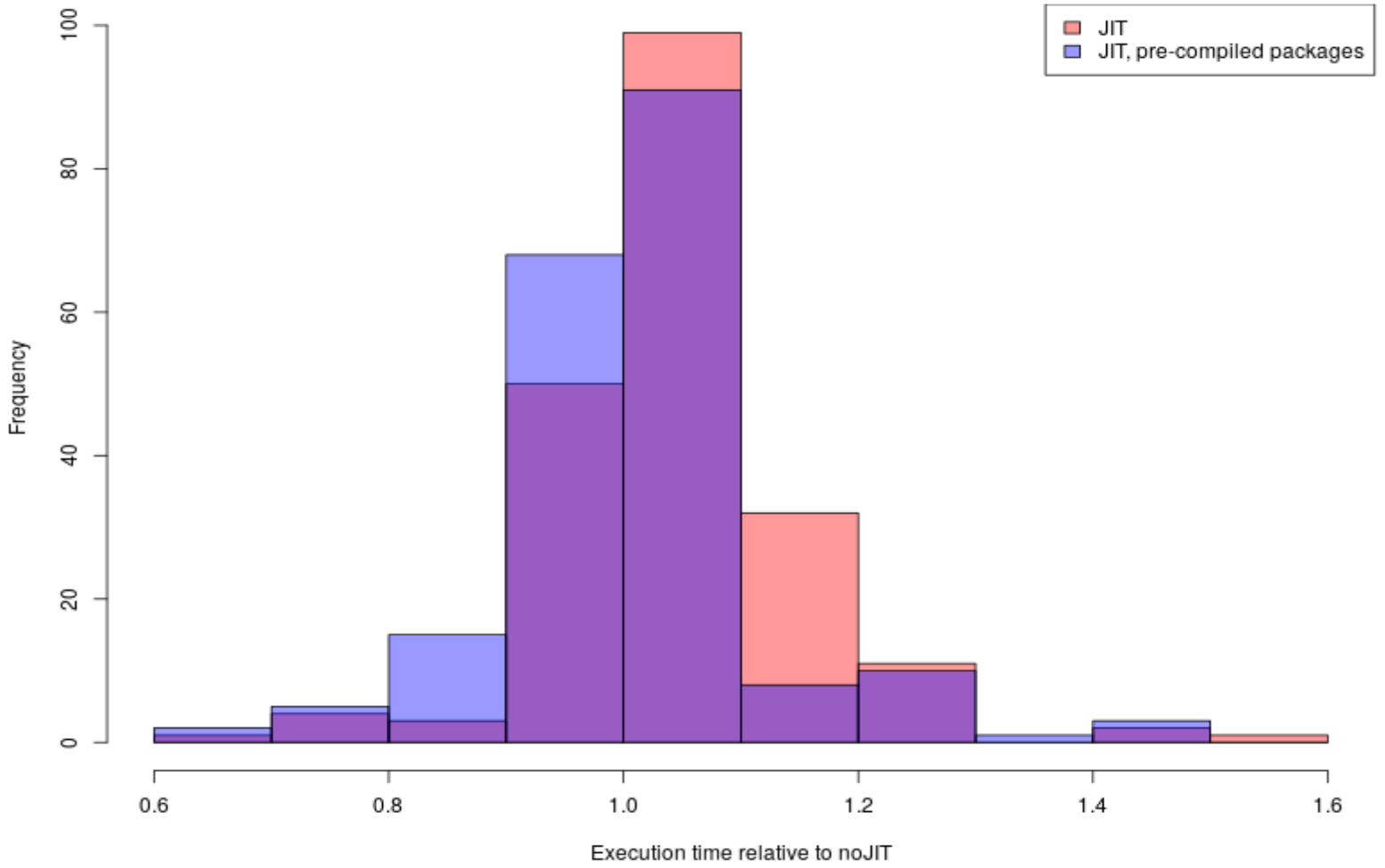
Mitigating compilation overhead

- JIT heuristics
  - Only compile functions likely to be executed often
  - Do not compile trivial functions, without loops, etc
  - Already in use, but can be more aggressive when JIT/AST compatibility improves
- Code cache
  - Re-use the same code if already compiled
  - Helps with code generation
- Precompilation
  - Compile package code and installation time
  - Not enabled by default for regular packages yet
  - Cons: compiling dead/unused code (and code not used by tests)
  - Performance issues with de-serialization to be resolved

Pre-compilation often helps

207 examples extracted from CRAN packages (runtime >5s, no downloading, set.seed)

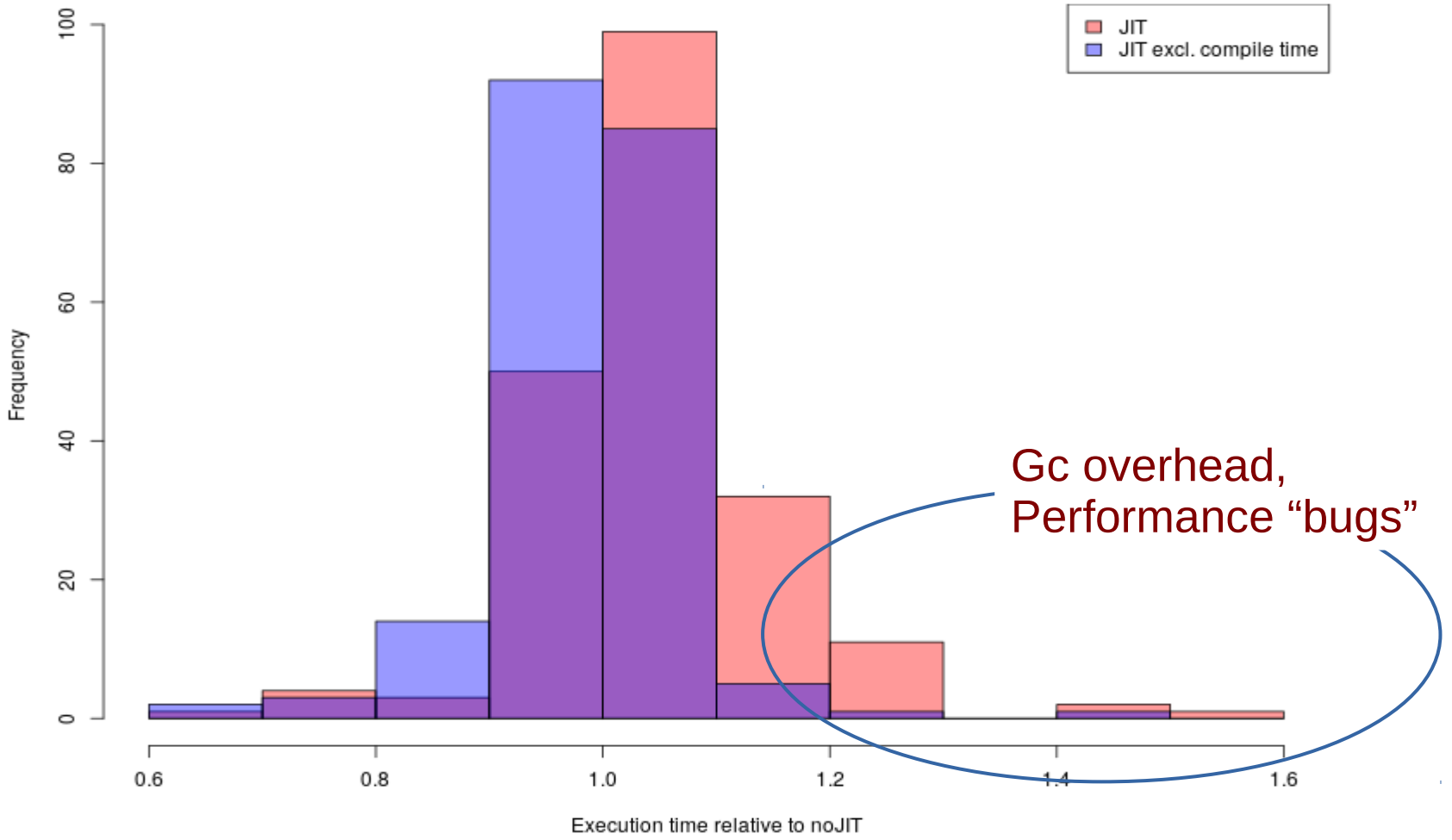
With JIT and pre-compiled packages, median performance change is 0.



Non-compilation slowdowns

207 examples extracted from CRAN packages (runtime >5s, no downloading, set.seed)

With compilation time excluded, there are still some slowdowns



Slowdown due to GC interaction

Excerpt from function lasso.stars (archived CRAN package)

```
fit = glmnet(x, y, lambda = lambda)

R.path = list()
for(k in 1:rep.num){
  ind.sample = sample(c(1:n), floor(n*sample.ratio), replace=FALSE)
  out.subglm = glmnet(x[ind.sample,],y[ind.sample], lambda = fit$lambda, alpha = alpha)
  R.path[[k]] = out.subglm$beta
  rm(out.subglm)
  gc()
}
```

gc() called in a tight loop:
85% of time is spent in GC tracing live heap
16% slowdown with JIT over AST

The indirect GC overhead may also be due to indirect impact on heap expansion when there is nothing wrong with the package, such as in AIM, cv.cox.interaction

Slowdown due to code generation

CRAN package mixtox

```
dichotomy <- function(fun, a, b, eps){
  expr <- parse(text = fun)
  execFun <- function(xx){}
  body(execFun) <- expr
  <...>
  flag <- sign(execFun(ab2) * execFun(a))
}

for (i in seq(lev)){
  for(j in seq(pointNum)){
    fun <- as.character(1)
    for (k in seq(fac)){
      if (model[k] == 'Hill_two')
        fun <- paste(fun, '-', pctEcx[k, i] * conc[j], '/ (', param[k, 1],
                     '* xx / (', param[k, 2], '- xx))', sep = '')
      else if (model[k] == "Hill_three")
        fun <- paste(fun, '-', pctEcx[k, i] * conc[j], '/ (1 /', param[k, 3],
                     '* (1 + (', param[k, 1], '/ xx)^', param[k, 2], '))', sep  = '')
  …
    root[i, j] <- dichotomy(fun, lb, ub, eps)
```

Source code generation and parsing

“eval(parse(text=))”

Slowdown due to improper use of “digest”

digest::digest computes hash including internal object state, but is used when visible state is needed. Internal state of a closure includes JIT bits and byte-code.

CRAN package R.cache

```
# 1. Generate cache file
key <- list(what=what, ...);
pathnameC <- generateCache(key=key, dirs=dirs);

# 1. Look for memoized results
if (!force) {
  res <- loadCache(pathname=pathnameC, sources=sources);
  if (!is.null(res)) return(res)
}

# 2. Otherwise, call method with arguments
res <- do.call(what, args=list(...), quote=FALSE, envir=envir);

# 3. Memoize results
saveCache(res, pathname=pathnameC, sources=s

# 4. Return results
res;
```

Computes digest from closure object

Invokes the closure, changing its internal state

Summary: how to improve performance with byte-code

- Package pre-compilation
  - Maintainer can enable selectively
  - Eventually should be turned on by default
- JIT heuristic
  - Compile later (after more calls)
- GC heap sizing heuristic
  - Take GC load into account
- Package fixes

  “debugging performance”