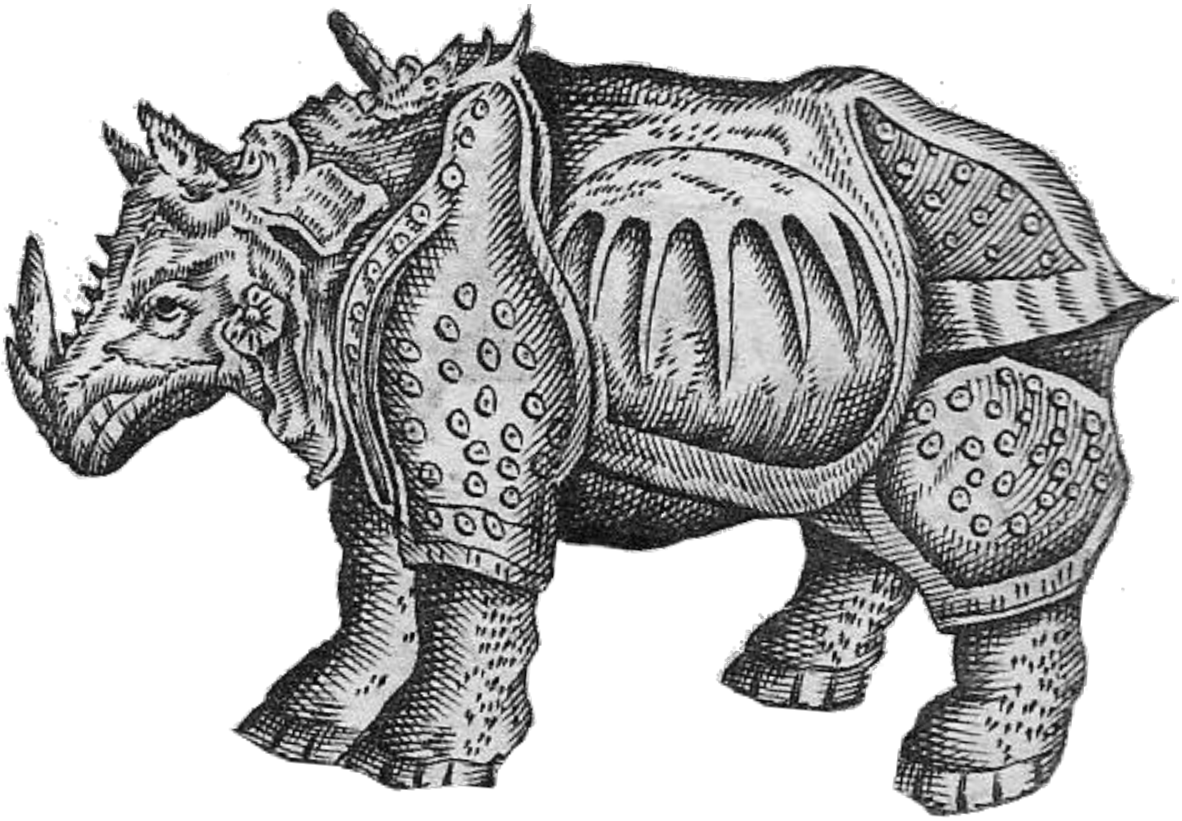
Version 0.1-0



Programming with Big Data in R

Guide to the pbdPAPI Package

Performance Analysis Tools for R

pbdR Core Team

Guide to the **pbdPAPI** Package

Performance Analysis Tools for R

May 16, 2014

Drew Schmidt
National Institute for Computational Sciences
University of Tennessee

Christian Heckendorf
Student, Boston College
Chestnut Hill, Massachusetts

Wei-Chen Chen
Department of Ecology and Evolutionary Biology
University of Tennessee

George Ostrouchov
Computer Science and Mathematics Division,
Oak Ridge National Laboratory

Pragneshkumar Patel
National Institute for Computational Sciences
University of Tennessee

Version 0.1-0

Acknowledgement

Heckendorf was generously supported by Google for Google Summer of Code 2014.

Chen was supported in part by the Department of Ecology and Evolutionary Biology at the University of Tennessee, Knoxville, and a grant from the National Science Foundation (MCB-1120370.)

Cover art is *A Rhinoceros*, Richard Hall, ca. 1740.

This publication was typeset using LaTeX.

Contents

1 Introduction

The value of profiling code is indisputable; if you disagree, stop pretending that you actually care about data. R's own `Rprof()` function is extremely useful, but its profiling capabilities are limited to simple timings of R functions. This is a very good starting point in performance analysis and can quickly help the R programmer focus in on bottlenecks. But for more experienced developers (especially those working with compiled code) additional performance information can be invaluable. Access to low-level hardware counter data can have tremendous impact when trying to understand and optimize performance of compiled code.

The **pbdPAPI** (**?**) package offers access to this low-level hardware counter information by way of the high-level C library **PAPI** (**?**). Therefore, an installation of **PAPI** is required in order to use the package. For convenience, we bundle **PAPI** version 5.3.0 with **pbdPAPI**, which will install by default. However, with appropriate configure arguments, one can easily build **pbdPAPI** with an existing system installation of **PAPI**. See Section 2 for details.

The current main features of **pbdPAPI** include:

1. Simple, high-level interfaces that mimic R's own profiling syntax.
2. A low-level interface that mimics **PAPI**'s native calls, with extremely general functionality.

Note that the **pbdPAPI** package is not officially affiliated with the **PAPI** project in any way.

2 Installation

In this section, we will describe the various ways that one can install the **pbdPAPI** package.

2.1 WITHOUT a System Installation of PAPI

This is the default method of installation. Here, the **PAPI** library will automatically be built first as a static library, and then the **pbdPAPI** package will be built and linked against that static library. All of this is handled completely transparently, and should only go wrong if your system is not supported by **PAPI**. This is the simplest approach, and should cover most users. Simply build the package as you would any other:

Shell Command

```
R CMD INSTALL pbdPAPI_0.1-0.tar.gz
```

and using the **devtools** package:

```
library(devtools)
install_github(username="wrathematics", repo="pbdPAPI")
```

2.2 WITH an Existing System Installation of PAPI

If you already have a system installation of **PAPI** available, it makes more sense to link with that existing library. The one catch is that the static library *must* have been compiled with `-fPIC`, which is non-standard. To build an external **PAPI** library in this way, you should do so by first setting:

Shell Command

```
export CC="${CC} -fPIC"
```

Assuming that `CC` is set; if not, you can use `cc` in the right hand side.

To link with an external installation of **PAPI**, from the command line, execute:

Shell Command

```
R CMD INSTALL pbdPAPI_0.1-0.tar.gz \
    --configure-args="--enable-system-papi \
    --with-papi-home=location/of/PAPI/install"
```

and using the **devtools** package:

```
library(devtools)
install_github(username="wrathematics", repo="pbdPAPI",
    args="--configure-args='--enable-system-papi
          --with-papi-home=location/of/PAPI/install'")
```

3 Performance Measurement

3.1 flips and flops

flips This is intended to measure *instruction* rate through the floating point pipe with no massaging.

flops Perhaps the more well-known measurement is the rate of floating point *operations*.

Calling `system.flops(expr)` on a valid `R` expression `expr` will produce system and processor timings, the number of floating point operations, and the Mflops.

The `pca` demo shows off this functionality with principal components analysis. Executing:

```
demo("pca", "pbdPAPI")
```

on an Intel Sandy Bridge Core i5 produces the following outputs:

```
      m  n  measured theoretical difference pct.error   mflops
1 10000 50 211875901   202500000    9375901  4.425185 1637.195
```

Theoretical flops

flops = (# cores) ∗ (# of SSE units per core) ∗ (cycles / second) ∗ (# SSE operations per cycle)

single precision (divide by 2 for double precision).

So for this Intel Sandy Bridge Core i5 again as a reference, the

$$\begin{aligned} \text{Mflops} &= (4) * (2) * (3200Mhz) * 2 \\ &= 25600 \end{aligned}$$

or about 25 Gflops (1,000,000,000 flops).

3.2 Cache Misses and Cache Hits

Memory and Cache Computers operate at *billions* of cycles per second. Of course, those operations occur on data. A useful abstraction we use in thinking about processing data is you load the stuff up into ram and then the processor does things to it. This is usually fine, or at least convenient, but it's not accurate, as you are probably aware.

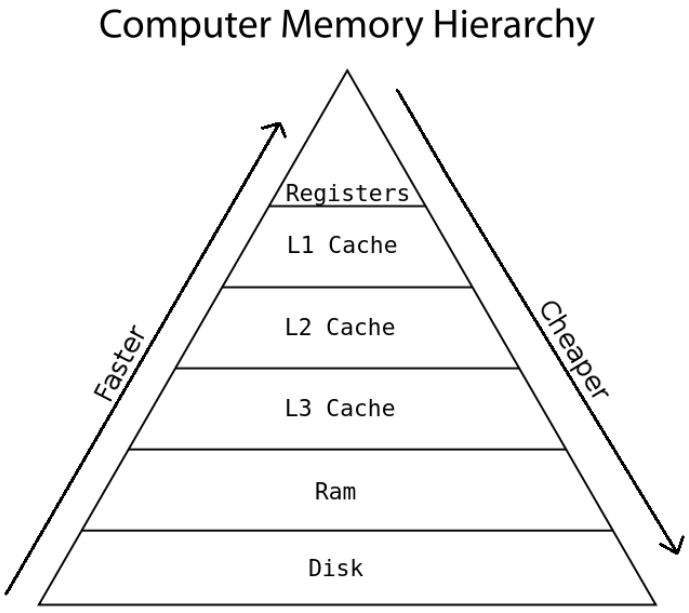


Figure 1: Memory Hierarchy

Another more accurate abstraction is that shown in Figure 1. In a sense, the magic really happens when things get into the CPU registers. But something that's in ram that you want to operate on, as it's headed to the CPU, it gets cached into various levels of (comparatively) fast access storage along the way. Not understanding this (simplified from reality) architecture can

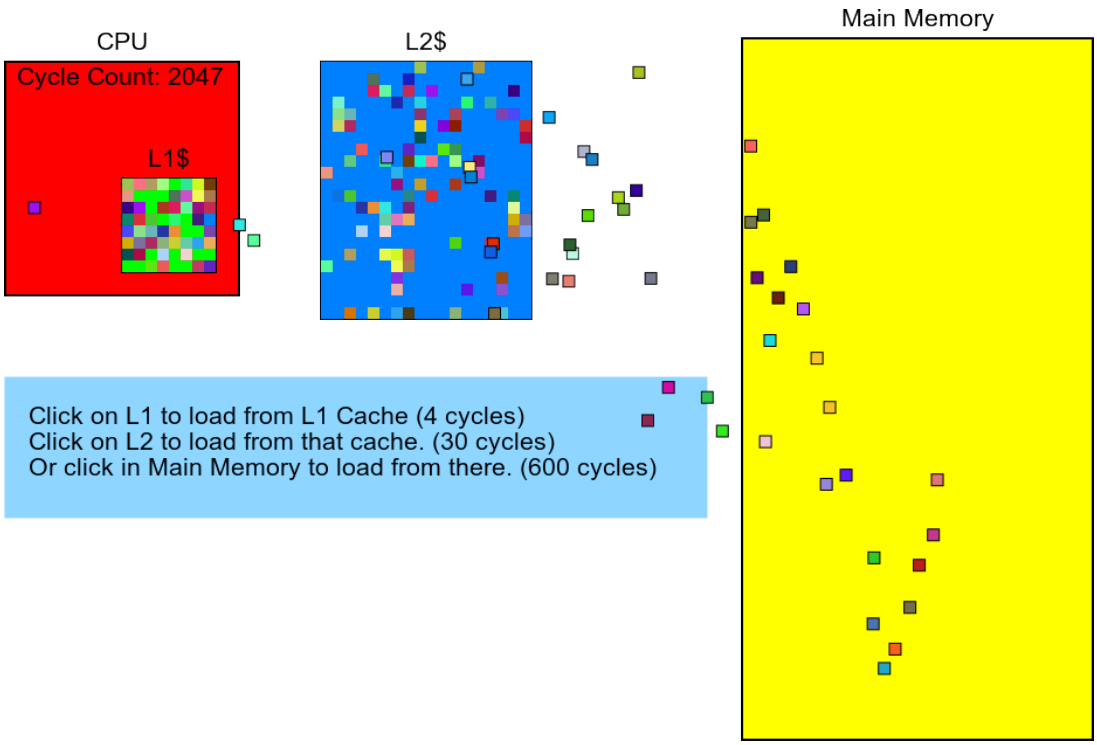


Figure 2: Still from Interactive Visualization Showing Relative Memory Access Speeds

have can cause some pretty nasty performance side-effects. on your code. If you're unfamiliar with this, I would strongly encourage you to check out this really cool interactive visualization showing (relative) speeds of cache misses, also shown in Figure 2. It too involves some hefty simplifications of how modern hardware actually works, so if this is at all confusing, let us all take a moment to pity the tragic life of the computer engineer.

Latency Numbers

Cache Misses Fundamentally, a cache miss occurs when the cache needs some piece of data to pass along to registers, but it isn't immediately available and has to go digging through ram (or god help you, disk) to get it. Cache misses are bad and reduce performance. You can't get rid of them, unless your entire problem fits into cache (which is glorious when it happens), but you can eliminate *unnecessary* cache misses being aware of how your data and algorithms interact with cache. When people tell you things like "R's matrices are column-major" or that you should loop over columns then rows, this is exactly what they are talking about.

Consider the following example, where we will fill a matrix with 1's, first by looping over rows then columns, and then by looping over columns then rows. For maximum effect, we will be dropping to C by way of **Rcpp**. If you do not have **Rcpp** installed on your system, you can still follow along (even if you don't know C++), but you will not be able to recreate the timings locally.

```
library(inline)
```

```
bad_cache_access <- "
  int i, j;
  const int n = INTEGER(n_)[0];
  Rcpp::NumericMatrix x(n, n);


  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      x(i, j) = 1.;

  return x;
"

good_cache_access <- "
  int i, j;
  const int n = INTEGER(n_)[0];
  Rcpp::NumericMatrix x(n, n);


  for (j=0; j<n; j++)
    for (i=0; i<n; i++)
      x(i, j) = 1.;

  return x;
"


bad <- cxxfunction(signature(n_="integer"), body=bad_cache_access,
   plugin="Rcpp")
good <- cxxfunction(signature(n_="integer"), body=good_cache_access,
   plugin="Rcpp")
```

A quick check of run times shows something drastically different happening:

```
n <- 10000L

system.time(bad(n))
#    user  system elapsed
#   1.016   0.232   1.259

system.time(good(n))
#    user  system elapsed
#   0.201   0.155   0.357
```

So even though we (mathematically) are doing the exact same thing, the run times differ by a factor of 3.5. **pbdPAPI** allows us to more thoroughly see what's happening. If we use `papi.cache()` to check the L1, L2, and L3 cache misses for each of these functions:

```
library(pbdPAPI)
```

```
n <- 10000L

papi.cache(bad(n))
#$L1.total
#[1] 193580295
#
#$L2.total
#[1] 159442230
#
#$L3.total
#[1] 16895275

papi.cache(good(n))
#$L1.total
#[1] 15552007
#
#$L2.total
#[1] 11580023
#
#$L3.total
#[1] 801150
```

it should be readily apparent what is going on now. The L1 cache misses differ by more than an order of magnitude, 194 million to 16 million!