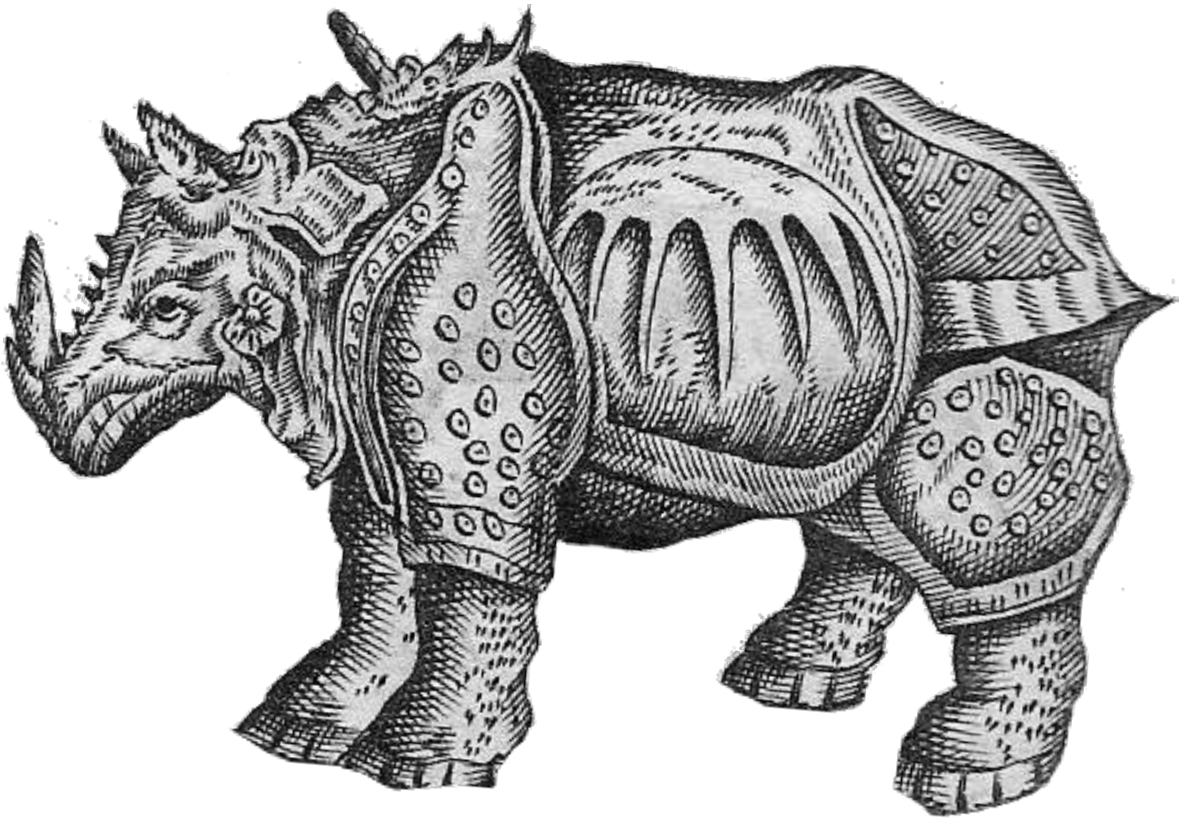
Version 0.1-0



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

September 17, 2015

Drew Schmidt
Business Analytics and Statistics
University of Tennessee

Christian Heckendorf
Boston University
Boston, Massachusetts

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

Version 0.2-2

Acknowledgement

Schmidt was supported in part by the National Institute for Mathematical and Biological Synthesis, sponsored by the National Science Foundation, the U.S. Department of Homeland Security, and the U.S. Department of Agriculture through NSF Awards #EF-0832858 and #DBI-1300426, with additional support from The University of Tennessee, Knoxville.

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.)

We thank James Ralph from the Innovative Computing Laboratory at the University of Tennessee for his helpful conversations about integrating **PAPI** into R.

Disclaimer

The findings and conclusions in this article have been formally disseminated neither by the U.S. Department of Energy, nor by the University of Tennessee, and as such should not be construed to represent any determination or policy of any University, Agency, and/or National Laboratory.

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

This publication was typeset using LaTeX.

Contents

Acknowledgement **3**

1 Introduction **1**
1.1 Installation Issues and Reporting Bugs 1

2 Installation **1**
2.1 Important Notes 2
2.2 Installing WITHOUT a System Installation of PAPI 2
2.3 Installing WITH an Existing System Installation of PAPI 3
2.4 Installing with Intel PCM 3

3 Performance Measurement **4**
3.1 flips and flops 4
3.2 Cache Misses and Cache Hits 5

4 Using pbdPAPI **7**
4.1 High-Level Interface 7
4.2 Low-Level Interface 7

5 Examples **8**
5.1 Floating Point Operations 8
5.2 Cache Misses 9

6 Additional Information **11**
6.1 Calling Context 11
6.2 Parallel Code 11

References **14**

1 Introduction

The value of profiling code is indisputable. 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** (Schmidt *et al.*, 2014) package offers access to this low-level hardware counter information by way of the high-level C library **PAPI** (Mucci *et al.*, 1999). 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.
3. A comprehensive set of package demonstrations.
4. This vignette.

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

1.1 Installation Issues and Reporting Bugs

Section 2 contains useful information about how to install **pbdPAPI**. Additionally, Sections 4, 5, and 6 offer useful information about using **pbdPAPI**.

If something goes wrong with installation or package use, please feel free to make use of our google group `http://group.r-pbd.org/`. We also actively monitor a mailing list if you wish to contact us directly: RBigData@gmail.com

For installation issues, please try to give us as much information as reasonably possible to help you. If **pbdPAPI** installs but you have trouble using it (for example, a function explicitly encourages you to report an error to us), please provide us the output of `pbdPAPI:::hw.info.internal()` (note the three colons) and `hw.info()`.

2 Installation

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

Please be aware that the full features of **pbdPAPI** are not supported on Windows or Mac at

this time. This is because the **PAPI** library itself does not support these platforms. Until **PAPI** officially supports these platforms, an alternative option is provided for some Intel CPUs in **pbdPAPI** through Intel Performance Counter Monitor . If we become able to support more platforms, we will do so immediately.

For a complete list of supported platforms for PAPI, see the official PAPI documentation.

2.1 Important Notes

It is possible for **PAPI** and/or **pbdPAPI** to build, but still have no access to any counter information. For example, the package will build on the popular ARM architecture device Raspberry Pi[1]. However, the device does not have any hardware counters, and as such, **pbdPAPI** effectively can not be used. If the package installs without error, you can see how many hardware counters you have available by calling:

```
library(pbdPAPI)
system.ncounters()
```

However, this too has some caveats. Those with Intel Sandy Bridge and/or Intel Ivy Bridge architectures should be aware that flops counts are unreliable on these platforms. This is a problem with the hardware returning incorrect values, not with **PAPI** or **pbdPAPI**. For more details, see: the **PAPI** documentation on this matter.

Additionally, not all hardware platforms support all counters. For the high-level interface, this simply means that some things you may wish to try simply can not physically work, and as such will return an error.

For a detailed list of all counters and whether or not your particular platform supports them, simply call

```
library(pbdPAPI)
papi.avail()
```

2.2 Installing 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:

[1] http://www.raspberrypi.org/

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

2.3 Installing 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. Not all versions of **PAPI** are supported. **pbdPAPI** package was built and tested with **PAPI** 5.3.0.

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'")
```

2.4 Installing with Intel PCM

If you are using an Intel CPU on a platform unsupported by **PAPI**, you may try using **Intel PCM** with **pbdPAPI**. The **Intel PCM** software requires additional drivers to function. On Linux and FreeBSD, these drivers are typically already available in standard kernels. Required drivers for OS X and Windows are built automatically when this package is built. Please also note that root or administrator privileges may be required for building, installing, and running **pbdPAPI** with **Intel PCM** enabled.

Shell Command

```
R CMD INSTALL pbdPAPI_0.1-0.tar.gz \
    --configure-args=--enable-ipcm \
    --no-test-load
```

and using the **devtools** package:

```
library(devtools)
install_github(username="wrathematics", repo="pbdPAPI",
    args="--configure-args=--enable-ipcm")
```

Running **pbdPAPI** may require setting the LD_LIBRARY_PATH environment variable to include the path where the **Intel PCM** library has been installed.

Shell Command

```
export
   LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:/usr/local/lib/R/site-library/pbdPAPI/li
```

3 Performance Measurement

In this section, we will outline some basics of several of the more useful profilers available in **pbdPAPI**.

3.1 flips and flops

pbdPAPI offers two high-level utilities for measuring floating point performance data: `system.flips()` and `system.flops()`. The former captures floating point instruction measurements; a *flip* is the rate of floating point instructions (flpins), or the number of flpins per second. Perhaps the more well-known measurement is the rate of floating point *operations*. Like its cousin `system.flips()`, the **pbdPAPI** function `system.flops()` will measure both the number of floating point operations as well as their rate — the number of floating point operations per second, or flops.

Generally, reports of flops (or flips) are not given, but Mega-flops (Mflops); as the name implies, 1 Mflop is 1,000,000 flops.

Theoretical flops A processor has a theoretical peak number of flops, which is typically much higher than what is found experimentally. Still, understanding the peak flops of a system can be useful in understanding "good" flops performance of a program (just understand that it will always be lower in practice than the theoretical peak).

The equation below demonstrates how to compute the peak Mflops of a processor:

$$\text{Mflops} = (\#\text{ cores}) * (\text{Speed in Mhz}) * (\#\text{ of SSE units per core}) * (\#\text{ SSE operations per cycle})$$

single precision Mflops (divide by 2 for double precision). So for this Intel Sandy Bridge Core i7 as a reference, the theoretical peak is:

$$\text{Mflops} = (4) * (2800Mhz) * 2 * 2$$

which is roughly 45 single precision Gflops, or 22.5 double precision Gflops — 22,500,000,000 floating point operations per second!

See where your computer stacks up against the fastest supercomputers in the world at top500.org. For instance, again using this laptop as a reference, and using theoretical flops as a proxy for running the Linpack benchmark (which is not really fair, we grant), we would crush every supercomputer on the list from June 1997, but wouldn't make the cut for June 1999.

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.

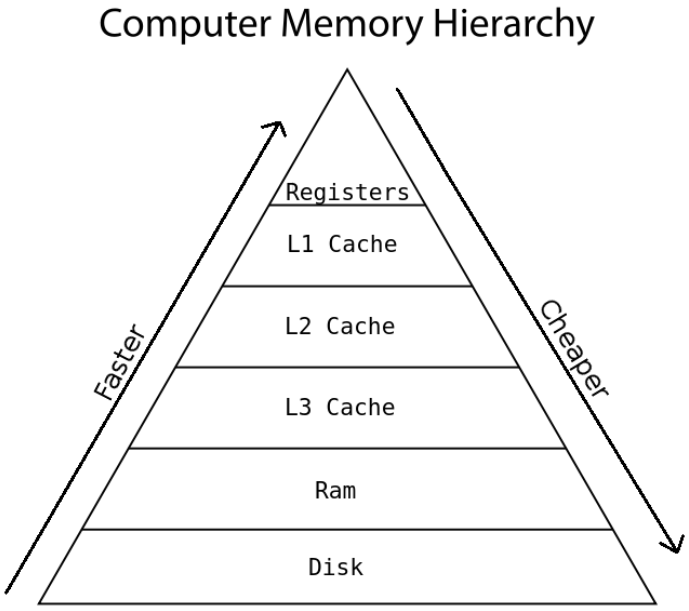


Figure 1: Computer Memory Hierarchy

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, gets cached into various levels of (comparatively) fast access storage along the way. Understanding this behavior, and writing your code to take advantage of it, can have *tremendous* impacts on performance.

A more detailed presentation of Figure 1 is that found in Figure 2. Here we see a more accurate presentation; for example, L3 cache (if it exists) is shared by all cores on a moder CPU, with L2 private, and L1 split into data and instruction caches. The lookup costs in terms of processor cycles are provided for each type of memory. These "times" are presented in terms of the CPU cycle for mostly historical reasons, though there is benefit to thinking in these terms (for

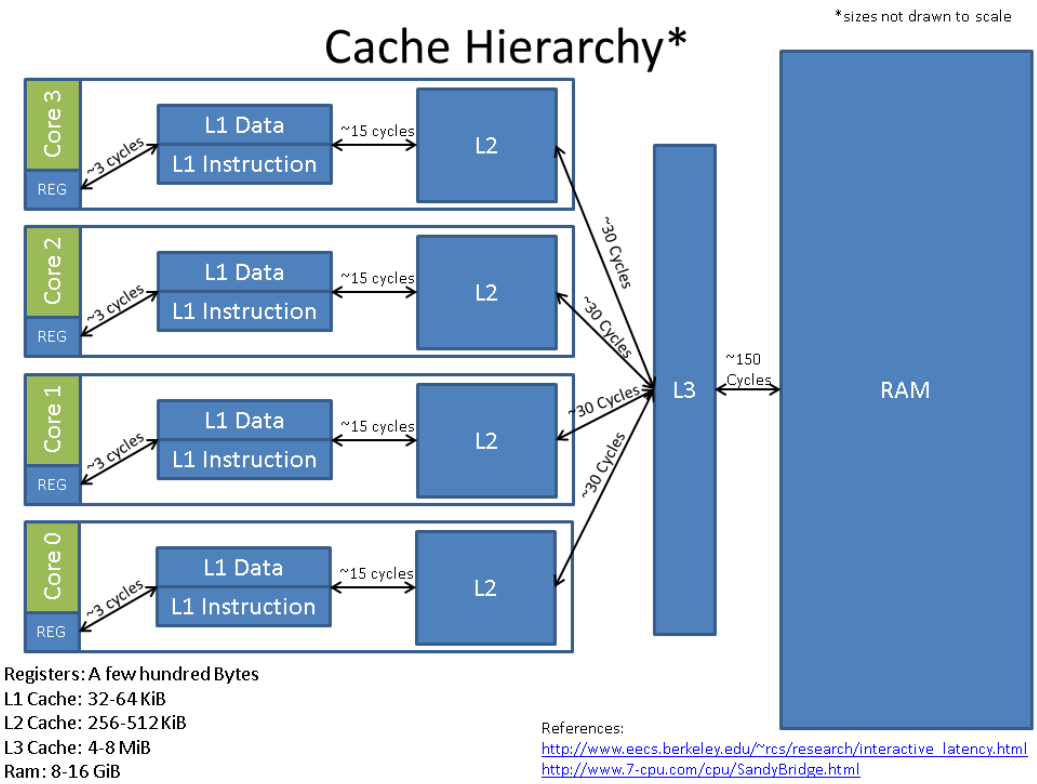


Figure 2: Detailed Computer Memory Hierarchy

example, when trying to minimize wait cycles). Though note that a faster processor could actually *increase* these costs, specifically RAM to L3 lookups.

If you are unfamiliar with the importance (or existence) of cache, I would strongly encourage you to experiment with this great interactive visualization showing (relative) speeds of cache misses. It too involves some 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. Another great resource is this interactive visualization showing memory and cache latency numbers by year.

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 the computer 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 completely, unless your entire problem — copies and all — comfortably fits into cache, but you can eliminate *unnecessary* cache misses being aware of how your data and algorithms interact with cache. See the demo `cache_access.r` in **pbdPAPI** for an example of good versus bad cache access.

4 Using pbdPAPI

The **pbdPAPI** package has 2 distinct interfaces for gathering performance data, one low-level and "one" high-level. The low-level interface is reminiscent of **PAPI**'s own native C interface, and "the" high-level interface is a collection of similar-looking wrappers around this interface.

Note that not all utilities will function on all hardware platforms. Those which are not supported for a given hardware platform will return an error. You can check which profiler events are available on your hardware platform using the **pbdPAPI** utility `papi.avail()` called with no argument.

4.1 High-Level Interface

We offer numerous useful simplified interfaces for profiling R code, shown in the table below:

Function	Description of Measurement
`system.cache()`	Cache misses, hits, accesses, and reads
`system.cpuormem()`	Characterize code as CPU or RAM bound (see docs for definition)
`system.epc()`	Events per cycle
`system.flips()`	Time, floating point instructions, and Mflips
`system.flops()`	Time, floating point operations, and Mflops
`system.idle()`	Idle cycles
`system.utilization()`	CPU utilization (see documentation for definition)
`ipcm.cache()`	Cache misses and hits for Intel PCM users

Each of these utilities behaves somewhat like R's own `system.time()` function. Simply pass a valid R expression in for the `expr` argument for any of these functions and the appropriate measurement will be returned.

See Section 5 and/or the package demos for some example usage.

4.2 Low-Level Interface

The low-level interface is provided for users who more familiar with hardware and may have questions about the operation of a block of code that do not easily fall into the the set of wrappers that make up the high-level interface. Even this comes in two forms. The first is the `system.event()` interface, which looks like the high-level interface functions, except that you must explicitly supply a **PAPI** events vector, with event names stored as strings. For a list of all possible events, and how to determine which are supported on your platform, see the R help `?papi.avail`.

As an example, suppose you wanted to measure the level 1 data cache misses and L1 data cache hits. After looking up the names for these events in the help file mentioned just above, you would simply call

```
events <- c("PAPI_L1_DCM", "PAPI_L1_DCH")
system.event(1+1, events=events)
```

Here, replace `1+1` with your preferred computation.

The final form of the low-level interface is essentially a deconstruction of the code making up the body of the `system.event()` function. This would allow you to skip several forms of error checking (if you want to do that for some reason). Primarily its purpose is to resemble the native C-level **PAPI** interface. We really don't recommend you use this, but it's there if you really believe you need it.

Revisiting the above example, you would call:

```
events <- c("PAPI_L1_DCM", "PAPI_L1_DCH")

papi.start(events=events)
1+1
papi.stop(events=events)
```

5 Examples

5.1 Floating Point Operations

Principal Components Analysis is a common statistical analysis technique. The implementation of PCA typically involves computing the singular value decomposition (SVD) of the input data and then projecting the data onto the right singular vectors. It is known that an SVD requires $6mn^2 + 20n^3$ floating point operations and that the projection onto the right singular vectors requires an additional $2mn^2$ operations (Golub and Van Loan, 1996). Finally, as PCA is usually performed on centered (and often scaled) data, we require an additional $2 * mn + 1$ operations for centering the data.

The **pbdPAPI** package contains the example `pca.r` as a package demo, which will perform a PCA on random data consisting of $10,000$ observations and 50 predictors. The analysis uses R's ordinary `prcomp()` routine, and the performance is measured by **pbdPAPI**'s `system.flops()`. You can run this demo by calling

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

An example output from this machine is:

```
      m  n  measured theoretical difference pct.error   mflops
1 10000 50 212538720   203500001    9038719   4.25274 2284.257
```

The **pbdPAPI** package has several other demonstrations that, like this, compare a theoretical floating point operations count against a measured count, and then display the Mflops the computation achieved. These demos include an inner product calculation `inner.r`, a matrix-matrix product `matprod.r`, and the fitting of a linear model in `regression.r`.

5.2 Cache Misses

To see the full source code described in this example, see the `cache_access` demo in **pbdPAPI**.

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.

You are probably aware that R matrices are stored in column-major fashion. What this means is that the data, as it is laid out in physical memory, is easier to go from the entry with index (i, j) to index $(i + 1, j)$ than it is to go to index $(i, j + 1)$.

An example of accessing data poorly is the following:

```
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;
"
```

Accessing data in this way maximizes the amount of work the computer needs to do in searching for the data your program needs. Accessing your memory correctly will help minimize cache misses:

```
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;
"
```

To prove this, we can build these functions using the **inline** and **Rcpp** packages (Sklyar *et al.*, 2013; Eddelbuettel and François, 2011) and then profile them with **pbdPAPI**.

```
library(inline)

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 is happening between the two implementations:

```
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 to the data, the run times differ by a factor of 3.5. **pbdPAPI** allows us to more thoroughly see what's happening. We can use `system.cache()` to check the L1, L2, and L3 (total) cache misses for each of these functions:

```
library(pbdPAPI)
n <- 10000L

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

system.cache(good(n))
#$L1.total
#[1] 15552007
#
```

```
#$L2.total
#[1] 11580023
#
#$L3.total
#[1] 801150
```

The L1 cache misses differ by more than an order of magnitude, 194 million to 16 million!

Perhaps a more useful measure is the *cache miss ratio*, which is the total cache misses divided by the total cache accesses (in the low-level interface syntax, this is the return of event `PAPI_L2_TCM` divided by the return of event `PAPI_L2_TCA`, for level 2).

Measuring this too is simple:

```
system.cache(bad(n), events="l2.ratio")
# L2 cache miss ratio
#            0.815597

system.cache(good(n), events="l2.ratio")
# L2 cache miss ratio
#           0.7156862
```

6 Additional Information

Your gradeschool teacher who told you that there is no such thing as a bad question? A liar and an intellectual fraud. Before beginning this project, a member of the **PAPI** team explained that "someone who *really* understands the hardware knows that there are certain questions you just don't ask." In R, we generally prefer to abstract away from the hardware as much as possible.

Hardware manufacturers aren't trying to make profiling as simple as possible; they're trying to execute instructions as fast as possible. As such, there are some special caveats worth making clear.

6.1 Calling Context

PAPI is designed to "listen in" on hardware counters only from its calling context. In particular, this means that system noise should not interfere with profiler data gathered by **pbdPAPI**.

6.2 Parallel Code

pbdPAPI currently does not support the profiling of parallel code. Attempting do to so will almost certainly return incorrect values. The exception would be SPMD-written code using MPI, where each R process is single threaded.

This is very easy to see with R's `mclapply()` form the **parallel** package. Consider this simple example:

```
system.flops(lapply(1, function(i) as.double(1:1000)/100))$flpops
# [1] 1008

system.flops(mclapply(1:2, function(i) as.double(1:1000)/100))$flpops
# [1] 94
```

Here we measure the number of floating point operations performed in dividing the (float) numbers 1 through 1000 by 100. In the first example, we get 1000 and some change (the **PAPI** + **pbdPAPI** + R overhead). In the second, we get... 94? Where did the others go? **PAPI** is designed to profile only the calling context, and `mclapply()` does its computation in forks, outside the scope of **PAPI**'s ability to profile.

This is (usually) true as well of multi-threading, for example, via OpenMP, although this is more difficult to describe, and frankly is platform dependent. Consider this example:

```
n <- 5000
x <- matrix(rnorm(n*n), n, n)

system.flops(x %*% x)
```

If we run this with OpenBLAS using 4 threads, on this reference Sandy Bridge hardware, we find:

```
$real_time
[1] 7.485725

$proc_time
[1] 7.457387

$flpops
[1] 13

$mflops
[1] 1.743238e-06
```

This is obviously not what we intended to find. Using the serial Atlas BLAS on the same machine, we find:

```
$real_time
[1] 20.52688

$proc_time
[1] 20.50697
```

```
$flpops
[1] 110667824

$mflops
[1] 5.396596
```

which is almost certainly closer to reality.

References

Eddelbuettel D, François R (2011). "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. URL http://www.jstatsoft.org/v40/i08/.

Golub GH, Van Loan CF (1996). *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)(3rd Edition)*. 3rd edition. The Johns Hopkins University Press.

Mucci PJ, Browne S, Deane C, Ho G (1999). "PAPI: A portable interface to hardware performance counters." In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pp. 7–10.

Schmidt D, Heckendorf C, Chen WC (2014). "pbdPAPI: Programming with Big Data – Performance Analysis Tools for R." R Package, URL http://cran.r-project.org/package=pbdPAPI.

Sklyar O, Murdoch D, Smith M, Eddelbuettel D, Francois R (2013). *inline: Inline C, C++, Fortran function calls from R*. R package version 0.3.13, URL http://CRAN.R-project.org/package=inline.