

Profiling R on a Contemporary Processor

Shriram Sridharan
University of Wisconsin–Madison
shrirams@cs.wisc.edu

Jignesh M. Patel
University of Wisconsin–Madison
jignesh@cs.wisc.edu

ABSTRACT

R is a popular data analysis language, but there is scant experimental data characterizing the run-time profile of R programs. This paper addresses this limitation by systematically cataloging where time is spent when running R programs. Our evaluation using four different workloads shows that when analyzing large datasets, R programs a) spend more than 85% of their time in processor stalls, which leads to slower execution times, b) trigger the garbage collector frequently, which leads to higher memory stalls, and c) create a large number of unnecessary temporary objects that causes R to swap to disk quickly even for datasets that are far smaller than the available main memory. Addressing these issues should allow R programs to run faster than they do today, and allow R to be used for analyzing even larger datasets. As outlined in this paper, the results presented in this paper motivate a number of future research investigations in the database, architecture, and programming language communities. All data and code that is used in this paper (which includes the R programs, and changes to the R source code for instrumentation) can be found at: <http://quickstep.cs.wisc.edu/dissecting-R/>.

1. INTRODUCTION

R is a statistical programming environment that has become the de facto standard for data analytics. R is the most commonly used data analysis language after SQL [22], and the most popular data mining tool [7]. In addition, many large enterprise database vendors have started to provide R as an interface alongside SQL for in-database analytics [4, 12]. There are four main reasons for this widespread adoption of R: open-source, ease of use, a strong community of users (both academic and enterprise), and a huge repository of CRAN (Comprehensive R Archive Network) packages. As of June 2014, the CRAN repository has 5684 packages. CRAN is one of the key strengths of R as it enables developers to contribute their code back to the community, and allows R users to leverage this rich body of work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Hawai'i Island, Hawaii.

Proceedings of the VLDB Endowment, Vol. 8, No. 2

Copyright 2014 VLDB Endowment 2150-8097/14/10... \$ 10.00.

While there are a number of different environments to run R, the most common setting is to run R in a single thread on datasets that fit in memory. In this paper, we focus on this common environment for running R, and profile how R uses memory and processor components using four characteristic workloads. The paper aims to provide a better understanding of where does time go when running R programs today. We also discuss how these insights lead to a number of open research issues. Furthermore, for some of the key issues, we also present an estimate of the potential benefits that R programs could see if some of these issues are addressed.

To the best of our knowledge, this is the first paper to characterize the execution time breakdown of R when running on a contemporary processor with four different workloads — a machine learning workload, a statistical modeling workload, a cluster analysis workload, and a primitive linear algebra operation. Since R is open-source, we were also able to drill through the code and execution profile to determine the factors that characterize the execution time breakdown. We make three key observations, as summarized below.

First, for all the workloads, more than 85% of the total execution time is spent in stalls. This result shows how poorly the R kernel performs on these workloads, and implies that there is huge potential for performance improvement by exploring new implementation techniques (in the R kernel), micro-architectural features, or automatic compiler optimization.

Second, stalls due to other components (branch misprediction and frontend stalls) contribute from 6-18% of the total execution time. Hence, the scope of methods that target reducing these stalls from within the R kernel is limited.

Third, memory stalls account for nearly 75% of the total execution time in all the workloads. The key reason for memory stalls is the frequent triggering of the garbage collector. The garbage collector in R produces a random memory access pattern as it traverses lists of pointers. This access pattern reduces the spatial locality in the CPU caches, and hence increases the memory stall time. Thus, methods to improve the garbage collector have the potential to significantly improve the performance of R programs.

Another reason for memory stalls is the *Create-Copy-GC* loop in R. When executing code, the R kernel creates a large number of temporary objects, a majority of which are either unnecessary or duplicates of existing objects. This behavior increases the memory pressure, and in-turn triggers the garbage collector frequently. This large use of main memory is also the key reason behind R swapping to disk even when the input datasets are far smaller than the main memory

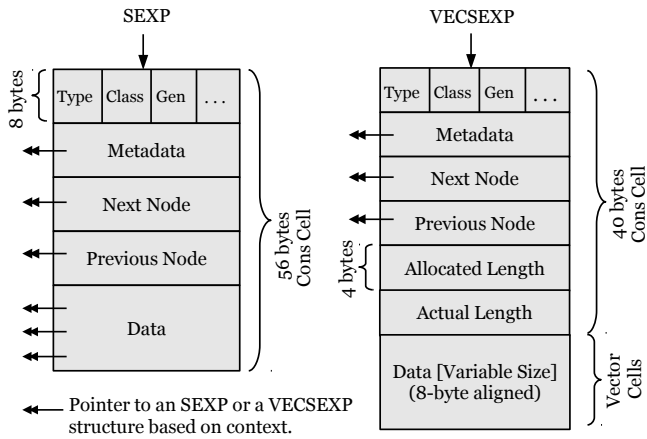


Figure 1: High-level model representing the internal data structures in a 64-bit build of R.

size. Hence, reducing the number of temporary objects that are created can significantly improve the performance of R.

R only provides a column-oriented storage layout and linear algebra operations that access them in a row-oriented manner suffer from memory stalls due to loss of spatial locality in the processor caches. Implementing additional storage layouts, or modifying linear algebra operations into cache-conscious algorithms can help minimize these stalls.

The observations highlighted above shed insights into the behavior of R on a contemporary processor. These insights can be used by database vendors who package R with their database products to better understand how the R “co-processing engine” works, and thus allows better packaging of R in database appliances and/or database cloud deployments. These insights might also be valuable to the database research community to consider exploring the use of database storage and query processing techniques to improve the R kernel. We hope that the architecture and the programming language communities also find this work useful as it allows them to consider architecture and compiler techniques to improve the performance of R programs. Also, with the increasing sizes of DRAM and decreasing costs [26], improving the R kernel presents big opportunities for much faster in-memory data analytics.

The remainder of this paper is organized as follows: Section 2 introduces preliminary concepts. Section 3 presents the processor model that we use to characterize the execution time breakdown. The experimental results are presented in Sections 4 and 5, and Section 6 presents opportunities to improve the R kernel. Section 7 discusses related work, and Section 8 contains our concluding remarks.

2. PRELIMINARIES

In this section, we describe key components that are used in the R kernel.

2.1 R Internals

R is an implementation of the S programming language (which is object-oriented). Internally, R implements constructs derived from both Lisp and Scheme, which are functional languages. Thus, R has both a functional and an object-oriented flavor.

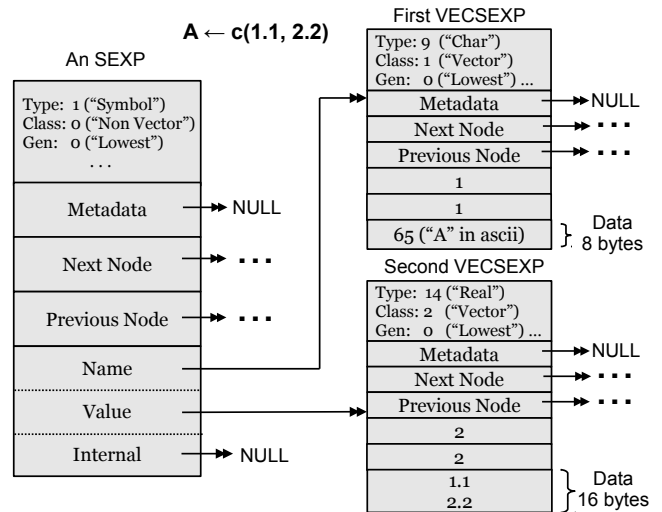


Figure 2: Usage of data structures in R.

Data Structure

R uses S-expressions (also referred to as a node or an object) as introduced and popularized by Lisp to represent all of its internal structures. S-expressions in R can be classified into vector and non-vector nodes. A *vector node*, which is represented internally using a VECSEXP, is used to store data. A *non-vector node*, which is represented internally using an SEXP, is used for all other structures like symbols, parse trees and linked lists. A high-level representation of an SEXP and a VECSEXP is shown in Figure 1.

SEXP and VECSEXP are both pointers to structures that contain a header and data. The first field of the header identifies important information like the type of the node (e.g., symbol, function, datatype, etc), the node class, generation of the node for garbage collection, and the reference count to this node. Subsequent fields contain pointers to the metadata, the next node, and the previous node. The header of a vector node also contains two extra fields to indicate the allocated and the actual length of the vector that is stored in the node. The data portion of a non-vector node is a union of structures that can represent a symbol, function, linked list, etc. depending upon the context. Each structure contains three pointers which can point either to an SEXP or a VECSEXP depending upon the type of the non-vector node. A vector node, however, does not contain pointers to data. Instead, the data follows contiguously in memory after the header.

Nodes in R also belong to one of eight node classes. Non-vector nodes (by default) belong to class 0. Vector nodes (classes 1-7) are divided into small vector nodes and large vector nodes depending upon the amount of data that is stored in the node. R allocates small vector nodes from pages that are maintained by R to reduce internal fragmentation. However, large vector nodes are directly allocated from the heap. As shown in Figure 1, R also has the notion of *cons cells* and *vector cells*. Every SEXP and the header of a VECSEXP is a cons cell. Data allocated in a VECSEXP is 8-byte aligned and each such 8-byte is a vector cell.

Figure 2 shows how these data structures are used to represent a simple vector ($A \leftarrow c(1.1, 2.2)$) in R. In this example, A is a symbol whose data is a real vector with two

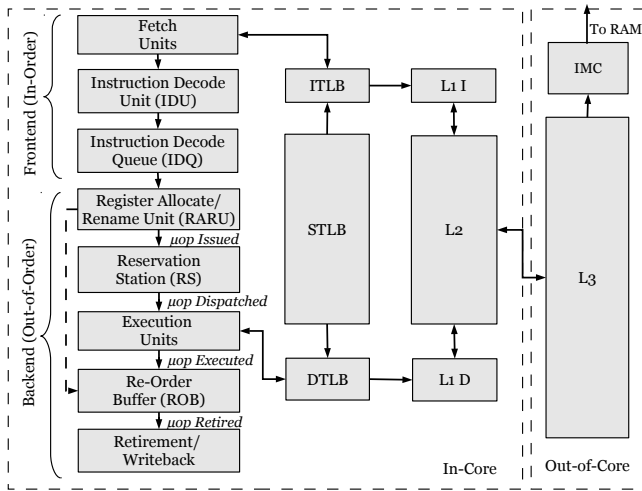


Figure 3: Overview of the processor architecture used to characterize the execution time.

values: 1.1 and 2.2. The variable name is stored in a VECSEXP of length 1 (*First VECSEXP* in Figure 2) and the values are stored in a VECSEXP of length 2 (*Second VECSEXP* in Figure 2). An SEXP of type *Symbol* is used to associate the variable name with its value. Note that the data portion of the SEXP contains fields that represent a symbol structure. Figure 2, however, is a simplified representation as R internally uses another level of indirection (using hash tables) for associating variables with their values. We omit these details in the interest of space.

Garbage Collection

R has an in-place generational garbage collector with two older generations. The garbage collector is based on [13]. This garbage collector maintains a circular doubly linked list of nodes for each generation of each node class (using the next node and the previous node pointers shown in Figure 1). Initially, when a node is allocated in R, it is snapped into the lowermost generation of the doubly linked list corresponding to the node’s class. Lower generation objects get more frequently collected than higher generation objects, and objects that are not garbage collected during a collection get promoted to the next higher generation. This mechanism is based on the observation that objects that survive more collections are likely to be referenced for a longer time, and hence should not be garbage collected in the near future.

The garbage collector in R is triggered either when the number of cons cells currently allocated crosses a threshold, or the number of vector cells currently allocated crosses a threshold. The thresholds are dynamically adjusted after a full collection (i.e., after all generations are collected) based on the current memory usage of R.

Garbage Collection (GC) happens in two phases — Mark and Sweep. During the Mark phase, all nodes reachable from a root node are marked by traversing the circular doubly linked lists of pointers. During the Sweep phase, all the unmarked nodes in the generation(s) that is being collected are freed. After every full collection, all the small nodes allocated in pages are sorted by reorganizing the pointers so that the locality of reference is improved.

Reference Counting

R has a naive reference counting implementation to provide copy-on-write semantics, where a copy of an object is only made if the object has multiple references, and there is a write from some reference to this object. However, this reference counting implementation just indicates whether the object has single or multiple references (i.e. it does not keep an actual count of the number of references to the object).

2.2 The Read-Eval-Print Loop

An R script can contain many lines of code. Since R is interpreted, each line in the script is executed independently in a Read-Eval-Print Loop (REPL). The following example illustrates how the REPL mechanism works.

Example: The R code to sum two objects and assign the result to another object is: `A <- B + C`. Executing this line of code triggers the following sequence of actions:

1. The *read* function parses the line of R code into an SEXP structure, which is a linked list based on the prefix notation. In our example, the parsed code in prefix notation is: `<- A + B C`, where each element is an SEXP.
2. The *eval* function takes the linked list parsed by the read function and processes it. In this example, it evaluates the SEXP `<-` to be a builtin function, and hence calls it with the remaining elements of the list as arguments. The `<-` function in-turn evaluates its arguments. Here, the first argument is the symbol `A` and the second argument is the function `+`. Evaluating the second argument results in a call to the `+` function which again evaluates its arguments, computing the sum, and returns the result back to `<-`. The `<-` function performs the assignment and returns control back to the original *eval* function. Note that both functions and data are passed as arguments, thus the functional programming paradigm in R.
3. The *print* function outputs the results, if necessary. In this example, no printing is needed, and control returns to the interactive shell to evaluate the next line of R code.

3. THE PROCESSOR MODEL

In this section, we explain the different hardware components that contribute to the execution time. The processor architecture that we use to characterize the execution time breakdown is shown in Figure 3, and is based on [31].

The processor is a pipeline of different units with queues in between them to hold instructions. The pipeline can be divided into the in-order frontend and the out-of-order backend. The frontend components have the following responsibilities:

1. The Instruction Fetch Unit fetches the next instruction to be executed, with the help of the Branch Prediction Unit, from the L1 I-Cache and provides it to the Instruction Length Decoder (ILD). The ILD prepares the instruction for execution and puts it into an instruction queue.
2. The Instruction Decode Unit (IDU) takes the instruction from the instruction queue, converts it into a set of μ ops (micro-operations), and puts them into the Instruction Decode Queue (IDQ). A μ op represents the smallest unit of operation that can be executed in the execution units.

The backend components have the following responsibilities:

Variable	Description
T_C	Computation time
T_{Br}	Branch misprediction stall time
T_{Fe}	Frontend stall time
T_{ILD}	Instruction Length Decoder stalls
T_{IDU}	Instruction Decode Unit stalls
T_{RARU}	Register Allocate and Rename Unit stalls
T_{L1I}	Stall time due to L1 instruction cache misses that hit the L2 cache
T_{ITLB}	Stall time due to ITLB misses that result in a STLB hit or cause a page walk
T_{Be}	Backend stall time
T_{L1D}	Stall time due to L1 data cache misses that hit the L2 cache
T_{L2}	Stall time due to L2 misses that hit in the L3
T_{L3}	Stall time due to L3 misses that hit in the DRAM
T_{DTLB}	Stall time due to DTLB misses that result in a STLB hit or cause a page walk
T_{ALU}	Stall time due to the ALU execution units

Table 1: Execution time components.

1. The Register Allocate and Rename Unit (RARU) allocates resources required for the μop like a ROB (Re-Order Buffer) entry, RS (Reservation Station) entry, load or store buffers before the μop is *issued* to the Reservation Station. The Re-Order Buffer (ROB) stores the μops temporarily, which can be executed out-of-order.
2. The Reservation Station (RS)/Scheduler is responsible for dynamically dispatching μops to one of the execution units. The execution units are fully pipelined and perform ALU and load/store operations. μops can be scheduled for execution in parallel and out-of-order as long as the program correctness is not affected.
3. Once all the μops corresponding to an instruction have been executed, then the instruction is said to have completed execution and can be safely retired (usually done by write-back of state to the architectural registers).

The model for characterizing execution time: A μop goes through different stages – Issued, Dispatched, Executed, Retired – during its execution (see Figure 3). We use the μop *issue* flow within the pipeline to characterize the execution time based on [3], resulting in the following breakdown:

1. Computation Time: Issued μops that subsequently retire contribute to the computation time (T_C).
2. Branch Misprediction Stalls: Issued μops that do not retire contribute to the “wasted work” done by the execution units due to a wrongly predicted branch (T_{Br}).
3. Frontend Stalls: μops that were not issued because of a stall in any unit in the frontend of the pipeline contribute to the frontend stalls (T_{Fe}).
4. Backend Stalls: μops that were available in the Instruction Decode Queue but were not issued because of resources being held-up in the backend contribute to the backend stalls in the pipeline (T_{Be}).

Putting all of this together, the time taken to execute a single line of R code (T_R) is the sum of the computation time and the various stall times. Thus,

$$T_R = T_C + T_{Br} + T_{Fe} + T_{Be}$$

Characteristic	L1	L2	L3
Cache Size	32 KB Data (D), 32 KB Instr. (I)	256 KB	24 MB
Associativity	L1 I – 4 way, L1 D – 8 way	8 way	24 way
Inclusivity	Non-inclusive	Non-inclusive	Inclusive
Miss penalty (cycles)	10 (hits in the L2)	35 (hits in the L3)	180

Table 2: Intel Xeon E7-4850 cache characteristics.

The stall time due to the frontend and the backend can be further divided into different components as shown in Table 1. Note that the RARU stalls (T_{RARU}) are accounted under frontend stalls, even though it is a backend component as we use the μop issue flow to characterize the execution time breakdown. Moreover, since the L2 and the L3 caches are unified, T_{L2} and T_{L3} are not only the stall times due to data fetch misses but also due to instruction fetch misses. Note that, some of the stall times due to the smaller stall components can overlap with instruction execution in the pipeline because of techniques like dynamic instruction scheduling, speculative execution, and data prefetching. Hence, the contribution of their stall cycles to the total execution time is approximate and an upper bound. We discuss this issue further in Section 4.3.

4. EXPERIMENTS

In this section, we describe our hardware platform, the workload, and the measurement methodology that we use.

4.1 Hardware platform

For the experiments, we used a 2GHz Intel Xeon E7-4850 processor (we will just refer to this as the Xeon processor) based on the Nehalem micro-architecture. While this processor has 10 cores, R can only use one of the cores as it is single-threaded. We also pinned the R process to a physical core and disabled hyper-threading. The cache characteristics of the Xeon processor are shown in Table 2. Note that the L3 cache is inclusive, i.e. the cache lines found in the L1 and L2 caches (across all the cores) are also present in the L3 cache. The L1 cache has a separate instruction (I) and data (D) cache, whereas the L2 and L3 cache are unified and cache both instructions and data. The processor has its own integrated memory controller (IMC) to connect to the DRAM via a 1066 MHz bus. The test machine has 64GB DRAM, and we used R-3.0.2 (64-bit version) running on Scientific Linux 6.

4.2 Workload

We used four different workloads for our study. The workloads consist of a machine learning technique — Decision Trees (Classification), a statistical modeling technique — Linear Regression (Regression), a cluster analysis technique — KMeans (Clustering), and a linear algebra operation — Matrix Multiplication. The reason for choosing classification, regression and clustering is because they are common and popular analysis methods. We also refer to these three collectively as the *data analysis workloads*. We also choose a primitive linear algebra workload because it is used as a building block in many data analysis algorithms, and R is also widely used to evaluate linear algebraic operations.

Workload	Package	Algorithm
Decision Tree	rpart [30]	Recursive Partitioning
Linear Regression	stats	QR Decomposition
KMeans	stats	Lloyd’s Algorithm

Table 3: Data analysis workloads, their packages, and the algorithms used for their implementation.

Dataset. We used the Airline On-time Dataset [1] for the data analysis workloads. The dataset consists of flight arrival and departure details for all commercial flights within the USA, from October 1987 to April 2008. There are nearly 120 million rows with 29 different columns in the dataset. The on-disk size of the uncompressed dataset is 11GB after preprocessing. The in-memory size when the complete dataset is loaded in R is 27GB because of the binary file type representation. For every data analysis workload, we use the maximum dataset size that can be analyzed in-memory for that workload without swapping to disk. In some cases, the workload was able to analyze the complete dataset without swapping. To stress these workloads, we increased the number of rows in the original dataset by repeating the dataset. For matrix multiplication, we use uniform sampling to construct two large matrices.

Table 3 shows the list of data analysis workloads along with the package they are found in, and the algorithm that is used. Matrix Multiplication is exposed via the *base package* in R. All these packages are available as part of the native R distribution and work completely in-memory.

All the workloads are invoked via an R script. Each R script consists of one or more lines of Extract-Transform-Load (ETL) operations, followed by explicit garbage collection, and then followed by calling the actual R function that implements the workload. The ETL operations are necessary to load the data and set it up for the workload under study. To focus on the core computation, we only load the required columns for the workload, and not the whole dataset. We use explicit garbage collection so that unnecessary data allocated during the ETL operations does not trigger a garbage collection event during the execution of the workload (the main target of this study). Note in the performance results presented in the later sections, we only measure the workload (the last line of R code in every script); i.e. the ETL and the garbage collection times are not included. Also note that all the workloads, except the linear algebra operation, are R functions written using many lines of R code. The R functions can in-turn call their own C/Fortran code if efficient or custom processing is required. The R scripts are described below. Only part of the R ETL code is provided due to space constraints.

Decision Trees: We built a decision tree model to classify whether an airline can be late or not based on the day of the week and the departure delay. We term an airline as *Late* if its arrival delay is greater than 10 minutes. The R script is as follows:

```
A <- read.table(file="airline220MRows.csv", sep=";", ...)
A$Late <- A$ArrDelay > 10
A$Late <- as.factor(A$Late)
gc(T)
result <- rpart(Late ~ DepDelay + DayOfWeek, data = A)
```

Variable	Description	Value
T_R	Total execution time of the R script	Actual unhalted CPU cycles
T_C	Computation time	Estimated based on the number of μ ops retired
T_{Br}	Branch misprediction stall time	Estimated based on the number of μ ops issued and number of μ ops retired
T_{Be}	Backend stall time	Actual stall cycles
T_{L1D}	L1 D-cache stalls	#Misses * 10 cycles
T_{L2}	L2 stalls	#Misses * 35 cycles
T_{L3}	L3 stalls	#Misses * 180 cycles
T_{DTLB}	DTLB stalls	#DTLB misses * 7 cycles + Page walk cycles
T_{ALU}	ALU stalls	Actual stall cycles
T_{Fe}	Frontend stall time	$T_R - T_C - T_{Br} - T_{Be}$
T_{ILD}	ILD stalls	Actual stall cycles
T_{L1I}	L1 I-cache stalls	#Misses * 10 cycles
T_{ITLB}	ITLB stalls	#ITLB misses * 7 cycles + Page walk cycles
T_{MISC}	Other stalls	Estimated stall cycles

Table 4: Measurement methodology for each components’ stall time.

Linear Regression: Similar to the decision tree model, we built a linear regression model to characterize the arrival delay of an airline based on the day of the week and the departure delay. The R script is as follows:

```
A <- read.table(file="airline100MRows.csv", sep=";", ...)
gc(T)
result <- lm(ArrDelay ~ DepDelay + DayOfWeek, data = A)
```

KMeans: We used five numeric columns (removing missing values) to find two clusters in the dataset. The R script is as follows:

```
A <- read.table(file="airline150MRows.csv", sep=";", ...)
gc(T)
result <- kmeans(na.omit(A), 2, algorithm="Lloyd")
```

Matrix Multiplication: We used two matrices with 2^{30} elements each for matrix multiplication. Elements of the matrices were uniformly sampled from numbers between 1 and 100 and NA (to include missing values). R internally uses BLAS [24] libraries for linear algebra operations. However, in the presence of missing values, the R kernel implements its own matrix multiplication. Since the focus of this paper is to characterize the R kernel, we introduced missing values into the matrices. The R script is as follows:

```
A <- matrix(sample(c(1:100, NA), 230, replace=T), ncol=222)
B <- matrix(sample(c(1:100, NA), 230, replace=T), nrow=222)
gc(T)
result <- A%*%B
```

Our workload consists of a range of R scripts from those that allocate minimal memory which does not require any deallocation (matrix multiplication) to those that deallocate most of the allocated memory (data analysis workloads). The former accounts for scripts that are compute-bound and the latter for scripts that are memory-bound.

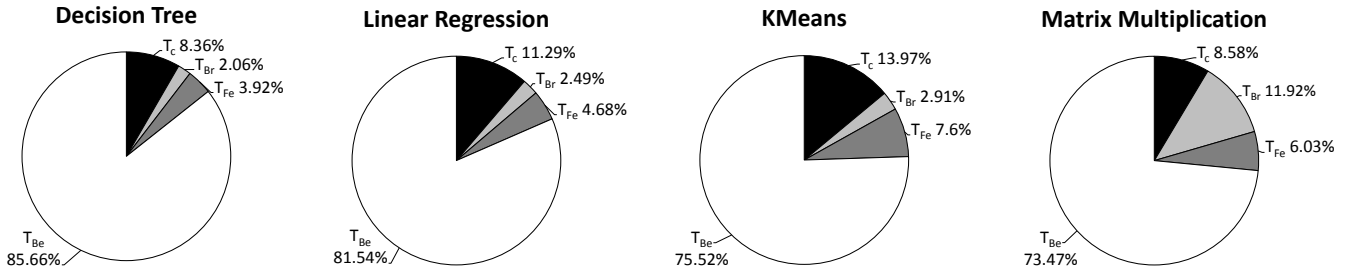


Figure 4: Execution time breakdown of the different workloads.

4.3 Measurement tools and methodology

We used PAPI [15] to measure the native performance counters of the Xeon processor. We measured 36 different performance events exposed by the Nehalem microarchitecture. Our measurements indicate the amount of time that R spends in the *eval* portion of the REPL. To do this, we modified the R source code to start collecting counters using PAPI just before *eval* is called, and stop collecting counters just after *eval* is stopped. We also needed to profile specific components of the R kernel like the garbage collector. We used PAPI in a very similar way to achieve this.

To increase confidence intervals, every workload was run multiple times and the maximum standard deviation for any significant performance counter across all the workloads was less than 3%. Table 4 shows the measurement methodology for the individual stall components. As can be seen:

1. The frontend stall cycles is measured as $T_R - T_C - T_{Be} - T_{Br}$. This equation holds from the processor model discussed in Section 3.
2. The backend stall cycles is approximated as the number of cycles no μop was executed in any of the ALU execution units, since when μop execution is stalled, the RS get backed up quickly and hence no more μop can be issued to the RS from the IDQ (cf. Section 3 and Figure 3).
3. Except for the ALU stalls, the measurement methodology for the individual components of the backend stalls assumes a sequential penalty model (i.e., the total backend stall cycles = $\sum_{\text{performance impacting events}} \text{number of occurrences of the event} \times \text{average penalty cycles for that event}$ [3]). This method overcounts due to the capability of the processor to hide some of these stalls by out-of-order execution. However, for simplicity, we use the computed stall cycles to only compare between the different individual stall components. We do not use it to compare with the total execution cycles or the backend stall cycles.
4. T_{IDU} and T_{RARU} have been replaced with T_{MISC} . This measure is estimated from the computed frontend stall cycles and the other frontend stall components.

5. RESULTS

We executed the workloads described in Section 4.2 on the hardware platform described in Section 4.1, and present our findings in this section.

5.1 Execution Time Breakdown

Figure 4 shows the execution time breakdown of the different workloads. The graphs show the distribution of the

computation time (T_C), the branch misprediction stall time (T_{Br}), the frontend stall time (T_{Fe}) and the backend stall time (T_{Be}) as a percentage of the total execution time for all the workloads. As can be seen, nearly 85-90% of the time is spent in stalls (T_{Br} , T_{Fe} and T_{Be}) for all the workloads – an alarmingly high component that indicates how poorly R performs in-memory data analysis for large datasets on current processors. On the other hand, this behavior also presents big opportunities for performance improvements in R. As DRAM sizes continue to increase while dropping in cost [26], improving the R kernel presents opportunities to analyze far larger datasets than is possible today (in R), and far more efficiently.

Backend stalls account for approximately 75-85% of the total execution time. The reason for the backend stalls can be either memory-bound stalls or ALU execution unit stalls like a divide, square root or any floating point operation. However, for all the workloads, the measured stall cycles due to the ALU execution units is $< 1\%$ of the backend stall cycles and hence we approximate the entire backend stalls to be memory-bound stalls. Thus, reducing the memory-bound stalls in R can dramatically improve the performance of the computation time in all the workloads. As we will see in the Section 5.2.1, an additional benefit is to considerably reduce the memory footprint for certain workloads.

5.2 Backend Memory Stalls

Figure 5 shows the decomposition of the backend memory stall time into four components T_{L1D} , T_{L2} , T_{L3} , and T_{DTLB} . Since the L2 and the L3 caches are unified, T_{L2} and T_{L3} also include the stall time due to instruction fetches that do not hit in the L2 cache. However, the number of instruction fetch requests that miss the L2 is four orders of magnitude smaller compared to the number of accesses to the L3 cache. Hence, T_{L2} and T_{L3} can be approximated to be the stall time only due to data fetches.

In our experiments, the L1D stalls accounts only for 1.5 - 3% of the total memory stall time, and the DTLB stalls only for 0.5 - 5% of the total memory stall time in all our workloads. DTLB stalls occur when the address translation for the given virtual to physical address misses in the DTLB. When this happens, the Xeon processor looks up in its STLB (“Second-level TLB”) cache, which is shared between the DTLB and the ITLB and also has many more entries than each of them. If the virtual address is not present even in the STLB, then it triggers a page-walking mechanism. However, in our measurements, the number of page-walk cycles (which is directly measured using a hardware counter) is comparatively low, probably due to fast hardware page-table-walkers present in the x86-64 based Xeon processor.

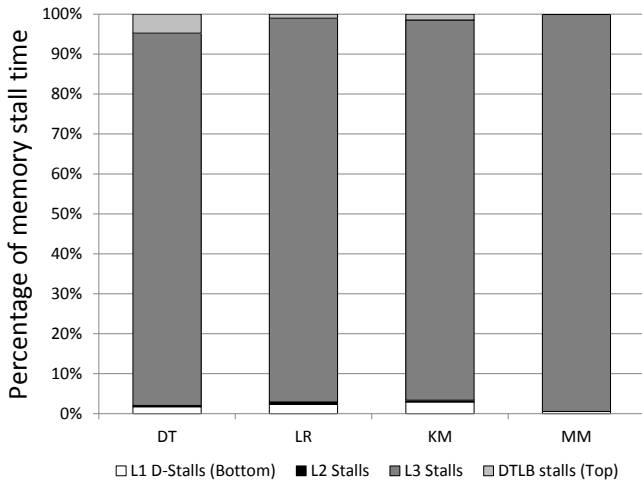


Figure 5: Backend memory stalls breakdown. DT: Decision Trees, LR: Linear Regression, KM: KMeans, MM: Matrix Multiplication.

This effect minimizes the contribution of the DTLB stalls to the memory stall time. Moreover, the high latency to service the L2 and L3 cache misses further diminishes the contribution of the DTLB and the L1D stalls to the memory stall time. Also, a miss in the L1D cache that hits in the L2 cache incurs very low latency, and can be overlapped with other computation due to the out-of-order pipelined execution units. Hence, the actual contribution of the L1D stalls to the backend memory stalls should be lower than reported. Since both the DTLB and the L1D stalls are very low, we do not consider them further in this paper.

For all the data analysis workloads, T_{L3} , which is DRAM latency-bound, is the dominant stall with nearly 95% of the memory stall time attributed to it. Even for the matrix multiplication workload, T_{L3} is the dominant stall component contributing nearly 99% of the memory stall time. However, matrix multiplication can also become L3 bound or L2 bound depending upon the size of the matrices. The reasons for stalls in both these workloads are discussed further in Sections 5.2.1 and Sections 5.2.2 respectively.

5.2.1 Stalls - Data analysis workloads:

As discussed above in Section 5.2, L3 stalls, which are bound by the DRAM-latency, contribute to about 95% of the memory stall time for the data analysis workloads. Although out-of-order execution of other instructions can overlap with some of these misses, the L3 miss stalls still dominate the stall time. This behavior is likely due to the exorbitant number of stall cycles for servicing a miss in the L3 cache from the DRAM. Past and current trends indicate that this number of cycles has only increased over the years, and this trend is likely to continue. With the current R kernel, L3 stalls will continue to worsen over time, and hence methods to focus on minimizing the number of L3 cache misses could pay rich (performance) dividends.

The L3 cache misses arise due to a lack of spatial locality in the access pattern of the allocated data. To better understand the reasons for this behavior from within the R kernel, we inspected and profiled the R kernel at various points of interest. Our investigation revealed that one of the

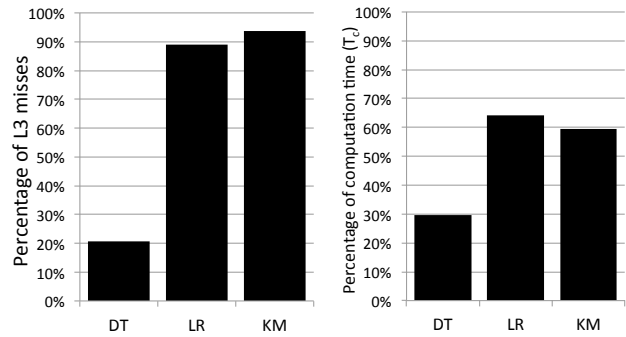


Figure 6: Percentage of L3 misses and percentage of computation time spent doing garbage collection. DT: Decision Tree, LR: Linear Regression, KM: KMeans.

major sources of the L3 cache misses is the garbage collector in R. As explained in Section 2, the garbage collector traverses circular doubly linked lists of pointers as part of its mark-and-sweep generational garbage collection method. Traversing a list of pointers has an adverse effect on the spatial locality of the processor caches, and hence increases the number of L3 misses. We also found that during the evaluation of the data analysis workloads, the R kernel allocates a large number of temporary objects from the heap, a majority of which are either unnecessary or duplicates of existing objects. These actions not only increase the memory pressure but also trigger the garbage collector frequently. We call this the *Create-Copy-GC* loop in R. This aspect is also the major reason that causes R to swap to disk even for datasets that are far smaller than the main memory size.

Impact of Garbage Collection

Figure 6 shows the impact of garbage collection on the L3 misses and the computation time (T_C) for the data analysis workloads. The linear algebra workload is not shown as it is compute-bound and does not trigger the garbage collector. As can be seen, garbage collection is the reason for nearly 90% of the L3 cache misses for the linear regression and the kmeans workloads. Also, both these workloads spend more than half their computation time doing garbage collection. Hence, minimizing the number of L3 misses due to garbage collection can greatly improve the performance of these two workloads.

When improving the R kernel will not greatly improve the performance of an R script? Figure 6 also shows that for the decision tree workload, only 20% of the total L3 misses are due to garbage collection, and this program spends 30% of its computation time in garbage collection. By further inspecting and profiling the *rpart* CRAN package, we find that it uses custom C code to implement certain parts of the recursive partitioning algorithm using the “Call” API, which allows calling custom C code from R. The custom C code manipulates data structures using pointers in a cache-insensitive manner and contributes to most of the remaining L3 cache misses. In such cases, optimizing the R kernel can boost performance only by a small amount. To get higher performance improvements, the decision tree implementation itself has to be modified to make it cache-conscious.

Computation Speed vs Memory Utilization Tradeoff. One possible way to reduce the number of L3 misses is to mini-

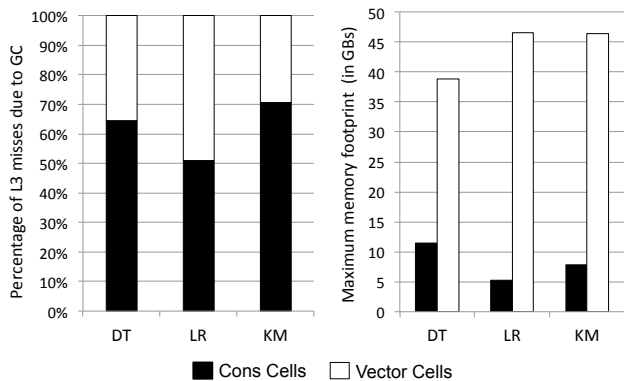


Figure 7: Left: Percentage of L3 misses caused by triggering the garbage collector due to cons cells and vector cells crossing their thresholds. Right: Maximum memory footprint of cons cells and vector cells. DT: Decision Tree, LR: Linear Regression, KM: KMeans.

mize the number of times that the garbage collector is triggered. Recall from Section 2 that the garbage collector is triggered when the number of cons cells allocated crosses a threshold (Thresh_{cons}), or the number of vector cells currently allocated crosses a threshold (Thresh_{vec}). R checks for both these conditions before it allocates any memory from the heap. These thresholds are also dynamically adjusted after every full collection to suit the memory demands of the current workload in R. Figure 7 shows the percentage of the L3 cache misses that are caused by triggering the garbage collector due to either of these two conditions. It also shows the maximum memory (in GBs) that is used by both the cons cells and the vector cells. It can be observed that although the maximum memory used by the cons cells ranges only between 10-30% of the total memory footprint, the percentage of L3 misses caused by the cons cells triggering the garbage collector is 50-70% of the total number of L3 misses across all the data analysis workloads.

To understand the reasons for this behavior, one needs to look into how R adjusts these thresholds. R uses a primitive model to increment or decrement the thresholds based on the current amount of heap memory that is allocated. The model for incrementing the thresholds is shown below. A very similar model is used to decrement the thresholds.

```

if(CurrentAllocatedcons > Threshcons * GROWFRACcons)
  Threshcons += GROWMINcons + GROWINCFRACcons * Threshcons

SizeNeededvec = CurrentAllocatedvec + RequestSizevec
if(SizeNeededvec > Threshvec)
  Threshvec = SizeNeededvec
if(SizeNeededvec > Threshvec * GROWFRACvec)
  Threshvec += GROWMINvec + GROWINCFRACvec * Threshvec

```

The CAPITALIZED variables in the above model are constants used by the garbage collector to control the values of the thresholds. For both the cons and the vector cells, when the currently allocated number of cells crosses a certain fraction of the current threshold, the threshold is incremented by a fixed value and a fraction of the current threshold. Note that Thresh_{vec} is also adjusted based on the allocation size requested when the garbage collector gets triggered. These

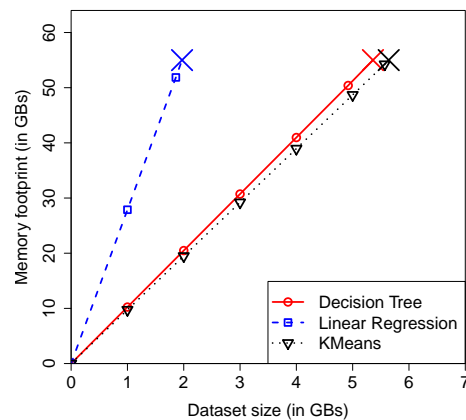


Figure 8: Memory footprint vs dataset size. The X mark indicates that the workload begins to swap.

thresholds provide a *space vs speed tradeoff* within the R kernel. The higher the thresholds, the less frequently the GC is triggered, but available memory decreases more rapidly, and vice-versa.

For our workloads, the $\text{GROWINCFRAC}_{cons}$ is set to a very low value compared to what is needed to deal with the rate at which cons cells are allocated. This setting causes Thresh_{cons} to get incremented slowly, and hence the GC is triggered frequently, thereby increasing the number of L3 cache misses for the cons cells. Setting this variable to higher values would have reduced the number of L3 misses due to cons cells¹. Although the R kernel provides options to change the values of these constants, it is cumbersome for a data scientist to determine how to choose these values. Wrong values can lead to increased memory consumption and hence cause the R program to swap quickly to disk, or increased computation time due to frequent garbage collection. Hence, an interesting direction for future work is to use a more robust model to tune the values of these constants automatically based on factors such as the available amount of main memory, the current allocated heap size, and the rate of memory allocation for the current workload.

We also conducted a sensitivity analysis of the GROWINCFRAC parameters – see [29] for more details.

Impact of temporary objects

Figure 8 shows the maximum memory footprint as the size of the dataset increases for the data analysis workloads. Note that the size of the dataset indicates the number of bytes occupied by the columns of the data frame (required for the computation) that are loaded in memory. As can be seen in the figure, R begins to swap for very small dataset sizes compared to the size of the DRAM. For example, the linear regression workload analyzes less than 2 GB of data before beginning to swap on our machine with 64 GB of RAM! This behavior is due to the large amount of memory allocated for temporary objects that are created during the evaluation

¹Increasing the $\text{GROWINCFRAC}_{cons}$ parameter for linear regression (to the exact value required by the algorithm) reduced the number of L3 misses due to cons cells to less than 1% of the total number of L3 misses. Decreasing it (to a small value) increased the number of L3 misses due to cons cells to 82% of the total number of L3 misses, and the computation time increased by 138%.

Workload	# of temp objects	Total malloc'ed (in GBs)	Total free'd (in GBs)
Decision Tree	235	247	226
Linear Regression	156	114	77
KMeans	93	92	66

Table 5: Number of temporary objects (>1MB) created, total memory allocated, and deallocated (in GB) in the data analysis workloads.

of the data analysis workloads. Table 5 shows the number of temporary objects (of size greater than 1 MB) that are created, the total amount of memory that is allocated, and the total amount of memory that is deallocated during the evaluation of the data analysis workloads. The total memory allocated is higher than the DRAM size because the garbage collector controls the maximum memory footprint that is in use at any point in time. Also, the total memory deallocated is during the evaluation of the workload; some objects may get deallocated during the next garbage collection cycle. Analyzing why R creates so many temporary objects revealed to us that while a few of them were necessary for running the R script, a majority of them were either unnecessary or duplicates of existing objects. Below, we present a *classification of such temporary objects* created in the R kernel during the evaluation of our workloads.

1. *Duplicates of existing objects.* The primary reason for creating duplicate objects stem from R’s functional programming paradigm where each function is supposed to be side-effect free. Being side-effect free means that the functions do not modify the state of the objects passed as arguments; instead, they create a duplicate copy of those objects (immutable objects). In some places, the R kernel does not properly decrement the reference count of an object when necessary. In such cases, duplicates get created due to the copy-on-write semantics. These duplicate objects get garbage collected later. Though immutability is a desired property for reasoning about the state of the objects, it incurs considerable overhead in terms of memory and processor utilization both for the copy and the garbage collection components. This behavior only gets worse as the size of the object increases. Our investigation shows that it is certainly possible to modify some of these objects in-place without affecting the correctness of the R script.

2. *Attributes that may not be required.* An example of this category is setting the *row.names* attribute on a data frame (A data frame is the equivalent of a SQL table in R). The *row.names* attribute provides a name to all the rows in a data frame (usually using the sequence from 1 to the number of rows). R creates such attributes by default even if these attributes may not be required by the user. Making such attributes optional can reduce the memory pressure.

3. *Intermediate objects during arithmetic operations.* An example of this aspect has been discussed in [33].

Example: `d <- sqrt((x-xs)^2+(y-ys)^2)`

We repeat part of their example as shown above. Here *x* and *y* are vectors, *xs* and *ys* are real values, `^2` finds the square of each element of the vector and returns a vector, `sqrt` finds the square root of each element of the vector and returns a vector. Based on the REPL explained in

Section 2.2, R creates intermediate objects for each of the following five expressions: *x-xs*, *y-ys*, $(x-xs)^2$, $(y-ys)^2$ and $(x-xs)^2+(y-ys)^2$. If unnecessary intermediate vectors could be overwritten, then the number of intermediate vectors could be brought down to one in this case (i.e, *y-ys* can be overwritten with $(y-ys)^2$, and *x-xs* can be overwritten with $(x-xs)^2$, followed by $(x-xs)^2+(y-ys)^2$, followed by the `sqrt` of that vector).

4. *Inefficient implementation.* An example of inefficient implementation occurs when some of our workloads check for missing values (NAs) in the input data frame. The goal of this step is to remove those rows that contain an NA. To do so, these scripts call an R function which does the following:

- First, it creates a logical vector (where each element is TRUE or FALSE) for each column in the data frame indicating whether the corresponding row in the column contains an NA.
- Second, it does a binary OR of these vectors, two at a time, to get the vector which indicates which rows in the data frame contain an NA value.
- Finally, it does a unary NOT of this vector to get the final vector that indicates which rows should be kept in the data frame.

This processing is inefficient as each operation creates multiple temporary objects, each of them the size of the number of rows in the data frame. An efficient implementation (which actually exists in R in the `stats` package, but not in the main kernel) is to check the whole data frame at once to find the rows that should be kept.

Another example of inefficient implementation is found when subsetting a data frame in R [9]. We explain this aspect using the following R script:

```
A <- data.frame(c(1,2,3), c(4,5,6))
B <- c(TRUE,FALSE,TRUE)
result <- A[B,]
```

The script creates a data frame (A), creates a logical vector (B), and subsets the data frame A using B. The first argument for subsetting indicates required rows (specified using the logical vector B) and the second argument indicates required columns (empty indicates all columns). During subsetting, the R kernel first creates an intermediate vector containing the required indices of the corresponding column based on the logical vector. It then uses these indices to create the vector that contains the actual values from the column. This step is repeated for every column in the example above, thus creating multiple intermediate vectors. Instead, it would be far more efficient to combine these two operations, thus avoiding the intermediate vectors.

Given this behavior of R, it is not surprising that R runs out of memory and begins to thrash quickly for datasets that are far smaller than the available main memory. This observation indicates that if R is to be used for large-scale data analytics, reducing the number of temporary/unnecessary objects should be a key area of focus going forward. However, the challenge lies in doing this in a user-transparent manner without affecting much of the ease-of-use of R.

5.2.2 Stalls in the Linear Algebra Workload:

As seen in figure 5, the L3 stalls dominate the matrix multiplication workload taking up more than 99% of the

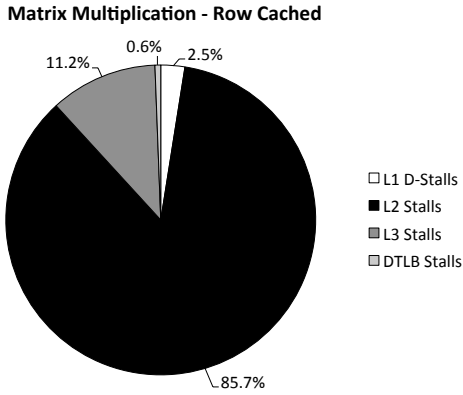


Figure 9: Memory stalls breakdown for the matrix multiplication workload where a row size fits into the L3 cache.

memory stall time. The matrix multiplication workload is compute-bound and does not trigger the garbage collector; thus, pointer-chasing is not the problem here. To understand the reasons for the stalls, one needs to look at the implementation of matrix multiplication in the R kernel. The C-pseudocode for multiplying two matrices X and Y , where NRX is the number of rows of X , NCX is the number of columns of X and NCY is the number of columns of Y , is shown below.

```

for (i = 0; i < NRX; i++)
  for (k = 0; k < NCY; k++) {
    sum = 0.0;
    for (j = 0; j < NCX; j++) {
      sum += X[i + j * NRX] * Y[j + k * NRY];
    }
    result[i + k * NRX] = (double) sum;
  }

```

Column-Oriented Storage Layout in R: Internally, a matrix in R is represented using an SEXP whose *metadata* contains an attribute named “class” with the value “matrix”. The elements of a matrix are stored in a VECSEXP after the header in a column-oriented layout contiguously in-memory. The data access pattern of the above algorithm is as follows: To calculate one row of `result`, the algorithm traverses one row of X and the entire matrix Y in a column-oriented manner. Hence, for the whole matrix multiplication, the entire matrices `result` and X are traversed in a row-oriented manner once. Whereas, the entire matrix Y is traversed in a column-oriented manner, NRX times.

When the processor caches cannot hold an entire row of the matrix X , each element accessed in X may result in an L3 cache miss. Also, when the entire matrix Y cannot be held in the cache, each cacheline (64 bytes) access of Y could miss in the L3 cache. In our workload, the number of elements in a row of matrix X is 2^{22} , and each element is 8 bytes. Hence, the row size is 32MB, which is greater than the L3 cache size (24MB) of our Xeon processor. Thus, accessing the matrix X in a row-oriented manner results in memory stalls due to L3 cache misses. However, each cacheline access of Y may not result in an L3 cache miss. This is due to the *hardware prefetchers* in the Xeon processor that can look at the access patterns of the load and store instructions and prefetch the required data. The Xeon processor contains a *Streamer* hardware prefetcher that monitors read, write and prefetch requests from the L1-D and L1-I cache, and when a forward

or backward stream of requests is detected, it prefetches the required cachelines [3]. To investigate this aspect, we also ran the matrix multiplication workload where a row size (and not the entire matrix) fits into the L3 cache (i.e. row size = 512KB, matrix size = 128MB). Figure 9 demonstrates the memory stall breakdown of this workload. Here, matrix multiplication becomes L3-bound indicating that most of the cachelines get prefetched into the L3 cache. However, they could also get prefetched into the L2 cache when the L2 cache is not heavily loaded with missing demand requests. In that case, the matrix multiplication workload would become L2-bound. This observation indicates that, if the matrix X was stored row-wise, then it would also get prefetched by the hardware prefetchers, and hence could reduce the number of L3 cache misses.

More broadly, providing alternate storage layouts for the users to choose from (or have a compiler choose it automatically), like a row-oriented storage layout, can help minimize stalls due to L3 misses for linear algebra operations that operate row-wise. Similar observations have been made before (e.g. [14]), but in a different setting of traditional analytic query processing. Also well-known techniques like blocking [23] could be used to speed up the implementation of the matrix operations in the R kernel.

5.3 Branch Misprediction and Frontend Stalls

Frontend stalls are infrequent and contribute 4-8% to the total execution time across all the workloads. The frontend can be stalled because of latencies in fetching or decoding the instructions, thus being unable to deliver the maximum capable number of four μ ops to the backend per cycle. For all our workloads, T_{ITLB} and T_{L1I} are an order of magnitude smaller than the other memory stall components, indicating that instruction fetching is not a bottleneck. Also, as discussed in Section 5.2, the number of instruction fetches that miss in the L2 cache is insignificant, probably because of the powerful instruction prefetching and branch prediction hardware units in modern hardware platforms that minimize the frontend stalls. The low ITLB misses are probably due to the smaller number of pages that are required to fit the instructions of the workloads.

Branch misprediction stalls are also low, ranging from 2-12% for all the workloads. This behavior is probably because the R kernel inlines most of its code with macros and inline functions, which reduces the number of branch predictions that are required. In addition, sophisticated hardware mechanisms are built into contemporary processors to reduce the cost of a branch misprediction, which further minimizes the branch misprediction stalls. In our Xeon processor, instructions and μ ops of incorrectly predicted paths are flushed as soon as a branch misprediction is detected. This method frees up resources in the backend so that the frontend can immediately start delivering μ ops of the correct branch, thus minimizing instruction starvation due to branch misprediction [31].

Since both these stalls depend upon the hardware and the compiler generating the sequence of instructions, the scope to optimize these stalls from within the R kernel is limited.

6. TOWARDS A MORE EFFICIENT R

In this section, we discuss ways to improve the processor and memory utilization of R. An obvious way is to improve the R kernel itself. Nevertheless, alternate ways to improve

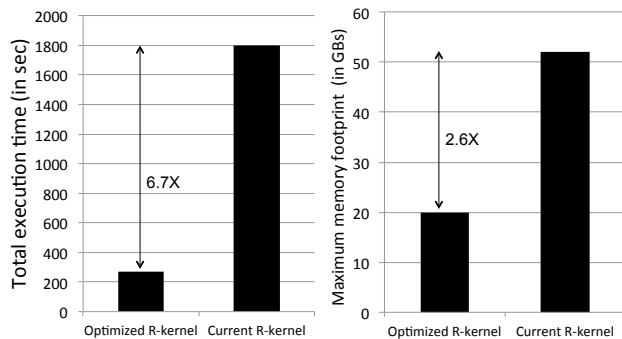


Figure 10: Left: Computation time comparison for the linear regression R-script in the optimized R kernel and the current R kernel. Right: Memory footprint comparison for the same.

at least the processor utilization are to provision R servers judiciously, and to consider (micro-) architectural changes. We discuss each of these aspects in this section.

6.1 Improving R: Research opportunities

Improving the R kernel not only improves the processor utilization but also the memory utilization of R, thus enabling analysis of larger datasets with the available main memory. Based on our study, we present the following research opportunities to enhance R.

1. The *Create-Copy-GC* loop is the major reason for R running out of main memory quickly for large datasets. Hence, compiler techniques to reduce the number of temporary objects can help bring down the memory pressure, and speed up the execution time for large datasets.
2. Triggering the garbage collector frequently is a major reason for the alarmingly high memory stalls in R. This behavior is because of the use of a primitive model for adjusting the thresholds responsible for triggering the garbage collector. Hence, applying ideas developed in the optimization and machine learning communities to adjust the thresholds dynamically using a more robust model can improve the performance of R programs.
3. Pointer-chasing during garbage collection is a major source of L3 cache misses in R. Ideas from the programming languages community (e.g. [17]) to place the garbage collection data structures in a cache-conscious layout could help minimize these L3 stalls.
4. R only provides a column-oriented storage layout for its matrices. Linear algebra operations, like matrix multiplication, that access them in a row-oriented manner suffer from memory stalls. Hence, ideas from the database and architecture communities to optimize the matrix storage layout, and using methods like blocked matrix operations could help minimize these stalls.

We hand-optimized the R kernel to not create unnecessary/duplicate temporary objects and tuned the thresholds to not fire the garbage collector frequently for the linear regression R script. Figure 10 shows the performance improvement achieved with these two optimizations in the R kernel for the linear regression R script. As can be seen, we were able to achieve a 6.7X decrease in the computation

time and 2.6X decrease in the memory footprint with these optimizations. However, the real challenge lies in realizing this potential automatically for all R scripts, and is a key goal for future work.

6.2 Provisioning R Servers: Deployment Optimizations

Processor utilization of the current R kernel can be improved by provisioning R servers in a judicious manner. Based on our experiments, we emphasize the following.

1. *Minimize the number of cores.* The R kernel is single-threaded and hence will only run on one core. Also, minimizing the number of cores reduces the complexity of cache coherency checks, and hence can reduce the access latencies to the processor caches.
2. *Maximize CPU clock cycles.* Improving the CPU frequency can speed up the serial execution. Again since the R kernel is single-threaded, this choice directly impacts the computation time.
3. *Minimize DRAM access latency.* Our experiments show that the L3 cache stalls dominate the memory stall time for most of the workloads. Hence, minimizing the DRAM access latency by using DDR memories with higher clock rate can bring down the stall time, and/or choosing configurations with larger L3 caches.

6.3 Hardware Enhancements: Research Opportunities

The R kernel, due to its Lisp origins, essentially operates on linked lists. Linked list traversal is a pointer-chasing operation that has a random access pattern with no spatial locality. Hence, each access can potentially lead to a cache miss. Existing prefetchers can only prefetch strided or sequential access patterns, and hence are not useful in this case. *Pointer prefetchers* are well studied in the computer architecture community to overcome this specific problem. However, they are not available in commercial hardware today. Proposals include to either use hardware only pointer prefetchers [18, 28], or to use hardware pointer prefetchers assisted by the software [16]. Examining some of these techniques for R can potentially minimize the number of cache misses due to pointer-chasing.

7. RELATED WORK

To the best of our knowledge, this is the first work to characterize the execution time breakdown of different algorithms in R on a contemporary processor. Much of the related work has focused on scaling R to larger datasets by making R work in out-of-memory scenarios (using DBMS, Hadoop, or simple files), or porting R to existing data processing environments.

The RIOT [33, 34] system optimizes I/O inefficiencies in R by implementing an optimization engine and a storage manager in an external R package. RIOT defines its own data structures that are complementary to R’s built-in vectors, matrices and arrays. Ricardo [19] and RHIPE [20] integrate R with Hadoop. Ricardo decomposes data analysis algorithms into parts that are executed by R, and parts that are executed by Hadoop. An R-Jaql bridge is used to communicate between R and Hadoop. RHIPE uses divide-and-recombine methods for statistical algorithms to parallelize large data computations in Hadoop. In all these cases,

existing algorithms have to be rewritten within the new environment. Also, Ricardo and RHIPE require setting up a Hadoop cluster which can be cumbersome, as opposed to using R as a standalone main memory program. Nevertheless, these out-of-memory implementations are aimed at extremely large datasets that do not fit in main memory. Our work is orthogonal to these as it focuses on identifying bottlenecks within the R kernel that limit the amount of data that can be analyzed in main memory. This main memory setting is common in many existing data science environments.

R has also been widely adopted in the enterprise. Revolution Analytics [6] offers a commercial distribution of R, and also incorporates parallel external memory implementations of popular data analysis algorithms that can scale and be distributed across nodes in a cluster. Similarly, pbdR [27] also offers external CRAN packages that enables higher levels of data parallelism across many nodes. Note that both Revolution and pbdR propose using R with multiple cores. Our work here is complementary to that work as it focuses on the single-threaded R kernel. Large enterprise database vendors like Oracle, IBM, Tibco, Pivotal, SAP and Teradata have incorporated R into their environments for large scale data analytics. Oracle R Enterprise (ORE) [4] integrates R with the Oracle database to provide in-database analytic capabilities for R and Oracle users. Teradata and IBM have partnered with Revolution Analytics to enable analytic algorithms to be run on their platforms. Tibco [10] has built an enterprise R runtime called TERR, which they claim provides better memory management capabilities than open-source R. Pivotal [12], like Oracle supports in-database analytics on their database, Greenplum. SAP has integrated R with their in-memory database [8], HANA, to allow using R for specific statistical functions. Vertica uses Presto [32] as it's R package. Our work complements these methods as efforts to improve the core R kernel carry over to these environments.

There is a growing list of CRAN packages that tackle the memory limitation of R [2]. Some notable packages are big-memory [21], biglm [25] and ff [11]. These packages store data in plain files on disk and provide efficient data analysis algorithm implementations by chunking the data, and swapping the required data in and out of memory. Although these packages do not require setting up a DBMS or Hadoop, their usage is limited to the algorithm that are implemented on these packages. They also have a disk storage layout that is different from R's internal data structure representation, and hence are not interoperable with most CRAN packages.

The concept of Reference Classes (RC) [5] in R was introduced recently. This concept enables an R user to control when to copy and/or modify objects in-place. However, the onus is now on the user to reason about the state of an object. Also, RC is not widely used, and the vast majority of the CRAN packages are still based on the (traditional) functional programming semantics in R.

8. CONCLUSIONS AND FUTURE WORK

This paper presents a dissection of R programs categorizing where time is spent when running these programs. We have identified interesting opportunities for future work including considering exporting and adapting database storage methods to R, presenting opportunities for architects to consider micro-architectural features that target improving

the performance of R programs, and opportunities for the programming languages community to enhance the R language for better performance. Our results indicate that the gap between the “bare metal performance” of the hardware today and what is exploited by R programs is large, presenting many opportunities for future research.

9. ACKNOWLEDGMENTS

This research was supported in part by a grant from the Microsoft Jim Gray Systems Lab, and by the National Science Foundation under grants IIS-1250886 and III-0963993.

10. REFERENCES

- [1] Airline On-time Dataset. <http://stat-computing.org/dataexpo/2009/>.
- [2] High Performance Task View Page. <http://cran.r-project.org/web/views/HighPerformanceComputing.html>.
- [3] Intel Performance Manual. <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>.
- [4] Oracle R Enterprise. <http://www.oracle.com/technetwork/database/database-technologies/r/r-enterprise/overview/index.html>.
- [5] R Reference Classes. <http://www.inside-r.org/r-doc/methods/ReferenceClasses>.
- [6] Revolution Analytics. <http://www.revolutionanalytics.com/>.
- [7] Rexer Analytics Data Miner Survey 2013. <http://www.rexeranalytics.com/Data-Miner-Survey-Results-2013.html>.
- [8] SAP HANA R Integration Guide. https://help.sap.com/hana/SAP_HANA_R_Integration_Guide_en.pdf.
- [9] Subsetting a data frame in R. <http://stat.ethz.ch/R-manual/R-patched/library/base/html/Extract.data.frame.html>.
- [10] Tibco TERR. <http://spotfire.tibco.com/discover-spotfire/what-does-spotfire-do/predictive-analytics/tibco-enterprise-runtime-for-r-terr>.
- [11] D. Adler, C. Glaser, O. Nenadic, J. Oehlschlagel, and W. Zucchini. *ff: memory-efficient storage of large data on disk and fast access functions*, 2013. R package version 2.2-12.
- [12] P. A. T. at Pivotal Inc. and with contributions from Data Scientist Team at Pivotal Inc. *PivotalR: R front-end to PostgreSQL and Pivotal (Greenplum) database, wrapper for MADlib*, 2014. R package version 0.1.15.1.
- [13] H. G. Baker. The treadmill: real-time garbage collection without motion sickness. *ACM Sigplan Notices*, 27(3):66–70, 1992.
- [14] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.
- [15] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
- [16] I. Burcea, L. Soares, and A. Moshovos. Pointy: a hybrid pointer prefetcher for managed runtime systems. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 97–106. ACM, 2012.
- [17] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–74, 2000.
- [18] J. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 62–73. IEEE Computer Society Press, 2002.
- [19] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: integrating r and hadoop. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 987–998. ACM, 2010.
- [20] S. Guha, R. Hafen, J. Rounds, J. Xia, J. Li, B. Xi, and W. S. Cleveland. Large complex data: divide and recombine (d&r) with rhipe. *Stat*, 1(1):53–67, 2012.
- [21] M. J. Kane, J. Emerson, and S. Weston. Scalable strategies for computing with massive data. *Journal of Statistical Software*, 55(14):1–19, 2013.
- [22] J. King and R. Magoulas. *2013 Data Science Salary Survey*. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA, 2014.
- [23] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74, 1991.
- [24] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.
- [25] T. Lumley. *biglm: bounded memory linear and generalized linear models*, 2013. R package version 0.9-1.
- [26] J. C. McCallum. Memory prices (1957-2013). <http://www.jcm.it/memoryprice.hta>.
- [27] G. Ostrouchov, W.-C. Chen, D. Schmidt, and P. Patel. Programming with big data in r, 2012.
- [28] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *ACM SIGOPS Operating Systems Review*, volume 32, pages 115–126. ACM, 1998.
- [29] S. Sridharan and J. M. Patel. Profiling R on a contemporary processor (Supplementary material). <http://quickstep.cs.visc.edu/pubs/dissecting-R-ext.pdf>.
- [30] T. Therneau, B. Atkinson, and B. Ripley. *rpart: Recursive Partitioning*, 2013. R package version 4.1-3.
- [31] M. E. Thomadakis. The architecture of the nehalem processor and nehalem-ep smp platforms. *Resource*, 3:2, 2011.
- [32] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: distributed machine learning and graph processing with sparse matrices. In *EuroSys*, pages 197–210, 2013.
- [33] Y. Zhang, H. Herodotou, and J. Yang. Riot: I/o-efficient numerical computing without sql. *arXiv preprint arXiv:0909.1766*, 2009.
- [34] Y. Zhang, W. Zhang, and J. Yang. I/o-efficient statistical computing with riot. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 1157–1160. IEEE, 2010.