Profiling Rexx with **bpf** and **perf**

René Vincent Jansen, Performance Architect, Department of Finance - Customs
The Netherlands
November 2021
BPF

- BPF (Berkely Packet Filter) is part of the kernel
  - Available in every modern Linux kernel
  - Perf and bpftrace are tools for working with bpf

- ‘Kernel VM’ programs, can be made with bcc (hard), bpftrace (easier)
- Perf is a command that is useful for most profiling/sampling/tracing actions
Top

Every Linux has this
Look at the system

- Top
- NMON
- Perf top
This needs installing.

It will also give you a performance overview on the process level.
**BPF and Perf**

BPF ("Berkeley Packet Filter") and perf-events are part of the Linux kernel. There is a ‘kernel VM’ that enables one to write small programs to be executed by the kernel in a controlled way.

**perf**, **top** and **iostat** are, nowadays, built on top of that. Perf and bpftrace need to be installed.

**bcc** and **bpftrace** are ways to make these kernel vm programs, the latter being modelled on **awk**.
Profile -why

- Good question!
- We want to know where the time is spent. Process-level is not enough to give developers clues on what to speed up.
- Not every program is optimally designed and implemented
- I will present a short series of (5) examples to give you an idea how much structures and algorithms can influence the performance of simple tasks
- [these are a bit contrived (running 10000 times) to show some mechanisms clearly.]
Perf stat

We want to know more than the ‘time’
Dhrystone(1.1) time for 50000000 passes = 4
This machine benchmarks at 11029411 dhrystones/second

Performance counter stats for './dhrystone':

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.743,86 msec task-clock</td>
<td>2.743,86 msec</td>
<td>1,000 CPUs utilized</td>
</tr>
<tr>
<td>9 context-switches</td>
<td>9</td>
<td>0.003 K/sec</td>
</tr>
<tr>
<td>1 cpu-migrations</td>
<td>1</td>
<td>0.000 K/sec</td>
</tr>
<tr>
<td>54 page-faults</td>
<td>54</td>
<td>0.020 K/sec</td>
</tr>
<tr>
<td>10.893.802.753 cycles</td>
<td>10.893.802.753 cycles</td>
<td>3,970 GHz</td>
</tr>
<tr>
<td>29.661.436.752 instructions</td>
<td>29.661.436.752</td>
<td>2,72 insn per cycle</td>
</tr>
<tr>
<td>3.752.113.617 branches</td>
<td>3.752.113.617</td>
<td>1367,460 M/sec</td>
</tr>
<tr>
<td>71.092 branch-misses</td>
<td>71.092</td>
<td>0.00% of all branches</td>
</tr>
</tbody>
</table>

2,744234624 seconds time elapsed
2,744167000 seconds user
0,000000000 seconds sys
We have a text

KJV= King James Version of the Bible

4521345 bytes (4.4 MB)

We try for 10000 times to find the last verse of Revelations

The first attempts are not particularly bright
We have a Class

This is a container. We make an instance of it for every line from the text, and use PARSE to fill it.

Only the later programs in the series are using it
First try: we loop with I/O

```java
class Read_Text_Loop
{
    method Read_Text_Loop()
    {
        main(args={}static
        r=Read_Text_Loop()
        textline=''
        loop for 10000
        source = BufferedReader(FileReader('./data/kjvdat.txt'))
        loop forever
        lastline=textline
        textline = source.ReadLine
        if textline = null then leave
        end
        end
        say lastline
    }
}
```

→ bpfprobes git:(master) perf stat java -XX:+PreserveFramePointer Read_Text_Loop
Rev|22|21| The grace of our Lord Jesus Christ be with you all. Amen.

Performance counter stats for `java -XX:+PreserveFramePointer Read_Text_Loop`:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>task-clock</td>
<td>54.821,02 msec</td>
</tr>
<tr>
<td>context-switches</td>
<td>33.517</td>
</tr>
<tr>
<td>cpu-migrations</td>
<td>4.825</td>
</tr>
<tr>
<td>page-faults</td>
<td>74.516</td>
</tr>
<tr>
<td>cycles</td>
<td>214.181.041.636</td>
</tr>
<tr>
<td>instructions</td>
<td>634.317.494.047</td>
</tr>
<tr>
<td>branches</td>
<td>141.624.924.841</td>
</tr>
<tr>
<td>branch-misses</td>
<td>1.353.808.195</td>
</tr>
</tbody>
</table>

54,334276354 seconds time elapsed

49,806402000 seconds user
5,269838000 seconds sys
Second Program: another sort of Read loop

Here we are assigning the content to an instance of ChapterAndVerse

Apart from the construct, does more or less the same. We read only once. Notation-wise is the base for the examples; it is much shorter; also this is the performance baseline.

```java
CLASS Read_Text_OneLine

PROPERTIES INHERITABLE
AL = ArrayList()

METHOD Read_Text_OneLine()
    REMXIO().FILE(’/data/kjvdat.txt’).FOREACHLINE(this.docid())

METHOD main(args=String[]) STATIC
    R=Read_Text_OneLine()

CLASS Read_Text_OneLine.docid DEPENDENT IMPLEMENTS LineHandler

METHOD HANDLE(in)
    PARSE IN BOOK ’|’ CHAPTER ’|’ VERSE ’|’ TEXTLINE
    A = ChapterAndVerse(book,chapter,verse,textline)
    PARENT.AL.ADD(A)

➜ bpfsprobes git:(master) perf stat java -XX:+PreserveFramePointer Read_Text_OneLine

Performance counter stats for 'java -XX:+PreserveFramePointer Read_Text_OneLine':

<table>
<thead>
<tr>
<th>Counter</th>
<th>Value</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>task-clock</td>
<td>312.48 ms</td>
<td></td>
</tr>
<tr>
<td>context-switches</td>
<td>369</td>
<td></td>
</tr>
<tr>
<td>cpu-migrations</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>page-faults</td>
<td>15,988</td>
<td></td>
</tr>
<tr>
<td>cycles</td>
<td>1,195,562,392</td>
<td></td>
</tr>
<tr>
<td>instructions</td>
<td>1,440,566,467</td>
<td></td>
</tr>
<tr>
<td>branches</td>
<td>264,599,270</td>
<td></td>
</tr>
<tr>
<td>branch-misses</td>
<td>7,326,681</td>
<td></td>
</tr>
<tr>
<td>2,012 CPUs utilized</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.001 M/sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.166 K/sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.051 M/sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.826 GHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.20 insn per cycle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>846,775 M/sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.77% of all branches</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0.155274554 seconds time elapsed
0.289902000 seconds user
0.024498000 seconds sys
Avoiding the I/O

We have an ArrayList

We add instances of ChapterAndVerse

Now we read only once and loop 10000 times through this, until we are at the end.

We are not looking or comparing yet.
Now look for the right verse in the ArrayList

We again loop through the ArrayList that we filled once

But now we search for the book of Revelations 22:21

Which is the last line we added to the ArrayList

```
CLASS Read_Text_OneLine2

PROPERTIES INHERITABLE
ACL = ArrayList()

METHOD Read_Text_OneLine2()
  RxxxId().File('./data/kyvdat.txt').forEachLine(this.docId)

METHOD main(args=String[]) STATIC
r=Read_Text_OneLine2()
loop for 10000
  loop i=0 to r_AL.size()-1
    a = chapterAndVerse(r_AL.get(i))
    if a.book='Rev' then
      if a.chapter='22' then
        if a.verse='21' then leave
      end
    end
  end
end
say a

CLASS Read_Text_OneLine2.docId DEPENDENT IMPLEMENTS LineHandler

METHOD handle(in)

parse in book '|' chapter '|' verse '|' textLine
a = chapterAndVerse(book,chapter,verse,textLine)
pARENT.AL.add(a)
```

> bpfspbrobes git:(master) perf stat java -XX:+PreserveFramePointer Read_Text_OneLine2

Rev 22 21  The grace of our Lord Jesus Christ be with you all. Amen.

Performance counter stats for 'java -XX:+PreserveFramePointer Read_Text_OneLine2':

```
27.941,67 msec task-clock  # 1,013 CPUs utilized
6.797 context-switches    # 0,243 K/sec
1.083 cpu-migrations      # 0,039 K/sec
92.130 page-faults        # 0,003 M/sec
110.927.691.194 cycles    # 3,970 GHz
240.753.224.323 instructions    # 2,17 insn per cycle
46.301.682.611 branches    # 1657,084 M/sec
49.056.142 branch-misses  # 0,11% of all branches

27.572957882 seconds time elapsed
27.810831000 seconds user
0,176500000 seconds sys
```
What happens when we want an earlier line

So so the time spent into searching in an Array is dependent of the position of the target string.
Let’s redo this with another structure, the TreeMap

A TreeMap is a keyed structure in which the keys and their values are stored in a sorted way.

This means looking up the key can be done by a binary search algorithm, that is conveniently hidden from us.
Now try to find an earlier verse

Because of binary search, this is not quicker than looking for the last verse.
sudo perf top

Because sometimes you cannot run programs in isolation, and need a broad picture about what’s happening
The Dhrystone benchmark

To generate some load for `perf top` and see where its time is spent.

The `-Wno-implicit-function-declaration` is only for the M1 Mac because the source is so old and `clang` does not like it. There are no symbols or debug options selected.

```c
#include <stdio.h>

int main() {
    int i, j, k;
    int iold = 0, jold = 0, kold = 0;

    // Dhrystone benchmark
    while (INTLOC1 < INTLOC2) {
        INTLOC3 = INTLOC1 - INTLOC2;
        PROC7(INTLOC1, INTLOC2, &INTLOC3);
        ++INTLOC1;
    }
    PROC8(Array1Glob, Array2Glob, INTLOC1, INTLOC3);
    PROC1(PTRLO);
    for (CHARINDEX = 'A'; CHARINDEX <= CHARINDEX + CHARINDEX; ++CHARINDEX)
        if (ENUMLOC == FUNC1(CHARINDEX, 'C'))
            PROC6(IDENT1, GENUMLC);
    INTLOC3 = INTLOC2 + INTLOC1;
    INTLOC2 = INTLOC3 / INTLOC1;
    INTLOC2 = INTLOC3 - INTLOC2 - INTLOC1;
    PROC2(GRUNTLOC1);

    // Stop timer
    while (INTLOC1 < INTLOC2) {
        INTLOC3 = INTLOC1 - INTLOC2;
        PROC7(INTLOC1, INTLOC2, &INTLOC3);
        ++INTLOC1;
    }
    PROC8(Array1Glob, Array2Glob, INTLOC1, INTLOC3);
    PROC1(PTRLO);
    for (CHARINDEX = 'A'; CHARINDEX <= CHARINDEX + CHARINDEX; ++CHARINDEX)
        if (ENUMLOC == FUNC1(CHARINDEX, 'C'))
            PROC6(IDENT1, GENUMLC);
    INTLOC3 = INTLOC2 + INTLOC1;
    INTLOC2 = INTLOC3 / INTLOC1;
    INTLOC2 = INTLOC3 - INTLOC2 - INTLOC1;
    PROC2(GRUNTLOC1);

    // Print time
    printf("Dhrystones(%d) time for %ld passes = %.0f\n",
           VERSION, (LONG) LOOPS, BENCHTIME);
    printf("This machine benchmarks at %.0f dhrystones/second\n",
           ((LONG) LOOPS) / BENCHTIME);

    // Get usage
    getrusage(RUSAGE_SELF, &ru);
    double t = (double)(endtime.ru_utime.tv_sec
                       - starttime.ru_utime.tv_sec
                       - nulltime.tv_sec) + (double)(endtime.ru_utime.tv_usec
                                                  - starttime.ru_utime.tv_usec
                                                  - nulltime.tv_usec) * 1e-6;
    printf("Dhrystones(%d) time for %ld passes = %.1f\n",
           VERSION, (LONG) LOOPS, t);
}
```
perf top of the functions within a process

We see exactly which functions in the program used the majority of the processor cycles. This gives us a handle on the optimising process.
Now let's run our first Java program

The one with the loop

Very disappointing, isn't it?

What happens here? We have only addresses, no human-readable symbols.

This is caused by the JIT process, which does Just-In-Time compiling to native (instruction set architecture dependent) machine code.
Symbols to the rescue

- https://github.com/jvm-profiling-tools/perf-map-agent
- Set JAVA_HOME
- cmake .
- make
Create a map with java-perf-map

- `java -XX:+PreserveFramePointer <your_class>`
- `sudo perf record -F 99 -p `pgrep java` -g -- sleep 10`
- `~/apps/perf-map-agent/bin/create-java-perf-map.sh `pgrep java``
- `sudo perf script >out.perf`
- But for now, we are going to look at `perf top`

Linux perf tools will expect symbols for code executed from unknown memory regions at `/tmp/perf-<pid>.map`. This allows runtimes that generate code on the fly to supply dynamic symbol mappings to be used with the perf suite of tools.
Life is better with symbols

The same program, but now it has a usable mapping from addresses to symbols, provided to us by the JIT compiler.

We see all the time is spent in I/O, in java/io/BufferedReader::readLine
Let’s revisit this one

We’ll run this through **perf top**

```java
// Read_Text_OneLine2.java

class Read_Text_OneLine2
{
    properties
    
    method Read_Text_OneLine2()
    
        MyClass.DoSomething();

    method main(args=String[]) static
    
        MyClass obj;
        obj.doSomething();

    loop for 10000
        MyClass obj;
        MyClass.DoSomething();

    end
}

class Read_Text_OneLine2.docid dependent implements LineHandler
{
    method handle(in)
    
        MyClass obj;
        MyClass.DoSomething();
}
```

**Performance counter stats for 'java -XX:+PreserveFramePointer Read_Text_OneLine2':**

```
27.941,67 msec task-clock # 1,013 CPUs utilized
6.797 context-switches # 0,243 K/sec
1.083 cpu-migrations # 0,039 K/sec
92.130 page-faults # 0,003 M/sec
110.927.691.194 cycles # 3,970 GHz
240.753.224.323 instructions # 2,17  insn per cycle
46.301.682.611 branches # 1657,084 M/sec
49.056.142 branch-misses # 0,11% of all branches

27,572957882 seconds time elapsed
27,810831000 seconds user
0,176500000 seconds sys
```
Hmm

We see that most time is spent in `Rexx::docompare` and `Rexx::OpAdd`

Why would that be?

Well, the compare is because of the comparison of every ArrayList element, which contains an instance of ChapterAndVerse, to the 3 strings.

The OpAdd is the loop counter. In this case …
We don’t need decimal loop counters

I changed that as can be seen on the right, using the DO ... BINARY block delimiter.

That alone shaves off about 20 seconds.

---

```java
class Read_Text_Donline2b

properties inheritable

AL = ArrayList()

method read_text_donline2b()
    regex().file('/data/kjvdat.txt').foreachline(this.docid())

method main(args=String[]) static
    r = read_text_donline2b()
    do binary
        loop for 10000
            loop i = 0 to r.al.size()-1
                a = chapterandverse r.al.get(i)
                if a.book == 'rev' then
                    if a.chapter == '22' then
                        if a.verse == '21' then leave
                    end
                end
            end
        end
        say a
    end

class Read_Text_Donline2b.DOCID dependent implements LineHandler

method handle(in)
    parse in book '|' chapter '|' verse '|' textline
    a = chapterandverse(book, chapter, verse, textline)
    parent.al.add(a)
```

→ `bpfprobes git:(master) perf stat java -XX:+PreserveFramePointer Read_Text_Donline2b`

Rev 22 21  The grace of our Lord Jesus Christ be with you all. Amen.~

Performance counter stats for 'java -XX:+PreserveFramePointer Read_Text_Donline2b':

```
<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>task-clock</td>
<td>8.362.75 ms</td>
<td></td>
</tr>
<tr>
<td>context-switches</td>
<td>680</td>
<td>1,027 CPUs utilized</td>
</tr>
<tr>
<td>cpu-migrations</td>
<td>55</td>
<td>0.081 K/sec</td>
</tr>
<tr>
<td>page-faults</td>
<td>16.418</td>
<td>0.007 K/sec</td>
</tr>
<tr>
<td>cycles</td>
<td>19.206.783.758</td>
<td></td>
</tr>
<tr>
<td>instructions</td>
<td>15.628.203.011</td>
<td></td>
</tr>
<tr>
<td>branches</td>
<td>3.228.597.088</td>
<td></td>
</tr>
<tr>
<td>branch-misses</td>
<td>9.766.693</td>
<td></td>
</tr>
<tr>
<td>elapsed</td>
<td>8,145946642 seconds</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td>8,325632000 seconds</td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td>0,044008000 seconds</td>
<td></td>
</tr>
</tbody>
</table>
```

8.362.75 ms task-clock # 1,027 CPUs utilized
680 context-switches # 0,081 K/sec
55 cpu-migrations # 0,007 K/sec
16.418 page-faults # 0,002 M/sec
19.206.783.758 cycles # 2,297 GHz
15.628.203.011 instructions # 0,81  insn per cycle
3.228.597.088 branches # 386,069 M/sec
9.766.693 branch-misses # 0,30% of all branches

8,145946642 seconds time elapsed
Profiling Rexx BIFs
/** REXX */
OPTIONS LEVELB
/* SUBSTR */
SAY "Look for SUBSTR OK"
/* DO 100000 */
/* These from the REXX book. */
/* SAY '='<SUBSTR('ABC',2)'=' */
IF SUBSTR('ABC',2) \= 'BC' THEN SAY 'FAILED IN TEST' 1
IF SUBSTR('ABC',2,4) \= 'BC ' THEN SAY 'FAILED IN TEST' 2
IF SUBSTR('ABC',2,6,')' \= 'BC....' THEN SAY 'FAILED IN TEST' 3
/* These from Mark Hessling. */
IF SUBSTR("FOOBAR",2,3) \= "OOB" THEN SAY 'FAILED IN TEST' 4
/* SAY '='<SUBSTR("FOOBAR",3)'=' */
IF SUBSTR("FOOBAR",3) \= "OABR" THEN SAY 'FAILED IN TEST' 5
IF SUBSTR("FOOBAR",3,6) \= "OABR " THEN SAY 'FAILED IN TEST' 6
IF SUBSTR("FOOBAR",3,6,"*') \= "OABR**" THEN SAY 'FAILED IN TEST' 7
IF SUBSTR("FOOBAR",6,3) \= "R " THEN SAY 'FAILED IN TEST' 8
IF SUBSTR("FOOBAR",8,3) \= " " THEN SAY 'FAILED IN TEST' 9
IF SUBSTR('1234567890',5) \= '567890' THEN SAY 'FAILED IN TEST' 10
/* SAY '='<SUBSTR('1234567890',5)'=' */
IF SUBSTR('1234567890',6,6,"'') \= '67890." THEN SAY 'FAILED IN TEST' 11
IF SUBSTR('ABC',2,4,"'') \= 'BC.' THEN SAY 'FAILED IN TEST' 12
IF SUBSTR('ABCDEFGH',1,2,"'') \= 'AB' THEN SAY 'FAILED IN TEST' 13
IF SUBSTR('ABCDEFGH',2,3,"é') \= 'BCD' THEN SAY 'FAILED IN TEST' 14
IF SUBSTR("René Vincent Jansen",1,4,"'') \= 'René' THEN SAY 'FAILED IN TEST'
IF SUBSTR("René Vincent Jansen",6,7,"'"') \= 'Vincent' THEN SAY 'FAILED IN TEST'
IF SUBSTR("12345678",5,6,"É") \= '5678ÉÉ' THEN SAY 'FAILED IN TEST' 17
/* SAY SUBSTR("12345678",10,6,"é") */
/* IF SUBSTR("12345678",10,6,"Éé") \= 'ÉÉÉÉÉÉ' THEN SAY 'NEED EXCEPTIONS FOR THIS */
/* END */
SAY "SUBSTR OK"
RETURN
<table>
<thead>
<tr>
<th>Function</th>
<th>Time (msec)</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBSTR*</td>
<td>4.72</td>
<td>ooRexx</td>
</tr>
<tr>
<td></td>
<td>1.36</td>
<td>BREXX</td>
</tr>
<tr>
<td></td>
<td>0.77</td>
<td>Regina</td>
</tr>
<tr>
<td></td>
<td>0.48</td>
<td>cREXX - Rexx Version</td>
</tr>
<tr>
<td></td>
<td>0.43</td>
<td>cREXX - RXAS version</td>
</tr>
</tbody>
</table>

* Unicode testcases skipped except for cREXX
1) We need to run this in a loop to see significant CPU usage

2) The memory overhead might be of that loop

3) Still, we clearly see the relative CPU profile of the called functions
SUBSTR in RXAS

Well, 3/4 of it

Based on algorithm in ANSI standard
SUBSTR in level B cREXX

.. for an impression, all the code is in:

https://github.com/adesutherland/CREXX

The clarity of this, coupled with the almost not measurable performance difference, made us decide to implement most BIF’s in Rexx.

(Which Peter subsequently did).
CPS: The Clauses Per Second Benchmark

ARM is on the move
The latest from MFC’s Speleotrove

<table>
<thead>
<tr>
<th>Date</th>
<th>RexxCPS</th>
<th>Hardware</th>
<th>Software environment</th>
<th>-----</th>
<th>---</th>
<th>---</th>
<th>---</th>
</tr>
</thead>
<tbody>
<tr>
<td>2021.10.28</td>
<td>23,774,392</td>
<td>M1 Mac ARM 64</td>
<td>Darwin ooRexx_5.0.0</td>
<td>6.05</td>
<td>14</td>
<td>Sep</td>
<td>2021</td>
</tr>
<tr>
<td>2015.03.06</td>
<td>19,413,819</td>
<td>IBM z13</td>
<td>CMS REXXC370</td>
<td>4.02</td>
<td>23</td>
<td>Dec</td>
<td>1999</td>
</tr>
<tr>
<td>2013.07.04</td>
<td>17,778,252</td>
<td>IBM zEC12 2827-789</td>
<td>CMS REXXC370</td>
<td>4.02</td>
<td>23</td>
<td>Dec</td>
<td>1999</td>
</tr>
<tr>
<td>2021.10.28</td>
<td>15,928,590</td>
<td>M1 Mac ARM 64</td>
<td>Unix Regina 3.9.3</td>
<td>5.00</td>
<td>5</td>
<td>Oct</td>
<td>2019</td>
</tr>
<tr>
<td>2012.01.01</td>
<td>14,766,746</td>
<td>Intel i5 2.5 GHz</td>
<td>Win7 DosCrx1.0</td>
<td>5.00</td>
<td>22</td>
<td>Apr</td>
<td>1999</td>
</tr>
<tr>
<td>2021.08.30</td>
<td>14,418,411</td>
<td>iMac Apple Silicon M1</td>
<td>Darwin</td>
<td>6.05</td>
<td>12</td>
<td>Aug</td>
<td>2021</td>
</tr>
<tr>
<td>2011.06.00</td>
<td>14,126,688</td>
<td>IBM z196 2817-742</td>
<td>CMS REXXC370</td>
<td>4.02</td>
<td>23</td>
<td>Dec</td>
<td>1999</td>
</tr>
<tr>
<td>2020.06.14</td>
<td>12,500,000</td>
<td>Lenovo T540-15ICB</td>
<td>Win10-64 ooRexx 4.2.0</td>
<td>6.04</td>
<td>22</td>
<td>Feb</td>
<td>2014</td>
</tr>
<tr>
<td>2020.01.27</td>
<td>11,494,253</td>
<td>Lenovo T540-15ICB</td>
<td>Win10-64 Regina 3.9.3</td>
<td>5.00</td>
<td>5</td>
<td>Oct</td>
<td>2019</td>
</tr>
<tr>
<td>2011.06.08</td>
<td>10,135,135</td>
<td>Intel i7 4.7 GHz</td>
<td>Win7 ooRexx 4.1.0</td>
<td>6.03</td>
<td>5</td>
<td>Dec</td>
<td>2010</td>
</tr>
<tr>
<td>2014.05.05</td>
<td>8,287,671</td>
<td>Pentium G3220 3 GHz</td>
<td>Win7-64 Regina 3.7</td>
<td>5.00</td>
<td>?</td>
<td>7</td>
<td>2014</td>
</tr>
<tr>
<td>2014.01.08</td>
<td>7,665,816</td>
<td>Intel Xeon 3.5 GHz</td>
<td>Win7 ooRexx 4.1.2</td>
<td>6.03</td>
<td>28</td>
<td>Aug</td>
<td>2012</td>
</tr>
<tr>
<td>2012.05.25</td>
<td>6,675,567</td>
<td>Intel i5 2.5 GHz</td>
<td>Win7 ooRexx 4.1.0</td>
<td>6.03</td>
<td>5</td>
<td>Dec</td>
<td>2010</td>
</tr>
<tr>
<td>2012.03.26</td>
<td>6,192,687</td>
<td>Xeon 3.1GHz 4-way</td>
<td>Linux jREXX</td>
<td>0.0.3</td>
<td>26</td>
<td>Mar</td>
<td>2012</td>
</tr>
<tr>
<td>2001.08.09</td>
<td>5,567,929</td>
<td>AMD Athlon 1.4 GHz</td>
<td>DosCrx1.0,16 bit</td>
<td>5.0</td>
<td>2</td>
<td>Dec</td>
<td>1999</td>
</tr>
</tbody>
</table>
Platforms

Different platforms, different scores

Need a performance regression section on the RexxLA Jenkins build machine.
The End. For the moment.