30 Years of CPS

The Rexx Clauses-per-second benchmark

René Jansen - RexxLA 2019
The Standard by MFC

- Measures Clauses Per Second
- A clause is ~ a line of instructions
- History of measurements since 1989
- Product of analysis of 1000’s of lines of real applications
- Multiplatform Classic Rexx
What does it do?

- Two loops
  - One for calibration of an (almost) empty loop
  - One that does the work
- Can specify # of averaging and measuring loops
- Total execution should be 1 sec elapsed time for dependable results
Calibrating the empty loop

CPS EXEC B1 V 130 Trunc=130 Size=141 Line=0 Col=1 Alt=0

!!!

... +1 +2 ... +3 ... +4 ... +5 ... +6 ...

00000 * * * Top of File * * *

00001 ------------------------ 39 line(s) not displayed ------------------------

00040 /* Calibrate for the empty do-loop */

00041 empty=0

00042 do i=1 to averaging

00043 call time 'R'

00044 do count; end

00045 empty=time('R')+empty

00046 end

00047 empty=empty/averaging

00048

00049 noterm=(system='CMS'); if pos('0',tracevar)=1 then noterm=0

00050 if noterm then do

00051 say 'Calibration (empty DO):' empty 'secs (average of: averaging')

00052 say 'Spooling trace NOTERM'

00053 'CP SPOOL CON * START NOTERM'; 'CP CLOSE CON PUR'

00054 end

00055 ------------------------ 87 line(s) not displayed ------------------------

00142 * * * End of File * * *
The timer loop

```c
/* ----- This is first of the 1000 clauses ----*/
flag=0; p0='b'
do loop1 to 14
    /* This is the "block" comment in loop */
    key='Key Bee'
    acompound.key1.loop=substr(1234"5678",6,2)
    if flag=acompound.key1.loop then say 'Failed1'
    do j=1.1 to 2.2 by 1.1 /* executed 28 times */
        if j=acompound.key1.loop then say 'Failed2'
        if 17<length(j)-1 then say 'Failed3'
        if j='foo_bar' then say 'Failed4'
        if substr(1234,1,1)=9 then say 'Failed5'
        if word(key,1)="'" then say 'Failed6'
        if j<5 then do /* This path taken */
            acompound.key1.loop=acompound.key1.loop+1
        end
        end
        iterate
        end /* j */
    avar.1='0'
    'loop
select
    when flag='string' then say 'Failed1'
    when avar.flag.2=0 then say 'Failed2'
    when flag=599.7 then say 'Failed3'
    when flag then avar.1.2=avar.1.2*1.1
    when flag=0 then flag=0
end
```

/* ----- This is last of the 1000 clauses ----*/
```
Hardware speed and REXX

As hardware speed increases, REXX is being used for a wider set of applications. Some informal figures:

- x86 systems—now over 27,000 REXX clauses per second (486/66)
- RISC systems—over 42,000 REXX cps (same interpreter)
- Mainframe systems—over 90,000 REXX cps
- REXX Compiler/370—up to 465,000 REXX cps
WE APPEAR TO HAVE NUMBERS STARTING 1983
<table>
<thead>
<tr>
<th>Year</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1992</td>
<td>0.000.000</td>
</tr>
<tr>
<td>2000</td>
<td>5.000.000</td>
</tr>
<tr>
<td>2005</td>
<td>10.000.000</td>
</tr>
<tr>
<td>2010</td>
<td>15.000.000</td>
</tr>
<tr>
<td>2015</td>
<td>20.000.000</td>
</tr>
</tbody>
</table>

The graph illustrates the overall increase in value from 1992 to 2015, with a significant rise in the last five years.
But which is the ‘standard’ standard?

Every interpreter delivers one

All ever so slightly different

Brexx and ooRexx cannot run each other’s rexxcps

When fixed, they run their own cps faster than the other’s

Canonical version on speleotrove.com
<table>
<thead>
<tr>
<th>Interpreter</th>
<th>Version</th>
<th>Architecture</th>
<th>OS</th>
<th>Virtualization</th>
<th>Compiler</th>
<th>Compiler Version</th>
<th>Compiler options</th>
<th>CPS score</th>
<th>Hardware</th>
<th>CPU</th>
<th>Date</th>
<th>Measures/Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>brexx</td>
<td>2.1.9</td>
<td>s390x</td>
<td>Red Hat Enterprise Linux Server 7.5 (Maipo)</td>
<td>z/VM</td>
<td>gcc</td>
<td>4.8.5</td>
<td>-O3 fno-stack-protector</td>
<td>6.961.707</td>
<td>IBM LinuxOne Rockhopper (Z13s)</td>
<td>2019-07-04</td>
<td>100/100</td>
<td></td>
</tr>
<tr>
<td>brexx</td>
<td>2.1.9</td>
<td>s390x</td>
<td>Red Hat Enterprise Linux Server 7.5 (Maipo)</td>
<td>z/VM</td>
<td>gcc</td>
<td>4.8.5</td>
<td>+ -mzarch</td>
<td>8.502.614</td>
<td>IBM LinuxOne Rockhopper (Z13s)</td>
<td>2019-07-04</td>
<td>100/100</td>
<td></td>
</tr>
<tr>
<td>brexx</td>
<td>2.1.9</td>
<td>s390x</td>
<td>Red Hat Enterprise Linux Server 7.5 (Maipo)</td>
<td>z/VM</td>
<td>gcc</td>
<td>4.8.5</td>
<td>+ -m-hard-float&quot; &quot;-m-hard-dfp&quot;</td>
<td>8.516.123</td>
<td>IBM LinuxOne Rockhopper (Z13s)</td>
<td>2019-07-04</td>
<td>100/100</td>
<td></td>
</tr>
<tr>
<td>brexx</td>
<td>2.1.9</td>
<td>s390x</td>
<td>Red Hat Enterprise Linux Server 7.5 (Maipo)</td>
<td>z/VM</td>
<td>gcc</td>
<td>4.8.5</td>
<td>+ &quot;-march=z13&quot;</td>
<td>8.567.324</td>
<td>IBM LinuxOne Rockhopper (Z13s)</td>
<td>2019-07-04</td>
<td>100/100</td>
<td></td>
</tr>
<tr>
<td>brexx</td>
<td>2.1.9</td>
<td>s390x</td>
<td>Red Hat Enterprise Linux Server 7.5 (Maipo)</td>
<td>z/VM</td>
<td>gcc</td>
<td>4.8.5</td>
<td>static executable</td>
<td>8.827.066</td>
<td>IBM LinuxOne Rockhopper (Z13s)</td>
<td>2019-07-04</td>
<td>100/100</td>
<td></td>
</tr>
<tr>
<td>REXX370</td>
<td>4.02</td>
<td>4.02</td>
<td>CMS 6.4.0</td>
<td>z/VM</td>
<td>gcc</td>
<td>4.8.5</td>
<td></td>
<td>3.246.319</td>
<td>IBM LinuxOne Rockhopper (Z13s)</td>
<td>2019-07-04</td>
<td>100/100</td>
<td></td>
</tr>
<tr>
<td>brexx</td>
<td>2.1.9</td>
<td>x86_64</td>
<td>Ubuntu 18.04.2 LTS</td>
<td>no</td>
<td>gcc</td>
<td>7.4.0</td>
<td></td>
<td>19.620.712</td>
<td>Dell Latitude 5480</td>
<td>2019-07-04</td>
<td>100/100</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BREXX</td>
<td>21104425</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ooRexx</td>
<td>13533448</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Regina</td>
<td>8238630</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jaxx</td>
<td>9278788</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 4
On-line CPU(s) list: 0-3
Thread(s) per core: 2
Core(s) per socket: 2
Socket(s): 1
NUMA node(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 142
Model name: Intel(R) Core(TM) i7-7567U CPU @ 3.50GHz
Stepping: 9
CPU MHz: 1000.053
CPU max MHz: 4000.0000
CPU min MHz: 400.0000
BogoMIPS: 7008.00
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 4096K
NUMA node0 CPU(s): 0-3
Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vmx flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsave dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp mce ccnews multidev acpica amphal coredump ioasid dsm gpat pmcmov pkpx mmca
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BREXX</td>
<td>2648138</td>
</tr>
<tr>
<td>ooRexx</td>
<td>1831838</td>
</tr>
<tr>
<td>Regina</td>
<td>2062977</td>
</tr>
</tbody>
</table>

```bash
current_numbers@jetson:~/data/brexx/src$ lscpu
Architecture:       aarch64
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  4
Socket(s):           1
Vendor ID:           ARM
Model:               1
Model name:          Cortex-A57
Stepping:            r1p1
CPU max MHz:         1428.0000
CPU min MHz:         102.0000
BogoMIPS:            38.40
L1d cache:           32K
L1i cache:           48K
L2 cache:            2048K
Flags:               fp asimd evtstrm aes pmull sha1 sha2 crc32
```
Current numbers on ARM 32 (armv7l)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BREXX</td>
<td>2816483</td>
</tr>
<tr>
<td>ooRexx</td>
<td>1914118</td>
</tr>
<tr>
<td>Regina</td>
<td>2043062</td>
</tr>
</tbody>
</table>

pi@kleene:~/apps $ lscpu
Architecture:        armv7l
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  4
Socket(s):           1
Vendor ID:           ARM
Model:               3
Model name:          Cortex-A72
Stepping:            r0p3
CPU max MHz:         1500.0000
CPU min MHz:         600.0000
BogoMIPS:            108.00
Flags:               half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm crc32
Why is ooRexx slower?

- Interpreter api
- Stems are slower than array objects
Why is BREXX faster?

- Limited numeric precision
- Not an exact implementation of the standard
Written in Intel assembler (masm) for DOS and Windows, very fast

Its purpose was to verify the ANSI (ISO) standard

Parts of it generated from the grammar in that document

Made with knowledge of Intel processor cache and pipelines
Parallel Rexx

☐ Maybe the future

☐ Nvidia and Intel have competing ‘AI’ processors

☐ CUDA in video cards exists for several years now, CUDA C++ can - with your assistance, that is - parallellize selected loops to execute hundreds times faster, consistent with the number of GPU’s available

☐ To be faster we cannot really trust the CPUs itself

☐ Experiments would be very interesting
NetRexx CPS

There never was one

Or was there?
From 1996, by Mike Cowlishaw, but never released until now

Optimizing compilers optimize a lot of code away, must make sense of the numbers

In NetRexx 3.08, there is an experimental version released

Uses Nanotime instead of millisecs

Currently evaluated on different architectures to see what actually is executed
NetRexx programs have a complex runtime

NetRexx is translated to Java source, which is compiled to bytecode, which is interpreted (sometimes) but mostly compiled just-in-time to native code

Optimization takes place on different levels, each can throw out code

Reaches extremely high numbers
## Current rexxcps.nrx

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Value 1</th>
<th>Value 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aarch64</td>
<td>2,407E+15</td>
<td>2_407_464_380_676</td>
</tr>
<tr>
<td>IA 86_64 I9</td>
<td>1,1489E+16</td>
<td>11_488_524_444_376</td>
</tr>
<tr>
<td>IA 86_64 I7</td>
<td>9,861E+15</td>
<td>9_861_203_229_267</td>
</tr>
</tbody>
</table>
How do we see what is executed?

- List the generated java
- List the generated bytecode
- List the generated assembler
.nrx yields .java

We can verify a 1:1 relationship between the nrx and the java sourcecode.
.java yields .class

We can verify that all bytecode for the benchmark has been generated.
With a special dll we can see what the HotSpot compiler generates as assembly.

We can also see and influence which parts will be compiled and which parts will be interpreted.
Parallellism

- Rexxcps.nrx also is a single tasking program
- As java can also be parallellized, the future might hold change
- Pipelines in NetRexx, for example are already multitasked over all available processors
Example of Parallelism

2.067 seconds are spent with 249% cpu, which makes for an elapsed time of 0.87 seconds