

CRX - This Year's Story

Introduction

For those of you who missed a previous symposium and have not read the accounts on the REXXLA website, here is a quick summary of previous presentations.

Compact REXX is a vehicle for experiments in implementing REXX.

- It is written in Assembler code for DOS because this is coding for fun and I am nostalgic about the programming methods of decades ago.
- The compiler part of CRX is table driven, using tables generated directly from the Backus-Naur Format description of REXX syntax in the American National Standard.
- The execution part of CRX interprets "pseudo-code" generated from the user's source program by the compiler part.
- The pseudo-code is a mixed sequence of operands and operators, where the order (reverse Polish) determines which operators are applied to which operands.
- Operators are one byte numbers indexing an array of the addresses of the routines which implement operators.
- Operands are two byte numbers indexing one of possibly several arrays of eight byte values, these being the values of the constants and variables of the executing program.
- A stack of eight byte values in memory, managed by the software, holds the values for intermediate results and arguments - REXX values that do not have associated symbols.
- Some eight byte values will contain short character strings, some will contain numbers in the hardware (Intel) representation of numbers, and some will contain pointers to long strings. The long strings may represent numeric values outside the range containable in eight bytes of binary.
- When necessary, a garbage collector scans for all values in use and copies them so that they are physically adjacent in memory. This releases the remaining memory for further allocation (or for DOS to use when running commands).
- Stemmed variables require an execution time lookup mechanism; an adaptive binary tree of the tail values is used which keeps recently looked up values near the root of the tree.
- Some operators are implemented using algorithms coded in REXX, taken from the ANSI standard specification. The algorithms are compiled into a variation of the pseudo-code that has one-byte operands, and that generated pseudo-code becomes part of the CRX implementation (by link-edit).

Execution Speed

Last year I showed you execution of a million REXX clauses per second on a 300MegaHertz Pentium II when running the usual performance benchmark, REXXCPS.

In retrospect I should maybe have said "enough is enough" but I have actually spent a lot of time trying to improve on that.

I don't know how to improve on the design above in any significant way. I do know how to move some of the work in running REXXCPS from execution time to compile time but this would be contrary to the spirit of the REXXCPS program. For example REXXCPS contains the clause:

```
if substr(1234,1,1)=9 then say 'Failed5'
```

If it made suitable arrangements about what happened if TRACE was turned on, an implementation could implement this as doing nothing at all at execution time. However, that would not be in the

right spirit because REXXCPS intends to measure SUBSTR performance. As an example of what is in the right spirit of REXXCPS, I think it is reasonable to move to compile time the check that ensures the second and third arguments of SUBSTR(V,1,1) are numbers, because REXX programs often contain constants in these positions. I think CRX does most of the optimisations that can be done in one pass and are in the spirit of REXXCPS.

So if there are no design changes that help, and no work that can be done away with, the only potential for speed-up is in the actual coding of the implementation.

The timings in the reference book for Intel instructions suggest there is potential. Several instructions can be recoded as two instructions that run faster, for instance JCXZ takes 5-8 cycle times whereas CMP CX followed by JZ can be 2-4 cycles. Also the reference book says that for conditional jump instructions the time is one cycle when the jump is not taken, three when it is. On this basis, the assembler code would go faster if re-arranged so that the likely path was the fall-through path, and the unlikely path was the jump. So I re-programmed along these lines, even though it made the coding less readable in places.

```
    cmp dh,al          cmp dh,al          MyEq:inc cx
    jne MyNe           je MyEq            jmp MyLoop
    inc cx             MyLoop:
MyNe:
```

The second and third columns here achieve the function of the first, with only the second column being executed in the normal case and the second and third columns executed in the rare case. The code of the third column is entered and left by a jump so it can be put somewhere away from the main line.

Variations like this sometimes seemed to improve speed and sometimes made it worse. The granularity of timings on DOS is coarse, with 16 clock ticks per second, but even with measurements of sufficient time to discount that factor there were apparently random variations in speed.

You might want to ponder on the cause of that, before reading on.

I guessed that the speed of code on a Pentium II is sensitive to the alignment of instructions, e.g. whether they start on odd or even memory addresses. So a change made in the coding could affect the speed of the program simply by changing the alignment of some unrelated part of the coding. You can imagine how difficult this would make improving the coding by trial and error.

At this point it might have been wise to admit defeat, but I decided that more science was called for. Instead of laboriously creating and measuring variations by hand, perhaps a utility written in REXX could be used to create and run them. The utility could not be expected to understand Assembler source which used macros so the first requirement was a version of the program in the simplest style of Assembler coding.

The utility had to construct a single assembler source program out of the nine assemblies that comprise CRX. (The implementation is made in parts and link-edited because otherwise the listing of the assembler output becomes too large for my DOS editors to handle.) The utility did this by scanning the multiple assembler listings and extracting what was necessary for a single assembly containing the same instructions. There are some difficulties in this because assembler listings were not designed to be read by a computer and have ambiguities. Also, in theory, it is impossible to tell without execution what parts of a program are data and what parts executable. However, seven hundred lines of REXX can do a reasonable job of understanding sensible programs.

The next step was to profile the execution of REXXCPS by CRX. Profiling consists of counting how often different pieces of the subject, here CRX, are executed when it is processing the test case, here REXXCPS. Many language tools provide a profiling feature but they tend to produce nicely

coloured histograms and digraphs from which you draw conclusions, rather than sets of numbers ready for computing from. Here the profiling problem was to annotate, with counters of execution frequency, the single complete-CRX assembly. The annotation was to be at the most detailed level, with a counter for each little bit of code isolated by a label or a jump instruction. I will call these "snippets" and there are some 2800 of them in CRX.

High-tech profilers work by overwriting the instructions of the subject with instructions that do a hardware interrupt. I chose the simpler approach of having the Rexx utility insert a call to a counting routine into every snippet. This is imperfect because it results in a larger program which in turn means there will be more garbage collections when the test case is run. However, over-counting of garbage collection was not something that mattered.

By combining the information gathered on the frequency of execution of each snippet with the number of cycles needed for execution of the snippet (as given by the hardware reference books and put on the listing by the assembler) we can arrive at a figure for the total number of hardware cycles needed to compile and run REXXCPS. This comes out at 899382865 cycles for 3 million Rexx clauses. Given a 300MHz machine this translates to 3.0 seconds. That is a reasonable match to a better-than-one megaclause per second rating from REXXCPS because it applies to all the processing for REXXCPS, including the compilation, not just the part of itself that REXXCPS times.

The profile of any program will be skewed - not all instructions in it will be executed with the same frequency. An interpreter of Rexx, particularly one like CRX which has table-driven parts, will be highly skewed. In fact, 90 percent of the execution is in less than 10 percent of the executable code. Here are the top ranked snippets that account for 24 percent of execution time.

1. Moving bits of a tailed reference like **acomound.key1.loop** into position to be looked up.
2. Comparing the tail on a tailed reference with other tails that have been used with that stem.
3. Deciding whether the next thing in the pseudo-code is an operator or an operand.
4. Making a call to code that implements an operator.
5. Dividing a value that is held in binary by ten. This is needed when the value is converted to a character string.
6. Loading hardware registers, with pointers to a short string Rexx value.
7. Testing whether the argument to something like the SAY instruction is already a string, or whether it needs conversion.
8. The loop that does UPPER for the PARSE instruction.
9. Testing whether an operand is a variable or a constant. (Hence which array does the value come from.)
10. Multiply by ten - used in character to binary conversion.
11. Move 8 bytes from somewhere to the software managed stack.
12. Scan the digits of a number that is in character string form.
13. Comparison of string values.
14. Move the right hand side part of a concatenation or abuttal.
15. Load the hardware registers with a binary value from the software managed stack.

There is nothing surprising there, except perhaps how high referencing a tailed variable comes in the list. Programming style may be relevant to this. How do you write an array of structures in Rexx? Consider something which in the "C" language would be:

```
struct Box { Height int; Length int; Depth int;} Boxes[99]
```

The Rexx equivalent will be more powerful because names as well as numbers can be used to identify individual boxes, but the "C" syntax does better at indicating that Height, Length and Depth are associated. Some Rexx programmers would use stemmed variables with two parts to the tail:

```
/* Set up box J */  
Box.J.0Height = 22  
Box.J.0Length = 33  
Box.J.0Depth = 44
```

The 0 in these references is just an idiom, to prevent Height, Length and Depth being confused with variables of those spellings. Any digit would serve as well as 0.

Alternatively, different stems could be used:

```
BoxHeight.J = 22  
BoxLength.J = 33  
BoxDepth.J = 44
```

The latter loses all connection between Height, Length and Depth, except for some artificial partial match in the names of the stems. However, it does change from references with two parts in the tail to references with just one part.

As you know, a reference like AAA.BBB.CCC does not mean look for things in AAA which have a first tail part matching BBB and then look in that set of things for something with a tail matching CCC. What it does mean is look in AAA for something with a tail of value BBB'.CCC. The two concatenations in constructing BBB'.CCC are what makes this reference relatively expensive, since string moves are expensive under the Intel architecture. In theory, programmers striving for speed might do best to avoid tails with more than one component, although I have no figures to prove that.

Let us now return to the more general problem of arranging the 300 or so performance relevant snippets into a program best arranged to get speed out of the Pentium hardware. The speed might come from putting them on particular alignments, or in the pattern of which ones have a fall through at the bottom into executing others. How can we get a good solution when the decisions are inter-related, we don't know which are important, and there are too many possibilities to try all the combinations?

Problems like this are not unusual. The "Travelling Salesman" problem, of visiting a set of cities with the least travelling, is an example. So are many problems of packing and scheduling. An approach that has given good results is to generate solutions randomly and look for features that characterise the high performers amongst those solutions. Then those features can be fixed when further solutions are evaluated.

So we need to generate a feasible random arrangement of the snippets, assemble the snippets in that arrangement, use the result to run REXXCPS, record the arrangement and its performance, then try again. REXX is ideal for this sort of combination of computing and driving other tools.

Well, I did get this automated trial and error working, but I still could not recognise a pattern in what improved speed and what did not. Belatedly, I sought some help from the Pentium II technical specifications. The light dawned. The information in the Technical Reference that I had been using, and the speed data printed in the assembler listings, was about the Intel engines of the x86 and Pentium time. The Pentium Pro and successors have a different approach to instruction scheduling. In particular, they predict that backward branches will be taken and forward branches will not. In the example above, if the code for the rare case is positioned earlier in the program than the code for the usual case then a Pentium II will predict the rare case as happening each time, and will take of the order of 10 cycles to recover when it does not.

After gathering information about the frequencies of branches between the snippets (which requires a bit more data than just the frequency of execution of the snippets themselves), the utility was made to create an arrangement of snippets matched to the Pentium II rules. This produced the best speed yet on REXXCPS. Experimenting with alignment did not improve speed further. This is where it is now:

(Demo 1.17 MegaClause per Second)

So that is the Speed story - lots of problem solving interest for me, a bit of technique that could improve any assembler program for the current Intel engines, but only 10% speed improvement. I

suspect CRX is near the limit for the combination of Rexx, Intel architecture, and the "spirit of REXXCPS".

Just one more comment on speed: the people who design engines like the Pentium and the Athlon try hard to discourage us from putting unpredictable branches in programs. The Athlon keeps a cache (with 2048 entries) of where branches went when they we last executed but this no better than a static guess for coping with a conditional branch that (pseudo-randomly) branches about half the time and falls through about half the time. A wrongly predicted branch takes ten times as long as a typical instruction so contortions to avoid a branch can pay. Consider:

```
if MyFlag then Alpha = Constant1; else Alpha = Constant2
```

The straightforward assembler rendition of this is:

```
bt Flags,MyFlag
jnc ToElse
mov Alpha,Constant1
jmp AfterElse
ToElse:mov Alpha,Constant2
AfterElse:
```

Many programmers would write this, which saves code size and probably doesn't cost speed on most architectures.

```
mov Alpha,Constant1
bt Flags,MyFlag
jc AfterElse
mov Alpha,Constant2
AfterElse:
```

The interesting question is how this compares for speed with the branchless version:

```
xor Alpha,Alpha; Clear a register
bt Flags,MyFlag; Set Carry
sbb Alpha,Alpha; Subtract with Borrow - result zero or all ones.
and Alpha,Constant1-Constant2; Zero or the constant difference.
add Alpha,Constant2;
```

Pseudo-Code Size

There is only one noteworthy change to the pseudo-code that CRX compiles. The usual format has two byte operands and one byte operands interspersed, so AA = BB would become two bytes to reference BB (and load its value) followed by an assign operator and the reference for AA. However, since a Rexx program with more than 8000 variables or more than 8000 constants would be a rare beast (and probably ought to be rewritten in separate external files) the two bytes of a reference only need 13 bits to index the relevant array. If we take the even/oddness bit to distinguish operators/references and another bit for constant/variable we still have a bit spare. If we take this bit to mean "store to this reference" then AA = BB can be rendered as just two bytes for BB and two bytes for AA, with the assignment operator omitted.

External Rexx Procedures

Rexx allows for a program that comes as one source file to make use of a procedure that is the content of a different source file. Early Rexx processors were "pure" interpreters which necessarily fetched the source file and syntax checked it each time the external procedure was invoked by the main program.

More recent processors, those that produce pseudo-code, have usually been designed to retain the pseudo-code so that it does not have to be regenerated each time the external procedure is called.

Sometimes it is held appended to the source, in the same file. For Warp, it is held in the "attributes" associated with the file. (Sadly, the Warp designers did not anticipate the attributes being used for something large so a size restriction cuts in at around the size of a program with 500 Rexx statements, with the unfortunate affect that programs which could most benefit from pseudo-code retention cannot use it.)

Implementers have expressed interest in allowing the pseudo-code to be retained in memory and the Standards Committee suggested an option RELOAD/NORELOAD to control this.

The two approaches could be used together - having the pseudo-code with the source is the quickest way of running the procedure in the first instance (which would be particularly relevant to transaction processing response) while having it retained in memory would be fastest for subsequent executions.

How much information to keep in the pseudo-code and how much to get on demand (from the original source file) when needed, to provide a TRACE for example, is a matter of balance in the design. CRX is at the low end in terms of what it keeps in the pseudo-code. So even though a DOS system has only about 600K of memory addressable, I thought having NORELOAD always enabled might be effective.

To explain my example for this, I need to go back to the time of IBM's Systems Application Architecture (SAA). To ensure the various implementations of Rexx gave the same results, they were all required to run the same test suite. It was known as the Vienna test suite because it was written there. One small part of the Vienna suite consists of a file where each line gives the expected result for some arithmetic operation.

{ Show the file }

During the development of the Rexx standard, IBM contributed this file to help debug the standard. The committee needed it because they had written an algorithm in Rexx defining the Rexx arithmetic in terms of the simpler operations of integer addition and subtraction. Also they wrote the algorithms for the builtin functions in terms of simpler Rexx. Let us call these programs ARITHOP and ANSIBIFS.

I am now going to show you the execution of a program that reads lines from the Vienna file, feeds the values from a line as arguments to a call of ARITHOP, and checks the answer for that line. This is Classic Rexx on Warp. In normal testing the answers would go to a log file but here they are directed at the screen. You will see that the first few lines are processed quickly, but then a slower pace sets in.

{ Run example }

What is happening is that the later lines are testing the power operator, '**'. The right hand side argument to this operator has to be a whole number. To do this DATATYPE(RHS,'W') test, the ANSIBIFS code is called as an external routine. Let's look at the sizes of the code involved.

{DIR ...}

We see that ARITHOP has it's compiled pseudo-code in it's attributes, but ANSIBIFS doesn't because it is too big for the WARP limit on attributes. That means ANSIBIFS has to be recompiled from source each time it is called, resulting in slowness.

Reprogramming a big Classic Rexx program into multiple smaller external routines is not always easy because shared values have to be passed as arguments. And in this particular example it would defeat the purpose of testing ANSIBIFS in the form it was written.

Going the other way and combining the test, ARITHOP, and ANSIBIFS into one giant program would actually make for faster execution since all the compiling effort would happen just once at the beginning. This requires some recoding, for example because of name clashes and because EXIT

means RETURN in an external routine and termination in an internal routine. I have made it work just to demonstrate:

{ Show that }

However, we certainly don't want to encourage monolithic programs - easy reusability is lost, editing can get unresponsive, there would be a demand for INCLUDE, and so on. Ideally, what we want is to make external routine calls to be as quick as internal routine calls.

Here is the case with the test harness, ARITHOP and ANSIBIFS, run by CRX. The arithmetic is not DIGITS 9 so it doesn't take advantage of the CRX binary arithmetic, but it does avoid recompilations by keeping compilation results in memory. You will see one pause, on the first of the power operations, which is when ANSIBIFS is compiled for the only time. This is DOS running under WARP.

{ Show that }

Is the memory price for this speed to high?

There are design choices about what is held for a program, after compilation. The source is needed, e.g. for the SOURCELINE builtin function, but it can be kept on the disk and fetched as necessary in execution. The pseudo-code needs to be in memory when it is being run. The list of symbols used in the program has to be in memory. So do the values of constants. The symbol lookup information, which quickly converts from a symbol to it's number in the compact numbering of symbols that pseudo-code uses, is optional. The information can be reconstructed from the list of symbols. If we assume that things which need the information (like INTERPRET and the VALUE builtin) are rare, then the information could be remade on demand. The following statistics are for when the information is retained, and for when it is not.

	Source-Bytes	Lines	Pcode-bytes	Symbol-bytes	Variables& Constants
Harness	2370	72	244	395	65
ArithOp	20933	623	3008	1116	169
AnsiBifs	104657	2891	15990	16077	1003

Overall, 13 bytes per line for the executables of a routine to be retained in memory.

(10.3 without quick lookup of variables and constants)

This is not a high price.

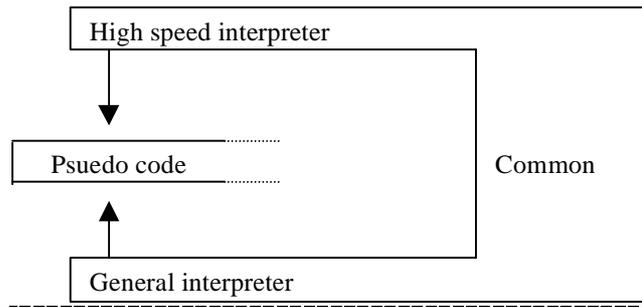
So compiling external routines the first time they are used by a program, and keeping the results in memory, will make subsequent calls to the external almost as fast as a call to an internal routine. We get the modularity and ease-of-use of external routines without undue performance pain. If this works well on DOS it should work even better on systems with more addressable memory.

Mechanisms for TRACE

We all know how useful TRACE is, and it is useful that it monitors the execution at a detailed level. In the case of TRACE T, it monitors after every variable or constant is accessed, after every operation, and at the beginning of every clause. If tracing was implemented simply, by seeing if

trace was active every time execution reached one of these places, it would be something of an unnecessary overhead since most programs, in most of their executions, are not being traced at all.

I have suggested before that the way to combine the trace facility with efficiency in the usual case was to have two interpreters in the implementation.



One of the interpreters does no tests for trace, except that when a TRACE statement or TRACE builtin function activates tracing it hands control to the other interpreter. The latter does extra testing while executing the pseudo-code, to produce the trace as necessary.

This two-interpreters approach works, but it has some limitations. Consider this clause from the REXXCPS program:

```
avar.=1.0''loop
```

The expression here has the same value as **1.0 || loop** where **||** is the abuttal operator. I suspect the reason that abuttal was not used was some concern about the portability of the vertical bar symbol. This is certainly a valid concern; there are three glyphs that look something like a vertical bar in the ASCII 8-bit character set and it is not always obvious from a keyboard which key will produce what, particular if the input is to be in EBCDIC and converted. In fact, the "Personal REXX" product I use today requires a different character than the Warp REXX does, for the vertical bar.

This raises an interesting language question - why does the language have an abuttal operator at all? We know we can do "concatenation-with-blank" by putting operands in succession, as with **ABC DEF** or **(Expression1) (Expression2)**. And we can do many cases of abuttal, which is "concatenation-without-blank" by closing the gap, as with **ABC'abc'** or **(Expression1)(Expression2)**. So it would be reasonable to do abuttal of a couple of variables with the **ABC' DEF** idiom, avoiding any abuttal operator.

One slim argument against avoiding **||** is that **1.0''loop** and **1.0 || loop** are not the same when traced.

```
2 *-* x = 1.0''loop
```

```
>L> 1.0
>L>
>O> 1.0
>L> LOOP
>O> 1.0LOOP
```

```
3 *-* x = 1.0 || loop
```

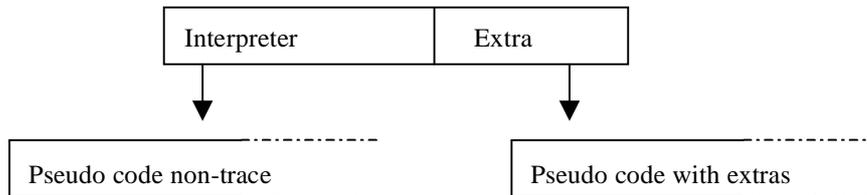
```
>L> 1.0
>L> LOOP
>O> 1.0LOOP
```

Returning to the implementation challenge, that means that the pseudo-code will have to reflect what was written - the compiler cannot compile **1.0''loop** as if the slightly faster **1.0 || loop** had been written. This particular case won't really be significant for speed but there are more common constructs where the potential to trace impedes code improvement. For example:

```
ABC = SomeExpression
If ABC > 0 then .....
```

To support tracing, the pseudo-code must contain the store to ABC and the load from ABC. But for tracing, the load would be unnecessary because the value of ABC is still handy on the stack - it was just stored from there.

Suppose that instead of a design with two interpreters and one pseudo-code we have a design with one interpreter and two lots of pseudo-code:



The first lot of pseudo-code will be optimized without concern for tracing. The second lot will have extra code compiled into it, with operation codes specific to tracing, wherever there is a need to monitor anything. Note that the interpreter has to support these extra opcodes, but the implementation of the non-trace opcodes does no testing for trace.

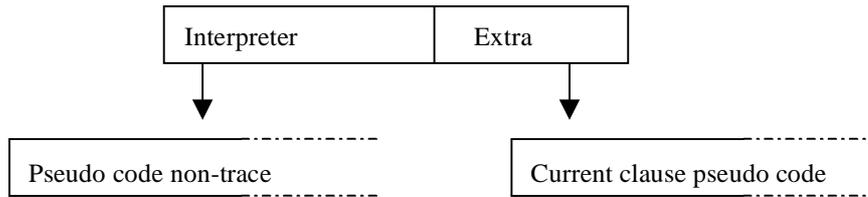
This approach removes the constraints noted above - `1.0 | loop` can be compiled into the fast lot of pseudo-code and `1.0 ' loop` compiled into the trace-supportive pseudo-code.

This is not all as simple as it may sound. There is a challenge for the implementor in arranging the changes from executing one lot of pseudo-code to the other, particularly as the TRACE builtin function can turn on tracing partway through an expression evaluation. Also the value of a label is moot, since a particular label in the source program will correspond to two different locations in pseudo-code, one in each lot of pseudo-code. When a CALL is executed, what location should be used as the return link, given that the trace mode may have changed by the time the RETURN is made? However, there are solutions, provided that we don't care that the tracing pseudo-code executes slowly.

This design allows for:

- More freedom in characteristics of the fast pseudo-code.
- Full TRACE functionality.
- Retention of the simple REXX model of program development, Edit-Run.

The price of the speed would be high if it meant that the size of the pseudo-code was more than double that of a simpler design. However, since we don't much care about speed of execution when tracing is turned on, there is no need to ever materialize the trace-supporting pseudo-code for the whole program. Pseudo-code can be compiled for a particular clause immediately before the clause is executed and discarded immediately after.



I will demo this with REXXCPS, as usual, firstly without trace. The version of CRX used is a touch slower than some because it hasn't been through the profiling I described earlier and it is running in the "DOS box" of WARP but it should do a MegaClause.

{Demo}

Now I edit a trace statement in, towards the end of REXXCPS.

{Demo}

The clauses-per-second figure is the same, or slightly altered by random effects, since that was executed by the same pseudo-code and same interpreter parts as before. However, when trace came on these clauses were run by doing compile-then-execute on a clause by clause basis.

There are some features of Rexx, like the INTERPRET instruction, which constrain the design of implementations of Rexx and hence cost all users something, whether or not they use the feature in their programs. What I have shown here is that TRACE is not like that - it need not cost anything except if and when it is used.

To end with, here is some untested speculation. If two ways of encoding REXX are better than one, would three be better than two? The notion with two is that one can be the basis and the other produced on the fly when more complicated situations arise. The reflection of this is to use one as a basis and the other when simpler and more favourable conditions arise. Taken to the extreme, this is what language processor people call "jitting" - the production of machine code Just_In_Time during execution when the dynamic conditions make it valuable. Perhaps the most efficient structure for Rexx is a basis pseudo-code, with another pseudo-code to be generated and run when tracing is active, and jitting of simple pieces of the program that can be done in binary and reasonably rendered as machine code.

Summary

Diminishing returns on speed improvements.

NORELOAD option to keep external procedures in main memory works well.

A second lot of pseudo-code, made clause by clause, allows trade between slowness with tracing turned on and quickness with tracing turned off.