STREAMLINING YOUR PIPELINES

Melinda Varian

Office of Computing and Information Technology
Princeton University
87 Prospect Avenue
Princeton, NJ 08544 USA
—.—
Email: melinda@princeton.edu
Web: http://pucc.princeton.edu/~melinda/
Telephone: (609) 258-6016

SHARE 92
Session 9241
February 1999

INTRODUCTION

At Princeton, by 1993 more and more of our applications were becoming extensive refineries of pipelines (for example, Rick Troth's CMS Gopher client and server and our own BITFTP server). The CPU utilization of heavily used applications is always a concern, so we wanted to try to optimize some of these applications. However, we found that we had no tools that would help us tune applications that made extensive use of *CMS Pipelines*. The excellent traditional tools for analyzing the CPU used by REXX programs, such as TRACEXEC (by Kent Fiala, of the SAS Institute) and HISTOREX (by Brick Verser, of Kansas State University), showed us only that most of the CPU used by these applications was due to PIPE commands, which we knew already.

So, in 1993, my management instructed me to write a tool for analyzing the performance of piped applications. The result was a tool called Rita (which was named for *Lovely Rita, Meter Maid*, since it is a program for reading meters). That name was suggested by John Hartmann, the author of *CMS Pipelines*, who also contributed much good advice and other assistance to this project.

John suggested that I start by turning on the 8K bit in the *CMS Pipelines* "msglevel" setting. That would cause *Pipes* to generate an accounting message each time a pipeline stage terminated:

```
pipmod msglevel 8207
Ready;
pipe (name test) literal hello | console
hello
Spent .017 milliseconds in stage.
... Issued from stage 1 of pipeline 1 name "test".
... Running "literal hello".
Spent .065 milliseconds in stage.
... Issued from stage 2 of pipeline 1 name "test".
... Running "console".
Spent .047 milliseconds in gnl ohd.
Spent .429 milliseconds in scanner.
```

```
Spent .244 milliseconds in dispatch.
Spent .451 milliseconds in wait.
Spent 2.491 milliseconds in acnt ohd.
Ready;
```

(You can find out about the various "msglevel" settings by issuing the command pipe ahelp pipmod.)

My first implementation was simply a REXX filter that could be used in a pipeline that invoked the subject pipeline with a runpipe stage. runpipe captured the *Pipes* accounting messages, and my REXX filter processed them using the then brand-new IBM 407 emulation in the spec stage.

```
'PIPMOD MSGLEVEL 8207'
'PIPE var pipe | runpipe | dobit8k | ...'
'PIPMOD MSGLEVEL 15'
```

The result was a display of the CPU utilization of each stage of each pipeline in the complex. This example shows the output for one subroutine pipeline, which had been invoked nine times from a loop in a REXX filter in a higher level pipeline:

```
23.9 ms in stage 1 of pipeline 1 "test27a": test27b
 0.1 ms in stage 2 of pipeline 1 "test27a": literal abc
 0.1 ms in stage 3 of pipeline 1 "test27a": totarget nlocate 1
 0.1 ms in stage 4 of pipeline 1 "test27a": literal xyz
 0.1 ms in stage 5 of pipeline 1 "test27a": hole

24.3 ms in pipeline "test27a" (9 invocations) <=====
```

The millisecond values are the CPU used by each stage exclusive of any CPU used by subroutine pipelines invoked by the stage.

This particular example taught me a good lesson about *Pipes* performance. I foolishly asserted that the *Pipes* accounting numbers must be wrong, because it couldn't possibly be so expensive to invoke that REXX filter nine times. The Piper suggested that I try EXECLOADing the filter. This was the result:

```
11.1 ms in stage 1 of pipeline 1 "test27a": test27b
 0.1 ms in stage 2 of pipeline 1 "test27a": literal abc
 0.1 ms in stage 3 of pipeline 1 "test27a": totarget nlocate 1
 0.1 ms in stage 4 of pipeline 1 "test27a": literal xyz
 0.1 ms in stage 5 of pipeline 1 "test27a": hole

11.5 ms in pipeline "test27a" (9 invocations) <=====
```

MORAL: If you find yourself issuing a CALLPIPE command in a loop, first ask yourself why the loop can't be absorbed into the pipeline, and if it really can't, then remind yourself to EXECLOAD any REXX filters (or other EXECs) used by the subroutine pipeline.

Although this prototype was useful, I soon found that I wanted a display of the CPU use for each stage both inclusive of and exclusive of the CPU used by its subroutines, such as TRACEXEC provides for EXECs. The *Pipes* accounting records don't supply enough information to determine which stage invoked a subroutine, so John suggested that I also process the "event

records" that runpipe creates when it is invoked with the EVENTS option. (The EVENTS option was added to *Pipes* in CMS 10 for use by PIPEDEMO. You can find out about the event records by issuing the command pipe ahelp pipevent.) So, I threw out my prototype and started over again to write what came to be called Rita. While I was writing Rita, John made several enhancements to the event records to make it easier for me to navigate the calling sequences in the subject pipeline. That was very helpful, especially since Gopher, for example, proved to be recursive. John also added a MSGLEVEL option to runpipe to provide finer control over the msglevel setting. (Both of these enhancements were included in CMS 11.)

The new tool consisted of two pieces, RITA EXEC, which contained a pipeline that invoked the subject pipeline, and RITA REXX, a filter that analyzed both the accounting messages and the event records as they were created. To use Rita, one replaced a PIPE command with EXEC RITA. Thus, the subject pipeline was the argument to RITA EXEC, which ran it with runpipe.

As soon as I began using Rita to measure our BITFTP server, however, I ran into the problem that stem and var stages in subroutine pipelines that referenced REXX variables in the main EXEC were now searching for those variables in RITA EXEC, because using RITA EXEC had inserted an additional EXECCOMM level below the main EXEC, between the level that had contained the PIPE command and the level of any REXX filters in subroutine pipelines. John quickly solved that problem by extracting my pipeline from RITA EXEC, encapsulating it in an Assembler program, and building RITA MODULE. So, then, to use Rita, one simply replaced a PIPE command with a RITA command. The result was a display similar to this one for each pipeline in the complex:

```
   36 (  36) ms in stage 1 of segment 1 "SendMail": var fromline
   22 (  22) ms in stage 2 of segment 1 "SendMail": locate 81
 1343 ( 972) ms in stage 3 of segment 1 "SendMail": 822cont
   14 (  14) ms in stage 4 of segment 1 "SendMail": faninany
  107 ( 107) ms in stage 5 of segment 1 "SendMail": stem fromline.

 1521 (1151) ms in pipeline "SendMail" (452 invocations) <=====
```

In this example, sendmail is a subroutine pipeline consisting of five stages, one of which, 822cont, is a REXX filter containing another subroutine pipeline. During this measurement run, the sendmail subroutine pipeline was issued 452 times and used 1521 milliseconds of CPU. Of that, 1343 milliseconds were used by its 822cont stage, and 972 milliseconds of that were used by something other than the subroutine pipeline in that stage.

A version of Rita that will work under CMS 10 and later has been included on the "Samples and Examples" disk (usually MAINT's 193) since VM/ESA 2.1.0. An even more current version of Rita can be found on the *CMS Pipelines* Web page (http://pucc.princeton.edu/~pipeline/) and at our anonymous FTP site (cd to anonymous.376 on pucc.princeton.edu).

CASE STUDY—A UUENCODE STAGE

Our BITFTP service machine runs at three sites, ten copies in all, so I have been eager to reduce its CPU utilization. BITFTP is a moderately large refinery:

- 1 PIPE
- 143 CALLPIPEs
- 2 ADDPIPEs
- 800+ stages
- 8000 lines of REXX

One of its real hotspots (accounting for twenty-five to thirty percent of its CPU use on some days) was its uuencode filter. However, after being hacked at off and on for a year, uuencode accounted for less than five percent of BITFTP's CPU use. In getting it to that point, I learned a few things about tuning a *Pipes* application, so I think it might make an interesting case study.

UUencoding is a poorly-defined process for encoding files so that they can be transmitted safely in email *via* hostile gateways. Basically, the process is:

- Expand each 6 bits to 8 (by inserting high-order 0s);
- Form into 60-byte records each preceded by a byte specifying the length of the data in the record before expansion (*i.e.*, 45);
- Add ASCII blank (x'20') to each byte to make it printable;
- Do something to prevent loss of trailing blanks; and
- Garnish the file with a header record and two trailer records.

Here are the input and output when uuencoding a one-record file, UUTEST TEXT:

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
begin 0600 uutest.text
M86)C9&5F9VAI:FML;6YO<'%R<W1U=G=X>7I!0D-$149'2$E*2TQ-3D]045)3M
'5%565UA96@  M
 M
end
```

Because I was working on VM, the encoded output would be EBCDIC, and it would be translated to ASCII when the mail went onto the net. Adding a blank (either EBCDIC or ASCII) to my EBCDIC characters would not be the same operation as adding an ASCII blank to an ASCII character, even after the resulting character had been translated to ASCII by the mail gateway. So, to add an ASCII blank to each of my EBCDIC bytes, I used the approach that John Fisher, of Rensselaer Polytechnic Institute, uses in his excellent ARCUTIL program. After the 6-to-8 expansion, each character is translated to the EBCDIC character that will assume its value plus x'20' after being translated to ASCII when going through the mail gateway. To prevent loss of trailing blanks, I use the scheme of suffixing a "guard character" to each line.

UUencode Phase 1

This was the first implementation of my uuencode filter:

```
'CALLPIPE (endchar ? name uuencode)',
      '*: |',                      /* Connect input to pipeline.  */
      'xlate 1-* e2a |',           /* Translate to ASCII.         */
      'spec 1-* 1 x0A next |',     /* Append line-feed to each.   */
      'fblock 45 |',               /* Reblock to 45-byte chunks.  */
   'd: drop last 1 |',             /* Last record may be short.   */
      'vchar 6 8 |',               /* Recode, 45 bytes to 60.     */
      'change //M/ |',             /* Data length (45 bytes).     */
      'spec 1-* 1 /M/ 62 |',       /* Append "guard" character.   */
   'f: faninany |',                /* Pick up last data record.   */
      'xlate 1-*' addspace '|',    /* Make data printable EBCDIC. */
      'preface var beginline |',   /* Prepend canonical header.   */
      'append literal  M|',        /* Indicate end of data.       */
      'append literal end|',       /* Indicate end of file.       */
      '*:',                        /* Connect output to pipeline. */
'?',
   'd: |',                         /* Process last data record:   */
   'c: count bytes |',             /* Calculate its size.         */
      'copy |',                    /* (To unblock "count".)       */
      'spec 1-* 1 x0000 next |',   /* Compensate for "vchar".     */
      'chop 45 |',                 /* Truncate compensation.      */
      'vchar 6 8 |',               /* Recode, each 3 bytes to 4.  */
   's: spec',                      /* Complete the formatting:    */
         'select 1',               /*   get data length;          */
         '1-* d2c 1.1 right',      /*   make it one byte binary;  */
         'select 0',               /*   get encoded data;         */
         '1-* 2',                  /*   put it next; and          */
         '/M/ next |',             /*   append guard character.   */
   'f:',                           /* Send back to mainline.      */
'?',
   'c: |',                         /* Count of bytes in last rec. */
   's:'                            /* Feed it to "spec".          */
```

UUencoding is used to transmit a file by enclosing it in mail. In the production version of this program, the first two stages are used only if the file is a text file, rather than binary. In that case, the recipient will expect the file to be in ASCII, not EBCDIC, once it has been decoded, so it must be translated before encoding, and its record structure must be preserved by appending a linend to each record before encoding. (Typically, the only reason to encode a text file is to preserve the record structure when the records are longer than 80 bytes.)

The encoding proper starts with the fblock 45 stage, which flows the records together and reblocks them into 45-byte chunks. The last record may be short, so drop last 1 is used to divert it to the second pipeline segment for special handling.

All records except for the last one are then fed into the vchar 6 8 stage, which expands each 6 bits to 8, thus expanding each 45-byte record to 60 bytes. (vchar treats each of its input records as a bit stream in which the size of an input "character" is defined by its first argument and the size of an output character is defined by its second argument. It steps through the input record taking an input character (6 bits in this case) and expanding or contracting it to an output character (8 bits in this case), padding with high-order zeros or truncating on the left, as required. The output record is also treated as a bit stream.)

The change stage prepends the character "M" to each record, because the records will be translated to ASCII when they go through a mail gateway, thus changing this "M" to x'2D' (decimal 45), which is the number of bytes in the record before expansion. The spec stage appends another "M", to guard against the loss of any trailing blanks. faninany gathers the records from both of its input streams. xlate translates each of the encoded characters to the character that will be x'20' greater when translated to ASCII. (That is, x'00' is translated to an EBCDIC blank, which will become x'20', an ASCII blank, when translated to ASCII; x'01' is translated to the EBCDIC character that will become x'21' when translated to ASCII; and so forth.) The canonical uuencode header record and trailer records are inserted into the pipeline, and the records flow out through the ending connector to the calling pipeline.

The last record, which may be short, is handled in the second pipeline segment. count bytes writes the length of that record to its secondary output. The copy stage puts a quantum delay into the pipeline to prevent a stall. spec appends two bytes of zeros to the record to compensate for a quirk of vchar, which discards any bits left over when it gets to the end of a record and hasn't enough bits left to make another full output character. chop 45 pares the record down to 45 bytes in case appending those two bytes of zeros made it longer than 45. vchar 6 8 expands each 6 bits of data to 8, thus transforming each 3 bytes into 4. The spec pulls in the count of data bytes from its secondary input, converts that to a one-byte binary number, then follows that by the expanded data and a guard character, after which this last record flows into the faninany in the first pipeline segment.

That worked, but it was slow:

```
   846.1 ms in stage  2 of segment 1: xlate 1-* e2a
  1520.5 ms in stage  3 of segment 1: spec 1-* 1 x0A next
   937.5 ms in stage  4 of segment 1: fblock 45
  1245.1 ms in stage  5 of segment 1: drop last 1
  5165.0 ms in stage  6 of segment 1: vchar 6 8
!  1355.7 ms in stage  7 of segment 1: change //M/
!  2227.2 ms in stage  8 of segment 1: spec 1-* 1 /M/ 62
   838.0 ms in stage  9 of segment 1: faninany
  1217.5 ms in stage 10 of segment 1: xlate 1-* 00 40 01 5A 02 7F...
     2.0 ms in stage 11 of segment 1: preface var beginline
     0.9 ms in stage 12 of segment 1: append literal  M
     0.9 ms in stage 13 of segment 1: append literal end
     1.1 ms in stage  2 of segment 2: count bytes
     0.6 ms in stage  3 of segment 2: copy
     5.2 ms in stage  4 of segment 2: spec 1-* 1 x0000 next
     0.8 ms in stage  5 of segment 2: chop 45
     1.1 ms in stage  6 of segment 2: vchar 6 8
     8.4 ms in stage  7 of segment 2: spec select 1 1-* d2c 1.1 right
                                         select 0 1-* 2 /M/ next
 15373.6 ms in pipeline "uuencode" (20 invocations) <=====
```

UUencode Phase 2

The first change to make was obvious. The function of the change stage that prefixed an "M" to each full line was merged into the spec stage that followed it:

```
   847.5 ms in stage  2 of segment 1: xlate 1-* e2a
:  1502.2 ms in stage  3 of segment 1: spec 1-* 1 x0A next
:   930.4 ms in stage  4 of segment 1: fblock 45
  1222.0 ms in stage  5 of segment 1: drop last 1
  5155.3 ms in stage  6 of segment 1: vchar 6 8
!  2680.3 ms in stage  7 of segment 1: spec /M/ 1 1-* 2 /M/ 62
   836.7 ms in stage  8 of segment 1: faninany
  1231.4 ms in stage  9 of segment 1: xlate 1-* 00 40 01 5A 02 7F...
     2.0 ms in stage 10 of segment 1: preface var beginline
     0.9 ms in stage 11 of segment 1: append literal  M
     0.9 ms in stage 12 of segment 1: append literal end
     1.1 ms in stage  2 of segment 2: count bytes
     0.6 ms in stage  3 of segment 2: copy
     5.1 ms in stage  4 of segment 2: spec 1-* 1 x0000 next
     0.8 ms in stage  5 of segment 2: chop 45
     1.1 ms in stage  6 of segment 2: vchar 6 8
     8.4 ms in stage  7 of segment 2: spec select 1 1-* d2c 1.1 right
                                           select 0 1-* 2 /M/ next
14426.7 ms in pipeline "uuencode" (20 invocations) <=====
```

MORAL: spec is relatively expensive, but if you need to use it, then you probably want to make it do as much as it can.

UUencode Phase 3

The next improvement also merged the function of two stages. block has an option C, which specifies that an EBCDIC linend is to be appended to each input record before reblocking. By using that option, I was able to delete the spec stage that had been appending an ASCII linend to each input record:

```
:  1197.6 ms in stage  2 of segment 1: block 45 c
   1123.1 ms in stage  3 of segment 1: xlate 1-* e2a
   1133.4 ms in stage  4 of segment 1: drop last 1
   5102.6 ms in stage  5 of segment 1: vchar 6 8
   2723.9 ms in stage  6 of segment 1: spec /M/ 1 1-* 2 /M/ 62
.   814.2 ms in stage  7 of segment 1: faninany
   1136.6 ms in stage  8 of segment 1: xlate 1-* 00 40 01 5A 02 7F...
      2.0 ms in stage  9 of segment 1: preface var beginline
      0.9 ms in stage 10 of segment 1: append literal  M
      0.9 ms in stage 11 of segment 1: append literal end
      1.1 ms in stage  2 of segment 2: count bytes
      0.6 ms in stage  3 of segment 2: copy
      5.2 ms in stage  4 of segment 2: spec 1-* 1 x0000 next
      0.8 ms in stage  5 of segment 2: chop 45
      1.0 ms in stage  6 of segment 2: vchar 6 8
      8.4 ms in stage  7 of segment 2: spec select 1 1-* d2c 1.1 right
                                           select 0 1-* 2 /M/ next
```

```
13252.3 ms in pipeline "uuencode" (20 invocations) <=====
```

MORAL: Again, merging the function of two stages into one paid off. Note, however, that the xlate e2a stage has become significantly more expensive, as the result of the reordering of the first few stages. Most of the records are originally longer than 45 bytes, so there are more records after reblocking, and processing more records takes more time. Because I hadn't yet written Rita when I made this change, I didn't notice this effect at the time. I was simply going by the change in total CPU utilization.

UUencode Phase 4

The next change was to replace faninany with fanin. There was no need for faninany here. I tend to use it, rather than fanin, out of habit, but all records from the primary stream will be received before the single record from the secondary stream, so fanin will do exactly the right thing in this case and will be less expensive:

```
  1187.0 ms in stage  2 of segment 1: block 45 c
  1115.3 ms in stage  3 of segment 1: xlate 1-* e2a
! 1138.8 ms in stage  4 of segment 1: drop last 1
  5116.5 ms in stage  5 of segment 1: vchar 6 8
  2706.7 ms in stage  6 of segment 1: spec /M/ 1 1-* 2 /M/ 62
.  598.6 ms in stage  7 of segment 1: fanin
  1117.8 ms in stage  8 of segment 1: xlate 1-* 00 40 01 5A 02 7F...
     2.1 ms in stage  9 of segment 1: preface var beginline
     0.9 ms in stage 10 of segment 1: append literal  M
     0.9 ms in stage 11 of segment 1: append literal end
     1.1 ms in stage  2 of segment 2: count bytes
     0.6 ms in stage  3 of segment 2: copy
     5.2 ms in stage  4 of segment 2: spec 1-* 1 x0000 next
     0.8 ms in stage  5 of segment 2: chop 45
     1.1 ms in stage  6 of segment 2: vchar 6 8
     8.4 ms in stage  7 of segment 2: spec select 1 1-* d2c 1.1 right
                                           select 0 1-* 2 /M/ next
13001.8 ms in pipeline "uuencode" (20 invocations) <=====
```

MORAL: Avoid unnecessary use of stages that use SELECT INPUT ANY. Watching for input on multiple streams is inherently more expensive than is simply reading all records from one stream before switching to another.

UUencode Phase 5

It was becoming obvious that the drop last 1 was relatively expensive. As there is a calculation of the number of output records earlier in the program, it was trivial to replace drop last 1 with take *n*-1:

```
  1193.5 ms in stage  2 of segment 1: block 45 c
  1122.2 ms in stage  3 of segment 1: xlate 1-* e2a
!  661.8 ms in stage  4 of segment 1: take 2999
:  5124.7 ms in stage  5 of segment 1: vchar 6 8
  2777.4 ms in stage  6 of segment 1: spec /M/ 1 1-* 2 /M/ 62
   600.0 ms in stage  7 of segment 1: fanin
```

```
 1140.9 ms in stage  8 of segment 1: xlate 1-* 00 40 01 5A 02 7F...
    2.1 ms in stage  9 of segment 1: preface var beginline
    0.9 ms in stage 10 of segment 1: append literal  M
    0.9 ms in stage 11 of segment 1: append literal end
    1.1 ms in stage  2 of segment 2: count bytes
    0.5 ms in stage  3 of segment 2: copy
    5.2 ms in stage  4 of segment 2: spec 1-* 1 x0000 next
    0.8 ms in stage  5 of segment 2: chop 45
    1.1 ms in stage  6 of segment 2: vchar 6 8
    8.4 ms in stage  7 of segment 2: spec select 1 1-* d2c 1.1 right
                                          select 0 1-* 2 /M/ next
12641.5 ms in pipeline "uuencode" (20 invocations) <=====
```

MORAL: take last and drop last must maintain a revolving buffer of the specified number of records. This may be an unnecessary expense in your application.

UUencode Phase 6

As you can see, vchar was becoming more and more conspicuously the main problem. Fortunately, then, the Piper came to lunch. At lunch, he asked all of the other guests around the table for their *Pipes* wishlists. One of the guests asked for vchar to be speeded up. He soon got his wish:

```
.  1157.9 ms in stage  2 of segment 1: block 45 c
.  1214.8 ms in stage  3 of segment 1: xlate 1-* e2a
    728.8 ms in stage  4 of segment 1: take 2999
:  1342.9 ms in stage  5 of segment 1: vchar 6 8
   2718.2 ms in stage  6 of segment 1: spec /M/ 1 1-* 2 /M/ 62
    595.9 ms in stage  7 of segment 1: fanin
   1130.7 ms in stage  8 of segment 1: xlate 1-* 00 40 01 5A 02 7F...
      2.1 ms in stage  9 of segment 1: preface var beginline
      0.9 ms in stage 10 of segment 1: append literal  M
      0.9 ms in stage 11 of segment 1: append literal end
      1.1 ms in stage  2 of segment 2: count bytes
      0.5 ms in stage  3 of segment 2: copy
      5.2 ms in stage  4 of segment 2: spec 1-* 1 x0000 next
      0.9 ms in stage  5 of segment 2: chop 45
      0.7 ms in stage  6 of segment 2: vchar 6 8
      8.3 ms in stage  7 of segment 2: spec select 1 1-* d2c 1.1 right
                                            select 0 1-* 2 /M/ next
   8909.7 ms in pipeline "uuencode" (20 invocations) <=====
```

This improvement was achieved by special-casing idioms that truncate records at byte boundaries. (It was included in CMS 11.)

MORAL: Feed the Piper.

UUencode Phase 7

By then, I had written Rita, so I did some measuring and finally noticed the problem I had introduced when I switched the order of the xlate and block stages. Since block can readily be made to append ASCII linends, this was easily remedied:

```
.   861.8 ms in stage  2 of segment 1: xlate 1-* e2a
.  1110.6 ms in stage  3 of segment 1: block 45 linend 0A
    726.5 ms in stage  4 of segment 1: take 2999
   1333.1 ms in stage  5 of segment 1: vchar 6 8
!  2730.0 ms in stage  6 of segment 1: spec /M/ 1 1-* 2 /M/ 62
    595.2 ms in stage  7 of segment 1: fanin
   1135.1 ms in stage  8 of segment 1: xlate 1-* 00 40 01 5A 02 7F...
      2.1 ms in stage  9 of segment 1: preface var beginline
      0.9 ms in stage 10 of segment 1: append literal  M
      0.9 ms in stage 11 of segment 1: append literal end
      1.1 ms in stage  2 of segment 2: count bytes
      0.6 ms in stage  3 of segment 2: copy
      5.3 ms in stage  4 of segment 2: spec 1-* 1 x0000 next
      0.9 ms in stage  5 of segment 2: chop 45
      0.8 ms in stage  6 of segment 2: vchar 6 8
      8.4 ms in stage  7 of segment 2: spec select 1 1-* d2c 1.1 right
                                           select 0 1-* 2 /M/ next
   8513.1 ms in pipeline "uuencode" (20 invocations) <=====
```

MORAL: All else being equal, it is cheaper to process a smaller number of larger records.

UUencode Phase 8

The spec that prepended the record length ("M") and appended the guard character ("M") was the obvious problem to tackle next. My first attempt was to replace it with two very simple stages, a change and a pad:

```
    855.3 ms in stage  2 of segment 1: xlate 1-* e2a
   1146.1 ms in stage  3 of segment 1: block 45 linend 0A
    717.1 ms in stage  4 of segment 1: take 2999
   1341.2 ms in stage  5 of segment 1: vchar 6 8
!  1376.2 ms in stage  6 of segment 1: change //M/
!   998.2 ms in stage  7 of segment 1: pad 62 M
    595.6 ms in stage  8 of segment 1: fanin
   1151.8 ms in stage  9 of segment 1: xlate 1-* 00 40 01 5A 02 7F...
      2.0 ms in stage 10 of segment 1: preface var beginline
      0.9 ms in stage 11 of segment 1: append literal  M
      1.0 ms in stage 12 of segment 1: append literal end
      1.1 ms in stage  2 of segment 2: count bytes
      0.6 ms in stage  3 of segment 2: copy
      6.2 ms in stage  4 of segment 2: spec 1-* 1 x0000 next
      0.8 ms in stage  5 of segment 2: chop 45
      0.7 ms in stage  6 of segment 2: vchar 6 8
      8.5 ms in stage  7 of segment 2: spec select 1 1-* d2c 1.1 right
                                           select 0 1-* 2 /M/ next
   8203.1 ms in pipeline "uuencode" (20 invocations) <=====
```

I was somewhat surprised that using pad to append a single character to records of a known length would be so inexpensive.

MORAL: It is sometimes cheaper to use two simple stages rather than one complex stage.

UUencode Phase 9

I needed a greater CPU reduction, however, so instead of using the change and pad, I replaced the spec stage with guard, a trivial Assembler filter whose only function was to copy 60-byte records from its input to its output, prefixing an “M” and suffixing an “M” to each:

```
ADDGUARD TITLE 'Pipeline stage to prefix length and suffix guard'
ADDGUARD MODBEG FREETYPE=CMS       Begin module.
 PIPEPVR (R5)                      Base the transfer vector.
GUARD    PROC  ,                   Define procedure.
OBUFFER  DC    C'M',CL60' ',C'M'   Output buffer.
GUARDED  EQU   OBUFFER,*-OBUFFER   Output record definition.
 PBEGIN  ,                         Begin procedure.
 REPEAT  ,                         Loop while there is input.
   PIPLOCX ,                       Examine next record (PEEKTO).
   MVC   OBUFFER+1(60),0(R1)       Copy record to output buffer.
   PIPOUTX GUARDED                 Emit the record (OUTPUT).
   PIPINPUT (,0)                   Consume input (READTO).
 UNTIL   NOTZERO                   Loop until no more input.
 PEXIT   RC=(R15)                  Exit.
 PROCEND ,                         End procedure.
 MODEND  ,                         End module.
```

Although guard was a very special-purpose routine with no parameter processing and no error checking and was not at all something one could be proud of, it had the advantage that it solved the problem with sixteen lines of code and about five minutes of work:

```
  888.6 ms in stage  2 of segment 1 xlate 1-* e2a
 1120.1 ms in stage  3 of segment 1 block 45 linend 0A
  674.4 ms in stage  4 of segment 1 take 2999
 1268.4 ms in stage  5 of segment 1 vchar 6 8
!  650.4 ms in stage  6 of segment 1 guard
  635.4 ms in stage  7 of segment 1 fanin
:  1167.6 ms in stage  8 of segment 1 xlate 1-* 00 40 01 5A 02 7F...
    2.1 ms in stage  9 of segment 1 preface var beginline
    0.9 ms in stage 10 of segment 1 append literal  M
    0.9 ms in stage 11 of segment 1 append literal end
    1.1 ms in stage  2 of segment 2 count bytes
    0.6 ms in stage  3 of segment 2 copy
    6.1 ms in stage  4 of segment 2 spec 1-* 1 x0000 next
    0.8 ms in stage  5 of segment 2 chop 45
    0.7 ms in stage  6 of segment 2 vchar 6 8
    8.5 ms in stage  7 of segment 2 spec select 1 1-* d2c 1.1 right
```

```
                                                 select 0 1-* 2 /M/ next
 6426.6 ms in pipeline "uuencode" (20 invocations) <=====
```

MORAL: It may be quite easy to move the worst part of the problem into Assembler.

UUencode Phase 10

It was obvious then to add a TR instruction and John Fisher's translate table to this Assembler filter to perform the function of the second xlate stage:

```
UUFORM   TITLE 'Pipeline stage to complete uuencode formatting'
UUFORM   MODBEG FREETYPE=CMS       Begin module.
 PIPEPVR (R5)                      Base the transfer vector.
UUFORMAT PROC  ,                   Define procedure.
OBUFFER  DC    C'M',CL60' ',C'M'   Output buffer.
GUARDED  EQU   OBUFFER,*-OBUFFER   Output record definition.
 PBEGIN  ,                         Begin procedure.
 REPEAT  ,                         Loop while there is input.
   PIPLOCX ,                       Examine next record (PEEKTO).
   MVC   OBUFFER+1(60),0(R1)       Copy record to output buffer.
:  TR    OBUFFER+1(60),UUETABLE    Add ASCII blank to each byte.
   PIPOUTX GUARDED                 Emit the record (OUTPUT).
   PIPINPUT (,0)                   Consume input (READTO).
 UNTIL   NOTZERO                   Loop until no more input.
 PEXIT   RC=(R15)                  Exit.
:  ******************************************************************
:  * The following translate table is designed to add the ASCII space *
:  * character, while allowing for subsequent translation from EBCDIC *
:  * to ASCII.  Thus, x'00' is translated to x'40', which will become *
:  * x'20' when the uuencoded file passes through an EBCDIC-to-ASCII  *
:  * gateway.  Similarly, x'01' is translated to x'5A', which will    *
:  * become x'21' when it passes through such a gateway.  I am        *
:  * indebted to John Fisher, of RPI, for this technique.             *
:  ******************************************************************
:  *                 0 1 2 3 4 5 6 7 8 9 A B C D E F
:  UUETABLE DC    X'405A7F7B5B6C507D4D5D5C4E6B604B61'  0
:           DC    X'F0F1F2F3F4F5F6F7F8F97A5E4C7E6E6F'  1
:           DC    X'7CC1C2C3C4C5C6C7C8C9D1D2D3D4D5D6'  2
:           DC    X'D7D8D9E2E3E4E5E6E7E8E9ADE0BD5F6D'  3
 PROCEND ,                         End procedure.
 MODEND  ,                         End module.
```

That made another significant reduction in the CPU use:

```
  835.4 ms in stage  2 of segment 1: xlate 1-* e2a
 1137.1 ms in stage  3 of segment 1: block 45 linend 0A
  714.6 ms in stage  4 of segment 1: take 2999
. 1286.3 ms in stage  5 of segment 1: vchar 6 8
: 849.9 ms in stage  6 of segment 1: uuformat
  591.5 ms in stage  7 of segment 1: fanin
    2.1 ms in stage  8 of segment 1: preface var beginline
    0.9 ms in stage  9 of segment 1: append literal  M
    0.9 ms in stage 10 of segment 1: append literal end
    1.0 ms in stage  2 of segment 2: count bytes
    0.6 ms in stage  3 of segment 2: copy
    6.2 ms in stage  4 of segment 2: spec 1-* 1 x0000 next
    0.9 ms in stage  5 of segment 2: chop 45
    0.7 ms in stage  6 of segment 2: vchar 6 8
    8.6 ms in stage  7 of segment 2: spec select 1 1-* d2c 1.1 right
                                          select 0 1-* 2 /M/ next
    1.6 ms in stage  8 of segment 2: xlate 1-* 00 40 01 5A 02 7F...

 5438.4 ms in pipeline "uuencode" (20 invocations) <=====
```

UUencode Phase 11

Doing that made me see how simple it would be also to do the 6-to-8 expansion in the Assembler routine for this special case of having an input record always an integral multiple of 3 bytes in length and always exactly equivalent to an output record:

```
ENCODE   TITLE 'Pipeline stage to uuencode 45-byte records'
ENCODE   MODBEG FREETYPE=CMS       Begin module.
 PIPEPVR (R5)                      Base the transfer vector.
ENCODEUU PROC  ,                   Define procedure.
OBUFFER  DC    C'M',CL60' ',C'M'   Output buffer.
GUARDED  EQU   OBUFFER,*-OBUFFER   Output record definition.
 PBEGIN  ,                         Begin procedure.
 REPEAT  ,                         Loop while there is input.
   PIPLOCX ,                       Examine next record (PEEKTO).
.  LA    R3,OBUFFER+1              Beginning of output buffer.
.  LA    R4,45/3                   Number of 3-byte chunks to do.
.  REPEAT  ,
.    ICM   R6,B'0111',0(R1)        Get three bytes to encode.
.    LA    R1,3(,R1)               Update input pointer.
.    STCM  R6,B'0001',3(R3)        Fourth output byte.
.    SLL   R6,2
.    STCM  R6,B'0010',2(R3)        Third output byte.
.    SLL   R6,2
.    STCM  R6,B'0100',1(R3)        Second output byte.
.    SLL   R6,2
.    STCM  R6,B'1000',0(R3)        First output byte.
.  UNTIL BCT=R4,INCR=(R3,4)        Loop thru 45 bytes of input.
.  NC    OBUFFER+1(60),=60X'3F'    Clear high-order bits.
   TR    OBUFFER+1(60),UUETABLE    Add ASCII blank to each byte.
   PIPOUTX GUARDED                 Emit the record (OUTPUT).
   PIPINPUT (,0)                   Consume input (READTO).
 UNTIL   NOTZERO                   Loop until no more input.
 PEXIT   RC=(R15)                  Exit.
 . . .
```

Thus, the vchar stage in the main portion of the pipeline could also be removed, again reducing CPU utilization:

```
  860.4 ms in stage 2 of segment 1: xlate 1-* e2a
 1100.0 ms in stage 3 of segment 1: block 45 linend 0A
!  639.6 ms in stage 4 of segment 1: take 2999
.  1176.4 ms in stage 5 of segment 1: encodeuu
  608.5 ms in stage 6 of segment 1: fanin
    2.0 ms in stage 7 of segment 1: preface var beginline
    0.9 ms in stage 8 of segment 1: append literal  M
    0.9 ms in stage 9 of segment 1: append literal end
    1.0 ms in stage 2 of segment 2: count bytes
    0.6 ms in stage 3 of segment 2: copy
    6.1 ms in stage 4 of segment 2: spec 1-* 1 x0000 next
    0.8 ms in stage 5 of segment 2: chop 45
    0.7 ms in stage 6 of segment 2: vchar 6 8
    8.5 ms in stage 7 of segment 2: spec select 1 1-* d2c 1.1 right
                                         select 0 1-* 2 /M/ next
    1.6 ms in stage 8 of segment 2: xlate 1-* 00 40 01 5A 02 7F...
```

```
 4408.1 ms in pipeline "uuencode" (20 invocations) <=====
```

UUencode Phase 12

But I was feeling insecure about the blind assumption of 45-byte input records. Worrying about that made me see that with one more small change, the Assembler stage could assume the function of the take stage and be made more robust at the same time:

```
ENCODE   TITLE 'Pipeline stage to uuencode 45-byte records'
ENCODE   MODBEG FREETYPE=CMS       Begin module.
 PIPEPVR (R5)                      Base the transfer vector.
PIPUUE   PROC  ,                   Define procedure.
OBUFFER  DC    C'M',CL60' ',C'M'   Output buffer.
GUARDED  EQU   OBUFFER,*-OBUFFER   Output record definition.
 PBEGIN  ,                         Begin procedure.
 REPEAT  ,                         Loop while there is input.
   PIPLOCX ,                       Examine next record (PEEKTO).
!  C     R0,=F'45'                 Right chunk size?
!  IF    NOTEQUAL                  No, we can't handle it.
!    PIPSEL  OUTPUT,1,EXIT=NOTZERO Select secondary output.
!    PIPSHORT ,                    SHORT input to secondary output.
!    GOTOEXIT ,                    All done; exit.
!  FI    ,                         Right chunk size; uuencode it.
   LA    R3,OBUFFER+1              Beginning of output buffer.
   LA    R4,45/3                   Number of 3-byte chunks to do.
   REPEAT  ,
     ICM   R6,B'0111',0(R1)        Get three bytes to encode.
     . . .
```

With the Assembler filter shorting its input to its secondary output when it encountered a record not exactly 45 bytes long, there was no need for the take stage:

```
  844.5 ms in stage 2 of segment 1: xlate 1-* e2a
 1109.2 ms in stage 3 of segment 1: block 45 linend 0A
! 1156.1 ms in stage 4 of segment 1: pipuue
  597.8 ms in stage 5 of segment 1: fanin
    2.1 ms in stage 6 of segment 1: preface var beginline
    0.9 ms in stage 7 of segment 1: append literal  M
    1.0 ms in stage 8 of segment 1: append literal end
    1.1 ms in stage 2 of segment 2: count bytes
    0.6 ms in stage 3 of segment 2: copy
:   6.3 ms in stage 4 of segment 2: spec 1-* 1 x0000 next
:   0.8 ms in stage 5 of segment 2: chop 45
:   0.8 ms in stage 6 of segment 2: vchar 6 8
:   8.7 ms in stage 7 of segment 2: spec select 1 1-* d2c 1.1 right
:                                        select 0 1-* 2 /M/ next
    1.6 ms in stage 8 of segment 2: xlate 1-* 00 40 01 5A 02 7F...
```

```
 3731.7 ms in pipeline "uuencode" (20 invocations) <=====
```

MORAL: Assembler filters are really very easy to write, and there can be a considerable payoff.[1]

UUencode Phase 13

I was quite happy with my Assembler filter, despite the inelegance of its handling only the common case, and I was ready to leave uuencode alone for a while, but then I got a note from a user complaining that BITFTP's uuencoder was sometimes padding records excessively. He quoted the nearest thing there is to a standard for uuencode, a BSD man page, which said that the encoded portion of each record must be padded to a multiple of 4 bytes. My filter was sometimes padding the last, short record of a file further than that. So, I added a straightforward REXX filter to force the data portion of the last record to be an integral multiple of 4 bytes in length:

```
/* POSTUUE REXX:     Position uuencode guard character properly  */

Signal On Error

'PEEKTO record'                    /* Examine next record.        */
Parse Var record datalen +1        /* Data length in binary.      */
reclen = (((c2d(datalen) + 2) % 3) * 4) + 1    /* UUencoded + 1. */
'OUTPUT' Left(record,reclen) || 'M'   /* Append guard character. */
'READTO'                           /* Consume processed record.   */

Error: Exit RC*(RC<>12)            /* RC = 0 if end-of-file.      */
```

It was fortunate that this filter was executed only once per file, because even with it compiled and EXECLOADed, it was relatively expensive:

```
  846.3 ms in stage 2 of segment 1: xlate 1-* e2a
 1090.3 ms in stage 3 of segment 1: block 45 linend 0A
 1135.7 ms in stage 4 of segment 1: pipuue
  590.7 ms in stage 5 of segment 1: fanin
    2.0 ms in stage 6 of segment 1: preface var beginline
    0.9 ms in stage 7 of segment 1: append literal  M
    0.9 ms in stage 8 of segment 1: append literal end
    1.1 ms in stage 2 of segment 2: count bytes
    0.6 ms in stage 3 of segment 2: copy
:   6.2 ms in stage 4 of segment 2: spec 1-* 1 x0000 next
```

---

[1] Writing filters in Assembler was finally supported officially in CMS 12, and a dumbed-down version of the interface was documented in the CMS 12 manuals. For an introduction to using the complete interface, see John P. Hartmann, *Writing Assembler Filters for Use with CMS Pipelines* (available from the *Pipelines* Web page).

```
:     0.8 ms in stage 5 of segment 2: chop 45
:     0.8 ms in stage 6 of segment 2: vchar 6 8
:     6.7 ms in stage 7 of segment 2: spec select 1 1-* d2c 1.1 right
:                                          select 0 1-* 2
:    26.9 ms in stage 8 of segment 2: postuue
      1.7 ms in stage 9 of segment 2: xlate 1-* 00 40 01 5A 02 7F...

   3711.5 ms in pipeline "uuencode" (20 invocations) <=====
```

UUencode Phase 14

Pipes had recently acquired the ability to run a REXX filter read from the secondary input of a rexx stage, so I tried implementing the same REXX code that way as well:

```
postuue = "/* */;'PEEKTO record';",
   "datalen=((c2d(Left(record,1))+2)%3)*4;",
   "'OUTPUT' Left(record,datalen+1)'M';'READTO'"

'CALLPIPE (endchar ? name uuencode)',
    . . .
   'r: rexx *.1: |',              /* Append guard character.     */
    . . .
'?',
   'var postuue |',               /* Small REXX stage to run.    */
   'r:'                           /* Feed it to "rexx".          */
```

Not surprisingly, that was measurably slower:

```
    853.6 ms in stage 2 of segment 1: xlate 1-* e2a
   1097.2 ms in stage 3 of segment 1: block 45 linend 0A
   1138.4 ms in stage 4 of segment 1: pipuue
    592.2 ms in stage 5 of segment 1: fanin
      2.1 ms in stage 6 of segment 1: preface var beginline
      0.9 ms in stage 7 of segment 1: append literal  M
      0.9 ms in stage 8 of segment 1: append literal end
      1.1 ms in stage 2 of segment 2: count bytes
      0.5 ms in stage 3 of segment 2: copy
:     6.2 ms in stage 4 of segment 2: spec 1-* 1 x0000 next
:     0.8 ms in stage 5 of segment 2: chop 45
:     0.7 ms in stage 6 of segment 2: vchar 6 8
:     6.4 ms in stage 7 of segment 2: spec select 1 1-* d2c 1.1 right
:                                          select 0 1-* 2
:    29.7 ms in stage 8 of segment 2: rexx *.1:
      1.7 ms in stage 9 of segment 2: xlate 1-* 00 40 01 5A 02 7F...
:     1.5 ms in stage 1 of segment 4: var postuue

   3734.1 ms in pipeline "uuencode" (20 invocations) <=====
```

MORAL: The compiler usually wins (but not always).

UUencode Phase 15

I knew there had to be a way to do it with built-ins, so I tried again, this time replacing the spec that appended trailing zeros and the chop 45 with a 407 emulation spec that built a CALLPIPE command with a pad stage that would pad the input record to an integral multiple of 3 bytes, thus assuring that the output of vchar would be an integral multiple of 4 bytes:

```
/* The following pipeline segment issues a CALLPIPE of the form:
   CALLPIPE (escape " name UUPAD) literal   """""""""| pad 12 00 |*:
*/
uuepad =,                          /* Length of encoded data must */
     'escape |',                   /*   be even multiple of 4.    */
  'p: spec',                       /* So, build CALLPIPE to make  */
        '/CALLPIPE (escape " name UUPAD)',
        'literal/ 1 1-* nw /||',   /*   input even multiple of 3. */
        'pad/ next',
           'select 1',             /* Get count of data bytes.    */
           'a: 1-* ignore',
           'compute #0=((a+2)%3)*3',/* Round up to multiple of 3.*/
           'counter 0 picture z9 nw /00 ||',
        '*:/ nw |',
     'pipcmd |'                    /* Issue the CALLPIPE command. */
```

That was faster than the REXX filter:

```
  856.4 ms in stage 2 of segment 1: xlate 1-* e2a
 1094.1 ms in stage 3 of segment 1: block 45 linend 0A
 1137.6 ms in stage 4 of segment 1: pipuue
  591.4 ms in stage 5 of segment 1: fanin
    2.0 ms in stage 6 of segment 1: preface var beginline
    0.9 ms in stage 7 of segment 1: append literal  M
    0.9 ms in stage 8 of segment 1: append literal end
    1.1 ms in stage 2 of segment 2: count bytes
    0.6 ms in stage 3 of segment 2: copy
:   0.6 ms in stage 4 of segment 2: escape
:  21.1 ms in stage 5 of segment 2: spec /CALLPIPE (escape " name
:                                   UUPAD) literal/ 1 1-* nw /|...
:   1.3 ms in stage 6 of segment 2: pipcmd
:   0.7 ms in stage 7 of segment 2: vchar 6 8
:   8.3 ms in stage 8 of segment 2: spec select 1 1-* d2c 1.1 right
:                                        select 0 1-* 2 /M/ next
    1.6 ms in stage 9 of segment 2: xlate 1-* 00 40 01 5A 02 7F...
:   0.9 ms in stage 2 of segment 3: fanout
:   0.7 ms in stage 3 of segment 3: copy

 3720.3 ms in pipeline "uuencode" (20 invocations) <=====
```

MORAL: Even the most contorted solutions with built-ins are usually faster than REXX filters.

UUencode Phase 16

Then, the Piper came back from holiday and reminded me that vchar 32 32 will truncate to an integral multiple of 4 bytes (that is, it will copy 32 bits from the input record to a 32-bit "character" in the output record, continuing until the end of the input record and discarding any leftover bits if there are not enough for a full output character):

```
  845.7 ms in stage 2 of segment 1: xlate 1-* e2a
 1091.5 ms in stage 3 of segment 1: block 45 linend 0A
 1136.1 ms in stage 4 of segment 1: pipuue
  590.3 ms in stage 5 of segment 1: fanin
    2.0 ms in stage 6 of segment 1: preface var beginline
    0.9 ms in stage 7 of segment 1: append literal  M
    0.9 ms in stage 8 of segment 1: append literal end
    1.1 ms in stage 2 of segment 2: count bytes
    0.6 ms in stage 3 of segment 2: copy
    6.2 ms in stage 4 of segment 2: spec 1-* 1 x0000 next
    0.7 ms in stage 5 of segment 2: vchar 6 8
:   0.8 ms in stage 6 of segment 2: vchar 32 32
:   8.6 ms in stage 7 of segment 2: spec select 1 1-* d2c 1.1 right
:                                        select 0 1-* 2 /M/ next
    1.6 ms in stage 8 of segment 2: xlate 1-* 00 40 01 5A 02 7F...

 3687.0 ms in pipeline "uuencode" (20 invocations) <=====
```

MORAL: vchar can be used to do almost anything.

UUencode Phase 17

I sent the Piper my Assembler filter and asked him to critique my use of PL/j. When he returned my filter, I was not surprised to find that he had both improved the PL/j and rewritten the inner loop to use an entirely different algorithm. I won't show that, however, because Rita said that his filter was slower than mine.

His response to that challenge was to come up with a faster algorithm and to handle the short record along with the others, resulting in a full-function and very fast uuencode filter:

```
  826.0 ms in stage 2 of segment 1: xlate 1-* e2a
 1081.2 ms in stage 3 of segment 1: block 45 linend 0A
 1284.9 ms in stage 4 of segment 1: pipuue
    2.2 ms in stage 5 of segment 1: preface var beginline
    0.9 ms in stage 6 of segment 1: append literal  M
    0.9 ms in stage 7 of segment 1: append literal end

 3196.0 ms in pipeline "uuencode" (20 invocations) <=====
```

MORAL: Challenge the Piper.

UUencode Postscript

John's uuencode filter never made its way into the product, so a few months later I received a request from a friend for a REXX uuencode. I started to give him my most current straight-REXX implementation, but realized that it was not really general-purpose enough, because of the take *n*-1 stage requiring that the output record count be known in advance. That led me to see whether I could find something that performed better than drop last 1 but did not require knowing the record count. After thinking about it a bit, I realized that what I really needed after the block 45 was not to special-case the *last* record, but rather to special-case any record shorter than 45 bytes. In those cases where the last record produced by block 45 is exactly 45 bytes long, there is no need for special processing at all. Thus, drop last 1 can be replaced by locate 45, which will divert any record shorter than 45 bytes to its secondary output for special processing.

My measurements using a June, 1995, version of *CMS Pipelines* showed that the CPU use of locate 45 fell about halfway between that of the drop and that of the take. Thinking I might use this as another state in the UUencode case study, I repeated the measurements with a June, 1993, version of *CMS Pipelines* to match the measurements I have just been showing you. What I found was that locate 45 used 22% less CPU in 1995 than it had two years earlier:

	June '93	June '95	Change
drop last 1	1117	1079	-3%
locate 45	1054	819	-22%
take 2999	614	602	-2%

The improvement appears to be due to changes made when support for "inputRanges" was added to locate. This sort of thing happens in *Pipes* all the time. John frequently gets ideas for improving the performance of individual stages or of *Pipes* as a whole and quietly implements them. For example, in 1996 he shaved twelve instructions off of one of the heavily-used paths in the *Pipes* dispatcher. My measurements show that his change reduced the time spent in the dispatcher by 3-7%. This change was included in CMS 12, as were several optimizations of the performance of individual stages.

MORAL: *Pipes* always gets faster, so always use the newest version you can get. The most current version of *Pipes* runs nicely all the way back to CMS 5, so consider porting it backwards if you are unable to move quickly to the most current level of CMS.

CASE STUDY—A NETDATA RECEIVE STAGE

One of my next projects was a service machine that needed to receive several hundred thousand NETDATA-format files per day. My preliminary work on this server made it clear that whether I used the CMS RECEIVE EXEC or a pipeline to get these files from the virtual reader, the time to receive the files would be the limiting factor for my server. Since I needed the data in a pipeline in order to process them, the tradeoff was between using RECEIVE EXEC to receive the file to disk and then loading it into the pipeline and erasing it from disk or using a pipeline to read the file straight from the virtual reader. I found the latter to require 3% more CPU but only half as much elapsed time, due to its doing far fewer I/Os.

So, I decided to see if I could speed up the standard receive pipeline filter. Given that that filter was the work of the Master Plumber, I knew that my only hope to make it run faster would be to use new function that had been added to *Pipes* during the years since he wrote his version of receive.

Receive Phase 1

One of his first cuts at a receive filter had been this:

```
'CALLPIPE (name Receive)',          /* Read and deblock NETDATA:   */
   'reader file' Arg(1) '|',        /* Read specified file.        */
   'find' "41"x'|',                 /* Just cards, please.         */
   'spec 2-* 1 |',                  /* Remove opcode.              */
   'pad 80 |',                      /* In case blanks stripped.    */
   'deblock netdata |',             /* Unravel DMSDDL damage.      */
   'find {|',                       /* Just real data.             */
   'spec 2-* 1 |',                  /* Delete control character.   */
   '*:'                             /* Here are the real records.  */
```

The reader stage moves the spool file specified in its argument from the virtual reader into the pipeline. The find stage selects records from that raw spool file based on the CCW opcode in the first byte. Selecting for records that start with x‘41’ has the effect of discarding the tag records from the spool file, which have no-op opcodes (x‘03’). The spec stage then discards that opcode byte by copying the input to the output starting with column 2. The pad 80 makes legal card images by restoring trailing blanks that have been lost along the way. Then deblock netdata reblocks its input to write out complete NETDATA record segments, both control records and data records, each prefixed by a flag byte. The data records, which are marked with a flag of x‘C0’ (“{”), are selected by the second find stage. The second spec stage removes that flag byte, at which point the file has been restored to the format it had before being transmitted.

That worked properly and was reasonably fast:

```
 31 ms in stage 1 of segment 1: reader file 0961
 55 ms in stage 2 of segment 1: find
147 ms in stage 3 of segment 1: spec 2-* 1
 56 ms in stage 4 of segment 1: pad 80
114 ms in stage 5 of segment 1: deblock netdata
 68 ms in stage 6 of segment 1: find {
177 ms in stage 7 of segment 1: spec 2-* 1
648 ms in pipeline “Receive0” (1 invocation) <=====

532 ms in dispatcher.
```

However, John soon saw how to remove the pad 80 stage by subsuming its function into the spec, and this version became the standard way of receiving a NETDATA file into a pipe:

```
 30 ms in stage 1 of segment 1: reader file 0961
 55 ms in stage 2 of segment 1: find
```

```
149 ms in stage 3 of segment 1: spec 2-* 1.80
110 ms in stage 4 of segment 1: deblock netdata
 68 ms in stage 5 of segment 1: find {
173 ms in stage 6 of segment 1: spec 2-* 1
585 ms in pipeline "Receive1" (1 invocation) <=====

462 ms in dispatcher.
```

MORAL: We have already seen the benefits of letting spec do as much as it can. This particular trick of using it to pad records is a good one to remember.

Receive Phase 2

I suspected that I could make this filter faster by replacing the two spec stages with change stages, but of course that required me to put back the pad:

```
'CALLPIPE (name Receive2)',       /* Read and deblock NETDATA:   */
   'reader file' Arg(1) '|',      /* Read specified file.        */
   'find' "41"x'|',               /* Just cards, please.         */
   'change 1.1 x41 // |',         /* Remove opcode.              */
   'pad 80 |',                    /* In case blanks stripped.    */
   'deblock netdata|',            /* Unravel DMSDDL damage.      */
   'find {|',                     /* Just real data.             */
   'change 1.1 /{// |',           /* Delete control character.   */
   '*:'                           /* Here are the real records.  */
```

Rita showed that the combination of change and pad was slightly slower than the first spec, but that the second change was faster than the spec it replaced:

```
 30 ms in stage 1 of segment 1: reader file 0961
 55 ms in stage 2 of segment 1: find
109 ms in stage 3 of segment 1: change 1.1 x41 //
 57 ms in stage 4 of segment 1: pad 80
110 ms in stage 5 of segment 1: deblock netdata
 68 ms in stage 6 of segment 1: find {
130 ms in stage 7 of segment 1: change 1.1 /{//
559 ms in pipeline "Receive2" (1 invocation) <=====

532 ms in dispatcher.
```

Despite the fact that the time charged to the individual stages was less than before, the EXEC as a whole actually used more CPU time. The difference is explained by another number that Rita reports, the time spent in the *Pipelines* dispatcher. As a rough estimate, the dispatcher time is proportional to the number of times a record is handed to a stage. Thus, when the Piper removed pad from his first implementation, dispatcher time dropped from 532 ms to 462, and when I put pad back, dispatcher time when back up to 532, which was enough to swamp the savings from using change rather than spec.

MORAL: Don't overlook the time spent in the dispatcher and other *Pipelines* overhead.

CAVEAT: On versions of *CMS Pipelines* before CMS 12 (*i.e.*, before internal level 1.0109001A) or when using a version of Rita that does not support the MASK option of runpipe, the dispatcher time reported by Rita will be so dominated by the time to dispatch Rita itself as to be useless for this sort of analysis. (Using CMS 12 and the CMS 12 version of Rita, dispatching Rita amounts to less than 5% of the total dispatcher time.) If you need to examine the dispatcher time using an earlier version of *Pipes*, simply issue the command PIPMOD MSGLEVEL 8207 and run your pipeline without Rita.

Receive Phase 3

To get the improvement I was looking for, I obviously needed to handle the records fewer times. For CMS 10, John made a very powerful enhancement to change that allows it to have a secondary output stream. When that stream is defined, change writes the *unchanged* records to it. The stream need not actually be connected; if it is defined but not connected, the unchanged records are discarded. Using that capability, I was able to replace each of the find | spec pairs in the original implementation with a change stage:

```
'CALLPIPE (end ? name Receive3)',/* Read and deblock NETDATA:    */
      'reader file' Arg(1) '|',  /* Read specified file.         */
   'c: change 1.1 x41 // |',     /* Select data; remove opcode.  */
      'pad 80 |',                /* Pad for DEBLOCK.             */
      'deblock netdata |',       /* Unravel DMSDDL damage.       */
   'd: change 1.1 /{// |',       /* Select data; remove ctl char.*/
      '*:',                      /* Here are the real records.   */
'?',
   'c:',                         /* Discard tags, etc.           */
'?',
   'd:'                          /* Discard NETDATA control recs.*/
```

The first change stage removes the x'41' opcode in column one and writes the changed records to its primary output. It also writes the unchanged records, the ones that did not have x'41' in column one, to its secondary output. That output is not connected to anything, so those records are discarded. Similarly, the second change selects the records that have a x'C0' in the first column and removes that byte, while discarding the records that don't begin with x'C0'. Although this implementation is a tad obscure, it is a definite win:

```
 29 ms in stage 1 of segment 1: reader file 0961
108 ms in stage 2 of segment 1: change 1.1 x41 //
 56 ms in stage 3 of segment 1: pad 80
109 ms in stage 4 of segment 1: deblock netdata
132 ms in stage 5 of segment 1: change 1.1 /{//
434 ms in pipeline "Receive3" (1 invocation) <=====

373 ms in dispatcher.
```

Both the time charged to individual stages and the dispatcher time are reduced significantly; overall, this solution uses half the CPU of the standard filter.

MORAL: *Pipes* always gets more powerful; check every new release for new built-in programs that may help you cool down your hot spots. (Issue the command pipe ahelp menu to find out what really is in the new release, because it is not all documented in the standard help files.)

Receive Phase 4

Somebody asked for a receive filter on the PIPELINE CFORUM, so I proudly appended my shiny new one, but the universal opinion was that it was just too arcane for general use. Steve Hayes, of IBM UK, proposed another solution that is quite nifty:

```
'CALLPIPE (name Receive4)',       /* Read and deblock NETDATA:   */
   'reader file' Arg(1) '|',      /* Read specified file.        */
   'find' "41"x'|',               /* Just cards, please.         */
   'not chop 1 |',                /* Remove opcode.              */
   'pad 80 |',                    /* Pad for DEBLOCK.            */
   'deblock netdata |',           /* Unravel DMSDDL damage.      */
   'find {|',                     /* Just real data.             */
   'not chop 1 |',                /* Delete control character.   */
   '*:'                           /* Here are the real records.  */
```

He used not chop 1, rather than spec, to strip the first byte from the records. (chop 1, by itself, copies the first byte to its primary output stream and the remainder of the record to its secondary output stream (or discards it, if the secondary output is not connected). not chop 1 reverses that action; that is, it copies the first byte to its secondary output or discards it if the secondary output is not connected, as in this case, and writes the remainder of the record to its primary output.) Using not chop was a very good idea, but the time in the dispatcher was surprisingly high:

```
CPU utilization of pipeline "Receive4":
   30 ms in stage 1 of segment 1: reader file 0961
   55 ms in stage 2 of segment 1: find
   91 ms in stage 3 of segment 1: not chop 1
   57 ms in stage 4 of segment 1: pad 80
  114 ms in stage 5 of segment 1: deblock netdata
   69 ms in stage 6 of segment 1: find {
  107 ms in stage 7 of segment 1: not chop 1
  523 ms in pipeline "Receive4" (1 invocation) <=====

CPU utilization of pipeline "not":
   91 ms in stage 2 of segment 1: chop 1
   91 ms in pipeline "not" (1 invocation) <=====

CPU utilization of pipeline "not":
  107 ms in stage 2 of segment 1: chop 1
  107 ms in pipeline "not" (1 invocation) <=====
```

```
718 ms in dispatcher.
```

Receive Phase 5

When I asserted (without doing any measurements) that the dispatcher overhead must be due to the complexity of not (which sets up a subroutine pipeline to run its target stage), Glenn Knickerbocker, of IBM, countered with the same logic implemented without not:

```
'CALLPIPE (end ? name Receive5)', /* Read and deblock NETDATA:   */
      'reader file' Arg(1) '|',   /* Read specified file.        */
      'find' "41"x'|',            /* Just cards, please.         */
   'c: chop 1',                   /* Remove opcode.              */
'?',
   'c: |',                        /* Here after opcode removed.  */
      'pad 80 |',                 /* Pad for DEBLOCK.            */
      'deblock netdata |',        /* Unravel DMSDDL damage.      */
      'find {|',                  /* Just real data.             */
   'd: chop 1',                   /* Delete control character.   */
'?',
   'd: |',                        /* Here after ctl char removed.*/
      '*:'                        /* Here are the real records.  */
```

As you can see, not was not the problem:

```
 30 ms in stage 1 of segment 1: reader file 0961
 55 ms in stage 2 of segment 1: find
 86 ms in stage 3 of segment 1: chop 1
 57 ms in stage 2 of segment 2: pad 80
113 ms in stage 3 of segment 2: deblock netdata
 69 ms in stage 4 of segment 2: find {
106 ms in stage 5 of segment 2: chop 1
516 ms in pipeline "Receive5" (1 invocation) <=====

715 ms in dispatcher.
```

The time in the dispatcher was essentially unchanged.

Receive Phase 6

I took another stab at replacing the spec stages, this time using strip leading with the count of the maximum number of bytes to strip set to 1:

```
'CALLPIPE (name Receive6)',       /* Read and deblock NETDATA:   */
   'reader file' Arg(1) '|',      /* Read specified file.        */
   'find' "41"x'|',               /* Just cards, please.         */
   'strip leading x41 1 |',       /* Remove opcode.              */
   'pad 80 |',                    /* In case blanks stripped.    */
   'deblock netdata |',           /* Unravel DMSDDL damage.      */
   'find {|',                     /* Just real data.             */
   'strip leading { 1 |',         /* Delete control character.   */
   '*:'                           /* Here are the real records.  */
```

That was the best non-arcane solution so far:

```
  30 ms in stage 1 of segment 1: reader file 0961
  55 ms in stage 2 of segment 1: find
  54 ms in stage 3 of segment 1: strip leading x41 1
  57 ms in stage 4 of segment 1: pad 80
 111 ms in stage 5 of segment 1: deblock netdata
  69 ms in stage 6 of segment 1: find {
  66 ms in stage 7 of segment 1: strip leading { 1
 441 ms in pipeline "Receive6" (1 invocation) <=====

 527 ms in dispatcher.
```

MORAL: There is more than one way to skin a cat. Become familiar with the entire repertoire of *CMS Pipelines* built-in programs to give yourself more ways of approaching problems.

Receive Phase 7

Then the Piper piped up with a small fix to chop that made it more efficient when it has only a secondary output stream connected, and that made the not chop and strip leading cases exactly equivalent:

```
CPU utilization of pipeline "Receive4":
   30 ms in stage 1 of segment 1: reader file 0961
   55 ms in stage 2 of segment 1: find
   54 ms in stage 3 of segment 1: not chop 1
   56 ms in stage 4 of segment 1: pad 80
  112 ms in stage 5 of segment 1: deblock netdata
   68 ms in stage 6 of segment 1: find {
   67 ms in stage 7 of segment 1: not chop 1
  443 ms in pipeline "Receive4" (1 invocation) <=====

CPU utilization of pipeline "not":
   54 ms in stage 2 of segment 1: chop 1
   54 ms in pipeline "not" (1 invocation) <=====

CPU utilization of pipeline "not":
   67 ms in stage 2 of segment 1: chop 1
```

```
   67 ms in pipeline "not" (1 invocation) <=====

  536 ms in dispatcher.
```

MORAL: It really does *always* get faster.

CASE STUDY—TUNING RITA

Rita is a pipeline, of course, and one that has needed quite a lot of tuning. The process of streamlining Rita did a good job of illustrating one of the most important concepts in tuning a pipeline, so I will use Rita itself as another case study.

Basically, the pipeline in Rita looks like this:

'PIPE var pipeline | runpipe events | rita | console'

The rita filter is 900+ lines of REXX that accumulates the numbers from the event and accounting records and formats them for display. So, optimizing RITA REXX was no different from optimizing any other REXX program. Tools like HISTOREX give one all the information that is needed to find the high-use areas and pound them down.

Unfortunately, after I had optimized the REXX code to the best of my ability, I was faced with the fact that running a pipeline under Rita still used at least an order of magnitude more CPU than did running it standalone. The basic problem was that every time the subject pipeline sighed, or even thought about sighing, runpipe events wrote an event record, and that record had to be processed by my REXX filter. I actually needed to see only a few of the many different event record types, so the obvious approach was to discard the unneeded record types before they got to my REXX filter. After a good deal of experimenting, I concluded that this was the most efficient way of doing that:

```
'PIPE (endchar ?)',
     'var pipeline |',            /* Load pipeline into pipe.    */
     'runpipe events |',          /* Run the subject pipeline.   */
  'l: lookup 1',                  /* Discard unneeded events.    */
'?',
     'strliteral x010203050708090a0c0d0e11 |',  /* Unused types. */
     'deblock 1 |',               /* Get one type per record.    */
  'l: |',                         /* Master file for "lookup".   */
      . . .                       /* More filtering.
     'rita |',                    /* Analyze event records.      */
     'console'                    /* Display results.            */
```

The event records have a one-byte record type field in the first column. The strliteral stage here lists the record types that are not of interest. deblock 1 breaks that list up into one-byte records that are fed into the secondary input of the lookup, to form its "master file". Each time runpipe produces an event record, that record is read into the primary input of the lookup. Since the

primary output of lookup is not connected, records that have a type matching any of those in the master file are discarded immediately. Records of other types are written to the secondary output of lookup, and are put through some sub-type filtering that isn't shown here before being fed into the rita stage. My measurements showed that using lookup to discard the unwanted records required only 18% as much CPU as doing it with a cascade of nfind filters.

Using this technique got Rita's overhead down far enough that I could begin sharing it with other people, but it was still terribly expensive to use. Then, however, John came to my rescue again and added the MASK option to runpipe. MASK allows one to specify the event record types that are to be suppressed. Using the MASK option greatly reduced the overhead of running a pipeline under Rita:

Bare EXEC:	12.4 secs
Under Rita without MASK option:	56.2 secs
Under Rita with MASK option:	16.3 secs

The versions of *Pipes* and Rita shipped in CMS 12 and later support the MASK option.

MORAL: If you know you will be discarding records, do it as early in your pipeline as possible.

ABSORBING LOOPS INTO THE PIPELINE

Whenever you see a PIPE or a CALLPIPE being issued in a loop, you have an opportunity to do some streamlining. Absorbing such a loop into the pipeline generally produces a dramatic reduction in CPU utilization. As an example of this process, I will discuss a very useful pipeline construct, the "sipping subroutine pipeline".[2]

Some of the *Pipes* selection filters divide their input stream into two pieces; that is, they select all the records up to some point and then discard the remainder. These are called "partitioning selection filters" (because they partition the input stream). Examples include take, tolabel, totarget, and whilelabel. When a partitioning selection filter is used in a CALLPIPE, it can select some number of records from its input stream and then terminate when it reaches the point where it would start discarding the remaining records. The subroutine pipeline created by the CALLPIPE then terminates, and the records that would have been discarded are left in the pipeline, ready to be processed by other commands in the REXX filter that issued that CALLPIPE. Thus, the subroutine pipeline created by the CALLPIPE simply "sips" the records it needs from the beginning of the input stream and leaves the rest for further processing.

Word Spill Using a Sipping Subroutine Pipeline

SPILL1 REXX is an example of a REXX filter that performs a "word spill" operation by invoking a simple sipping subroutine pipeline in a loop. spill1 issues CALLPIPE commands in a loop to process one input record at a time. It uses PEEKTO and STREAMSTATE commands to decide when it should exit from its Do Forever loop. The PEEKTO gets RC 12 when there is no more input; the STREAMSTATE OUTPUT gets RC 12 when the output stream is no longer connected. Thus, in either case, the loop is terminated.

---

[2] Ben Chi, of the University of Albany, coined this delightful term.

```
/* SPILL1 REXX:   Perform a word spill at the specified length.  */

Parse Arg length .                  /* Desired maximum lrecl.      */

Do Forever                          /* Do until end-of-file.       */

  'PEEKTO'                          /* Test for end-of-file.       */
  If RC /== 0                       /* Exit if no more input.      */
     Then Exit RC*(RC<>12)

  'CALLPIPE (end ? name Spill1)', /* Split up single record:     */
    '*: |',                         /* Input from pipeline.        */
    'take 1 |',                     /* Sip one record to process.  */
    'split |',                      /* Split it into words.        */
    'join * / /' length '|',        /* Rejoin to desired length.   */
    '*:'                            /* Output records to pipeline. */

  'STREAMSTATE OUTPUT 0'            /* Test for end-of-file.       */
  If RC == 12  Then Exit 0          /* Exit if output severed.     */

End
```

The subroutine pipeline itself is very simple. What it does is sip one record, split that up into individual words, and then rejoin those words into one or more records that are no longer than the length specified as an argument. The idea is to split the records at a word boundary, rather than in the middle of a word.

The reason for having coded this subroutine pipeline to process only one input record at a time was to prevent it from putting words from different input records into the same output record. That is, the join * would join all the words from all the records in the input stream, if it were not prevented from doing that by the take 1.

This technique has great power and simplicity, but it is not always the cheapest way to do things. Rita produced these numbers when this REXX filter was used to process 5,000 records:

```
CPU utilization of pipeline specification "DoSpill1":
   11 ( 11) ms in stage 1 of segment 1: < all notebook
   16 ( 16) ms in stage 2 of segment 1: take 5000
  932 (606) ms in stage 3 of segment 1: spill1 60
    7 (  7) ms in stage 4 of segment 1: hole
  966 (640) ms total in "DoSpill1" (1 invocation) <=====

CPU utilization of pipeline specification "Spill1":
   37 ( 37) ms in stage 2 of segment 1: take 1
  125 (125) ms in stage 3 of segment 1: split
  165 (165) ms in stage 4 of segment 1: join * / / 60
  326 (326) ms total in "Spill1" (5000 invocations) <=====

  966 ms attributed to stages; 6 virtual I/Os.
```

```
 1124 ms in scanner.
  191 ms in commands.
  733 ms in dispatcher.
```

The spill1 stage used 932 ms of CPU, 606 of which were not attributed to running its subroutine pipeline; that is, there was a lot happening in the REXX filter in addition to the running of the subroutine pipeline. Note also the fairly large *Pipelines* overhead numbers shown at the bottom of the Rita display, particularly the time spent in the pipeline scanner, which had to scan that CALLPIPE command 5,000 times.

Word Spill With Continuous Flow

The reason for doing the 5,000 CALLPIPE commands was to avoid mixing together words from more than one of the input records. The CALLPIPE loop was a simple, straightforward solution to that problem. Any solution that subsumes that loop into a single pipeline is likely to be more complex. However, if this is a high-use filter, you may well judge the increase in complexity to be offset by the reduction in cost.

Here is a way to perform the same function with continuous flow through a single CALLPIPE:

```
/* SPILL2 REXX:   Perform a word spill at the specified length.  */

Parse Arg length .                  /* Desired maximum lrecl.      */

'CALLPIPE (end ? name Spill2)',     /* Split up multiple records:  */
     '*: |',                        /* Input from pipeline.        */
  'o: fanout |',                    /* Divert second copy.         */
     'spec number 1.10 |',          /* Convert first to key.       */
  'j: juxtapose |',                 /* Prefix keys to words.       */
     'join keylength 10 / /' length+10 '|',  /* Join as desired. */
     'not chop 10 |',               /* Remove the record key.      */
     '*:',                          /* Output to pipeline.         */
'?',
  'o: |',                           /* Second copy to here.        */
     'split |',                     /* Split into words.           */
  'j:'                              /* Send to be prefixed.        */
```

As each record enters this subroutine pipeline through the input connector, it passes to a fanout stage that makes a second copy and sends that to the second pipeline segment, where it is split up into one-word records. The first copy of each input record goes from the fanout to the spec stage, which converts it into a record containing only a 10-digit sequence number. juxtapose then reads each of the one-word records from its secondary input and prefixes it with the corresponding sequence number record from its primary input, producing an output stream containing records that each contain a single word prefixed by the sequence number of the input record from which that word was derived. join keylength joins the records that begin with the same sequence number. It inserts a blank between each of the records it joins, and it joins only as many records

as it can without exceeding the specified output length. An output record contains the key (the sequence number) only once, at the beginning, no matter how many input records were joined to form that output record. Thus, each of the original input records may be converted into one or more output records, just as in the earlier implementation. not chop is used to strip off the sequence number, and the records are sent through the output connector to the calling pipeline.

You will note that the PEEKTO and the STREAMSTATE are not required here, because there is no longer a loop to terminate. When either the input stream or the output stream goes to end of file, the CALLPIPE command will terminate, resulting in spill2 as a whole terminating.

No loop is required because the trick of attaching the same sequence number to each piece derived from a given input record allows us to maintain the record boundaries while pulling the records apart and putting them back together. Thus, all the processing can be done in a single subroutine pipeline.

Rita shows interesting results for this implementation:

```
CPU utilization of pipeline specification "DoSpill2":
    7 (  7) ms in stage 1 of segment 1: < all notebook
   14 ( 14) ms in stage 2 of segment 1: take 5000
  538 (  1) ms in stage 3 of segment 1: spill2 60
    6 (  6) ms in stage 4 of segment 1: hole
  565 ( 28) ms total in "DoSpill2" (1 invocation) <=====

CPU utilization of pipeline specification "Spill2":
   27 ( 27) ms in stage 2 of segment 1: fanout
   65 ( 65) ms in stage 3 of segment 1: spec number 1.10
  178 (178) ms in stage 4 of segment 1: juxtapose
  163 (163) ms in stage 5 of segment 1: join keylength 10 / / 70
   19 (  0) ms in stage 6 of segment 1: not chop 10
   85 ( 85) ms in stage 2 of segment 2: split
  537 (518) ms total in "Spill2" (1 invocation) <=====

CPU utilization of pipeline specification "not":
   19 ( 19) ms in stage 2 of segment 1: chop 10
   19 ( 19) ms total in "not" (1 invocation) <=====

  565 ms attributed to stages; 6 virtual I/Os.

    1 ms in scanner.
    0 ms in commands.
  490 ms in dispatcher.
```

Running the spill2 subroutine pipeline once to process 5,000 records is actually more costly than running the spill1 subroutine 5,000 times, because spill2 does much more work on each record it processes. However, both REXX and *Pipes* overhead drop dramatically in the spill2 case, making the total cost of spill2 only one-fourth that of spill1. (With no loop in spill2, REXX had so little to do that it used only 1 ms. Without the loop, the pipeline scanner had only one CALLPIPE to scan, and the pipeline command processor did not have 5,000 PEEKTOs and STREAMSTATEs to process.)

Of course, the real solution is to use the spill built-in program added in CMS 12. spill requires only 18 ms to do a much nicer job of word-spilling these 5,000 records.

Other Techniques for Achieving Continuous Flow

In general, one writes a sipping subroutine in order to operate on a single record or a single group of records without getting them mixed up with other records. In cases where performance considerations require a different approach, there are several useful techniques, in addition to the one we've just seen of numbering the records. Two techniques you should keep in mind are:

1. Inserting a marker at the end of each group of input records to use later to recover the original record boundaries:

```
 . . .
'spec 1-* 1 x00 next |',          /* Delimit record for later.   */
'deblock 1 |',                    /* One record per character.   */
'unique first |',                 /* Don't want runs of the same.*/
'deblock linend 00 terminate |', /* Restore record boundaries.  */
 . . .
```

| This fragment deletes all but the first occurrence of runs of the same character. A character known not to occur in the data is inserted at the end of each original record. The records are split into one-byte records. Consecutive duplicates are discarded. And then deblock restores the original record boundaries, deleting the marker characters in the process.

2. Joining the records for each group together into a single record before processing them:

```
 . . .
'join * / / |',                   /* Join all NAMES records.     */
'split before string /:NICK./ |', /* Split into NAMES entries.  */
'locate /:LIST./ |',              /* Select those with lists.    */
 . . .
```

| This fragment process records from a NAMES file discarding those that do not contain :LIST tags. All the records from the file are joined into a single record with a blank inserted between the original records. That single record is split before the :NICK tags, thus producing one record per NAMES file entry, and entries not containing :LIST tags are discarded.

OPTIMIZING UNABSORBED LOOPS

It may often be the case that for reasons of simplicity you will prefer to issue a subroutine pipeline repeatedly in a loop rather than absorbing that loop into a single pipeline. In those cases, there are a few simple techniques that can be used to reduce REXX and *Pipelines* overhead.

Spill1 Phase 2: Reducing REXX Overhead

When looking at the SPILL1 REXX example, we noticed that REXX was doing a lot of work, 606 ms worth. Any of the usual methods for reducing REXX overhead are likely to help. The most obvious is to compile SPILL1 REXX:

```
CPU utilization of pipeline specification "Dospill1C":
   13 ( 13) ms in stage 1 of segment 1: < all notebook
   17 ( 17) ms in stage 2 of segment 1: take 5000
  606 (274) ms in stage 3 of segment 1: rexx (spill1 crexx) 60
    7 (  7) ms in stage 4 of segment 1: hole
  643 (312) ms total in "DoSpill1C" (1 invocation) <=====

CPU utilization of pipeline specification "Spill1":
   38 ( 38) ms in stage 2 of segment 1: take 1
  126 (126) ms in stage 3 of segment 1: split
  168 (168) ms in stage 4 of segment 1: join * / / 60
  332 (332) ms total in "Spill1" (5000 invocations) <=====

  643 ms attributed to stages; 6 virtual I/Os.

 1151 ms in scanner.
  196 ms in commands.
  745 ms in dispatcher.
```

Although that reduces the time in REXX from 606 ms to 274 ms, it has no impact on the *Pipes* overhead, since 5,000 CALLPIPE commands are still issued and scanned, 5,000 PEEKTO commands are still issued, and 5,000 STREAMSTATE OUTPUT commands are still issued.

Spill1 Phase 3: Reducing Pipelines Overhead with EOFREPORT

The *Pipelines* overhead of SPILL1 REXX can be reduced by reducing the number of *Pipelines* subcommands issued. CMS 14 introduced the EOFREPORT subcommand, the purpose of which is to facilitate writing REXX filters that propagate end-of-file "quickly". That is, EOFREPORT allows a filter to notice end-of-file on its output stream while it is waiting to read from its input stream and to notice end-of-file on its input stream while it is waiting to write to its output stream. An additional benefit of EOFREPORT is that it may provide a performance improvement by allowing one to issue fewer *Pipelines* subcommands. In the case of spill1, for example, we can insert an EOFREPORT ALL command before the loop and remove the STREAMSTATE OUTPUT command from within the loop:

```
/* SPILL1X REXX:   Perform a word spill at the specified length. */

Parse Arg length .                /* Desired maximum lrecl.      */

'EOFREPORT ALL'                   /* Watch for severed streams.  */

Do Forever                        /* Do until end-of-file.       */

  'PEEKTO'                        /* Test for end-of-file.       */
  If RC /== 0                     /* Exit if no more input.      */
     Then Exit RC*(RC<>12 & RC<>8)

  'CALLPIPE (end ? name Spill1X)', /* Split up single record:    */
    '*: |',                       /* Input from pipeline.        */
    'take 1 |',                   /* Sip one record to process.  */
    'split |',                    /* Split it into words.        */
    'join * / /' length '|',      /* Rejoin to desired length.   */
    '*:'                          /* Output records to pipeline. */

End
```

When the output stream is severed as the result of the take 5000 in the main pipeline, the PEEKTO will terminate with return code 8, giving us the same results as before but requiring 4,999 fewer *Pipelines* subcommands to be issued.

That reduces the time in REXX from 274 ms to 186 and the command-processing overhead from 196 ms to 115:

```
CPU utilization of pipeline specification "DoSpill1X":
   14 ( 14) ms in stage 1 of segment 1: < all notebook
   16 ( 16) ms in stage 2 of segment 1: take 5000
  516 (186) ms in stage 3 of segment 1: rexx (spill1x crexx) 60
    7 (  7) ms in stage 4 of segment 1: hole
  553 (223) ms total in "DoSpill1X" (1 invocation) <=====

CPU utilization of pipeline specification "Spill1X":
   37 ( 37) ms in stage 2 of segment 1: take 1
  126 (126) ms in stage 3 of segment 1: split
  167 (167) ms in stage 4 of segment 1: join * / / 60
  330 (330) ms total in "Spill1X" (5000 invocations) <=====

  553 ms attributed to stages; 7 virtual I/Os.

 1127 ms in scanner.
  115 ms in commands.
  708 ms in dispatcher.
```

Moving Stages Out of the Loop

If you are unable to absorb your loop into a single CALLPIPE command, you may still find it possible to move some of the stages out of the loop. Assume that you have a loop that issues a CALLPIPE something like this:

```
'EOFREPORT ALL'                    /* Propagate eof quickly.      */

Do Forever                         /* Do until end-of-file.       */

  'PEEKTO'                         /* Test for end-of-file.       */
  If RC /== 0                      /* Exit if no more input.      */
     Then Exit RC*(RC<>12 & RC<>8)

   'CALLPIPE (name Glenn1)',
     '*: |',                       /* Input from pipeline.        */
     'take 5 |',                   /* Sip a set of records.       */
     'xlate |',                    /* Upper-case them.            */
     'pick w1 >> w2 |',            /* Select the ones we need.    */
     'join * |',                   /* Join into one record.       */
     'duplicate |',                /* Make a second copy.         */
     '*:'                          /* Output to pipeline.         */

End
```

```
CPU utilization of pipeline specification "Glenn1":
   41 ms in stage 2 of segment 1: take 5
  100 ms in stage 3 of segment 1: xlate
   82 ms in stage 4 of segment 1: pick w1 >> w2
   28 ms in stage 5 of segment 1: join *
   12 ms in stage 6 of segment 1: duplicate
  263 ms total in "Glenn1" (2358 invocations) <=====

  670 ms in scanner.
   56 ms in commands.
  520 ms in dispatcher.
```

You will note that there is no real need for the function performed by the xlate and duplicate stages to be done within the loop. All input records could be upper-cased and all output records could be duplicated. And this could be done by prefixing an ADDPIPE to the stage's input stream and suffixing an ADDPIPE to the stage's output stream:

```
'EOFREPORT ALL'                    /* Propagate eof quickly.      */

'ADDPIPE *.input: | xlate | *.input:' /* Upper-case all input.   */
'ADDPIPE *.output: | duplicate | *.output:' /* Duplicate output. */

Do Forever                         /* Do until end-of-file.       */

  'PEEKTO'                         /* Test for end-of-file.       */
  If RC /== 0                      /* Exit if no more input.      */
     Then Exit RC*(RC<>12 & RC<>8)

   'CALLPIPE (name Glenn2)',
     '*: |',                       /* Input from pipeline.        */
     'take 5 |',                   /* Sip a set of records.       */
     'pick w1 >> w2 |',            /* Select the ones we need.    */
     'join * |',                   /* Join into one record.       */
     '*:'                          /* Output to pipeline.         */

End
```

Even though only two stages are moved out of the loop in this case, total CPU utilization drops 25%, because the scanner and the dispatcher both have less work to do:

```
CPU utilization of pipeline specification "NoName001":
   88 ms in stage 2 of segment 1: xlate
   88 ms total in "NoName001" (1 invocation) <=====

CPU utilization of pipeline specification "NoName002":
    8 ms in stage 2 of segment 1: duplicate
    8 ms total in "NoName002" (1 invocation) <=====

CPU utilization of pipeline specification "Glenn2":
   41 ms in stage 2 of segment 1: take 5
   83 ms in stage 3 of segment 1: pick w1 >> w2
   27 ms in stage 4 of segment 1: join *
  151 ms total in "Glenn2" (2358 invocations) <=====

  529 ms in scanner.
   54 ms in commands.
  435 ms in dispatcher.
```

Subroutine Pipelines Invoked by Subroutine Pipelines in Loops

If you have a subroutine pipeline invoked repeatedly within a loop, it is a good idea to check whether it uses any REXX filters and to take some precautions if it does.

Of course, the very first precaution is to make sure you are EXECLOADing those REXX filters. I had a sipping subroutine that invoked Joachim Becela's SPACE REXX in two places. space is the equivalent of the REXX Space() function; it contains this subroutine pipeline:

```
'CALLPIPE (name Space)',
  '*: |',                              /* Input from pipeline.      */
  'tokenize / / / / |',                /* Split and delimit.        */
  'joincont not leading / / / / |',    /* Rejoin Space'd.           */
  'strip leading |',                   /* Remove delimiting blanks. */
  'locate 1 |',                        /* Discard last delimiter.   */
  '*:'                                 /* Output to pipeline.       */
```

When my sipping subroutine was invoked 530 times, the times recorded for its two invocations of space were 538 ms and 341 ms. After I added an EXECLOAD SPACE REXX command to my EXEC, however, the time charged to the first invocation was reduced to 313 ms (as it had been being charged for the time spent loading the REXX file).

It is often the case that a REXX filter, such as this one, has no REXX logic but is just a segment of pipeline that has been packaged separately to hide complexity. If that is the case and *if* the REXX filter is being invoked from a sipping subroutine pipeline in a production application, you may want to consider pulling the pipeline segment up into the sipping subroutine to avoid the cost of repeatedly launching a REXX filter. When my sipping subroutine was invoked 551 times, the two invocations of space used about 700 ms (with SPACE REXX EXECLOADed). When I moved the four stages from the pipeline in SPACE REXX up into my sipping subroutine, the two occurrences of those four stages used only 100 ms altogether.

Near the end of a Rita report, you will see counts of the number of different pipeline specifications issued and of the total number of pipeline specifications issued. These were the numbers reported when I was invoking SPACE REXX as a separate filter:

```
    15 pipeline specifications used (126 stages).
  4969 pipeline specifications issued.
```

It is good to take note of the changes in these numbers when you are changing a complex pipeline. This is what I saw when I moved the stages from SPACE REXX up into the calling pipeline:

```
    14 pipeline specifications used (128 stages).
  3867 pipeline specifications issued.
```

As expected, 1102 fewer pipeline specifications were issued, which makes it not surprising that the time spent in the scanner and the dispatcher was reduced by ten percent.

Avoiding ALL in Loops

The all filter is a wonderful convenience in "quick-and-dirty" pipes, but for performance reasons you should avoid using it in loops in production applications. Originally, all simply generated the same sequence of locate, nlocate, and faninany stages you would have written yourself, so the only additional cost involved in using all was its time to generate that sequence (which seems to be about 13 ms on our 9672 for the all stage shown below). This was insignificant for one-time uses, but could mount up if the all were part of a subroutine pipeline that was being issued a large number of times in a loop. More recently, there have been significant enhancements to locate that have not been propagated to all, so you may now be able to generate a more efficient search sequence yourself than you would get using all.

For example, this stage in a subroutine pipeline that was invoked 619 times used 8,349 ms:

```
'all /(h/ ! /home/ ! /(H/ ! /Home/ ! /HOME/ |'
```

Because of recent enhancements to locate, that all stage could be replaced by this sequence:

```
      . . .
   'h: locate anycase /(h/ |',     /* Select "(h" or "(H".       */
   'H: faninany |',
      . . .
'?',
   'h: |',
      'locate anycase /home/ |',   /* Select home/Home/HOME/etc. */
   'H:'
```

These three stages used only 37 ms, a reduction of 99.6%.

CACHING THE RESULTS OF EXPENSIVE OPERATIONS

One of the most powerful techniques for streamlining your pipelines is to cache the results of expensive operations. This is usually easy to do using the lookup stage, which has been greatly enhanced in recent releases of CMS. For example, if you needed to do a DNS lookup on the hostnames in records passing through a pipeline and there were likely to be duplicates, you might do well to cache and reuse the output of the hostbyname stage rather than doing the same DNS lookups repeatedly. This is a simple filter to do DNS lookups and cache the responses (its input is a stream of hostnames; its output records have the IP address appended):

```
'CALLPIPE (listerr endchar ? name CachedDNSLookup)',
     '*: |',                          /* Hostnames to look for.      */
  'l: lookup 1-* word 1 master |',/* Already know about it?      */
  'y: faninany |',                    /* Bring in new cases.         */
     '*:',                            /* Names & IP addrs to caller. */
'?',
  'l: |',                             /* Unmatched hostnames here.   */
  'o: fanout |',                      /* Divert copy for new master. */
  'h: hostbyname |',                  /* Can we resolve it?          */
  'i: faninany |',                    /* Bring in ones we couldn't.  */
     'copy |',                        /* (Unblock FANOUT.)           */
  's: spec 1-* 1 select 1 1-* nw |', /* Build response/master.   */
  't: fanout |',                      /* Divert copy for response.   */
     'copy |',                        /* (Unblock LOOKUP.)           */
  'l:',                               /* Into LOOKUP's reference.    */
'?',
  'o: |',                             /* Unmatched hostnames here.   */
  's:',                               /* Into SPEC for new master.   */
'?',
  'h: |',                             /* HOSTBYNAME errors here.     */
  'i:',                               /* Return error to caller.     */
'?',
  't: |',                             /* Copy of HOSTBYNAME result.  */
  'y:'                                /* Out to the caller.          */
```

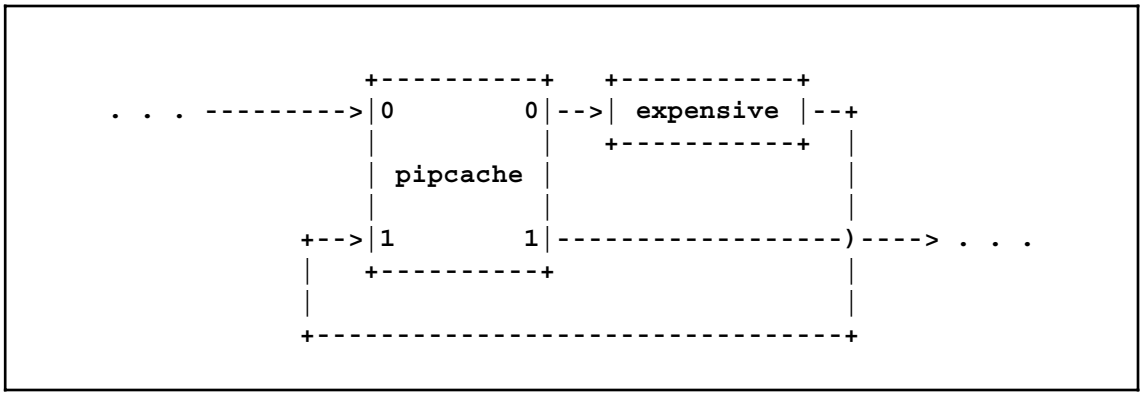
lookup's reference (its "master file") serves as the cache here. The master records contain two words, the hostname and the corresponding IP address. Each time a hostname record is read from the input, lookup tries to match it with a record in its reference. When a match is found, the record from the reference (the "master record") is written to lookup's primary output and is sent from there out to the caller.

The reference starts out empty, as lookup's secondary input stream is unconnected. So, at least the first hostname will not be matched. When a hostname is not matched, the detail record is written to the secondary output of lookup, and passes from there to hostbyname, which determines the IP address for that host. hostbyname's answer is fed into the tertiary input of lookup (the stream for masters to be added to the reference) and is also sent out to the caller. The next time that hostname is encountered on the primary input, lookup will find a match in its reference and will write the result directly to the caller.

The originally-empty lookup reference accumulates a new record each time another hostname is resolved, and no hostname need be resolved more than once, no matter how often it occurs in the input stream. Depending on the proportion of duplicates, using this technique can result in dramatic reductions in both CPU and elapsed time.

If you are not yet comfortable building feedback loops for lookup yourself, Rob van der Heij has provided a production-strength cache stage called PIPCACHE REXX that is ready to be used right out of the box. The pipcache stage requires a somewhat unusual topology:

```
'PIPE . . . | p: pipcache | expensive-operation | p: | . . .'
```



pipcache assumes that the stages between its primary output and its secondary input perform the operation whose results are to be cached. When pipcache reads a record from its primary input, it searches for a match in its cache. If a match is found, pipcache writes the result value from its cache to its *secondary* output. If no match is found, pipcache copies the input record to its *primary* output, where it is put through the expensive operation. The result of that operation is fed into the secondary input of pipcache, which caches the result and copies it to its secondary output. Thus, whether or not there was a cache hit, the result is written to the secondary output of pipcache. In the process, the result is cached if it was not already in the cache.

As an example of a case in which one would want to use pipcache, let's look at this fragment that processes the output of a CP QUERY RDR command with the ISODATE option:

```
 . . .
'spec w8 1  1-* nw |',              /* Prepend copy of isodate.  */
'dateconvert w1 isodate rexxb |',   /* Convert to Rexx Date('B').*/
 . . .
```

Word 8 of the response (the date in isodate format) is copied to the beginning of the record, so that the dateconvert stage can be used to convert it to the REXX base date while leaving the original record intact.

With 35,000 spool files, using dateconvert turned out to be extremely expensive:

```
  865 ms in stage 7 of segment 1: spec w8 1  1-* nw
58606 ms in stage 8 of segment 1: dateconvert w1 isodate rexxb

 1662 ms in dispatcher.
63006 ms total.
```

Since there were known to be many repeats of the dates, the obvious solution was to cache the converted dates and execute dateconvert only once for each unique date:

```
      . . .
   'd: spec w8 1  write',          /* Just date to primary, and   */
        'outstream 1  1-* 1 |',    /*   whole record to secondary.*/
   'p: pipcache |',                /* Cache for DATECONVERT.      */
      'dateconvert isodate rexxb |',        /* To Rexx Date('B'). */
   'p: |',
      'copy |',                    /* (Unblock first SPEC.)       */
   's: spec 1-* 1  select 1  1-* nw |',      /* Prefix Date('B'). */
      . . .
'?',
   'd: |',                         /* Original records to here.   */
      'copy |',                    /* (Unblock first SPEC.)       */
   's:'                            /* Feed into second SPEC.      */
```

Because pipcache treats its entire input record as the key when looking up values in its cache, we must give it records containing only the date to be converted. So, the spec stage is modified to write only the date to its primary output and to write the original record to its secondary output. Once the isodate record has been converted to the base date, another spec is used to reunite the two records, giving the same output as before.

As pipcache reads each isodate record from its primary input, it looks for a match in its cache. If a match is found, the cached base date is written to the secondary output. If no match is found, the isodate record is copied to the primary output, where dateconvert reads it and converts it to the base date. pipcache then reads the base date from its secondary input, stores it in its cache, and copies it to its secondary output. In either case, the second spec stage reads the base date from its primary input and prefixes it to the original QUERY RDR output record, which it reads from its secondary input.

Although this more complex solution requires twice as much dispatcher time, the total cost of the pipeline is reduced by 86%:

```
  807 ms in stage  7 of segment 1: spec w8 1  write  outstream 1  1-* 1
 1300 ms in stage  8 of segment 1: pipcache
  400 ms in stage  9 of segment 1: dateconvert isodate rexxb
  171 ms in stage 11 of segment 1: copy
  798 ms in stage 12 of segment 1: spec 1-* 1  select 1  1-* nw
      . . .
  160 ms in stage  2 of segment 3: copy

 3480 ms in dispatcher.
 8787 ms total.
```

Exactly the same scheme can be used anywhere your pipelines perform an expensive or slow-running operation that doesn't really need to be repeated when its input is the same. PIPCACHE REXX is shown on the next page.

```
/*--------------------------------------------------------------------*/
/*  PIPCACHE REXX                          Rob vd Heij                */
/*                                                                    */
/*      Records containing values to be converted by the subsequent   */
/*  stage are read from the primary input stream.  Records with the   */
/*  results of the conversion are written to the secondary output.    */
/*  The stage that does the conversion gets its input from the        */
/*  primary output of this stage and writes its output to the         */
/*  secondary input of the stage.  The conversion is bypassed (the    */
/*  record from the primary input is not written to the primary       */
/*  output) if an earlier record had the same value, in which case    */
/*  the result can be found in the cache maintained by this stage     */
/*  and can be written directly to the secondary output, saving the   */
/*  cost of doing the conversion again.                               */
/*      The master records in the first LOOKUP stage below are input  */
/*  values that have not been "converted" by being fed through the    */
/*  (expensive) stage that follows this one.  Each of these master    */
/*  records is preceded by a 10-character sequence number key.        */
/*  The master records in the second LOOKUP stage use that same       */
/*  sequence number key to denote the corresponding converted value.  */
/*      If you are running a version of CMS Pipelines that does not   */
/*  support the "propagateeof" global option, change the CALLPIPE     */
/*  to ADDPIPE.                                                       */

'CALLPIPE (listerr endchar \ name PipCache propagateeof)',
   '\     *.input.0: ',               /* Values to be converted.     */
   '| l1: lookup 1-* 11-* master ',   /* Value already converted?    */
   '| l2: lookup 1.10 master',        /* Yes, use key to find result.*/
   '|     not chop 10',               /* Remove seq nr from result.  */
   '| f2: faninany',                  /* Either matched or converted.*/
   '|     *.output.1:',               /* Output all results.         */
   '\ l1:',                          /* Here if value not in table. */
   '|     spec number 1 1-* n',       /* Assign sequence number key. */
   '| f1: fanout stop anyeof',        /* Divert copy to be converted.*/
   '|     copy',                      /* (Unblock 1st LOOKUP/FANOUT.)*/
   '| l1:',                          /* New master to first LOOKUP. */
   '\ l2:',                          /* (Placeholder for 2ndry.)    */
   '\ f1:',                          /* Key + value to be converted.*/
   '| nr: chop 10',                  /* Keep numeric key here.      */
   '|     copy',                      /* (Unblock CHOP.)             */
   '|  s: spec 1.10 1 select 1 1-* n',/* Append result of conversion.*/
   '| l2:',                          /* New master to second LOOKUP.*/
   '\ nr:',                          /* Value to be converted.      */
   '|     *.output.0:',               /* To primary output so other  */
   ,                                 /* .. stage can convert it.    */
   '\     *.input.1:',               /* We read result from here.   */
   '| f3: fanout',                   /* One copy of the result to   */
   '| f2:',                          /* .. go to secondary output.  */
   '\ f3:',                          /* Another copy to feed into   */
   '|  s:'                           /* .. 2nd LOOKUP's reference.  */

error: return rc * ( rc <> 12 )
```

OTHER TIPS FOR STREAMLINING YOUR PIPELINES

1. Become familiar with the notes in the author's help files, which often mention performance ramifications. For example, pipe ahelp faninany displays this advice regarding the performance of the STRICT option:

   > 3. Depending on the number of input streams, STRICT may add significant overhead. Use it only when you can prove from the topology that it really is needed.

2. Be sure to note differences in the virtual I/O count when comparing alternate implementations. Rita displays the I/O count near the end of its detail output:

```
480 ms attributed to stages; 11 virtual I/Os.
```

3. Note differences in virtual memory utilization when comparing alternate implementations. Recent versions of Rita display the highwatermark memory utilization for each stage in each pipeline:

```
  2 (  2) ms (65K) in stage 1 of segment 1: < all notebook l
  3 (  3) ms (<1K) in stage 2 of segment 1: take 1000
141 ( 88) ms ( 3K) in stage 3 of segment 1: spill1x 60
```

4. Always keep an eye open for ways to reduce the number of records flowing through your pipelines. One way to do that is to discard unneeded records as early in your processing as feasible, but another is to *combine* records as early as possible. For example, a sipping subroutine that was invoked 537 times included these stages:

```
      . . .
   'H: fanin |',              /* Bring in the mail body.      */
      'join * x0D25 |',       /* Make single record w/CRLFs.  */
      . . .
'?',
      . . .
      'buffer |',             /* Hold the mail body here.     */
   'H:'                       /* Mail body following headers. */
```

   The buffer held the body of a mail file until fanin had read the mail headers from its primary input. The streams were then merged and the records were delimited by carriage return and linefeed and joined into a single record. The times for those three stages were 14, 99, and 69 ms, respectively. When the buffer was replaced with another join * x0D25, which both buffered the records and combined them into a single record, the times for the three stages became 14, 40, and 80 ms, a reduction of 26%.

5. Use xlate rather than change for single-character changes and look for ways to do more than one thing with a single xlate stage. For example, this:

```
 . . .
'xlate upper |',                 /* Make sure it's uppercase.   */
'change //WHALE DISPLAY / |',    /* Make it a WHALE command.    */
'change /./ / |',                /* Tokenize.                   */
 . . .
```

can be replaced by this:

```
 . . .
'xlate upper . 40 |',            /* Upper-case and tokenize.    */
'change //WHALE DISPLAY / |',    /* Make it a WHALE command.    */
 . . .
```

6. Learn to use the arithmetic capability of spec (also known as the “407 emulation”). Using the 407 emulation to perform arithmetic operations seems to cost about half what it costs to do them in interpreted REXX, but more than it costs to do them in compiled REXX. (You can learn about using the 407 emulation by issuing the command pipe ahelp spectut, which will display a tutorial, or pipe ahelp specref, which will display reference material.)

7. Keep lookup in mind whenever you need to do a series of operations on the same field of your records. For example, lookup can often replace a cascade of locate or change stages.

8. Become familiar with vchar, which can be used to do many functions involving acting on every *n*-th byte. For example, to delete every third byte, starting with the first one, use vchar 24 16. (In this case, vchar takes 24 bits from its input stream and places those in a 16-bit output “character”, discarding the eight high-order bits, thus deleting every third byte.) The examples displayed by pipe ahelp vchar will give you some idea of the power of this stage.

9. If you need to read a minidisk file into two separate pipelines in the same EXEC, don’t load the file into a REXX stem; read the file twice! The cost of using EXECCOMM to build and break down a large stem is almost always much greater than the cost of reading the file twice.

   And if a file must be read multiple times, consider EXECLOADing it and using < with no filemode to read the in-storage copy.

   Or, if you want to get fancy, you may be able to store the file in memory using the instore stage. This REXX filter (which was written before dam became available) reads its input stream and passes all of the records to its output, if any of them contains the string specified in its argument:

```
/*   Copy all input to output or none (if no matches found).   */

'ADDPIPE',                          /* Prefixed ADDPIPE.          */
   '*.input: |',                    /* Connect to our input stream.*/
   'instore |',                     /* Store all records.         */
   '*.input:'                       /* Write pointer to our input. */

'PEEKTO pointer'                    /* Read but don't consume.    */

'CALLPIPE',                         /* Examine the passed file:   */
   'var pointer |',                 /* Pick up copy of pointer.   */
   'outstore |',                    /* Expand file into pipeline. */
   'locate /'Arg(1)'/ |',           /* Any hits?                  */
   'take 1 |',                      /* Make answer Boolean.       */
   'count lines |',                 /* Make answer numeric.       */
   'var found'                      /* Store answer.              */

If found  Then                      /* If got any hits, ...       */
   'CALLPIPE',                      /* ... output entire file.    */
      'var pointer |',              /* Load pointer again.        */
      'outstore |',                 /* Expand file again.         */
      '*.output:'                   /* Write to output this time. */

'READTO'                            /* Now consume the pointer.   */

Exit
```

instore reads all the records from its input stream and stores them in memory. Once it has read all of its input, it writes a single output record that describes the in-storage file. This file token ("pointer") record produced by instore can be used repeatedly to load the file into a pipeline with an outstore stage, but once that record has been consumed, the in-storage copy of the file no longer exists. In this example, the ADDPIPE prefixed to the stage's input stream stores its input in memory and writes the file token record to the stage's input stream. The stage is careful not to consume that file token record until after the file has been read twice.

10. When searching for a string at the beginning of records, use find or strfind rather than locate. find uses about 20% less CPU than locate 1.n uses.

11. Consider whether operations in a spec stage can be combined. These two spec stages produce the same output, but the second one uses 17% less CPU:

```
'. . . | spec  1.10  1   /overlay/  11   18-*  18 | . . .'

'. . . | spec   1-*  1    /overlay/  11 | . . .'
```

12. Become familiar with the possibilities for using tokenize to parse your records. For example, the tokenize stage below comes from a pipeline that removes strings enclosed in square brackets from a file containing text. It performs the same operations as the cascade of stages in the next line, but the cascade uses 4.5 times as much CPU as the tokenize:

```
'. . . | tokenize /<>/ / / | . . .'

'. . . | spec 1-* 1 write / / 1 | split before anyof / <>/ | split after anyof / <>/ | . . .'
```

TIPS FOR USING RITA

1. Rita produces the best results when all CALLPIPEs and ADDPIPEs have the NAME option specified and when any specifications that have the same name are identical to one another.

2. If a user-written pipeline stage goes into a wait state outside of the control of *Pipes*, the wait time will be counted as CPU time. The most common case of this (one that can be very misleading) is a REXX stage that uses the REXX Say instruction. The time during which the user's terminal is in the "MORE" condition will be reported by Rita as CPU time used by the REXX stage.

3. The numbers displayed by Rita do not include the CPU utilization of other pipeline sets started with PIPE commands or runpipe stages from within the pipeline set Rita is measuring.

4. To use Rita with a pipeline that contains a ldrtbls stage, NUCXLOAD RITA MODULE so that invoking Rita doesn't alter the loader tables.

5. Rita is much less costly to use if RITA REXX has been compiled.

6. There is a bug in the version of Rita shipped with CMS 12 that causes it to loop when the subject pipeline uses an input console stage. If you need to measure such a pipeline, get a newer version of Rita from one of the archives.

CONCLUSION

Tuning a heavily-piped application is no different from tuning any other sort of application (or system). First you find out where the hot spots are and then you find ways to cool them down.