

Tuning the BBJ Performance Analyzer

By Jim Douglas

For several years, BBJ®, PRO/5® and Visual PRO/5® developers have optimized their applications with the help of the Performance Analyzer. Recently, we discovered many have used it to process trace files as large as hundreds of megabytes. These file sizes are much larger than anyone anticipated, which resulted in long import times. We investigated the trace file import process and found there was room for significant improvement. This article addresses the review process and details the improvements in the Performance Analyzer for the BBJ 6.0 release.

To begin testing, we inserted a SETTRACE in the Performance Analyzer import code (File→Open), then we imported a typical 10 MB trace file of about 108,700 lines of code. Next, we loaded this trace file into the Performance Analyzer itself.

Program	Line	Count	Total	Average	Percent	Statement
1105	435,913	222,523.27	.5105	16.0649%	[1105] index=ind(tracefile,err=endloop),newpct=round(index*100/size,0)	
1106	435,913	56,224.98	.1290	4.0591%	[1106] if newpct>pct then Progress!.set!Value(index); pct=newpct	
1109	435,912	51,446.39	.1180	3.7141%	[1109] goto loop	
1138	108,697	51,059.10	.4697	3.6862%	[1138] workrec.average=workrec.total/workrec.count	
1104	435,913	45,024.72	.1033	3.2505%	[1104] loop:	
1140	108,697	44,918.66	.4132	3.2429%	[1140] write record (workfile)workrec\$	
1114	326,934	43,852.71	.1341	3.1659%	[1114] if pos(">EXITING TO:"=rec\$)=1 then newprog=cvs(rec\$(13),3),level=level-1; if level=0 and main\$="" and newprog<>"" and n	
1113	326,934	43,242.14	.1323	3.1218%	[1113] if pos(">RUNNING:"=rec\$)=1 then newprog=cvs(rec\$(10),3); if level=0 and main\$="" and newprog<>"" and newprog<>"NO	
1117	326,656	41,666.01	.1276	3.0080%	[1117] if colon=5 and colon<=6 and pos(rec\$(1,colon-1)=line)=1 then num=num(rec\$(1,colon-1),err=not_line_number),rec\$=rec\$(colon+1	

Figure 1. Results from importing a typical 10 MB trace file into the Performance Analyzer

Importing that 10 MB trace file into the Performance Analyzer took about 23 minutes. About 16% of that total, shown in **Figure 1** was spent tracking the import process to update the progress bar on the screen. Based on this finding, we changed the import process to only perform that calculation once for every 1,000 records read from the trace file. This change cut about a minute from the total import time and reduced the import process to more than 6.7% of the total. **Figure 2** shows the new results.

Program	Line	Count	Total	Average	Percent	Statement
1107	435,913	90,603.24	.2078	6.8128%	[1107] read (tracefile,end=endloop)rec\$	
1106	435,913	89,418.53	.2051	6.7238%	[1106] if mod(counter,1000)=0 then index=ind(tracefile,err=endloop),newpct=round(index*100/size,0); if newpct>pct then Progress!.set!Val	
1108	435,912	63,043.22	.1446	4.7405%	[1108] on mask(rec\$) gosub build_record,write_record	
1138	108,697	55,940.76	.5146	4.2064%	[1138] workrec.average=workrec.total/workrec.count	
1109	435,912	53,998.64	.1239	4.0604%	[1109] goto loop	
1105	435,913	50,431.72	.1157	3.7922%	[1105] counter=counter+1	
1104	435,913	49,024.61	.1125	3.6864%	[1104] loop:	
1114	326,934	47,469.22	.1452	3.5694%	[1114] if pos(">EXITING TO:"=rec\$)=1 then newprog=cvs(rec\$(13),3),level=level-1; if level=0 and main\$="" and newprog<>"" and n	
1140	108,697	47,009.82	.4325	3.5349%	[1140] write record (workfile)workrec\$	
1113	326,934	46,760.68	.1430	3.5161%	[1113] if pos(">RUNNING:"=rec\$)=1 then newprog=cvs(rec\$(10),3); if level=0 and main\$="" and newprog<>"" and newprog<>"NO	
1117	326,656	46,699.85	.1430	3.5116%	[1117] if colon=5 and colon<=6 and pos(rec\$(1,colon-1)=line)=1 then num=num(rec\$(1,colon-1),err=not_line_number),rec\$=rec\$(colon+1	

Figure 2. Results of calculation performed every 1000 records

A review of the new profile showed that several lines of code executed more than 435,000 times. The trace file contains 108,700 lines of code, plus the same number of timing lines, for a combined total of 217,400 lines. So why is the Performance Analyzer reading over 435,000 records from the file?

Since this test ran on Microsoft Windows, each line in the trace file is delimited by \$ODDAS\$. The BBJ READ verb sees this as one line terminated by \$OD\$, followed by another (blank) line terminated by \$OAS\$. To eliminate excess processing, we modified the READ statement to check for

continued...



Jim Douglas
Software Engineer
Contractor

these empty lines. If a record is blank, the program skips most of the processing, and just goes back to get the next record. **Figure 3** shows the profile for this version.

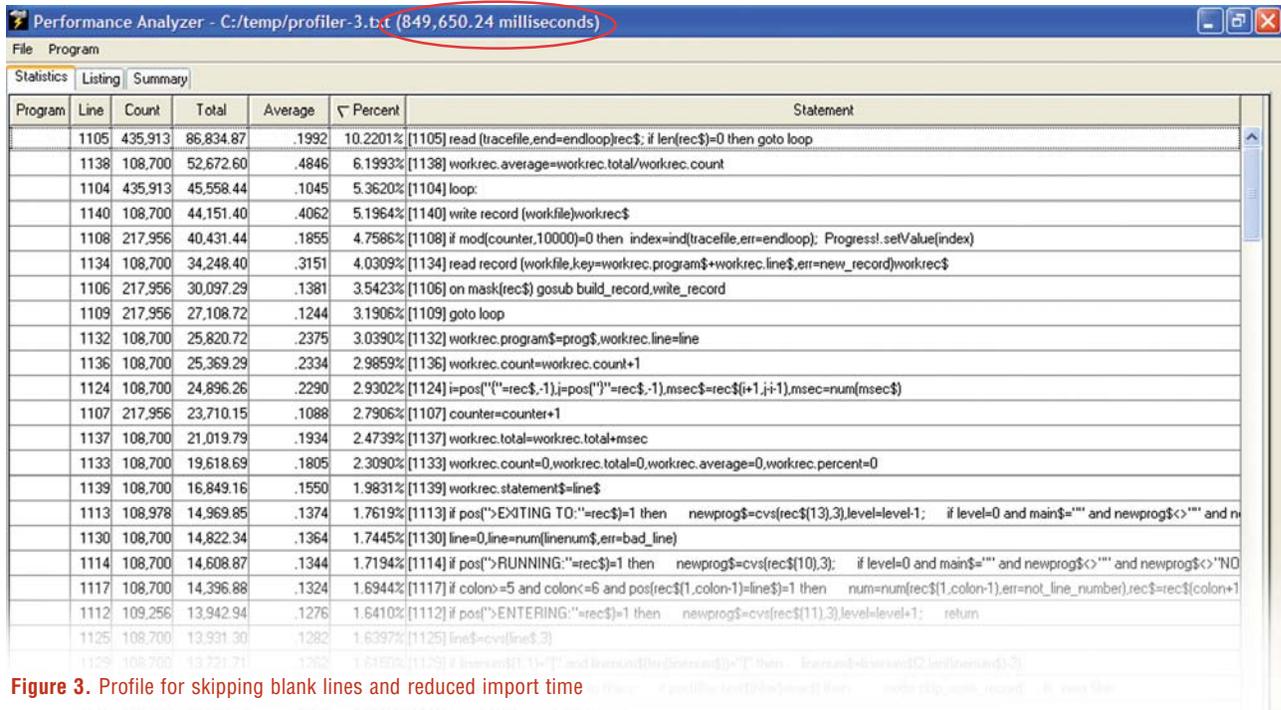


Figure 3. Profile for skipping blank lines and reduced import time

This last change trimmed the import process by another 8 minutes. In total, the import time was cut to 850 seconds (14:10). This is almost 9 minutes (535 seconds) faster than our original test – a 39% reduction. That’s a significant improvement for a few hours work, but it was still taking 14 minutes to import a moderately sized trace file. Since developers work with trace files as large as 500 megabytes, these incremental improvements were not enough.

Examine Your Assumptions

At this point, we thought about the overall structure of BBJ trace files and the Performance Analyzer. As originally designed, the Performance Analyzer reads through a trace file and stores summary information in an MKEYED file. When it finishes reading the trace file, it attaches the MKEYED file to a data-aware grid. This means that it has to read and update an MKEYED record for each line processed from the trace file. The trace file used in our testing contains 108,700 total lines of code, but only 2,027 unique lines of code. This is fairly typical; as most programs execute the same few lines of code many times. Though it is not practical to keep information about hundreds of thousands of lines of code in memory, we can easily calculate the summary information for a few thousand lines of code. By performing all of the processing in memory, we can completely eliminate the MKEYED file. Instead of attaching an MKEYED file to a data-aware grid, we can update the summary information directly from memory to a standard grid.

Ultimately, we decided to implement this approach using a Java utility program (referenced at the end of this article). However, the most dramatic improvements did not come from rewriting in Java; they came from examining our basic assumptions and taking a new approach.

Figure 4 shows the final test using the Java implementation. Total import time is less than 3½ seconds – nearly 250 times faster than our best time with the previous approach identified in **Figure 3**, and almost 400 times faster than the original version shown in **Figure 1**.

BBj 6.0 Performance Analyzer Enhancements

Improved performance is the most obvious change in the BBJ 6.0 Performance Analyzer – it loads trace files hundreds of times faster than the original Performance Analyzer. We also added several usability enhancements, many of them requested by BBJ developers.

continued...

Program	Line	Count	Total	Average	Percent	Statement
	1256	1	1,996.45	1,996.4500	59.5415%	[1256] Profiler.parse(tracefile\$)
	1258	1	103.74	103.7400	3.0939%	[1258] List! = Profiler.esList!
	1352	1	45.19	45.1900	1.3477%	[1352] grid.sortByColumn(4,-1)
	1299	1	17.43	17.4300	.5198%	[1299] grid.setCellText(0,0,statistics!)
	1261	1	12.11	12.1100	.3612%	[1261] statistics=new com.basis.bbj.client.datatypes.BB(Vector(list!))
	1354	1	2.21	2.2100	.0659%	[1354] row=grid.getRowModelIndexFromViewIndex(0)
	1318	1	2.12	2.1200	.0632%	[1318] list!=profiler.getSummaryList!
	1312	1	1.77	1.7700	.0528%	[1312] grid.setSelectedColumn(5)
	1253	1	.40	.4000	.0119%	[1253] print (gb._sysgui"context!gb._win.main,title!title\$+" - Loading "+tracefile!,"disable!0,"setcursor!BUSY)

Figure 4. Final test using the Java implementation

Client-Side Grid Sorting

The original Performance Analyzer stored statistical information in an MKEYED file and presented it using a data-aware grid. Clicking in a column header caused the program to generate a sorted SELECT command and attached that new SELECT command to the data-aware grid. Though this process was quick, there was a noticeable pause of a second or two and a visible repaint of the screen. The BBj 6.0 Performance Analyzer handles all sorting directly on the client. This is faster and visually smoother than the original approach. It turns out that this new approach is also easier to implement; it uses less code – and simpler code – than the original file-based approach. For more information on sorting, refer to the article, Grid to Order on page 16.

Program	Line	Count	Total	Average	Percent	Statement
gml	15310	3,953	3,862.97	.9772	28.8008%	15310 LET TF\$=SENDMSG(C,E_ID\$,54,0,T\$,GML.CONTEXT%!GR.SET%!
gml	15160	3,914	807.56	.2063	6.0209%	15160 LET GML_GR\$=GML.GR\$(GML.GR.TOT%*2)+1,0)(((FNDX(ROW!
gml	15170	3,914	583.22	.1490	4.3483%	15170 LET T.BUF\$=GML.DN.D\$(COL%+1); IF T.BUF\$="" OR GML_GC\$(
gml	15140	3,953	362.70	.0918	2.7042%	15140 SWITCH MODE_ID\$,5)
gml	15320	3,953	264.00	.0668	1.9683%	15320 NEXT COL%
gml	15130	3,953	260.23	.0658	1.9402%	15130 LET T.ROW%=ROW%
gml	15120	3,953	207.21	.0524	1.5449%	15120 LET T.COL%=COL%
gml	8490	2,120	201.88	.0952	1.5051%	8490 IF LEN(GML_TPL.T_COLORS\$(GML.COL%))=3 THEN LET GML_UPDA
gml	8420	2,120	165.87	.0782	1.2367%	8420 IF GML_TPL.MEMBER%=1 THEN IF GML_TPL.COL\$(GML.COL%)<>
gml	15150	3,914	161.50	.0413	1.2041%	15150 CASE 1
fmua	0470	1	156.16	156.1600	1.1643%	0470 LET BR\$=CTRL(C,5002,0); LET BR_Y%=BR_Y%; LET BR\$=CTRL(C,0
fmua	5250	3	148.38	49.4600	1.1063%	5250 PRINT (C)"CONTEXT"(%,)GRID(GR_ID\$,4,2,BR.W%-10,BR.H%-30,S

Figure 5. Color highlighting

Highlights

The Performance Analyzer presents a great amount of information, making it hard to decide on which lines of code to focus. The new color-highlighting feature, available from the Program→Highlights menu, enables developers to choose which criteria are important and highlight those specific pieces of information. The example shown in Figure 5 highlights the following values:

- all lines executed more than 10,000 times appear highlighted in yellow (none in this sample)
- all lines responsible for more than 1,000 milliseconds (1 second) of total execution time appear highlighted in green
- all lines that took more than 100 milliseconds to execute (on average) appear highlighted in blue
- all lines responsible for more than 2% of total execution time appear highlighted in red

Figure 5 shows how the lines that meet the highlighting criteria jump off the screen, directing the viewer's attention to the areas of the program worth considering for optimization. The color highlighting also appears in the listing view when loading a program listing (by double-clicking on any line of the Statistics or Summary grids).

continued...

Also new in BBj 6.0 are Filters, accessible from the Program→Options dialog. Filters, shown in **Figure 6**, omit lines containing specified strings from the statistical calculations.

To illustrate how this works, **Figure 7** shows a complete profile of the Performance Analyzer from the beginning of the program through to the parsing of a trace file. It is hard to focus on specific execution bottlenecks because about 59% of the measured time in this trace is spent waiting in the 'fileopen' dialog or otherwise waiting for user input (reading events from the event queue).

Figure 8 shows the trace without the program lines that only measure user typing time ('fileopen' and reading from the SYSGUI event queue). This gives us a better understanding of the real performance bottlenecks.

continued...

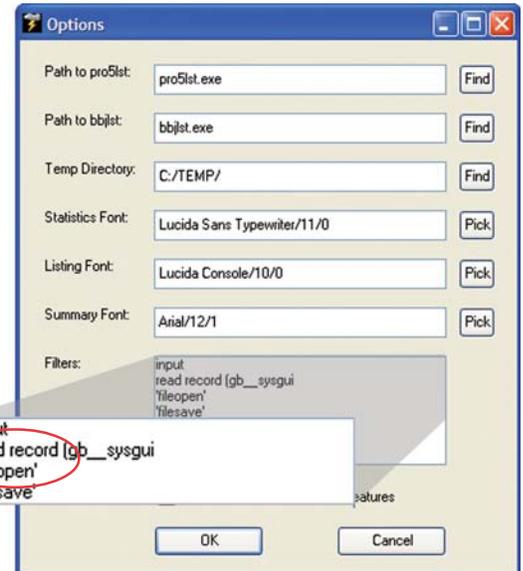


Figure 6. Program options showing filter's applied

Program	Line	Count	Total	Average	Percent	Statement
2035	1	13,340,163	13,340,1600	35.5952%	[2035]	print 'fileopen' Open Trace File",tracedir\$.",'
0789	31	8,785,778	283,4123	23.4428%	[789]	read record (gb_sysgui,siz=gb_event,err=gb_ever
1521	1	2,427,77	2,427,7700	6.4779%	[1521]	Profiler!.parse(tracefile\$)
0492	1	1,906,44	1,906,4400	5.0869%	[492]	print (gb_sysgui)'context'(gb_form_context
1576	1	1,556,31	1,556,3100	4.1527%	[1576]	grid!.sortByColumn(5,-1)
1578	1	1,266,28	1,266,2800	3.3788%	[1578]	row=grid!.getRowModelIndexFromViewIndex(0)
1572	1	1,028,12	1,028,1200	2.7433%	[1572]	if percent_highlight then bgcolor!= bbjapi().getS
1570	1	939,18	939,1800	2.5060%	[1570]	if average_highlight then bgcolor!= bbjapi().getS
0060	1	748,93	748,9300	1.9983%	[60]	gb_sysgui-unt; open (gb_sysgui,err=gb_cannot_ope
1566	1	695,57	695,5700	1.8560%	[1566]	if count_highlight then bgcolor!= bbjapi().getSys
1564	1	674,92	674,9200	1.8009%	[1564]	grid!.setCellText(0,0,statistics!)
1568	1	649,18	649,1800	1.7322%	[1568]	if total_highlight then bgcolor!= bbjapi().getSys
0513	1	632,53	632,5300	1.6878%	[513]	print (gb_sysgui)'context'(gb_form_context
1523	1	601,25	601,2500	1.6043%	[1523]	List! = Profiler!.getList()
0531	1	549,81	549,8100	1.4670%	[531]	print (gb_sysgui)'context'(gb_form_context
1617	1	117,41	117,4100	.3133%	[1617]	grid!.sortByColumn(4,-1)
1908	1	116,32	116,3200	.3104%	[1908]	print (gb_sysgui)'size'(grid,w,max(y-40,10))
1941	2	112,19	56,0950	.2994%	[1941]	screen\$=ctrl(gb_sysgui,0,0,gb_win.main)
0540	1	111,21	111,2100	.2967%	[540]	print (gb_sysgui)'context'(gb_form_context
0750	1	54,11	54,1100	.1444%	[750]	if len(statistics_font\$) then print (gb_sysgui)'opti
0910	31	52,21	1,6842	.1393%	[910]	if !(gb_eoj) then gb_eoj=1; for gb_wind
0546	1	48,74	48,7400	.1301%	[546]	gb_sysgui_fin\$=fin(gb_sysgui)
1526	1	46,53	46,5300	.1242%	[1526]	statistics!=new com.basis.bbj.client.datatypes.BBjVe
0792	31	40,96	1,3213	.1093%	[792]	gb_win_id\$=gb_popup_id=gb_event.x*(gb_event
0407	53	36,66	.6917	.0238%	[407]	if gb_window_context[xx_window]=xx_context then
0537	1	34,42	34,4200	.2242%	[537]	gb_sysgui_fin\$=fin(gb_sysgui)
0528	1	33,71	33,7100	.2196%	[528]	gb_sysgui_fin\$=fin(gb_sysgui)
0510	1	32,47	32,4700	.2115%	[510]	gb_sysgui_fin\$=fin(gb_sysgui)
0549	1	29,68	29,6800	.1933%	[549]	print (gb_sysgui)'context'(gb_form_context(6),'resource'
1960	14	23,28	1,6629	.1516%	[1960]	ini_statistics.colcol=dec(sendmsg(gb_sysgui.grid,38,col-1,\$
0519	1	21,24	21,2400	.1384%	[519]	gb_sysgui_fin\$=fin(gb_sysgui)

Figure 7. A sample profile without filters

Figure 8. Profile showing the effect of filtering out the 'fileopen' and user input lines

Other Enhancements

- The Performance Analyzer remembers the most recent directories in which the developer opened trace and program files, setting those directories as the defaults until otherwise changed.
- The program listing grid is sortable, and it shows the same color highlighting as the statistics grid.
- The summary grid is now sortable. This is useful for picking which program of many is responsible for the biggest portion of total execution time.
- The initial presentation of the statistics and summary grids now appear in descending order by the Percent column. This is the most commonly used view for focusing on program hot spots.
- The Performance Analyzer now shows complete program statements. The original Performance Analyzer wrote statistical data to fixed-length MKEYED records, limiting the display to only the first 156 characters of each program line.

Summary

The new Performance Analyzer will be available in BBj 6.0. A preview version is available in the BBj nightly build at www.basis.com/devtools/bbj/download_nightly.html. 



Download, compile, and run the sample Java utility program referenced in this article at www.basis.com/advantage/mag-v9n2/PA.zip.

BASIS first released the Performance Analyzer in BBj version 2.01. Read *Tuning Your Code With the Performance Analyzer* at www.basis.com/advantage/mag-v6n3/tuning.html.