Overview

In 4D, memory management is often a minor factor in a development project, if it is even considered at all. But, as with any software development environment, failure to consider memory management early in your project design, and all the way through the development cycle, can result in flawed applications; the best case would be that your 4D application merely runs slower and less efficiently than it potentially could. Just a little consideration throughout the development cycle for memory management can make the difference between a failed 4D project and a wonderfully successful, robust 4D application.

In this course, attendees will learn the basic knowledge needed to properly manage memory in their 4D projects. Considerations are made from the very beginning of project specification right through the coding, testing, and implementation when developing with the 4D RDBMS. Topics to be covered include structural design considerations, address tables, parameter management, variable scoping, recursivity, and much, much more. Many different practical techniques will be provided that can be implemented immediately in any 4D project. And, insights into the internal functionality of many of the commonly used areas of 4D will be made available so that any level of 4D programmer can begin to make their projects run more efficiently and reliably.

Source code will be provided to clearly demonstrate many of the memory management techniques presented in this session. And, code libraries will be made available to make development in 4D more practical for every 4D programmer.

An intermediate level of understanding of all areas of the 4D RDBMS is a requirement for this course.

What is the quickest and least expensive way for you to solve memory problems in your 4D applications?

Buy more RAM!

Then, go buy some more RAM!

Then, and only then, you should: buy even more RAM!

There is no single solution easier, less expensive, or more reliable. No other solution can possibly provide results as consistently positive as buying more RAM for your machines.

One stumbling block which I often hear about is the need to justify to "management" the savings afforded by purchasing RAM instead of investing in extraordinary efforts in programming. The easiest solution to this dilemma is to show the actual numbers to "management", proving that the purchase of RAM is by far the least expensive solution.

For instance, assume that you have a 4D application that requires just under 64M to run in. But, users occasionally experience out of memory warnings. And, you know that your application will run much faster if the application were given more RAM, and thereby have more cache space to utilize. Assume you have 30 users of this application. If you were to purchase just 64M more RAM for each user's machine, this would cost approximately $2400 US ($80 per RAM DIMM, for 30 users). The RAM can be installed in one evening, taking maybe 4 hours of your time.

Now, depending on your salary/consulting rates, this can turn out to be the equivalent in cost to about a week of your programming time (40 hours a week, $65 US per hour).

Ask yourself, and management: in the time span of a single week, can enough programming services be provided to make your application run as efficiently in half the amount of RAM that it currently has?

For everyone, without question, the answer has to be "No". Even "technophobic management" must be able to realize this indisputable fact.

This consideration does not even account for the increased stability that will be afforded to your application, and every other application that your users run. Everyone will be more efficient because their

machines run out of memory less often and applications have to be quit less often (to make room for other applications that need to be used).

The increase in performance afforded to 4D when using a larger internal cache instead of swapping to the disk is immeasurable. But, you must be able to take advantage of it.

First stop: always buy more RAM for your machines.

Once you have the RAM, you have to make certain that 4D can take advantage of it...

Memory Allocation to Macintosh Applications

How It Works

On a Mac, memory is allocated to an application by the OS when it is launched. There is no dynamic allocation of memory to an application by the OS "on the fly" when an application needs it. The application has settings to determine how large of a partition of memory it needs to launch and run. This memory must be allocated to the application by the OS in a single, contiguous block when the application launches. The application can not "return" any of this memory for use by the OS or any other application at any time. Only when the application quits is the memory returned for use by the OS, and thereby other applications.

If you choose the "About This Computer" menu item, under the Apple menu within the Finder, you may see a window that looks something like Figure 1, below:

figure 1: About this Macintosh window

An interesting thing to note about this particular dialog is that the largest contiguous RAM space, the "Largest Unused Block", that is available is different from the total amount of free RAM on the machine. If you do the math, subtracting the amount of space taken by all of the open applications from the total amount of built-in memory, there should be a total of 652.3M of RAM that is available to run other applications. But, instead, the single largest contiguous block of memory available is only 580.3M.

Since, on a Mac, an application must be allocated a single, contiguous block of RAM to operate in, there are some obvious limitations. If you have an application that requires 600M to run in, you will not be able to launch it on the machine depicted in Figure 1, as it currently stands. Though there may be over 650M of RAM available for running other applications, the largest single block is only 580M, smaller than the 600M your application requires.

Figure 2 depicts another view of the situation. The image was provided by a small application entitled MemMapper. This

application draws graphically the layout of RAM as it is used on a Mac. It displays also the actual application list and assignments of blocks of memory as they currently exist on any Mac. MemMapper is included on the 4D Summit 2000 CD for this class; as well, it is available from 4D Zine at http://www.4dzine.com/

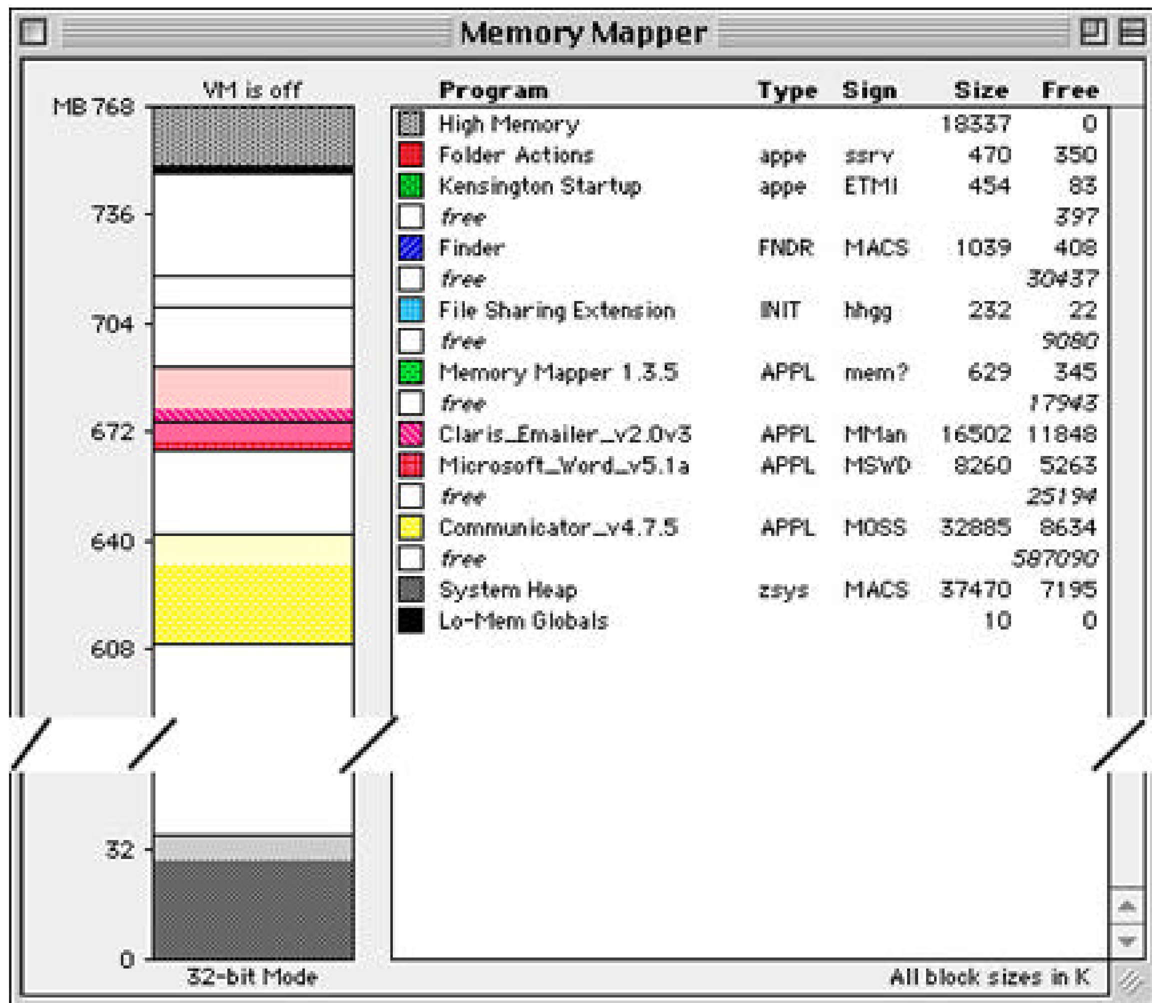

figure 2: Memory Mapper view of memory allocation on a typical Macintosh

Essentially, what is displayed in Figure 2 is fragmentation of memory, at the OS level. The total amount of RAM in the machine is not being used in the most efficient way possible. There is nothing wrong with this, though there are steps which you can take to minimize this from happening on a Mac.

When an application on a Mac is launched, the OS will try to allocate its RAM partition in the most intelligent location possible while fulfilling its minimum RAM requirements. If there is a contiguous

block of RAM available just above the System Heap or just below the High Memory allocation, then the OS will assign it to the application being launched. If a contiguous block that is large enough for the launching application is not found in either of these two locations, then the OS will assign a random block which is next to another application that is already running. The OS will purposefully not allocate a block of RAM to an application which would create a free block of RAM immediately above and below. Subsequent quitting of other applications may create such a state, but the OS will not purposefully allocate RAM in such a state.

The easiest way for you to control the fragmentation of memory within a Mac is to watch the order of application launches and quits. Be intelligent about how memory is allocated to applications on a Mac and make certain that you minimize the amount of fragmentation. If you have an email program which you always are running on your development machine, make it the first application you launch each time you restart your machine; if you use it frequently enough, consider leaving it open all of the time instead of continually launching and quitting it.

For production machines running 4D applications, consider taking a look at how the memory is used on the machine. For instance, the machine depicted in Figure 2, above, obviously has File Sharing running. If it was used in production for 4D Server, File Sharing could probably be turned off, saving the memory and the fragmentation it causes (as well as speeding the machine up; File Sharing is taxing on the CPU). And, restarting a production server just before relaunching the 4D application is the best way to return to the least fragmented state within memory on the machine.

The 'SIZE' Resource

Every application running on System 7 and above on a Macintosh should contain a size ('SIZE') resource. One of the principal functions of the 'SIZE' resource is to inform the OS about the memory size requirements for the application so that the OS can set up an appropriately sized contiguous partition for the application. The 'SIZE' resource is also used to indicate certain scheduling options to the OS, such as whether the application can accept suspend and resume events, among other low level settings.

A 'SIZE' resource consists of a 16-bit flags field followed by two 32-bit size fields. The flags field specifies operating characteristics of the application, and the size fields indicate the minimum and preferred partition sizes for the application. The minimum partition size is the actual limit below which your application will not run. The preferred partition size is the memory size at which an application can run most effectively and which the OS attempts to secure upon launching an application. If that amount of memory is unavailable, the application is placed into the largest contiguous block available, provided that it is larger than the specified minimum size.

If the amount of available memory is between the minimum and the preferred sizes, the Finder displays a dialog box asking if the user wants to run the application using the amount of memory available. If an application does not have a 'SIZE' resource, it is assigned a default partition size of 512K, which for all cases is unacceptable to 4D applications.

When defining a 'SIZE' resource, one should give it a resource ID of -1. A user can modify the preferred size in the Finder's information window for an application. If the user does alter the preferred partition size, the OS creates a new 'SIZE' resource having resource ID 0. The OS also creates a new 'SIZE' resource when the user modifies any of the other settings in the resource.

The user can also modify the minimum size in the Finder's information window for an application. If the user alters either the minimum or the preferred partition size, the OS creates two new 'SIZE' resources, one with resource ID 0 and one with resource ID 1.

At application launch time, the OS looks for a 'SIZE' resource with ID 0 for the preferred size and looks for a 'SIZE' resource with ID 1 for the minimum size; if these resources are not found, it uses the original 'SIZE' resource with ID -1.

Memory Allocation to Windows Applications

How It Works

On Windows, the allocation of memory to an application (an executable) functions much differently. With a Macintosh application, memory is allocated in a single, large, contiguous block to the application and it can not change without quitting the application. On Windows, an application is allocated a series of two (2) or more blocks of memory. The size of a block and the maximum number of blocks allowable are both configurable for each application.

4th Dimension makes it a simple matter to set the maximum number of allocatable blocks and the individual block size. Within the 4D application, under Database Properties in the Design environment, the application memory settings are available. Figure 3 shows these settings.

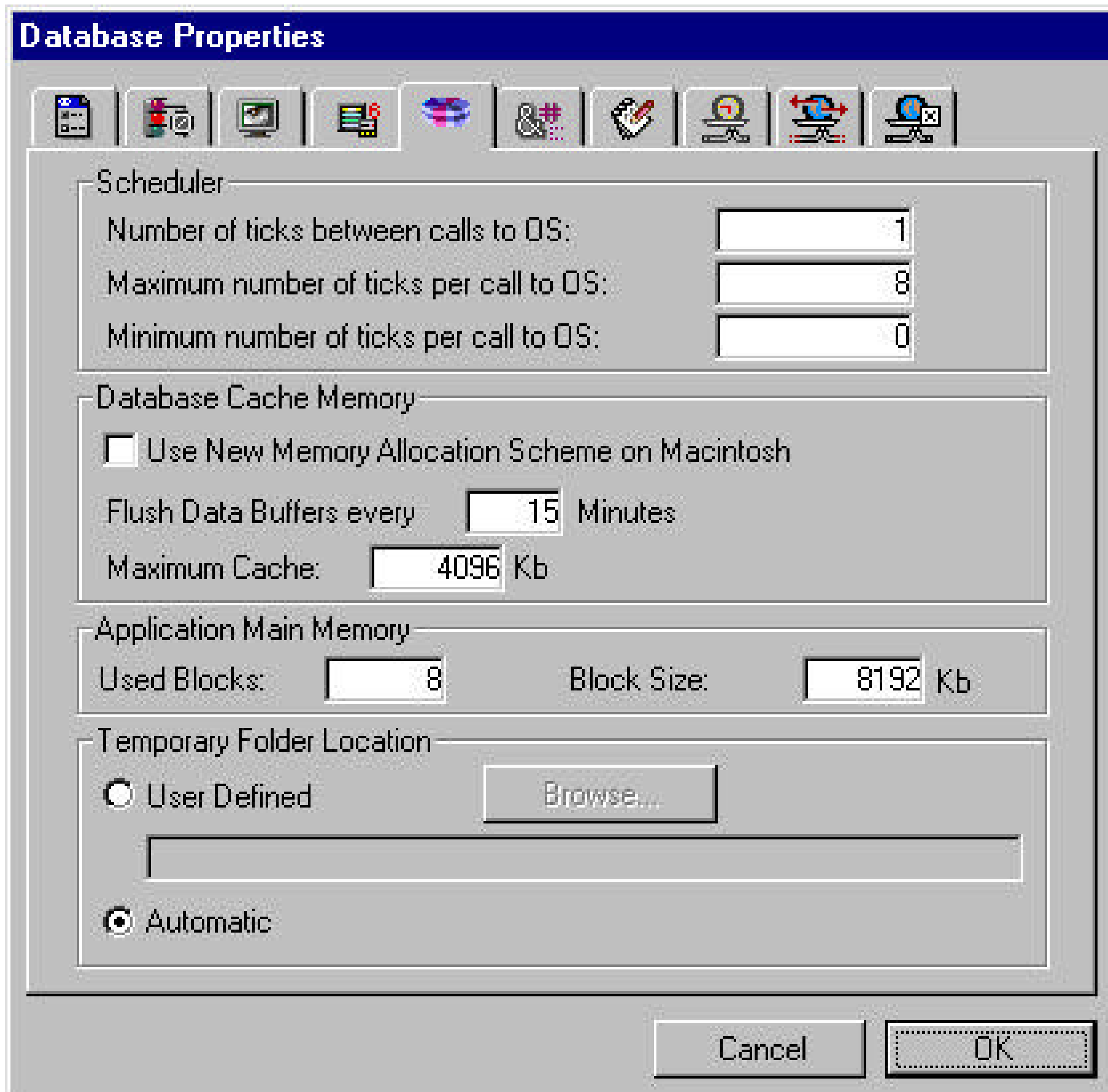

figure 3: Database Properties window within 4th Dimension

You can also use Customizer to set the memory settings for a 4D based application on Windows. Figure 4 shows the Preferences window available from within Customizer.

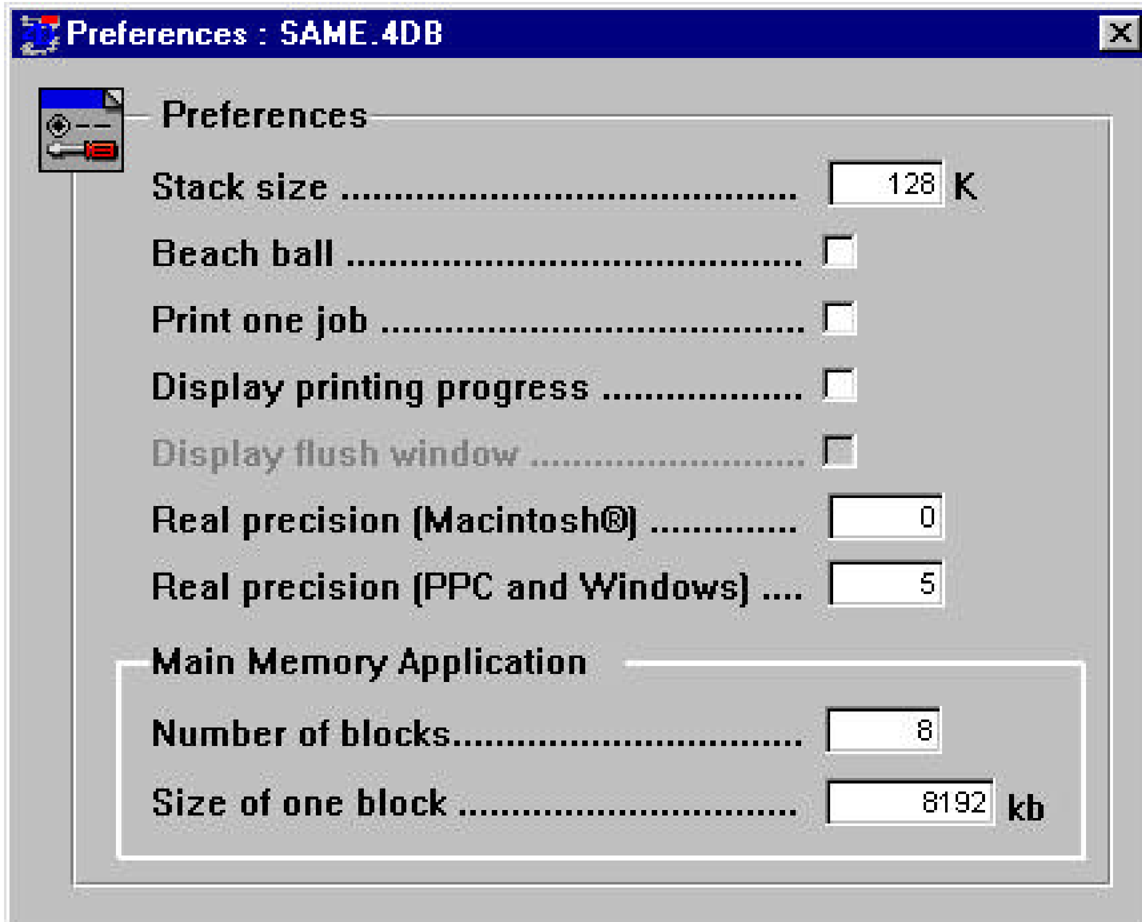

figure 4: Preferences window within Customizer

When a Windows application launches, memory will be allocated to the application by the OS on an *as needed* basis. Each block of memory assigned to the application will be of the size specified for the application. As the Windows application needs more memory, the OS will assign successive blocks of memory. This will continue up to the maximum number of block which the application is set to utilize.

In many ways, the memory allocation scheme on Windows is much more flexible and robust than that available on a Macintosh. Memory is assigned in smaller blocks in most cases to Windows applications, where a Macintosh application must have memory assigned to it at launch in one contiguous block. On Windows, the fragmentation of memory is not nearly as significant a problem. With large memory block assignments, memory fragmentation will have a much reduced impact on the total memory available on the computer for the OS and for other applications.

Current Memory Settings Location

For 4th Dimension based applications on Windows, there is a variety of places which the application main memory settings can be set. For all of these locations, only one can be the utilized by an application at any given time.

Application main memory settings can be made in any of the 4th Dimension applications. Using Customizer, the Preferences settings can be made within the applications themselves. Also, Customizer can be used to set the application main memory settings directly within a 4th Dimension structure document. The settings which Customizer makes in a structure document are the same settings which are available within the 4th Dimension Database Properties window. Structure document settings can be made with Customizer for all forms which a structure document can exist in: uncompiled (.4db), compiled (.4dc), and merged with Engine (.exe). Finally, application main memory settings can be made within the preference document for any of the 4th Dimension applications.

The rules for which application main memory setting is used at any given time is very simple. If settings have been made within the preference document for the current application, these settings will always take priority over any other settings available. If there are no application main memory settings available within the preferences document, then the application main memory settings within the actual application will be used. At no time will the application main memory settings stored in the structure document ever be used, except when the structure has been merged with Engine and there are no settings in the preference document.

The Customizer manual from 4D SA is an excellent source of information for these memory setting rules. It is a good idea to read this manual thoroughly if you do any amount of development or deployment on Windows.

Customizer

After considering the issues of application memory on your delivered platform, your next stop for handling memory issues is Customizer. Customizer allows you to manipulate more specific memory settings within the 4th Dimension product line. There are many different settings available within Customizer, and each has a very different impact on the performance and stability of your 4th Dimension projects.

It is worth mentioning that all of the values which can be set with Customizer follow the same usage rules as outlined for the application main memory settings for Windows within 4th Dimension. That being:

a) Settings in the preference document for the current application will be the first to be used, if they exist;

b) Settings in the current application will be used only if the same settings do not exist in the preference document;

c) Settings in the structure document, to a large extent, are not used at all unless the structure has been merged and is running with Engine;

Preferences, Stack Size

The Preferences window in Customizer provides an interface for setting the application main memory block size and block count. It also provides an interface for setting the default stack size (see figure 4).

The stack size parameter specifies how much memory is allocated to the stack for the User/Custom Menus process when the program is launched. Increasing this value will increase the number of methods or form call levels that can be nested successfully in this process. Each time you call a method from within another method, all passed parameters, local variables and code is the calling method are placed on the stack. Records that are pushed are also placed on the

stack. The number of nested methods that can be called successfully is limited only by the available stack space for the current process.

Whenever you receive a "stack is full" error during method execution in a process, you should immediately consider increasing the stack size for the current process. The stack size value is read in units of kilobytes (K). It is a good idea to increment the value by steps of 32. The default value when 4th Dimension ships is 32K (this is true still within 4D v6.7.x). Immediately setting this value to 128K is a safe thing to do early in your development in 4th Dimension.

There have been plenty of misconceptions about the setting of the stack size within 4th Dimension applications. A very common thread concerns the optimal setting of this value as being 640K or even 2048K (2 megabytes!). In actuality, setting the stack size this large will not even be recognized by 4th Dimension; there is a maximum stack size that 4th Dimension will allow and it is well below this "optimal" 2M value (this maximum allowed value is 1.2M). In very, very rare circumstances, there may be a need to increase the stack size to 192K. But, most definitely, there is no reasonable need to have a stack size above 256K. It is a good rule of thumb to use 128K as your stack size value. Any reasonably developed 4th Dimension based system will find this amount of stack space to be more than reasonable.

The same considerations which are used for setting the default stack size with Customizer should be used when you set the stack size for new processes in your code. When using any of the 4th Dimension commands that launch new processes (e.g. New process, Execute on server, etc.).

Stacks

There are many other processes which 4th Dimension can spawn beyond just the User/Custom Menus process. This includes an On Event Call process, Web Server processes, and 4D Backup processes, among others (see figure 5). The Stacks window in Customizer allows you to set the process stack size for these processes.

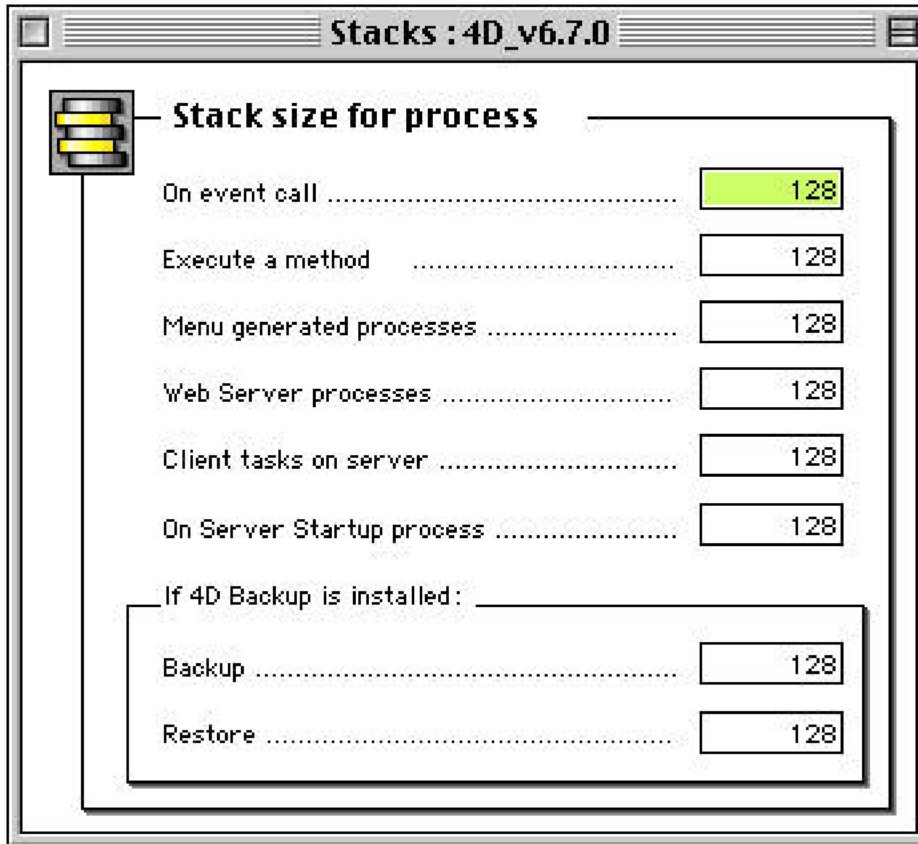

figure 5, process specific stack settings available in Customizer

The same considerations for stack size as used for the default stack size should be used for setting all of these individual stack sizes. Code recursion is obviously the largest effective variable in determining how large the stack sizes should be set to.

Properties, Database Cache Memory

In the Properties window of Customizer, a section entitled Database Cache Memory exists. Figure 6 shows the Properties window as available within Customizer.

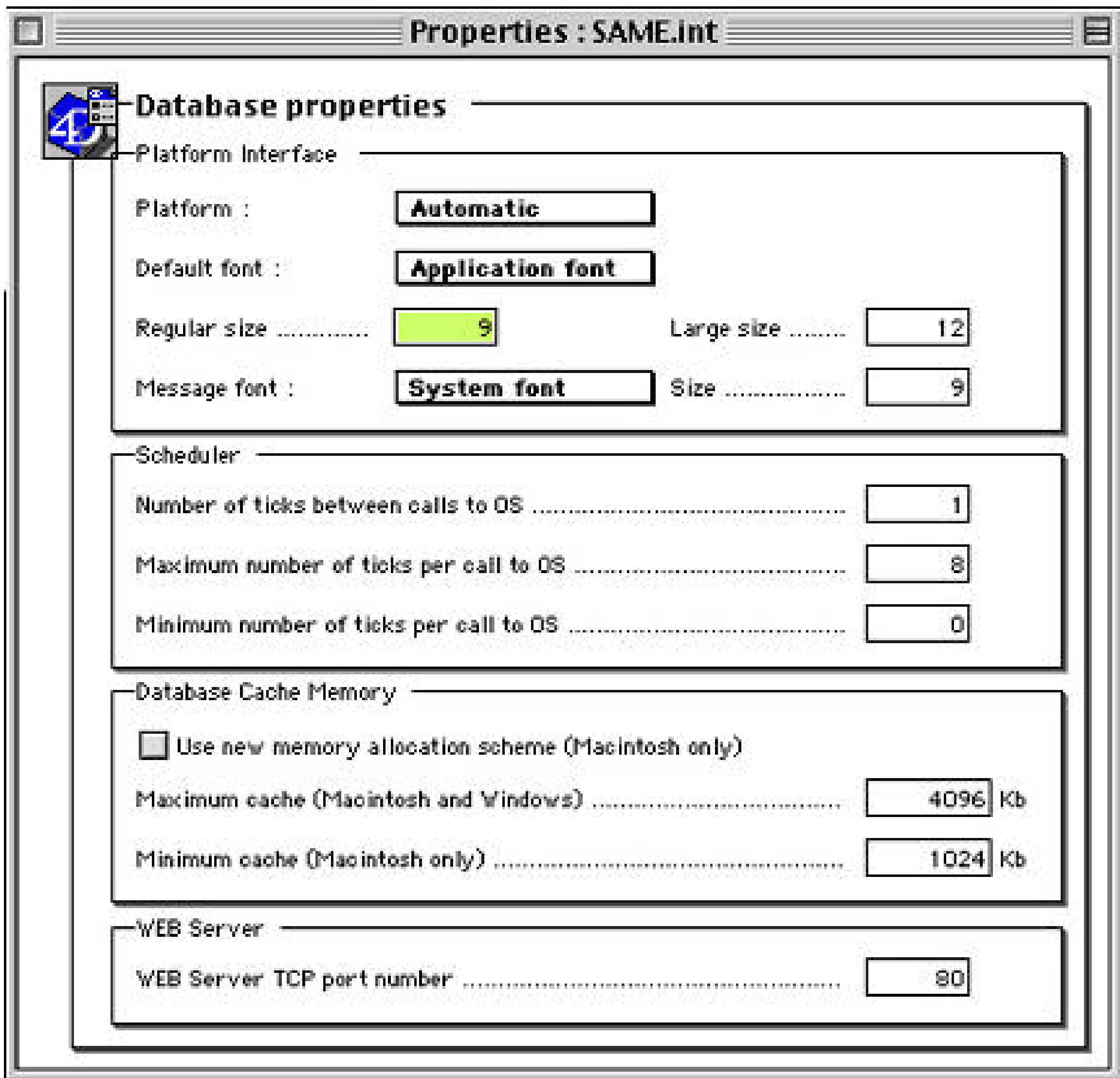

figure 6, database cache memory settings available in Customizer

The same value settings exist in the 4th Dimension Database Properties window within the Design environment. Figure 7 shows the values as available within 4th Dimension.

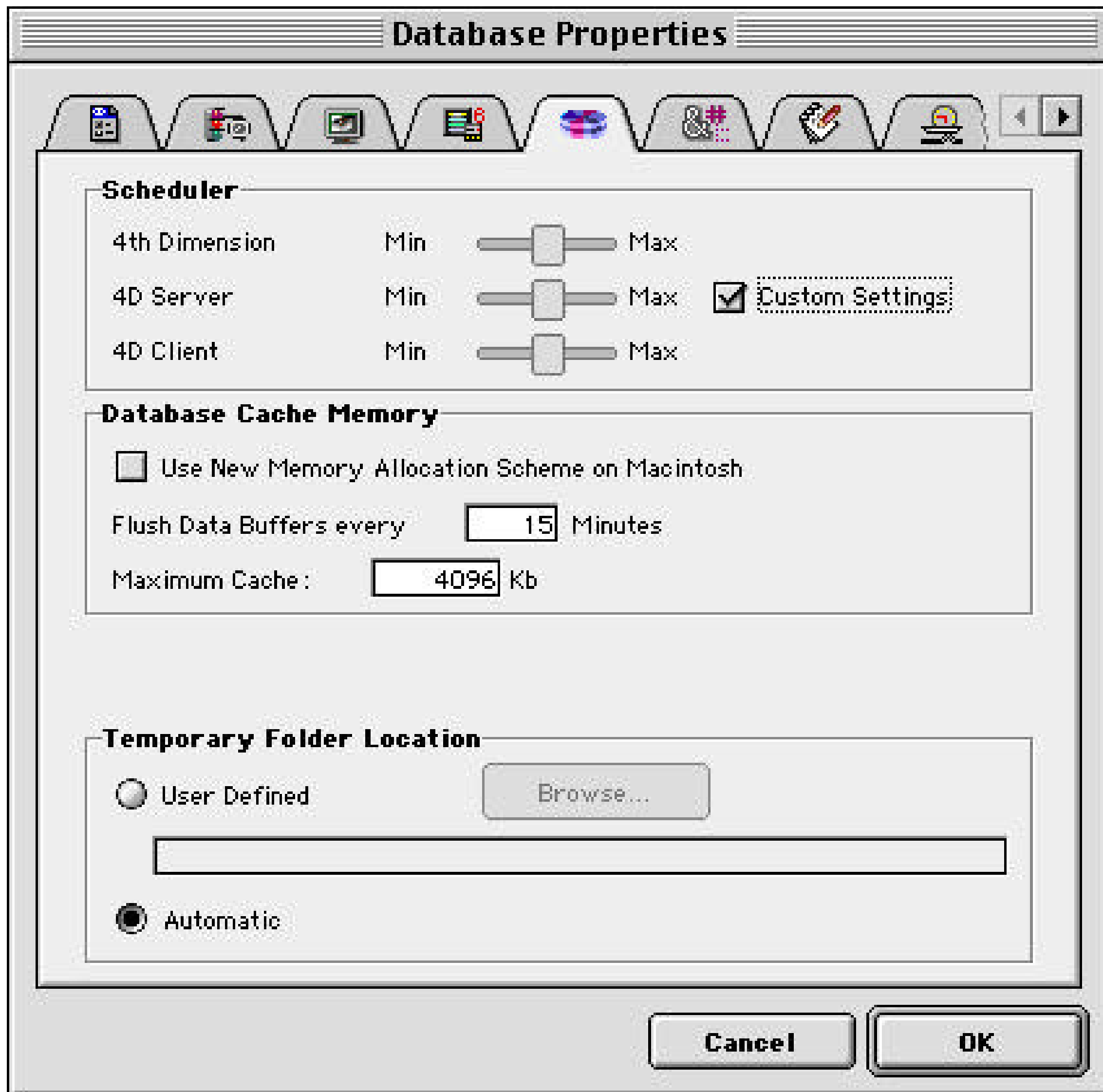

figure 7, database cache memory settings available in Database Properties window within 4th Dimension

If the flag for New Memory Allocation Scheme is set, then the minimum and maximum cache values set within this area will be used to control the database cache size used by 4th Dimension. Setting the New Memory Allocation Scheme flag will tell 4th Dimension to use memory available from the OS, memory not specifically allocated to the 4th Dimension application, for use as the database cache.

The database cache is a block of memory which is used to store frequently accessed records. Property setting of an optimized database cache can significantly improve the performance of a 4th Dimension application. By caching records which are frequently

accessed within memory, 4th Dimension does not have to repeatedly access the hard disk so much to retrieve data. Since the hard disk is *significantly* slower than memory, this translate into a very noticeable performance boost within your 4th Dimension applications.

If the New Memory Allocation Scheme flag is not set, then the cache will be created within the memory allocated to the 4th Dimension application, instead of memory managed by the OS. This can increase the amount of memory which your 4th Dimension application requires to operate in efficiently. But, the advantage is that your database cache can be protected from inadvertent corruption by other applications which run on the same machine.

From experience, there has never been a noticeable improvement by using the New Memory Allocation Scheme within 4th Dimension. Merely increasing the amount of memory that the 4th Dimension application uses for itself and then disabling the New Memory Allocation Scheme flag (thereby forcing the database cache to be part of the application assigned memory block) has proved to be more reliable for almost all developers and administrators.

When the New Memory Allocation Scheme is not enabled, the other settings within the Database Cache Memory are no longer used. Instead, the settings within the Memory window of Customizer will be used to determine the size of the database cache within the 4th Dimension application memory block. The Memory window in Customizer is detailed below.

Whenever the New Memory Allocation Scheme is enabled, you must restart the machine before it will be used by 4th Dimension.

Memory, Cache Memory, Kernel Memory, & Minimum Flushed

Figure 8 shows the Memory window in Customizer.

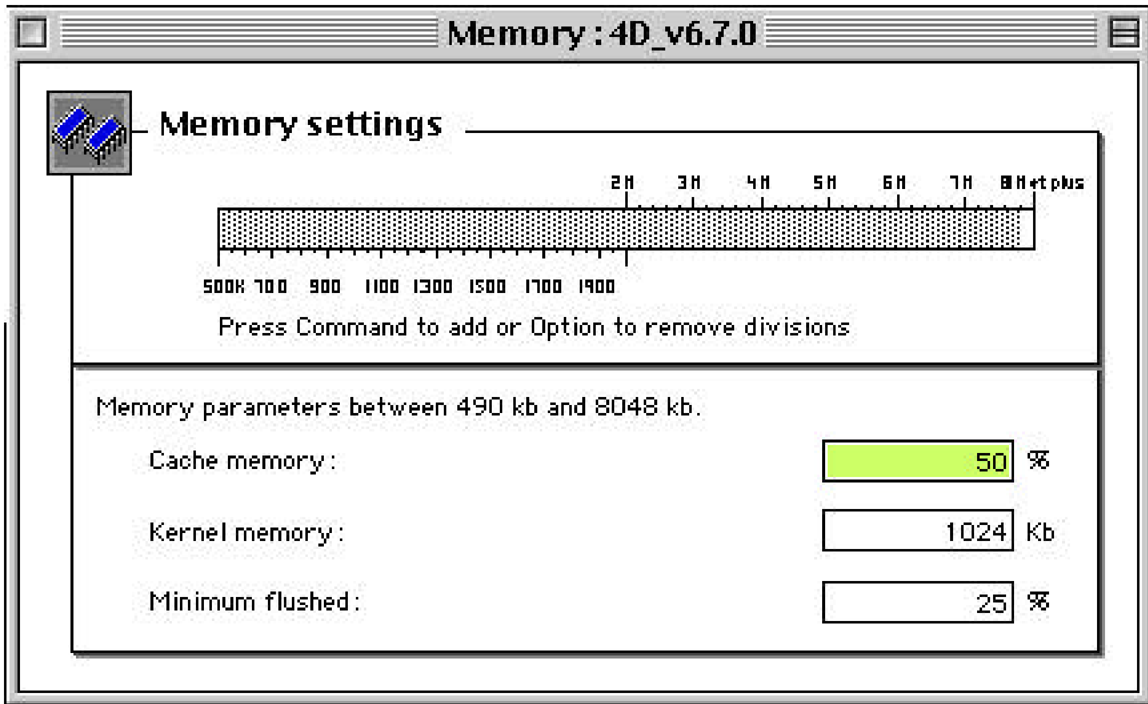

figure 8, Memory window in Customizer

The first item within the Memory window to note is the Kernel Memory value. The Kernel Memory value is the maximum amount of memory reserved for use by 4th Dimension's internal routines. Increasing this value can improve the performance of 4th Dimension application as it will reduce the amount of kernel segment swapping required within the application's core code. By default, this value is set to 512K. Setting it below this amount will reduce performance severely. Increasing it to 1024K or above will provide a reasonable level of performance increase, as it will allow more of the core code within 4th Dimension to stay loaded at any given time.

To understand the Cache Memory value, it is necessary to understand how the memory within a 4th Dimension application is allocated when launched. The following is a very simplistic explanation of how memory is assigned within 4th Dimension internally:

For however much memory is assigned to the 4th Dimension application at the OS level, the first thing done is a block of this memory is set aside by 4th Dimension and reserved for use by the internal core code. This block is approximately the size of the Kernel Memory setting. The remaining memory available to the 4th Dimension application is available for as the database cache (when the New Memory Allocation Scheme flag is not set) and heap space

for your forms, menus, form images, and methods. The amount of space reserved for use as the database cache is a percentage of the remaining memory space available after the Kernel Memory assignment has been made.

For instance, assuming the settings which exist in figure 8 and an application allocation of 30M to the 4th Dimension application, then the database cache would be approximately 14.5M; that is, 30M, less 1024K for the Kernel Memory, with 50% of the remaining being assigned as the database cache.

Depending on the functionality available within your application, you should be able to make a fairly knowledgeable guess as to percentage to assign to the database cache. If your application manipulates very little data and instead provides a great deal of functionality (e.g. a word processing application), then you do not need much of a database cache. If your application is constantly manipulating many pieces of data, many records in the data file (e.g. an accounting application), a larger database cache would probably provide much better performance.

The Runtime Explorer within 4th Dimension provides a means to determine how well your cache is being utilized. Under the Watch tab in the Runtime Explorer, within the Cache Statistics disclosure, there is an item entitled Global Hit percentage. This value indicates what percentage of 4th Dimension record access attempts are done from the database cache. Figure 9 shows where this value can be seen in the 4th Dimension Runtime Explorer.

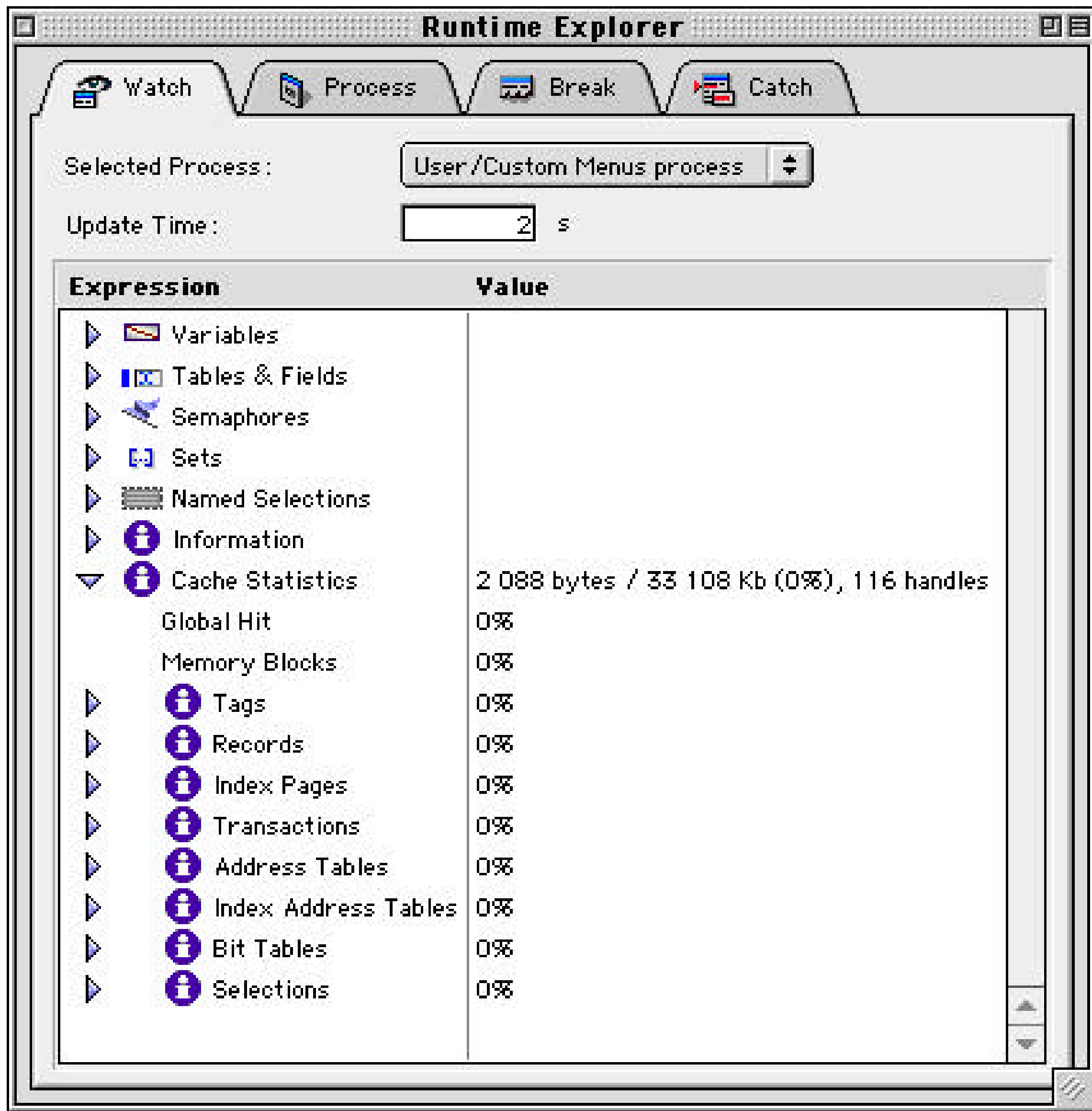

figure 9, Runtime Explorer showing the global hit value for the database cache

The same value can be seen as a thermometer within the 4D Server main window. Figure 10 shows this window, with the cache hit ratio thermometer towards the upper right hand corner of the window.

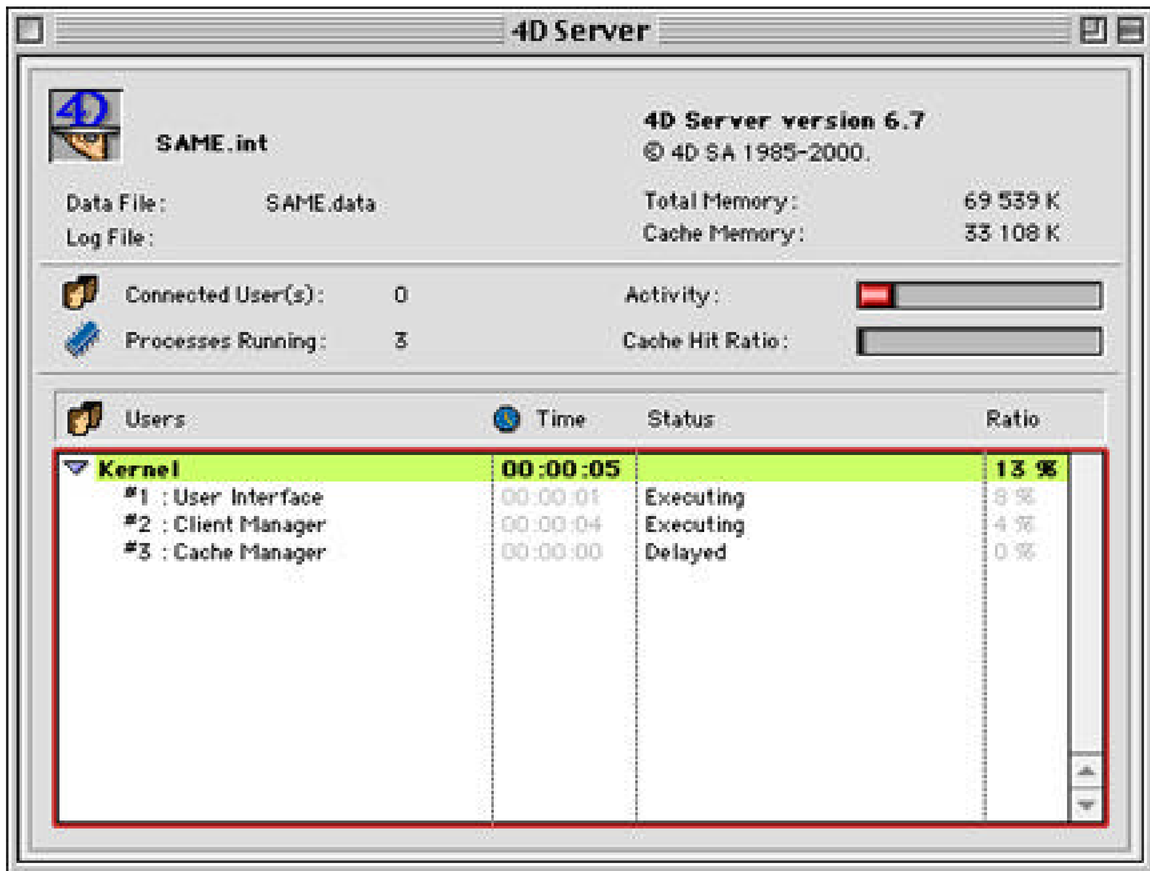

figure 10, the 4D Server main window

Using the cache hit ratio value can help you determine if adjusting your Cache Memory value within Customizer will improve the performance of your application. If the cache hit ratio is very low, especially during an operation which requires access to many records, then it might be a good a idea to increase the database cache size.

Be careful, though, as your cache hit ratio might be low, at times, and it could just as easily be an indication that your database cache is too large. For instance, if you know that very few records are ever really accessed within your 4th Dimension application, and your cache hit ratio is fairly low, then it very well good be that the database cache is too large.

Finally, the Minimum Flushed value is the minimum amount of data flushed from the database cache when it is full. Decreasing this amount causes more frequent, albeit shorter, interruptions to flush the database cache. Increasing this value causes less frequent, albeit longer, interruptions to flush the database cache. The correct

value to set here is dependent on the functionality and usage of your application. Enabling the Flush Window to be displayed will help you determine what percentage value will work optimally for normal usage of your 4th Dimension application.

Variables

Fixed Length Data Types

Most of the data types which are available in 4th Dimension can be classified as fixed length data types. The complete list of fixed length data types is as follows:

boolean
date
integer
longint
pointer
real
string
time

Fixed length data types are always of the same size. Variables of a fixed length data type will always occupy the same amount of memory, regardless of the value.

In general, the following table can be used as a reference for how many bytes each of the fixed length data types will occupy:

Type	Compiled
boolean	4
date	6
integer	4
longint	4
pointer	16
real	10
string	((declared length+2)\2)*2
time	4

Fixed length data types are distinguishable from other data types because the actual values which are stored in the variables are stored directly. This means, for local variables, the values are stored directly in the variable table for the method which the process is running in; for process variables, the values are stored in the process variable table for each process; for interprocess variables, the value is stored directly in the interprocess variable table.

There is no advantage to be gained by zeroing out variables of a fixed length data type. The memory for storing the value will still be allocated and is not recovered, even for strings which are set to an empty value (i.e. "").

Fixed length data types often have a much larger impact on the size of a variable table than variable length data types. Specifically, string data types can quickly make a variable table grow to be larger. Also, fixed length data types can force a stack to fill very quickly. The pushing of records containing many fixed length data types, stacking called methods which use many fixed length data types internally, and parameters from stacked methods which are fixed length data types will all be stored on the process stack. If there are many strings and reals, this can add up quickly and cause memory problems in your code.

It is a good idea to always make certain that you are explicitly declaring your data types. The default data types applied within Compiler is real for both buttons and numbers. These default settings can and should be changed to longints. Beyond this, though, you should be able to compile your 4th Dimension applications with the All Variables are Typed option, as you should be managing all of your variable types explicitly so that your stacks do not fill up unnecessarily.

Variable Length Data Types

There are a few data types in 4th Dimension which are not fixed length variables; quite often, variables or fields of this type are called variable length data types. The complete list of variable length data types is as follows:

array
blob
graph
picture
text

A variable length data type is distinguishable from a fixed length data type because the actual value within the variable is not stored in a variable address table of any kind. Instead, a reference is stored in the variable address table, and the actual value is stored in the application heap. This schema makes it much easier, and efficient, for 4th Dimension to manage the changing contents of these variables. Fixed storage space can be assigned then by 4th Dimension for the variable tables it uses, and the actual values for these variable length data types can be dynamically managed within the application heap.

Variable length data types do occupy space within a variable address table. For each of the variable length data types, 4 bytes of space are used within a variable address table. This may seem like a lot, but is actually very little when you consider that they reference values which can be almost limitless in size.

For arrays, the storage requirements for the values of the elements will actually vary upon the different data types allowed for arrays. The following table will help you calculate how much memory is required for different types of arrays:

Array Type	**Memory Usage Formula (EC is the element count)**
Boolean	(31+EC)\8
Date	(1+EC)*6
Integer	(1+EC)*2
Longint	(1+EC)*4
Picture	(1+EC)*4 + Sum of the size of each picture
Pointer	(1+EC)*16
Real	(1+EC)*10
String	(1+EC)*(((declared length+2)\2)*2)
Text	(1+EC)*6 + Sum of the size of each text
2D	(1+EC)*12 + Sum of size

of each array

Because of this architecture, though, there are certain considerations which you must take in your coding to make sure you are utilizing the 4th Dimension in the most efficient manner possible. As well, the exact nature of your code can have a significant impact on the amount of fragmentation which occurs within the application heap.

Each time a variable length data typed value is changed, it may result in fragmentation in the heap. Increasing the size of a variable length data type value may require that a BlockMove of memory be performed, as the value must be stored in a single contiguous block of memory. Decreasing the size of a variable length data type value will result in holes in your application heap which may be very difficult to recover without extra work by the CPU; often, holes in the application heap are recoverable only by the application compacting the memory space, which requires great deal of processor time both juggling the memory, updating application handles, and recovering fragmented memory space.

For instance, the following routine is a horrible way to write a routine:

```
C_LONGINT($i)
C_TEXT($xFlags)
$xFlags := ""
For ($i;1;30000;1)
   If (myArray{$i} = 0)
      $xFlags := $xFlags + "0"
   Else
      $xFlags := $xFlags + "1"
   End if
End for
```

This snippet of code requires the text variable to be each pass through the loop. This can result in many block memory moves internally within 4th Dimension, something which is vary inefficient.

A much better way to achieve the exact same functionality would be:

```
C_LONGINT($i)
```

```
C_TEXT($xFlags)
$xFlags := 30000 * "0"
For ($i;1;30000;1)
  If (myArray{$i} = 1)
    $xFlags[[$i]] := "1"
  End if
End for
```

With this alternative approach, the memory needed to store the complete text variable is set aside at the beginning of the routine. Then, byte by byte the values within the text variable are assigned. There is absolutely no memory assignments or movement needed within the long loop, making the routine much more efficient.

The same principles apply to the bytes within a BLOB and the elements within an array. Assigning a maximum value to these beforehand to reduce the number of memory block assignments and memory movement can significantly increase the speed and efficiency of your 4th Dimension code as well as reduce the amount of memory fragmentation within the application heap.

Even when you have a situation in which you do not know the exact resulting size of a variable length data type, there is still a better alternative than successively increasing the size. Instead, you can build a system wherein a set "block size" is used to increase the size of a variable length data type in larger chunks; then, when your routine is completed, you can reduce the size of the variable length data type value to discard the extra unused space. The following example illustrates this point clearly:

```
C_LONGINT($i)
C_BLOB($zFlags)
C_LONGINT($iBlockSize)
C_LONGINT($iCurrentByte)
SET BLOB SIZE ($zFlags;0)
$iBlockSize := 100
$iCurrentByte := -1
For ($i;1;Size of Array(myArray);1)
  If (myArray{$i} = 1)
    $iCurrentByte := $iCurrentByte+1
    If (BLOB Size($zFlags) = $iCurrentByte)
      SET BLOB SIZE($zFlags;$iCurrentByte+
                 $iBlockSize-1;0x0000)
```

```
      End if
      $zFlags{$iCurrentByte} := ASCII("1")
    End if
End for
If (BLOB Size($zFlags) > ($iCurrentByte+1))
    SET BLOB SIZE($zFlags;$iCurrentByte+1)
End if
```

You can see that the flags BLOB is increased in 100 byte increments as needed. This is continued until the loop is completed. Throughout the loop, the number of used bytes within the flags BLOB is maintained. After the loop is completed, the number of used bytes is compared to the size of the flags BLOB and any extra bytes within the variable length flags BLOB are removed.

This format for coding within 4th Dimension requires a certain level of consideration for how memory is handled internally. Optimizing the structure of your code segments can lead to significant improvements in speed and reduced memory fragmentation. These considerations should always be kept in mind while you are coding in 4th Dimension.

Cleanliness is Next to...

There are variety of objects within 4th Dimension which should be managed tightly. These objects, if not managed properly, can easily occupy memory space unnecessarily. Some of these objects, if managed incorrectly, can even be a source for memory leaks within your 4th Dimension applications (a memory leak occurs when memory has been allocated for a specific purpose, was never properly deallocated, and the memory space is in a state which is can not be recovered).

Variables

Variable length data types should be cleared out when you are done using them. Since BLOBs, pictures, text variables, etc., all occupy space in the application heap, you should recover this space as soon as you are done using it. This will allow for the large blocks of memory which these objects occupy to be reused efficiently by other

objects within 4th Dimension. This will help decrease, too, the amount of memory fragmentation, as the heap will be compacted more tightly and more quickly with fewer objects in it.

For instance, assume you have a routine which imports a large tab delimited document into a BLOB for parsing and storage in records within 4th Dimension. Once you are done parsing through the whole BLOB and all of your records have been created and saved, you should make certain you set the size of the BLOB to zero. This will free the memory space for subsequent usage.

There is a 4th Dimension v6.7.x component entitled "BASh" included with this course. It is available on the 4D Summit 2000 CD in both Macintosh and Windows formats. This component is also available on the 4D Zine web site at:

http://www.4dzine.com/4dz.acgi$freeware_show_00000254

The BASh component includes a method entitled NULL_Set_Variables. A variable number of referenced parameters can be passed to this method. All of the referenced variables passed to the method will be cleared out; BLOBs and pictures will be reduced to zero bytes, text values will set to empty ("").

Another routine in the BASh component entitled ARR_Clear will do the same for arrays. For a single referenced array which is passed to this routine, the size of the referenced array will be set down to zero.

These routines make it very simple to clear out your variable length data typed variables quickly and efficiently. Feel free to use the routines in this component in all of your 4th Dimension applications.

Hierarchical Lists

Every time that you work with hierarchical lists in 4th Dimension, you are often actually working with a hierarchical list reference. This hierarchical list reference is just a unique identifier for the hierarchical list object which is stored separately in the application heap.

It is important to always call CLEAR LIST when you are done working with an hierarchical list. The CLEAR LIST command takes a single parameter, the hierarchical list reference. CLEAR LIST will release the memory which the list actually occupies within the application heap so that other 4th Dimension objects can use the memory. Once the hierarchical list has been cleared, your hierarchical list reference should no longer be used.

If you do not consistently clear hierarchical lists which are used in your 4th Dimension application, you will have a memory leak. Basically, you will have a situation in which an area of the application heap can become locked and can never be unlocked. Eventually, this will lead to your 4th Dimension application running out of memory.

It is a good idea to make certain that any method which can create an hierarchical list also contains the code to clear the hierarchical list. This way, you can clearly and simply make certain that you are never leaking memory because of poor hierarchical list management.

NOTE: coding techniques of this sort, and many others, will be discussed in the 4D Summit 2000 class entitled ***Styles Vary***.

A common mistake made with hierarchical lists is to leave the list references dangling. For instance, it may seem natural to write the following method:

```
C_LONGINT($iListRef)
C_LONGINT($iUserSelected)
$iListRef := Load list ("My BS List")
  {let the user select from the list}
$iUserSelected := Selected list item ($iListRef)
  {store what the user selected}
`end of method
```

The problem with this method is that the hierarchical list was never cleared. It is easy to make the mistake of assuming that just because the hierarchical list reference was stored in a local variable, allowing the method to end would clear the list. In actuality, the hierarchical list reference is clear from memory, but the hierarchical list that was referenced is still stored in the application heap, the memory which the hierarchical list is occupying is still locked, and you now have a memory leak in your

4th Dimension application. Merely calling CLEAR LIST at the end of the method would have prevented the dangling list reference and the subsequent memory leak.

Selections

Each record which is in a selection for a table will occupy 4 bytes of space. This may not seem like a lot of memory. But, consider, that it is not uncommon to have selections which contain tens of thousands of records. And, within 4th Dimension, you can have a different selection on every table. And, for each process in your 4th Dimension application, you can maintain different selections on the same table. Worst of all, selections are maintained concurrently on 4D Server in a client/server environment.

Clearly, this is a place in which memory can be needlessly wasted. It is good practice to always make certain that your selections are cleared after you are done with them. The REDUCE SELECTION command is ideal for this. You can call REDUCE SELECTION on any table and reduce the number of records in the selection to zero. This effectively clears the whole selection from memory, freeing the memory for use by other 4th Dimension objects.

Properly clearing selections in a client/server in 4th Dimension can free up a tremendous amount of RAM on 4D Server. Often, when a 4th Dimension based client/server application is using too much memory on the 4D Server machine, it is because of poor management of selections.

Again, just like with hierarchical lists, it is a good practice to make certain that you clear your selection in the same method which it was originally set.

Named Selections

Named selections are merely selections which are referenced by name. A named selection can be maintained separately from the

current selection on the target table. The name of the named selection is the only identifier for the named selection object.

Consideration should be paid to whether you should be using the CUT NAMED SELECTION or COPY NAMED SELECTION to create a named selection. If you need to continue working with the current selection as it currently exists, you should use COPY NAMED SELECTION to create your named selection; this will physically make a copy of the named selection in memory. But, if you do not need to continue using the current selection, use CUT NAMED SELECTION to create your named selection; this will not make a block move of memory to create your named selection, instead only a couple of handles maintained internally will be updated; as well, you will not be using any extra memory to create your named selection.

A named selection can be created at any time. Depending on the way which a named selection was created, you may need to clear the named selection yourself. If a named selection is created with the command CUT NAMED SELECTION, then subsequently calling either USE NAMED SELECTION or CLEAR NAMED SELECTION will release the named selection from memory. If a named selection is created with the command COPY NAMED SELECTION or CREATE SELECTION FROM ARRAY, the only way to release the named selection from memory is to call CLEAR NAMED SELECTION.

Just like hierarchical lists, named selections are a common source of memory leaks in 4th Dimension. But, again, by managing named selections properly, memory is managed properly, as well. Including the clearing of named selection within the same method which they can be created is a simple means to prevent memory leaks involving named selections. And, pay attention to dangling named selection references.

Sets

Sets are similar to named selection in that they are objects referenced solely by their name. But, the differences that do exist with referencing and managing sets actually make them easier to deal with than named selections.

CLEAR SET is the only command that will release a set object from memory. Wherever a set can be created in your 4th Dimension code, just make certain that the set is subsequently released with the command CLEAR SET. This will prevent memory leaks where sets are involved. And, again, pay attention to dangling set references.

Documents, Communication Streams, Binds, etc.

There are plenty of other objects which require proper management. It would be impossible to list them all, though. Here are a few of the more commons ones, though:

a) Documents; if you create a new document or open a document, make sure you subsequently close it;

b) Communication Streams; any form of communication stream involves an opening of the stream (e.g. Open Channel, TCP_Open, ODBC_Connect, etc.); make sure you subsequently close the stream;

c) Binds; when using almost any of the connectivity packages, data binds are a common means to map data transfer between the local and the remote data source; creating these binds often takes a considerable amount of system resources, application resources, and memory; make certain that you subsequently release the binds when you are done with them.

Miscellaneous

Resources

Any resource which you create to be used should always have the purgeable flag set. The purgeable flag in a resource is an indicator to the Resource Manager in the OS that the memory occupied in the application cache by your resource can be purged if needed. If the purgeable flag is not set, once your resource is loaded, it will have

to remain in the application cache and can not be released. This will reduce the amount of space available for other objects and code used internally by 4th Dimension.

Processes in client/server

In a client/server environment, processes can have a significant impact on memory, especially on the 4D Server machine. For each global process that exists on a client machine, a corresponding process is created on the server machine. Where it may not seem to bad to have four or five processes on a client machine, consider that this many process will exist on the server for every single client connected.

So, if you have thirty (30) clients connected to the same server, and each client has only four (4) processes open, this will be 120 processes on the server machine. This can be very stressful on the server, not just in terms of processing speed within the system but also with respect to memory.

Be judicious in your design and coding decisions within your 4th Dimension applications. Within client/server systems, consider not just how memory is affected on the client machine but also how it is affected on the server. In many instances, small problems in a 4th Dimension single user application are amplified tremendously in a client/server environment, especially on the poor server machine.

Consider making some of your global process local processes if you can. Or, limit the number of actual global processes a client can create. Make certain you utilize a good memory management design throughout your 4th Dimension application development, as it will show in a client/server environment.

Parameters

When passing parameters to methods, serious consideration should be given to whether you need to pass a copy of the value or the original value. Passing a copy of the value will result in a larger use

of memory within your 4th Dimension applications. Oppositely, if you pass a reference (pointer) to the original value, there is very little extra memory required.

The following line of code will call a method somewhat inefficiently:

```
xText := MyCustomReplaceMethod (xText)
```

With this format, the value within xText is copied inside of the method MyCustomReplaceMethod. Then, the resulting value is again copied as the return value to the method. This is very inefficient and slow.

Instead, you could restructure your method slightly and make a call like:

```
MyCustomReplaceMethod (->xText)
```

This would be much more memory efficient, as the original value would be operated on directly within the method MyCustomReplaceMethod.

Where many 4th Dimension programmers have a problem with such a format for methods though is the apparent lack of pointers to local variables. For instance, the following line of code is not allowed in 4th Dimension:

```
MyCustomReplaceMethod (->$xText)
```

It may seem then that the most efficient way around this would be instead go back to the original method calling format. Or, as another possibility, you could just use a process level variable in place of the local variable $xText, even though this variable is not used in any other method. But, using a process level variable for such a situation would be immensely wasteful. There would eventually be too many locations in your code which use process level variables in place of local variables, and your variable address tables would become too large.

A better option is the use a few of the routines which are available within the BASh component (on the 4D Summit 2000 CD and 4D Zine). Consider the following piece of code instead:

```
C_POINTER($pMyText)
$pMyText := DSS_Get_Variable_by_Type (Is Text)
  {do some stuff}
MyCustomReplaceMethod ($pMyText)
  {do some other stuff}
DSS_Return_Variable ($pMyText)
  `end of method
```

The method DSS_Get_Variable_by_Type will return a reference to a variable of any type specified. This variable can be used just like any other variable, passed to subroutines, used for storing values within a process, or even to share data across processes. The reference to the variable is the key to identifying the shared variable and manipulating its contents. The DSS_Get_Variable_by_Type method can be called again and again to get more variables of any type, even multiple variables of the same type. Each variable returned by DSS_Get_Variable_by_Type will be distinct from each other and no two calls to DSS_Get_Variable_by_Type will return the same variable until a variable is returned to the pool of available variables. To return a variable to the pool of available variables, merely call the method DSS_Return_Variable with the variable reference.

These DSS, Dynamic Stack Space, methods can provide an immense degree of memory savings, variable address table savings, and efficiency within your 4th Dimension application. Methods can be writing to handle parameters in the most efficient manner possible. And, variable space can be reduced by using the temporary variables which are available through the DSS calls.

Consider also what level of efficiency and memory savings can be had by using the DSS methods when interacting with many of the popular plug-in APIs. The DSS methods support all data types, including arrays. The only data types which are not supported are two dimensional arrays and graphs (this is true as of v1.4.0 of the BASh component; this may change in future versions as the need arises).

Please feel free to use the DSS libraries, and the other routines, which are available in the BASh component. They are free for you to use in all of your 4th Dimension projects.

Useful Documentation

Allocating more than 128 MB to 4D Server , by Ingvar Josefsson

http://www.4dzine.com/4dz.acgi$freeware_show_00000011

Checking Memory with the Runtime Explorer and the Debugger , by Tim Tonooka, composed 08/06/1999

http://www.acius.com/tech%5Ftips/tips99%2D107.html

Keeping the Memory Clean - Part 1 , by Walt Nelson

http://www.acius.com/acidoc/cmu/cmu79967.htm

Keeping the Memory Clean - Part 2 , by Walt Nelson

http://www.acius.com/acidoc/cmu/cmu79966.htm

Kernel and cache memory settings in 4D and 4D Server , by Kevin Callahan and Jim Goshorn

http://www.4dzine.com/4dz.acgi$freeware_show_00000123

"Kernel memory" setting in Customizer Plus , by Tim Tonooka, composed 08/13/1999

http://www.acius.com/tech%5Ftips/tips99%2D111.html

"Memory Load" item in the Runtime Explorer, by Tim Tonooka, composed 4/21/2000

http://www.acius.com/tech%5Ftips/tips00%2D289.html

Memory optimization in 4D, by Jens Blomster

http://www.4dzine.com/4dz.acgi$freeware_show_00000026

Memory usage of 4D and 4D Server versus RAM disks, by Ingvar Josefsson

http://www.4dzine.com/4dz.acgi$freeware_show_00000120

Setting RAM partitition on Macintosh, by Steven G. Willis

http://www.4dzine.com/4dz.acgi$freeware_show_00000111

Setting the Stack Size for Web Processes, by Eric Saltzen, composed 8/25/2000

http://www.acius.com/tech%5Ftips/tips00%2D397.html

What is the "Cache Hit Ratio"?, by Tim Tonooka, composed 4/28/2000

http://www.acius.com/tech%5Ftips/tips00%2D293.html

Useful Software

BASh v1.4.0 , by Steven G. Willis

http://www.4dzine.com/4dz.acgi$freeware_show_00000254

MemMapper v1.3.5 (aka Memory Mapper)

ftp://ftp.4dzine.com/applications/

Special Thanks

Bryan Green, bryan@greensw.com

I must express a special thanks to Mr. Bryan Green of Green Software, Inc. Mr. Green provided an invaluable level of insight regarding the gathering of information for this class and the notes. Mr. Green's knowledge of the internal workings of 4th Dimension, especially as it relates to memory management and memory allocation, is clearly second to none in the developer community. No degree of thanks would be sufficient for his knowledge and expertise in assisting with this class.