VisualWorks Optimization

This module demonstrates how to overcome the performance bottlenecks in your VisualWorks application.

Performance problems are usually due to a bad choice of algorithm, or poor implementation of the algorithm. This module will show you how to find where your application is spending its time and provide some tips and techniques to improve its performance.

1. Motivation

Is there a performance problem in VisualWorks? Not usually: modern implementations are very fast, and often quite fast enough for practical purposes.

If performance is really a problem in your case, your should *profile* your application to determine which parts are really taking the time, and perhaps *time* how long those parts actually take. Remember the "90–10" rule:

In most programs, 90% of the time is spent in 10% of the code.

When this is the case, it's pointless improving the performance of the infrequently used 90% of the program – even making it infinitely fast would only improve overall performance by 10%!

Therefore you should first identify the bottlenecks, and then attack those. Performance problems are usually due to a bad choice of algorithm, or poor implementation of the algorithm. This course shows you how to find *where* your application is spending its time. It also gives some clues as to how you might improve performance through a variety of "tricks", but it is beyond the scope of this course to suggest how you should select appropriate algorithms.

Recommended textbooks in this area are by Jon Bentley:

"Programming Pearls", Addison–Wesley, 0-201-10331-1, 1986.
"More Programming Pearls", Addison–Wesley, 0-201-11889-0, 1988.
"Writing Efficient Programs", Prentice–Hall, 0-13-970244-X, 1982.

2. The VisualWorks Virtual Machine

A Smalltalk system consists of two parts (see Fig.1):

- The *virtual image* ("Image"), containing all the objects in the system.
- The *virtual machine* ("VM"), consisting of hardware and software to give dynamics to objects in the image.

The VM is responsible for three functions:

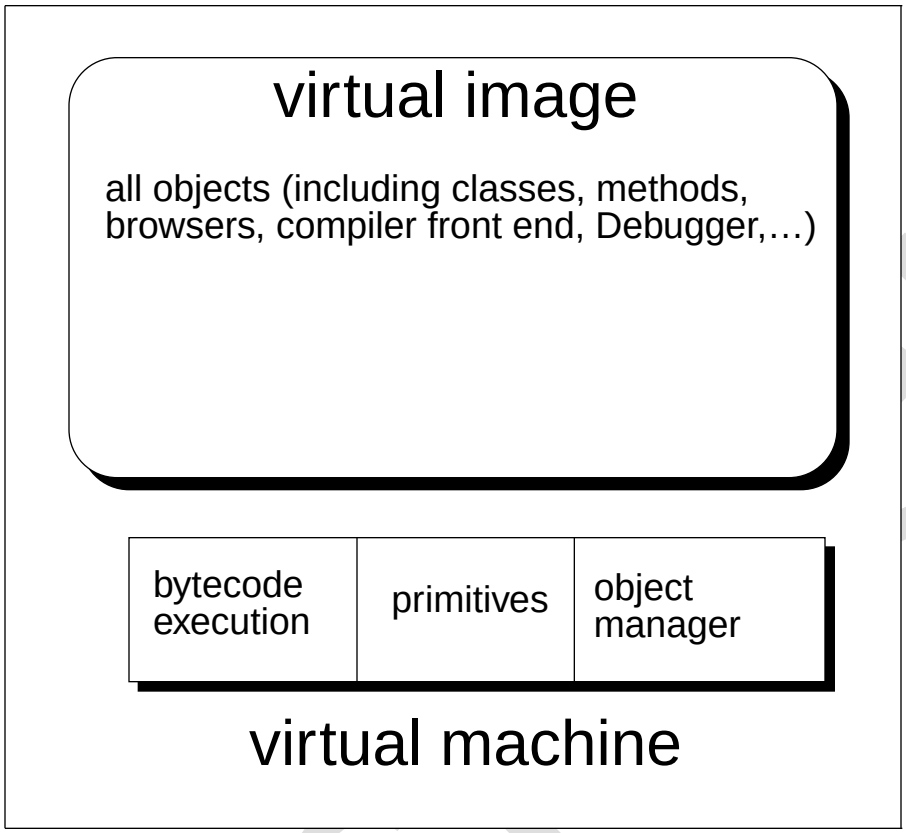

Figure 1: The Virtual Image and Machine

1. Execution of methods and blocks — the VM must execute the instructions that constitute the code of methods and blocks.
2. Primitive Methods — some methods cannot be written in Smalltalk, and must be implemented in the VM. Others are implemented in the VM for efficiency. Collectively, these are know as primitive methods, or just primitives.
3. Managing Object Memory — the object memory stores the objects in the image. The VM must organize space in the object memory to satisfy allocation requests, and must also collect garbage objects.

Let's look at these in more detail.

2.1. Bytecodes

Smalltalk methods are translated by the *compiler* (partly written in Smalltalk itself) into sequences of instructions (called *bytecodes*). The bytecodes for a method or block are then placed in an instance of a (subclass of) CompiledCode. Part of the compiler can be browsed in the categories System-Compiler-*. (See Fig.2.)

Note that the bytecodes are *not* interpreted in VisualWorks. When a method or block is evaluated for the first time, an internal compiler (hidden from the user) translates the bytecodes into native code (e.g., SPARC or PowerPC instructions). The native code is used subsequently.

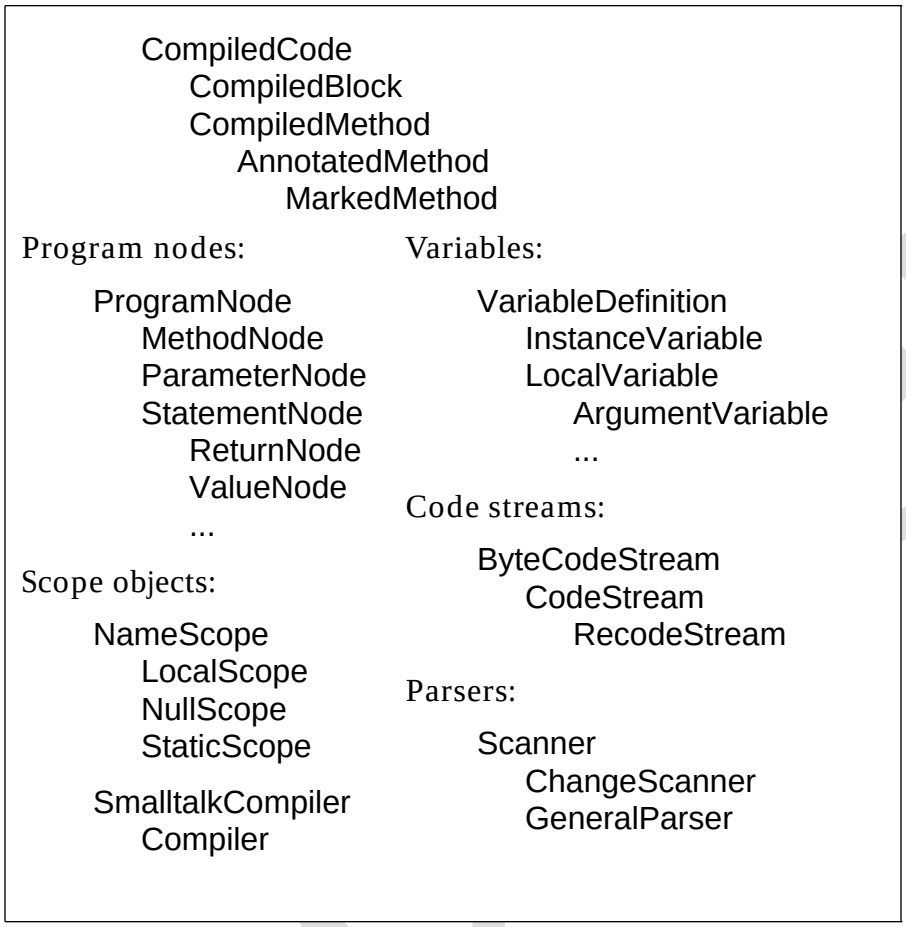

Figure 2: Bytecodes

A *compiled code cache* keeps a limited amount of native code (the native code is much bigger than the equivalent bytecodes). When the cache fills, an old method is discarded from the cache; the native code will be regenerated later if the method is evaluated again. The advantages of this approach are those of speed (interpretation costs are traded for compilation costs), while maintaining portability (an image will run on any platform for which a VM is available).

2.2. Contexts

The Virtual Machine presents the state of execution to the programmer by making visible representations of processes, and their stacks. The stacks are made up of Context objects, one per method or block in progress. Method activations are represented by MethodContexts, block activations by BlockContexts. This is used to advantage in the Debugger and Profilers, and the implementation of multi–processing and exception handling.

Each context contains a reference to the context from which it is invoked, the receiver, arguments and temporaries in that context, and the method or block being executed.

In reality, the VM avoids creating these objects if possible, i.e., when the programmer is unaware of their existence. Instead, it keeps this information on the stack, and creates the objects only when necessary (e.g., if the method refers to the thisContext pseudo–variable, which obtains a reference to the current context). This avoids placing undue stress on the garbage collector, and speeds the system substantially.

Ex 1. Inspect the compiled code of a method using the expression:

aClass compiledMethodAt: *#selector*

Can you match the instructions to the source?

2.3. Primitive Methods

Some bytecodes cause *primitive* operations to be performed by the VM. These are indicated in the source by the construct

<primitive: *n*>

at the beginning of a method —*n* is an integer identifying the primitive in the VM. A normal method body follows this construct. For example (taken from class Float):

```
* aNumber
    "Answer a Float that is the result of multiplying the receiver by the
    argument, aNumber. The primitive fails if it cannot coerce the argument
    to a Float"

    <primitive: 49>
    ^aNumber productFromFloat: self
```

Each primitive has a unique number to identify it. You can add your own primitives (See the VisualWorks User's Guide for more information).

When a primitive method is encountered, first the implementation in the VM is tried. This can either *succeed* or *fail*. If it fails, the Smalltalk code following the primitive construct is evaluated.

2.4. Object Memory Management

We can make a few observations about object memory:

- Most objects are small (much less than 100 bytes on average). Only a small percentage of objects (e.g. Images, and the System Dictionary) are over 1 Kbyte.
- Objects are created very frequently (e.g. Points); most of these objects have a short lifetime. Hence we need effective garbage collection.
- Object memory can become fragmented, especially if many large objects are created. The garbage collector may need to *compact* memory, in order to allocate space for more objects.

Some objects do not require space for an object body — they are *immediates*, and their value is encoded into their identifiers. Examples in the VisualWorks system include SmallIntegers and Characters. (Example: try Character instanceCount.) You cannot send the message become: to an immediate.

VisualWorks partitions memory into a number of different *spaces* with different management strategies for each space (Fig.3). Recently–created objects reside in *NewSpace* (which is itself partitioned), older objects reside in *OldSpace*. The *scavenger* reclaims garbage objects in NewSpace. When an object has lived long enough, it is *tenured*, i.e., moved into OldSpace. Old garbage objects are either reclaimed by an *incremental* garbage collector, or by a *compacting* garbage collector.

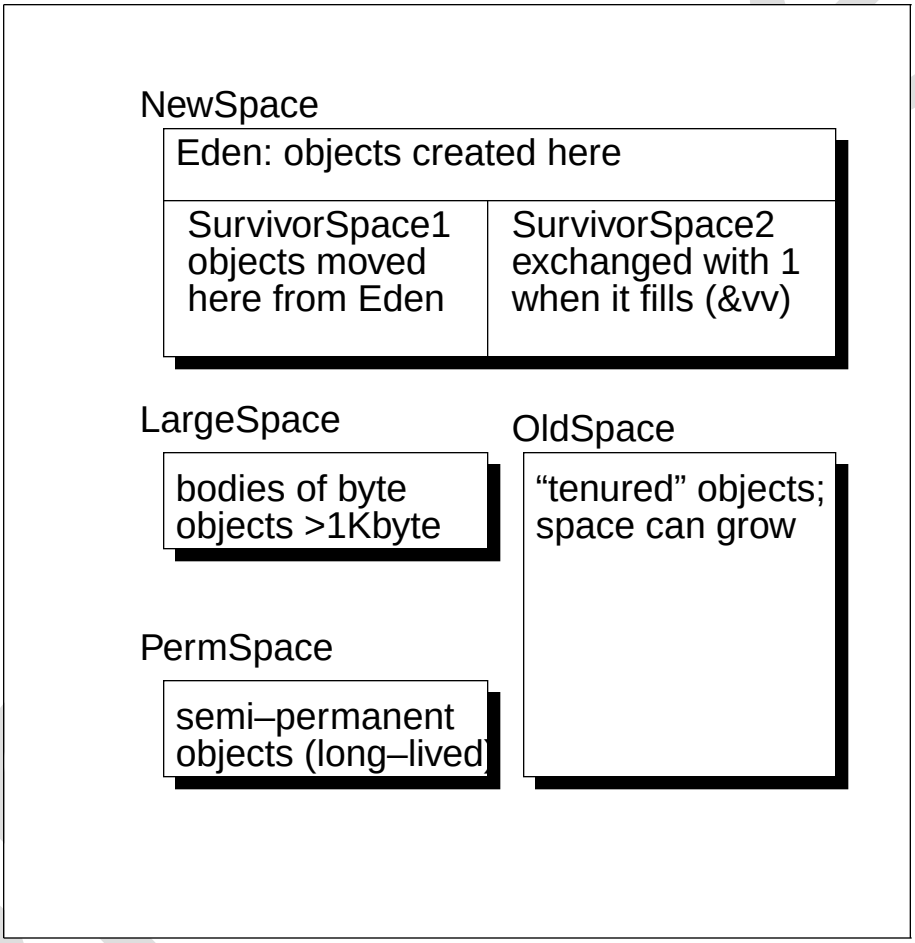

Figure 3: Object Memory

The programmer has a limited amount of control over when some garbage collection tasks take place.See the VisualWorks User’s Guide for more details (pp.341–6).

3. Measuring Performance

The main techniques for measuring the performance of a Smalltalk application are:

- Timing of code fragments, to get realistic indications of the overall amount of time being taken.
- Profiling (and Tracing) tools, to determine where the time actually is spent.

Also useful may be:

- Benchmarks, for comparing VisualWorks implementations and platforms.

3.1. Timing Code Fragments

VisualWorks provides a simple means of measuring the time taken for an arbitrary block of code:

```
Time millisecondsToRun: aBlock
```

Precision depends on the granularity of the clock (often 20 milliseconds). For good accuracy, we must time code which takes an appreciable amount of time (some seconds).

Timing parts of your code can be useful in comparing the relative times taken by different phases of an application, and in measuring how much effect a change has on performance. If you decide to modify code for a performance improvement, *always* measure the performance in some way before and after the change — otherwise you may just as well be guessing.

You can get wide variations in timings, by very large factors — as much as 2:1. These are probably caused by changes in the state of the run–time system (e.g., whether code is in the compiled code cache, the state of object memory).

To counter this you may want to force a garbage collection before timing the code, additionally it's worthwhile repeating the timings to get a variety of raw figures, then decide whether you're interested in either:

- Best time — if the timed code is likely to be run very often, and therefore will be in the compiled code cache, or
- Average Time — for less frequently run code[1]

If an individual time is very small (say less than 50 ms), the timing code itself may be a substantial proportion of the whole time. To counter this, time an empty block and subtract this from your results.

Examples:

```
Time millisecondsToRun: [100000 timesRepeat: [5.0/2]].

Time millisecondsToRun: [100000 timesRepeat: [5/2.0]].

Time millisecondsToRun: [100000 timesRepeat: [5.0/2.0]].

Time millisecondsToRun: [100000 timesRepeat: [5/2]].

Time millisecondsToRun: [100000 timesRepeat: [5//2]].
```

1. Using worst time is rare, because if the code is run very infrequently then its performance is not likely to be a cause for concern.

Ex 2. Run the examples to test the speed of division. Explain the difference in findings.

Ex 3. Write some message expressions to concatenate the class comments of all subclasses of class Object. How long it takes to perform this operation? (Make a note.)

3.2. Profiling

In principle, there are two different kinds of Profilers:

Tracers

A tracer monitors the complete control flow, including every message–send and primitive operation. It can give a precise, detailed picture of the flow of control through a program, as well as the amount of activity (time taken) in each method. Tracers usually require recompilation of the program; the resulting code is often much slower.

In VisualWorks, the Debugger provides some of this functionality, but execution is very slow, and it cannot be used to get a true picture of the time taken under normal evaluation.

Sampling Profilers

A sampling Profiler will interrupt the monitored process at regular intervals, and record which method is currently being evaluated (and what is below it in the stack). The information is only approximate, but there is only a small impact on execution speed, and no recompilation is required.

3.3. Using the ObjectKit Profilers

The Advanced Programming ObjectKit (APOK) package of utilities for VisualWorks includes a sampling Profiler to measure execution profiles, and a tracer that monitors every object allocation. Both are implemented as subclasses of Profiler (TimeProfiler and AllocationProfiler):

- The Time Profiler captures information about what percentage of time is being spent executing methods. It can be used to discover bottle necks;
- The Allocation Profiler captures information about the number and size of objects instantiated. It can be used determine memory overheads. However, it can also be used to increase code performance since object creation and destruction also takes time and is sometimes difficult to determine using the Time Profiler.

Both can be opened from a menu option on the ObjectKit launcher.

The Time Profiler

There are three programmatic ways of invoking the Time Profiler:

1. Evaluate the expression TimeProfiler profile: [“Some code”] to profile the contents of the block argument and open a Time Profiler window on the results (this is the “usual” usage);

2. Evaluate the expression TimeProfiler open to open a special workspace, in which any code evaluated using "do it" will be profiled (the slider at the top of the window adjusts the sampling period). Once profiled a Time Profiler window is opened on the results;

3. Evaluate the expression TimeProfiler profile: ["Some code"] reportTo: aFilename to profile the contents of the block argument and report the results (as text) in the specified file.

If using the first approach, the code to be profiled (Fig.4) should be placed in a block, and passed as an argument to the message self profile:. Ensure that the code runs for a reasonable amount of time (at least 100 samples, preferably thousands).The slider can be used to set the sample frequency; don't bother setting a delay much below 10 milliseconds, as the system clock is usually quantized to around this value.

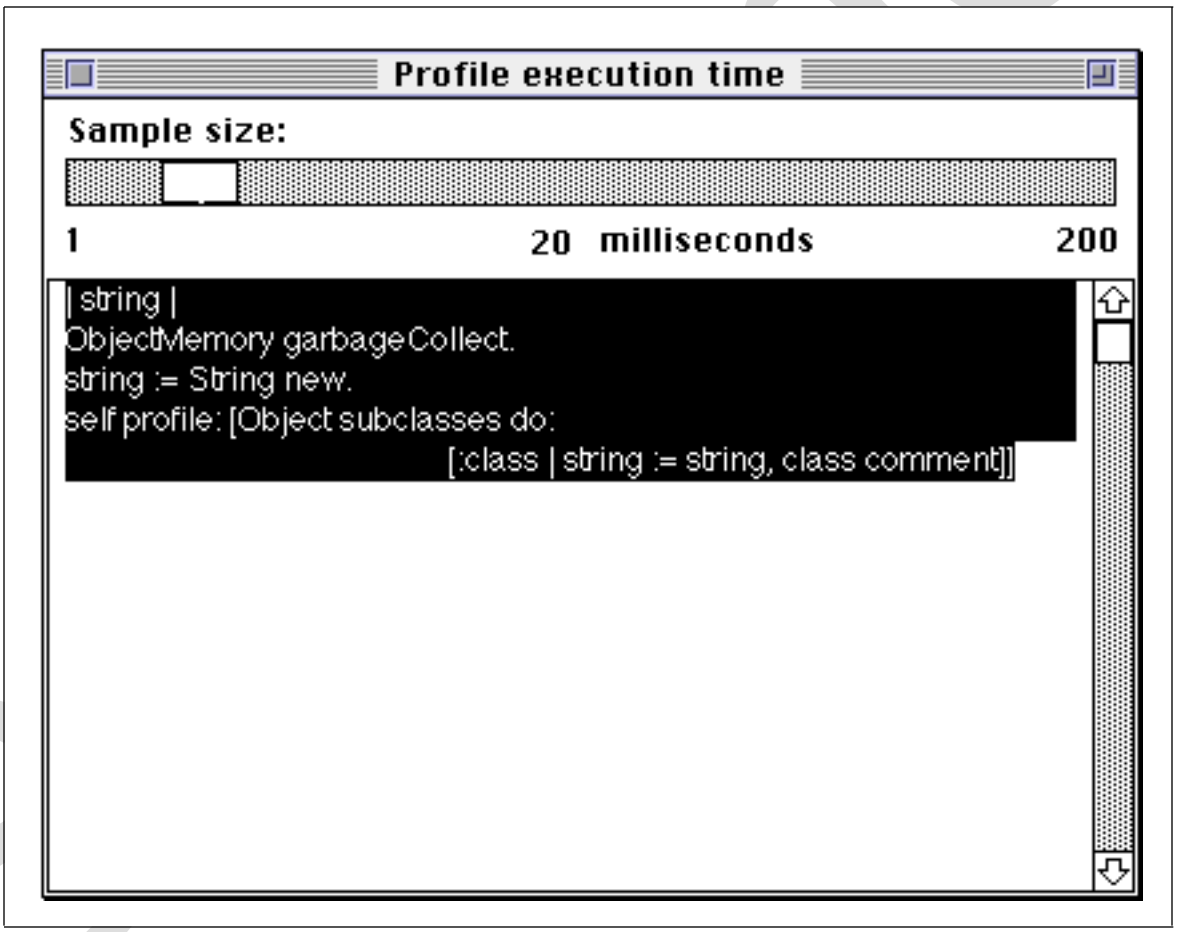

Figure 4: Time Profiler

The Time Profiler works by sampling the process executing the code periodically and recording which methods are executing in the context. The statistics are then gathered and normally displayed in the Time Profiler window. The Advanced User's guide describes in detail how to use the profiler, but here is presented a brief description of the tool.

After profiling, a summary view is opened in which the information may be displayed in one of two modes:

- 'tree' — this mode displays a hierarchical tree of contexts. Each line in the display has a number and the name of a method (or block in a method). The number represents the percentage of the total time that this method was active at this point in the context tree during the sampled period (Fig.5).

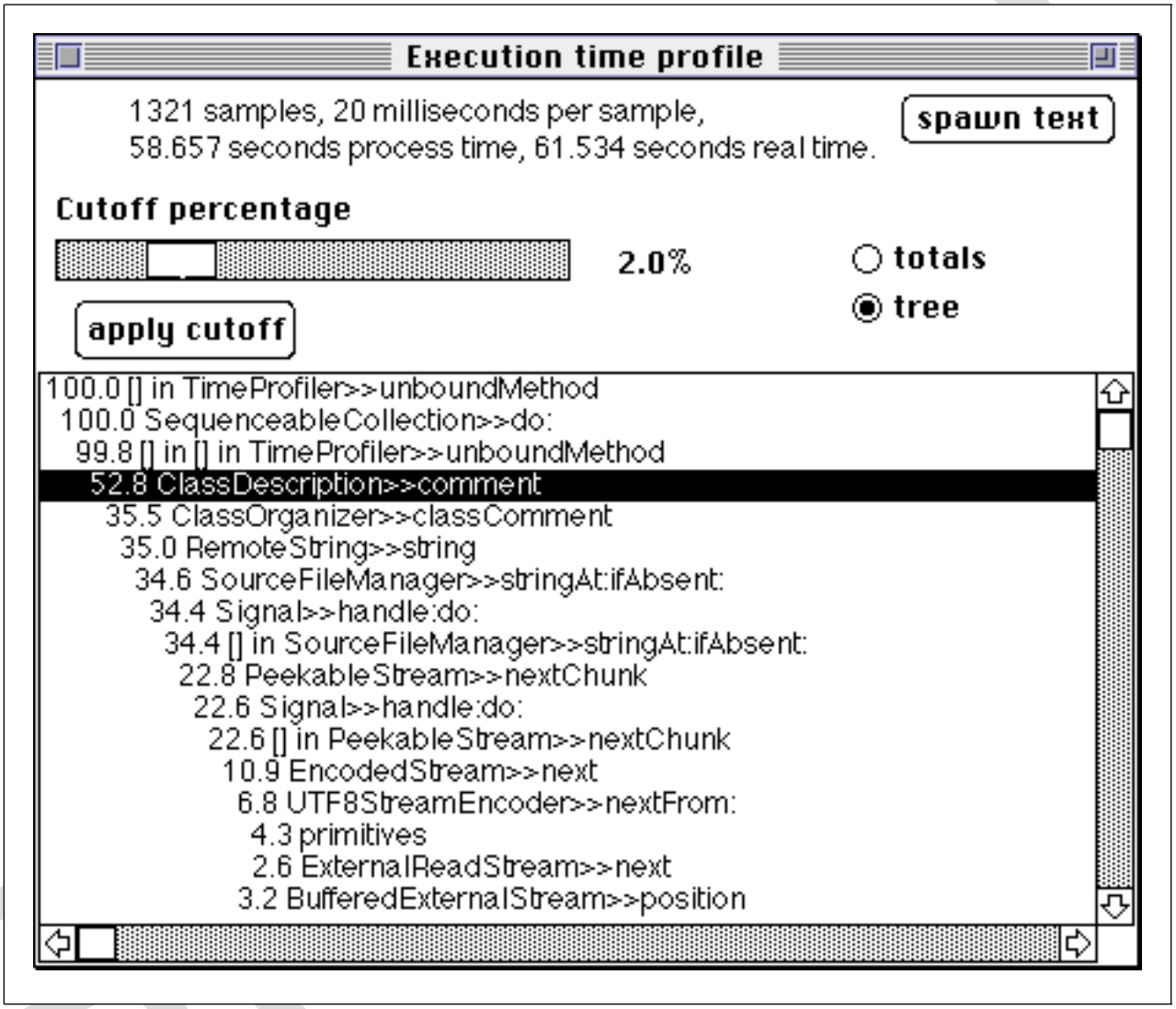

Figure 5: Time Profiler Tree

- 'totals' — this mode displays the total times spent in each method, sorted in order of duration (Fig.6). The display lines are the same format as the 'tree', but the number represents the total time spent in the method (as a leaf) during the whole sampled period (this is of limited use except in rare circumstances — it is somewhat misleading in its title)

Both summaries can be filtered to omit insignificant methods (by percentage of time) using the slider. The display of the context tree can be modified using the menu (Fig.7).

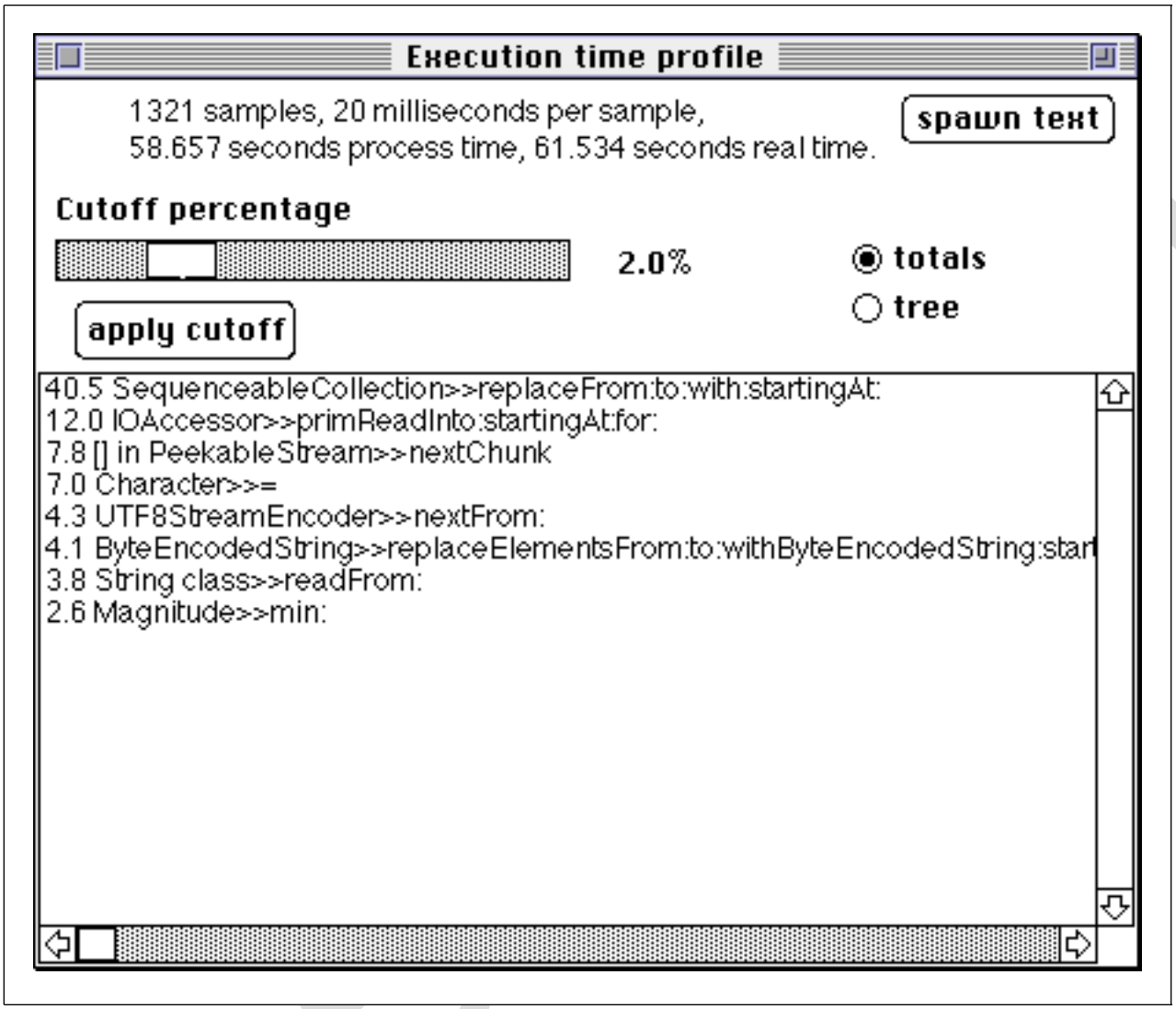

Figure 6: Time Profiler Totals

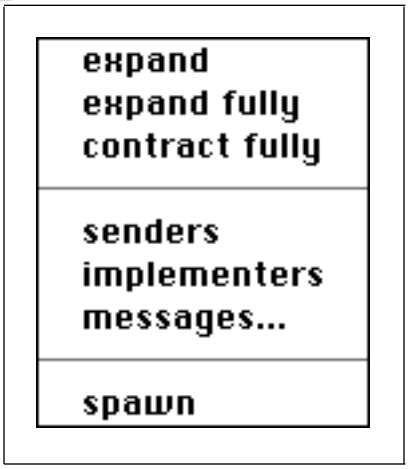

Figure 7: The Profiler menu

Of the two modes, 'tree' mode is the most often used. As can be seen from Fig.5, the contexts are displayed as an indented list. As shown in Fig.8, the display of the tree can be modified using the menu to highlight areas of interest.This shows that 99.8% of the time is spent in the inner block of the profiled block (the line
[] in [] in TimeProfiler>>unboundMethod). The messages sent in this block are also shown (52.8% in ClassDescription>>comment, and 47.0% in SequenceableCollection>>,). As expected, the percentage of time spent evaluating those methods sums to 99.8% (52.8% + 47.0%). Directly beneath ClassDescription>>comment, there are two methods (ClassOrganizer>>classComment and Object class>>readFromString:) whose percentage time adds up to less than the time spent in ClassDescription>>comment. At first sight this appears to be incorrect. However, there are two possible reasons for this difference:

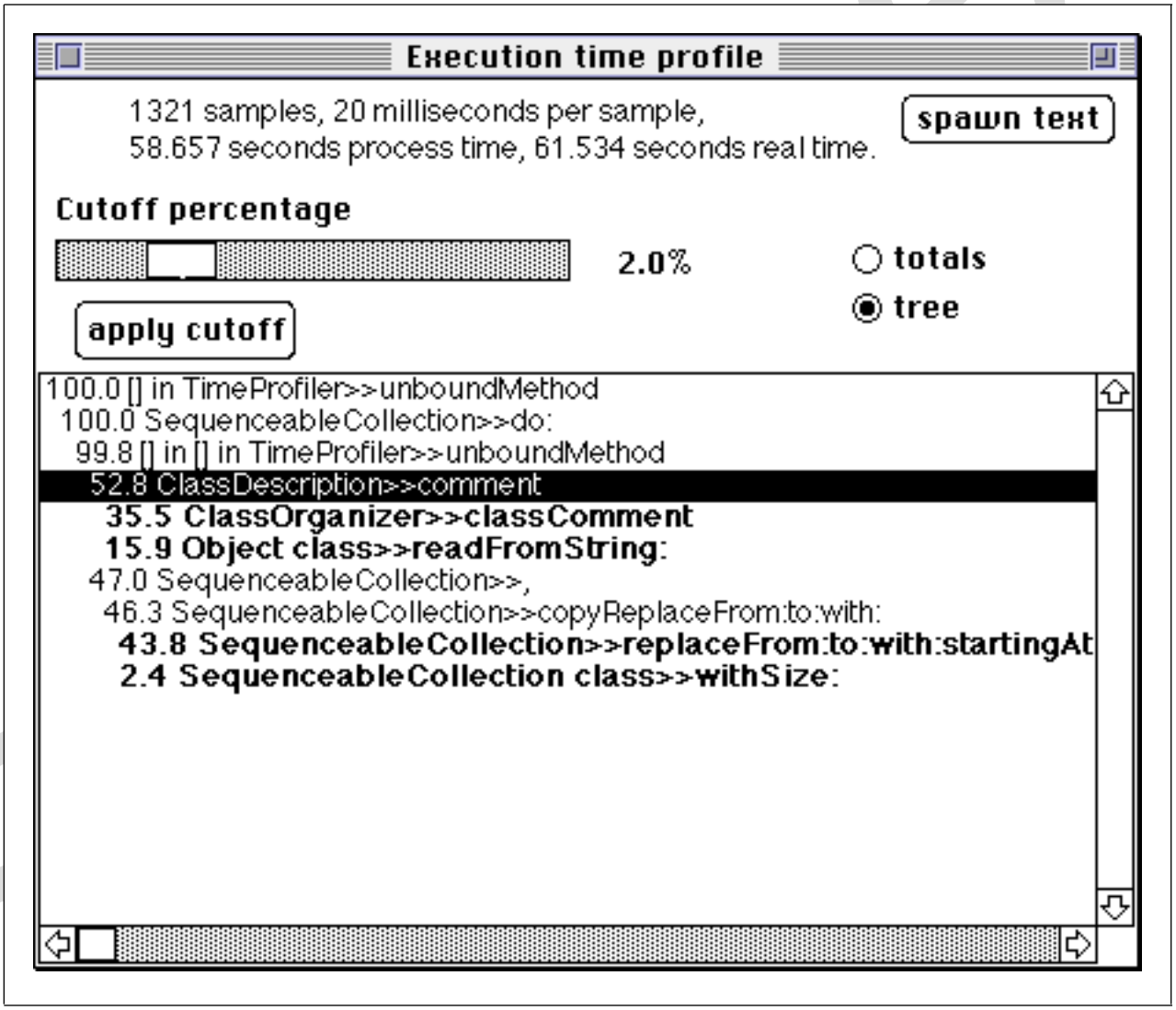

Figure 8: Changing the display of the Time Profiler Tree

1. the virtual machine takes some time to dispatch messages, allocate memory, etc.

2. the Time Profiler relies on a sampling mechanism which is not completely accurate

3. The slider in the upper of the window is set to only display those methods which take more than 2.0%

The Allocation Profiler

This Profiler intercepts all object allocation requests, and records the context in which they occurred, and the size, class and allocation type (word, byte, etc.) of each created object. (Because this is not a sampling Profiler, there is no need to repeat any test cases to improve precision of the data.)

The Profiler effectively samples the allocations, the sample being taken when a given number of bytes have been allocated since the last sample (set by the slider, Fig.9). As with the Time Profiler, larger sample sizes make for faster profiling (however, the context tree report is then less accurate).

The Allocation Profiler may be invoked using the same messages as those understood by the Time Profiler. However, there is one extra message, profileWithStatistics: ["some code"], which (after profiling) provides additional information about the total number and size of objects — the is the most common use.

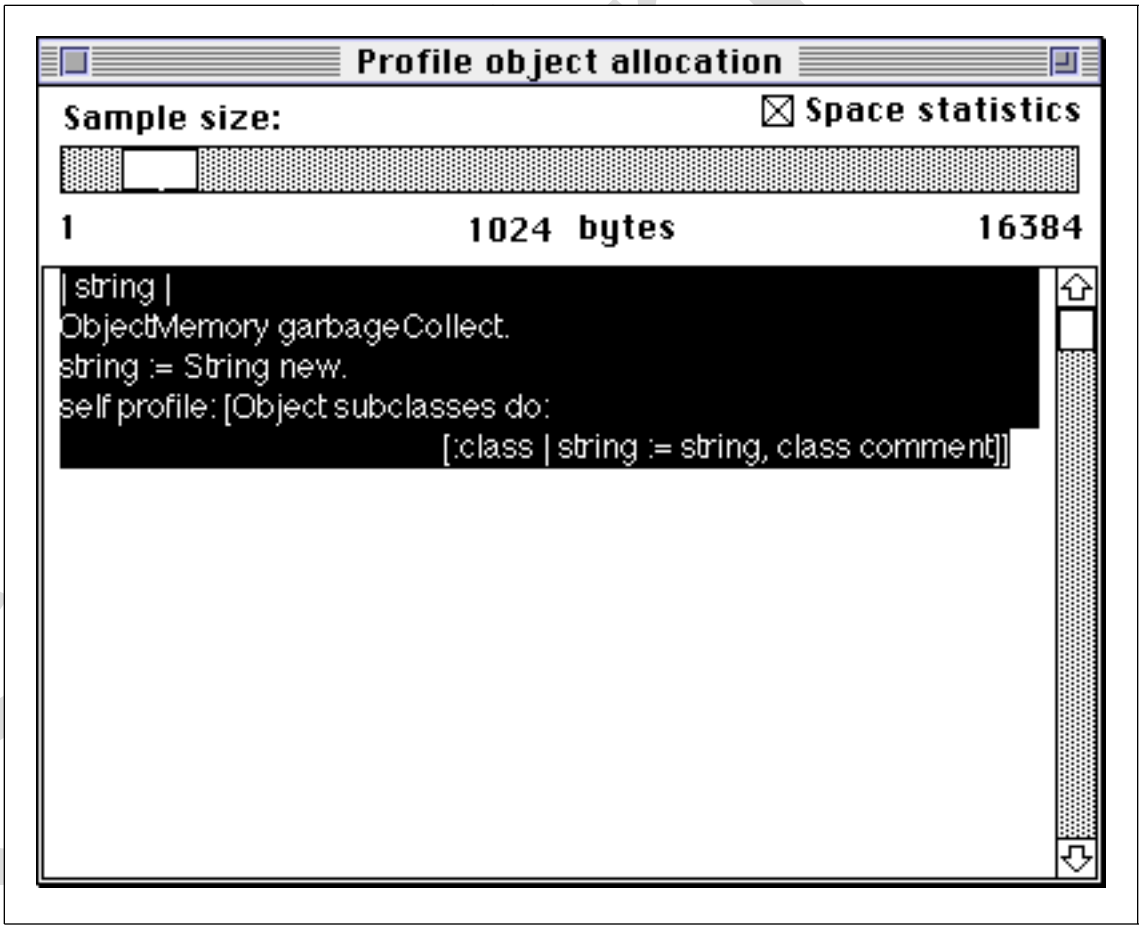

Figure 9: Allocation Profiler

The summary includes a tree of contexts to indicate where the allocations originated (Fig.10), summary totals of the objects created, and (optionally) a breakdown of the profile

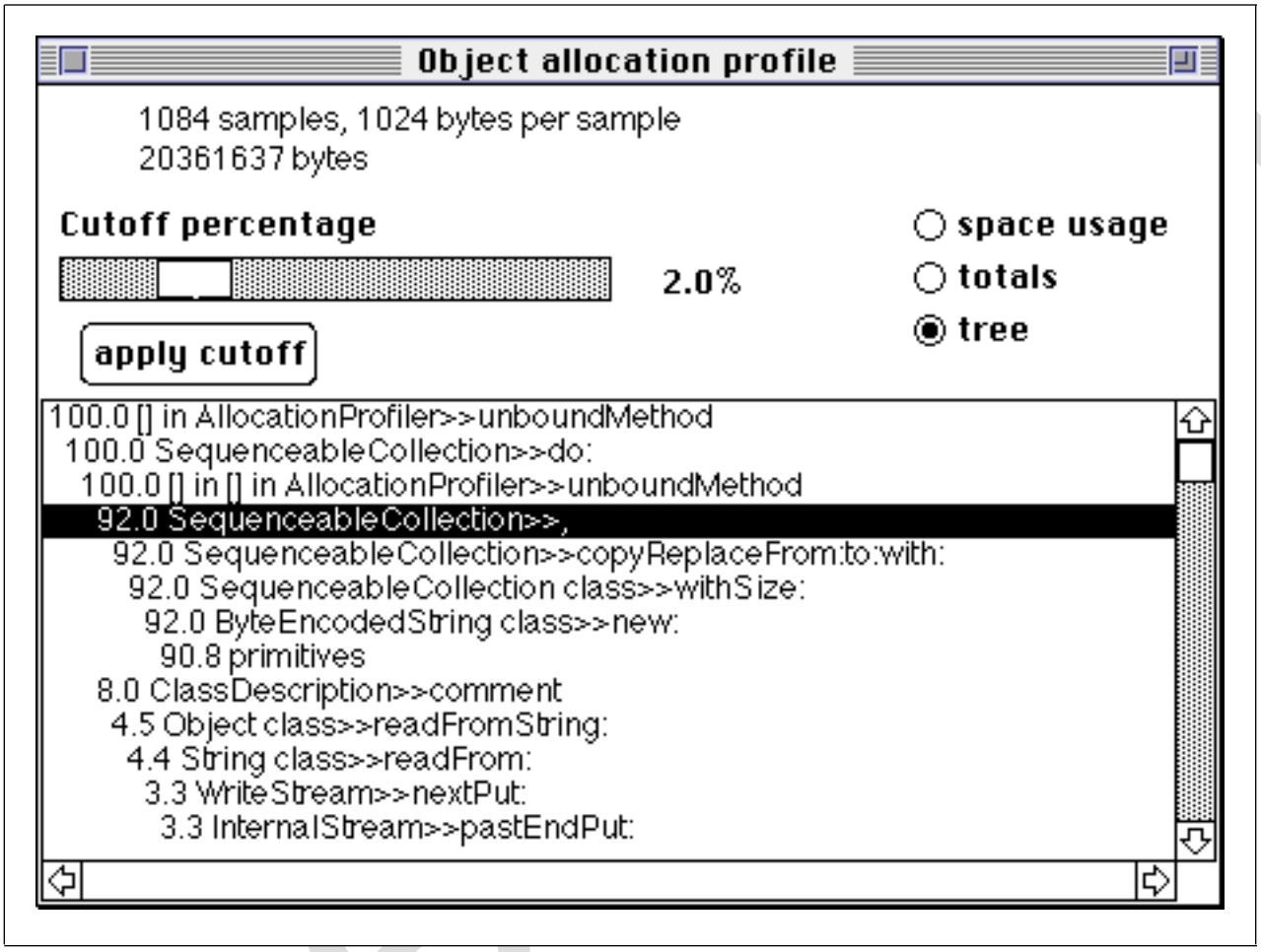

Figure 10: Allocation Profiler Tree

by class (Fig.11).

3.4. Problems with Recursive Code

The context tree is huge for deeply nested (e.g., recursive) code. It can be hard to interpret, and also takes a lot of space. For example, Fig.12 shows the Time Profiler tree as a result of profiling the following expressions:

```
| reps block |
block := [Browser allImplementorsOf: #at:put:].
reps := 60 * 1000 // (Time millisecondsToRun: block) + 1.
self profile: [reps timesRepeat: block]
```

The window displays the deep nesting of the message sends, highlighting two methods: Behavior>>allSubclassesDo: and Metaclass>>allSubclassesDo:. Trees that display this pattern are very difficult to interpret.

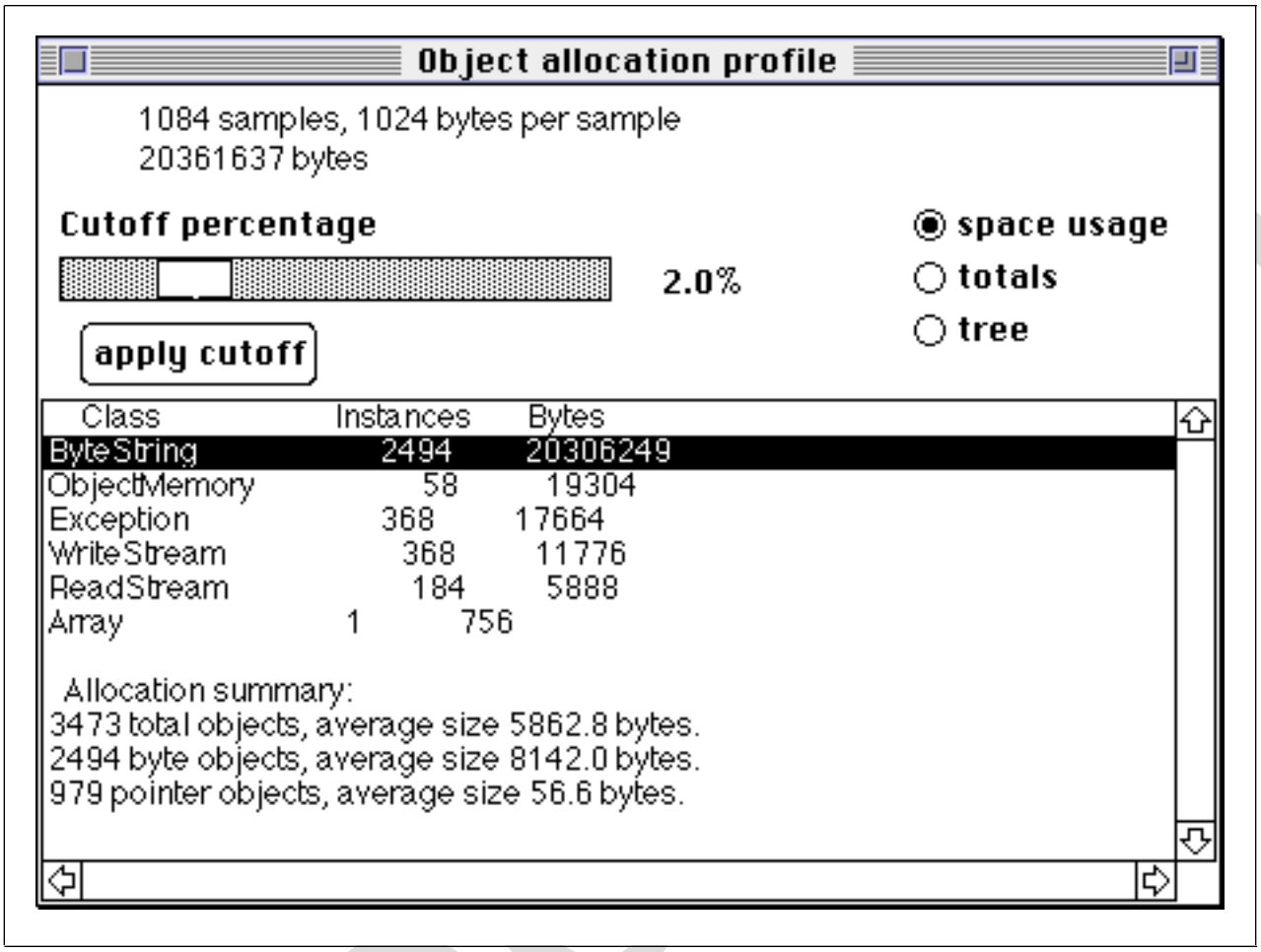

Figure 11: Allocation Profiler Space Usage

Furthermore, the **spawn** option from the menu only serves to complicate matters. In Fig.13, we can see the result of selecting the **spawn** menu option on Behavior>>allSubclassesDo: — a browser has been opened that provides an indented display of the methods who sent the selected message during the profiling, the total of their percentages against the method, and all the methods that method called with a summation of their percentages.

This shows that Behavior>>allSubclassesDo: was called in two places during the profiling, Behavior>>allSubclassesDo: and [] in SystemDictionary>>allSubclassesDo:. 307% of the overall time was spent in the first, 99.1% of the overall time in the second, summing to 406.1%. The indented methods underneath show that the selected method calls two methods and shows their percentages of the overall time, too. The figures can be misleading here, because the figures are summations from all the method branches during the profiling. Consequently it is possible to "double–count" some of the times (for example where recursive calls may occur). Numbers should only be compared relatively in any given browser, not between two different ones.

3.5. Other Possible Pitfalls and Problems

- They profile *all* the code evaluated by the block; it is hard to be selective.

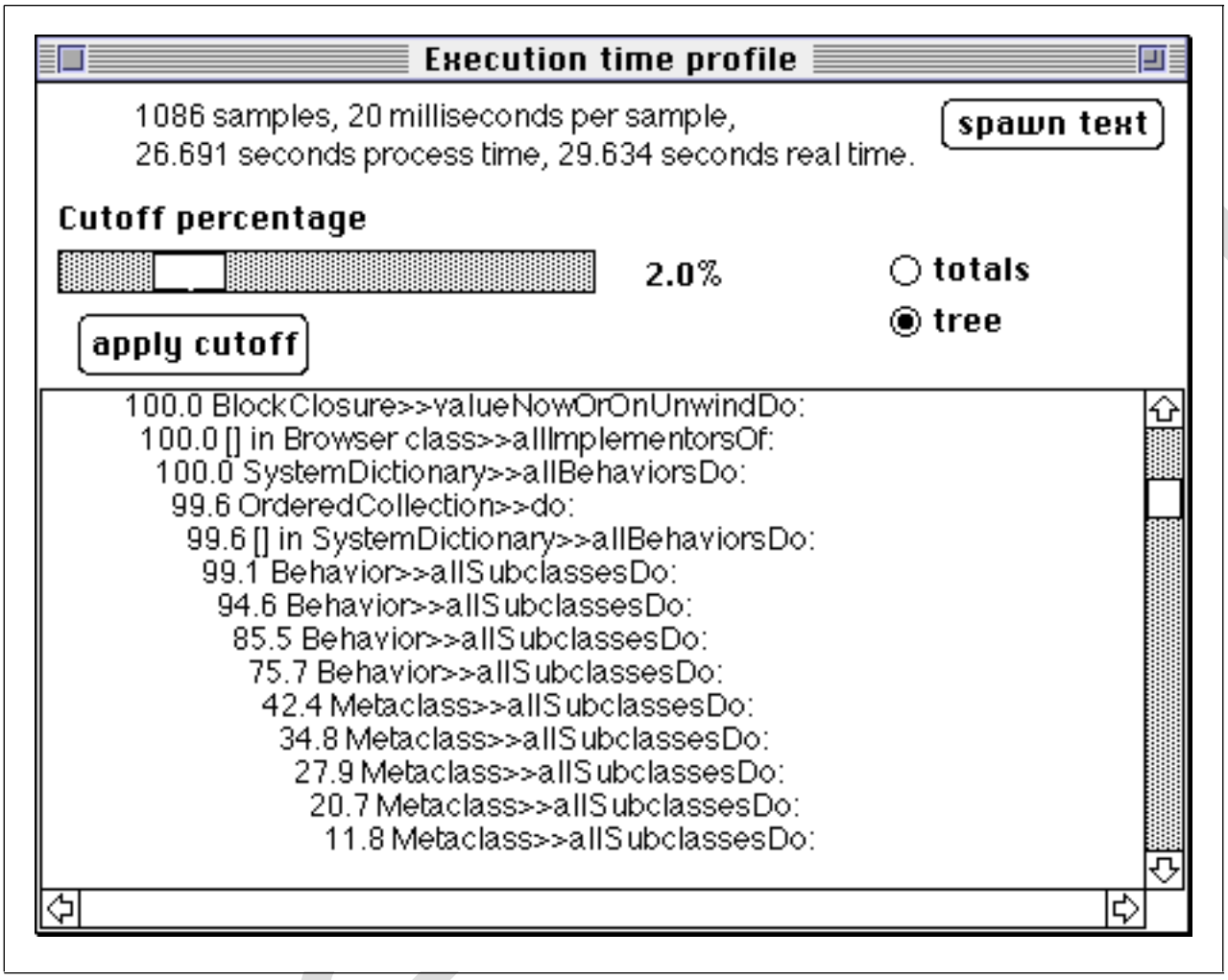

Figure 12: Recursive Tree

- The time Profiler has a subtle bug which can lead to over–estimates of time spent in deeply–nested contexts. Use a longish sample time (100 ms, say) to avoid this problem.
- The Allocation Profiler slows down object creation a lot.

Ex 4. Try out both Profilers; you may want to choose your own examples. How much slower do the examples run under each profiler?

Ex 5. Profile the code you wrote for Ex.3.

4. Improving Performance

In this section we will look at how to improve performance, concentrating on:

- Faster machine and / or more memory
- Program Changes

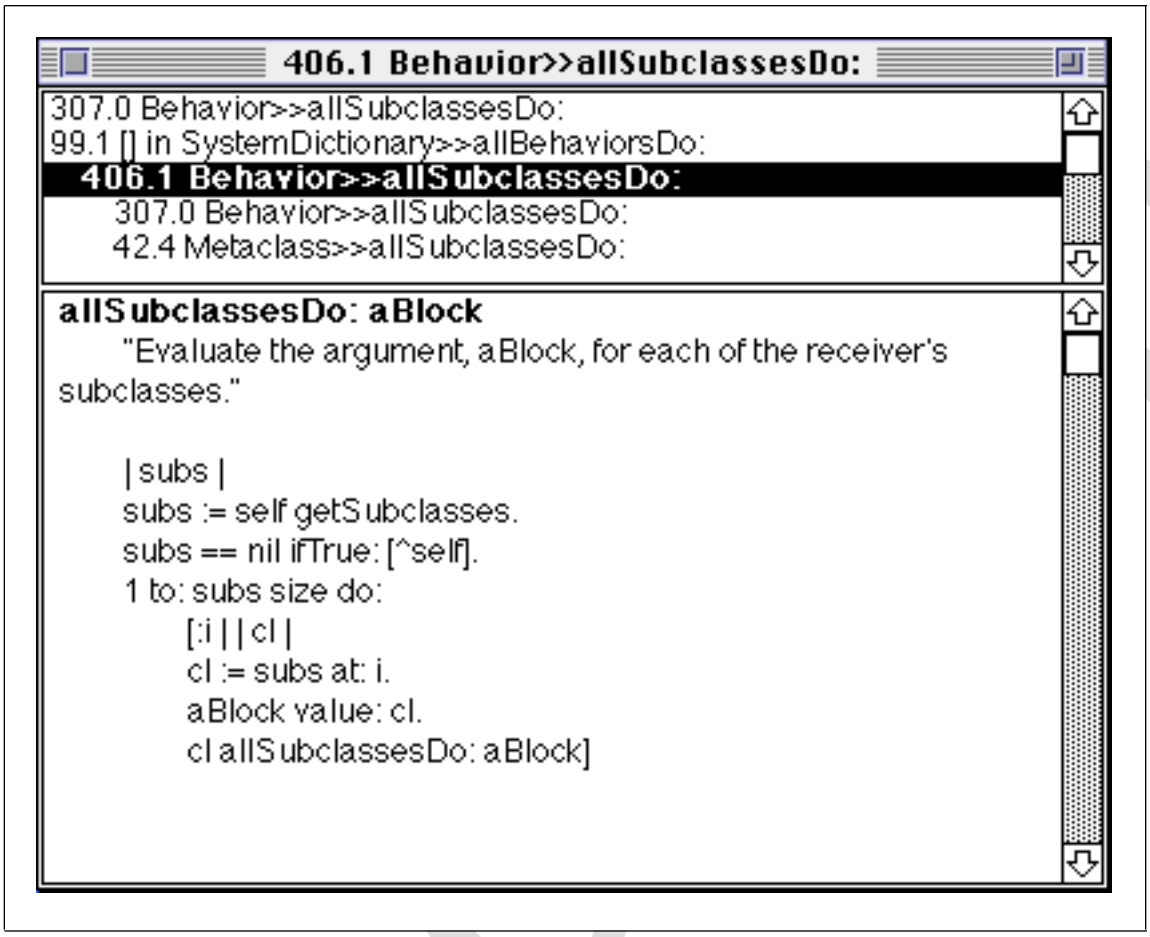

Figure 13: Spawning a Selected Method

- Tuning VM Parameters
- Using External Operations

4.1. Buy a faster machine, and/or more real memory

This may be the fastest, and / or most cost–effective performance improvement! Processor speeds go up yearly, and memory costs go down. Hence, paging (on virtual memory systems) is bad news; Smalltalk's behavior is rather different to that assumed by most virtual memory systems.

Extra memory can give huge increases in performance if it moves the working set into RAM.

4.2. Program changes

There are a number of changes that can be made to VisualWorks code that will optimize its performance. These are divided into 13 general changes, and specific changes to optimize the performance of collections, strings, blocks, etc.

General

1. In nested conditionals, put the most likely case first.

2. Don't use isKindOf: or isMemberOf:. Besides being slow, they represent bad object–oriented style (indicates the sender is taking responsibility for something that should be handled by the receiver).
3. Don't use respondsTo:. Besides being slow, it represents bad object–oriented style (for the same reasons as above).
4. Use IdentitySets and IdentityDictionaries instead of regular Sets and Dictionaries where possible.
5. Unless you are concerned about numerical accuracy (e.g., in monetary calculations), convert Fractions and FixedPoints to Floats before performing mathematical operations.
6. Where possible use SmallIntegers for all arithmetic. Reduce the amount of coercion by adopting the preferential order of arithmetic.
7. Use the following special selectors, which are optimized by the compiler:

   to:do:, ifTrue:ifFalse:, whileTrue:, and:, or:

   and others listed in the VisualWorks User's Guide (p.373).
8. and: is more efficient than & because it does not evaluate the argument if the receiver is false. Similarly, or: is more efficient than | because it does not evaluate the argument if the receiver is true. Both and: and or: are inlined by the compiler, so that no objects are created to represent the literal block arguments. So, unless evaluating the argument has side effects (which is, perhaps, bad style), use and: and or: instead of & and |.

   NOT:

   ```
   self sensor blueButtonPressed not & self viewHasCursor
   ```

   BUT:

   ```
   self sensor blueButtonPressed not and: [self viewHasCursor]
   ```
9. If a method requires repeated use of Character cr or Character space (for example), use the variables defined in pool dictionary TextConstants or its IOConstants subset to avoid repeated message sends. To gain access, list the dictionary as a pool dictionary.
10. Send changed: nil with: nil rather than the more general changed, which simply builds the same message for you. Similarly, implement update:with:from: rather than update:.
11. If the same message is being sent repeatedly inside a loop to access a constant, assign it to a temporary variable outside the loop.

    NOT:

    ```
    quantities inject: 0
            into: [:tot :qty | tot + qty * self getPrice]
    ```

    BUT:

```
| price |
price := self getPrice.
quantities inject: 0
        into: [:tot :qty | tot + qty * price]
```

12. Be careful with Transcript show: aString. Appending text to the Transcript is an expensive operation, so don't write out any more than necessary. Use Transcript nextPutAll: aString followed by Transcript flush instead.

13. Avoid generating Symbols. Avoid sending asSymbol.

14. Use Symbols as dictionary keys in preference to Strings.

Collection

When it is known that a collection is going to become quite large, create it using new:, supplying a guess at its final size. The default new only allocates between 2 and 10 elements (depending on the class), which can cause the collection to waste a lot of time growing (copying) itself.

Streams and Strings

1. Use a stream protocol rather than the concatenation operator to build a large collection from multiple subcollections.

   NOT:

   ```
   s := ''.
   $a asInteger to: $z asInteger
           do: [ :c | s := s, (String with: c asCharacter)]
   ```

   BUT:

   ```
   | s |
   s := (String new: 26) writeStream.
   $a asInteger to: $z asInteger
           do: [ :c | s nextPut: c asCharacter].
   s contents
   ```

2. Streaming over an Array which is larger than necessary and requesting the stream's contents will truncate the excess.

3. Reuse a stream by resetting it rather than creating a new stream.

Ex 6. Try out the streams and strings examples. What are the differences in space and time usage? How do these vary as the length of the loop is increased?

Blocks

A simple block that makes no references to private variables other than its own arguments or temporaries, is called a *clean* block. A simple block that makes no references to private variables other than its own arguments or temporaries, or self, instance variables, or arguments to any surrounding blocks or method is called a *copying* block.

Clean blocks are bound at compile time, and are the fastest kind. Copying blocks are slower, but still faster than the most general kind of simple block (known as *dirty* blocks) and continuation blocks (those that end with a return statement, for example: [:x :y | ^x + y]). In general, move the declarations of temporaries to the innermost possible block.

The special selectors mentioned above are *inlined* if literal blocks are used, so no block objects are created, nor are messages sent to evaluate the blocks, hence for those messages there is no need to worry about the clean / copying / dirty distinction.

Clean:

```
[:i | i performFunction]
```

Copying:

```
[:i | self performFunction: i]
```

Dirty:

```
| temp |
[:i | temp := temp + i performFunction]
```

Continuation:

```
[ ... ^nil]
```

Ex 7. Investigate the performance of the following:

```
100000 timesRepeat: []

(1 to: 100000) do: [:i| ]

1 to: 100000 do: [:i| ]

| b | b := []. 100000 timesRepeat: b

| b | b := [:i| ]. 1 to: 100000 do: b

true ifTrue:[]

| b | b := []. true ifTrue: b
```

Exceptions and Contexts

The exception–handling mechanism is mostly implemented in Smalltalk itself, with a little primitive support. It works by using the thisContext pseudo–variable to access the current context, and thence to access the stack of Contexts (MethodContexts and BlockContexts) in the current process's stack.

Hence, whenever an exception is raised, the stack has to be converted into object form. This can take a considerable amount of time. Hence it is advisable to only use exceptions for genuinely exceptional cases. Also, avoid using thisContext in performance–critical code.

Contexts also have to be converted to object form whenever a process is suspended (either due to a the message suspend being sent to Processor or the message wait being sent to a Semaphore). This puts a minimum overhead on process switching.

Ex 8. An exercise in the module "Blocks – Advanced Use" was to construct a block–exit method. Clearly, one use for this mechanism would be for loop exits:

```
[ :exit |
  coll do: [ :x | ...
    someCondition ifTrue: [exit value: nil].
    ...] valueWithExit
```

We defined a loop exit mechanism using exceptions in an exercise in the module "Handling Exceptional Conditions". Compare the performance of the methods:

a. What are the respective setup costs?

b. What are the costs of invocation? How do these vary with depth of stack?

Use object identity

Testing for object identity is very fast: the current VisualWorks compiler inlines the test, and uses no messages at all (this also means redefining == is completely ineffective). Hence, use == (and ~~) rather than = (and ~=) where safe to do so.

Much more effective is the use of identity–based collections (IdentitySet and IdentityDictionary). When building new keyed collections, consider providing equality– and identity–based versions. Alternatively, use objects whose definition of equality is identity (e.g., Symbols).

Ex 9. Investigate the relative performances of Set and IdentitySet when adding a large collection of Symbols (e.g., Smalltalk keys). What if the items added are converted to Strings first?

Avoid creating large, short–lived objects

There is always a tension when iterating over some aspect of a structure between building a collection of objects for the projection of that aspect, or building a special–purpose iterator. Example, in Behavior:

```
Object selectors do: [:selector | …]

Object selectorsDo: [:selector | …]
```

(The latter does not exist.)

The first of these constructs a Set which is discarded after the iteration is finished. This may be inefficient (especially as the selectors are already held as a separate Array). However, it is probably more flexible and reusable. Similarly note the difference between sending the message keysDo: to a Dictionary, rather than keys do:. Unfortunately there is no easy answer here — treat each case on its merits.

Ex 10. Use the Profiler(s) to compare the difference in times between:

```
Smalltalk keys do: [:k | … ]

Smalltalk keysDo: [:k | … ]
```

Then implement the selectorsDo: method in Behavior and compare with the version that builds a Set.

Avoid Recomputations – General

General principle: avoid repeatedly computing the same result, but keep the previously computed result.This is a space–time trade–off. This is especially easy in an object–oriented language, as you can often hide the cache inside an object or class, e.g., an instance variable, or a dictionary held by a class variable.

Often, use an instance variable which is either a useful recently–computed value, or nil. If the variable is nil, compute the value instead and retain in the variable. If the cached value becomes inappropriate, set the instance variable back to nil.

Examples:

SystemDictionary>classNames

BorderedWrapper>insetDisplayBox

CompositePart>preferredBounds

Browser *menu class variables*.

Avoid Recomputations — displays

If a view is composed of only a small number of different Images, generate them all, once, and retain them using an instance or class variable. Example: class variables in LabeledBooleanView.

Remember that the pixels in a Pixmap are stored externally to the object memory, whereas that in an Image is held by VisualWorks. This means displaying a Pixmap is likely to be much faster than an Image (especially if using X on a remote display). Class CachedImage is provided to switch between the two on demand.

Ex 11. Browse the methods that are listed as using a cache. Then try the examples in `CachedImage-Tutorial`.

Specialized Objects

Use of specialized subclasses of collection classes can give dramatic performance improvements.

Example: use of RunArray when Array would be inefficient (Primarily saves memory, but may improve speed if memory is tight).

Example: specialized Dictionary subclasses, optimized for storage space, insertion / removal, or search time.

A useful optimization is to specialize on the contents of a collection: (e.g., String is an optimized Array).

Ex 12. Optimize (or rewrite) the message expressions from Ex.3. What is your percentage speed improvement?

Ex 13. A common operation in a graphics system is to apply a transformation to every point in a list of points (e.g., adding a point to every point in the list). Profile

some examples using the obvious implementation (i.e., an Array of Points), then sketch an optimized point collection class that makes such batch operations efficient.

Encapsulate complex processes in objects

This is partly a style issue, but can also impact on performance.

If you are implementing a complex algorithm, operating on many objects, with many intermediate states, you should consider encapsulating and controlling the algorithm within a single object.

This can save much parameter passing and accessing of shared variables; both can lead to uglier code.

Examples: Compiler, scanners of all sorts.

4.3. Tuning VM Parameters

The sizes of various spaces (NewSpace, OldSpace, compiled code cache, etc.) can be controlled by the programmer, as can the thresholds which trigger the scavenger, or the timing of the incremental garbage collector.

These are all controlled by methods in ObjectMemory. An instance of ObjectMemory represents a snapshot of the state of object memory, and contains statistics about the space usage. The class MemoryPolicy is used to manage some aspects of the memory system.

If you think some of the performance problems are memory related, use the statistics from ObjectMemory to determine where the problems lie (e.g., if numScavenges is abnormally high, you might expand the NewSpace, or use the allocation Profiler to determine object creation rates). If your application has very unusual behavior, you can define your own MemoryPolicy for a different strategy.

See the VisualWorks User's Guide (pp.345–6) for more details.

4.4. Using External Operations

You can extend the VisualWorks system by calling external routines, usually written in C or C++. One use of these routines is to interface with existing libraries (e.g., device drivers in the operating system). Another is to rewrite operations more efficiently, in a language with less overhead (often due to less checking). For example, in C you do not pay the price of array–bound checking or garbage collection (at the expense of safety).

DLL and C Connect

This additional package from VisualWorks allows your Smalltalk application to invoke functions written in C to manufacture, modify and use C language datatypes, and to send messages to Smalltalk objects from your C code. The C functions can be statically linked into your application's executable, or dynamically loaded at run–time using the target platform's dynamic library loading facilities.