Benchmarking Performance in LiveCode

by Richard Gaskin

What is Benchmarking?

Wikipedia defines benchmarking as:

In computing, a benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it.

LiveCode's flexibility provides many ways to solve a given problem, often using algorithms which may sometimes vary quite broadly.

In order to determine the method which provides the best performance, it can be helpful to run benchmarking tests, usually relatively simple routines which isolate the thing you're testing into two or more tests, measuring the time it takes each one to complete the task.

Earlier xTalk implementations like HyperCard made this difficult to do because the finest measurement of time provided was a "tick", roughly one-60th of a second. The LiveCode engine provides time measurements in milliseconds, which is still often too course for many tasks run just once, but if run multiple times can provide useful measurements.

Why Benchmark?

There are many reasons why benchmarking can be useful. If you're making tools only for yourself, knowing the fastest way to solve a problem will minimize the time you spend waiting for your computer to finish it. And if you make commercial apps, delivering a more responsive system will likely increase your customers' overall satisfaction with your product.

In the modern world, the success of a software product is often driven by usability, in which responsiveness plays a role. The more fluid experience

you can provide, the less you interrupt the user's mental processes while using your app, and the more they can focus on the task they booted your app to accomplish.
Innovative features also play a key role in software success, and increasingly these include real-time updates to information displayed on screen. For example, in the olden days (more than a decade ago) window resizing was indicated by an XOR'd bounding rectangle, while the content region of a window was updated only when the resize operation was completed. In modern systems of course window resizing usually happens in real time, allowing the user to see objects adjust themselves within the window as its size is changing.
More significant examples include Inspector palettes, Master-Detail views, and other live updates which give the user a more immediate understanding of their data.
Moore's Law gives us a free ride for performance in many ways, with processor speeds increasing year over year. But relying on increases in hardware performance alone isn't enough for most apps to hone their performance, and in a very-high-level language like LiveCode performance can be even more critical.
Fortunately, the LiveCode engine provides enough language features to allow us to deliver apps with performance comparable to some lower-level languages for a wide range of common tasks, especially those involving GUI display and text processing. This is not all that surprising when you consider that most of the work is done within the engine in well-optimized routines written in C++, with your scripts merely acting as "glue" to tie these internal routines together.
The trick to getting the most out of the engine is to honor my favorite maxim:
Know the engine
Trust the engine
Use the engine
As you learn more about the engine, you can make ever better use of it to let it do more of the work.
The more clock cycles you can save in the day-to-day of writing code, the more you can spend later on as your app's features expand. You can think of benchmarking as a way of opening a savings account for new features: save cycles today so you can spend them on things that add true competitive value to your app tomorrow.

The more you know about how the engine works, the more you can easily adopt good coding habits that hone performance without any extra work as using faster methods increasingly become second-nature.
Benchmarking can help you better know the Rev engine, so you can make the most of the uncommonly good performance it can offer.

How to Benchmark

First and foremost, to get useful benchmarks you'll want to isolate what it is you're measuring. Within the context of a complex app it may be difficult to pin down exactly where a performance issue may be showing itself, but if you take the time to test specific parts of your program's operation you can usually find the weak spot in relatively short time.
Central to benchmarking of course is the measurement of time, so the first thing you want to do is store the time before the test is run, and then run your test and compare the store time to the current time when it's done:

```
on mouseUp
   put the millisecs into t
   RunTest
   put the millisecs - t into tResult
   -- Display the result:
   put tResult
end mouseUp
```

For many tasks, however, you'll find that running a test just once will take less than a millisecond, so the script above won't be very useful.
You might be tempted to think that if a task takes less than a millisecond it can't matter all that much. But consider that well-factored code often produces a code base in which many objects will call a relatively small number of commonly-used handlers. The more frequently those handlers are called, the more the time spent there adds up to eventually make a noticeable difference to the user.
So when benchmarking, you'll usually want to run your test multiple times, often hundreds or even thousands of times.
Incorporating this into our example above gives us:

```
on mouseUp
   put 1000 into tIterations
   put the millisecs into t
   repeat tIterations
```

```
      RunTest
   end repeat
   put the millisecs - t into tResult
   -- Display the result:
   put tResult
end mouseUp
```

Of course benchmarking a single algorithm is rarely as useful as comparing it to other ways to perform that task, so you'll usually have at least two tests run in your benchmarking script:

```
on mouseUp
   put 1000 into tIterations
   --
   -- Test 1:
   put the millisecs into t
   repeat tIterations
      RunTest1
   end repeat
   put the millisecs - t into tResult1
   --
   -- Test 2:
   put the millisecs into t
   repeat tIterations
      RunTest2
   end repeat
   put the millisecs - t into tResult2
   --
   -- Display the results:
   put tResult1 && tResult2
end mouseUp
```

When running multiple tests it can be helpful to provide identifiers in the results so you can easily identify which result was for which method you tested, so we can expand the result display at the end so the hander reads:

```
on mouseUp
   put 1000 into tIterations
   --
   -- Test 1:
   put the millisecs into t
```

```
   repeat tIterations
      RunTest1
   end repeat
   put the millisecs - t into tResult1
   --
   -- Test 2:
   put the millisecs into t
   repeat tIterations
      RunTest2
   end repeat
   put the millisecs - t into tResult2
   --
   -- Display the results:
   put "RunTest1: " & tResult1 &cr \
      & "RunTest2: "&tResult2
end mouseUp
```

You can copy the above into your benchmarking stacks to get you started, replacing RunTest1 and RunTest2 with your own handler names.
I find it's useful to save my benchmarking stack for future reference as the engine changes and new language features are added. I keep a folder named "Benchmarks" for this, where I keep my benchmark tests handy if I need them in the future. It's usually not hard to recreate most simple tests, but it's often even faster to just open one you've already made and run it, for the low cost of a few kbytes on your drive.

Shortcomings of Common Benchmark Tests

While most benchmarking is simple to do, there are some limitations which can be helpful to keep in mind:

Effects of other processes

As useful as benchmarking is, most benchmarking tests cannot tell the whole story of how performance will work in real-world applications. They're far more helpful than not, but keep in mind that many factors outside of your control can affect your timing, including networking and other background tasks running in OS threads, tasks performed by other open applications, and sometimes even the impact of running one test before another, which can subtly alter memory in ways that give advantage to earlier tests.

With these in mind, it can be helpful to minimize these effects by closing other applications while benchmarking, turning off networking, etc. while running your tests.
But more useful and usually much simpler is just to run your test multiple times, not just the iterations you established by the whole script itself should be ideally run at various intervals to help offset the effects of other threads affecting them.
If you see consistent trends running your tests at different times, you can have reasonable confidence that you've identified a good pattern worth adopting.

Effects of scaling data size

A more significant concern is scaling. Most tasks involve manipulating data, and data comes in various sizes. You may find that some algorithms work great on small data sets, but slow down disproportionately as the size of the data grows.
For example, using chunk expressions to parse text can compare favorably to using arrays in some circumstances, but in most cases you'll find arrays much faster as the size of your data grows, since most chunk expressions require the engine to count delimiters character by character as it traverses the data while arrays use hash tables which relate to specific memory locations for ultra-rapid lookups.

Effects of implementation-specific nuances

Arrays are a good example here, since their blinding speed is not without a few notable exceptions.
For example, if you're only using data in memory you'll find arrays hard to beat over most chunk expressions, but the very memory-location nature of their hash tables also makes them problematic for storage in ways that don't affect chunk expressions.
If you store an array in a stack file's custom properties, or if you write its data to a file using arrayEncode, you'll notice those operations take much longer than simply writing a block of text which would be suitable for parsing with chunk expressions. This is because the array's internal hash table has to be translated into a form that is no longer specific to memory location, since of course on disk any memory locations become irrelevant. This translation is a costly procedure, often taking several times longer to store than storing a single chunk of data of equivalent size.
Also consider that the number of dimensions you need to traverse in an array will add up. While a single lookup in a one-dimensional array is lightning fast, if you have a need to have four levels of nested arrays and

need to obtain the data from each of the inner-most elements, you'll probably find chunk expressions faster.

While a four-level deep array may sound unnecessarily exotic, consider the case of document data: you will not be able to know in advance how many documents will be open, so that's one dimension. You may not know the number of tables, so that's another. And the number of records will vary, as will the number of fields within each record, for a total of four levels. To get the data from a each field in each record of a given table you'd do something like this:

```
  repeat for each key tRecord in tRecordKeys
     repeat for each key tField in tFieldKeys
       get gDocumentData[tDocument][tTable][tRecord]
[tField]
     end repeat
  end repeat
```

If instead you stored simple tab- and return-delimited lists in custom properties, you'd find it surprisingly faster to do this:

```
  repeat for each line tRecord in tTable
      repeat for each item tField in tRecord
        get tField
      end repeat
  end repeat
```

While a seemingly rare case, it became critical in a system I was designing in which I had to store document data. In a database-style app where you'll likely be working with a single storage file, you can omit the document level array and bring the nesting count down to three, which will improve performance noitceably. But apps that handle documents are not so rare as to make this an irrelevant outlier.

That said, in light of the earlier discussion about scaling I should note that the performance you see will vary based on data size. Depending on the size of your data and what you're doing with it, it may not be a question of whether to use arrays or chunks, but by the time you're working with the amount of data where this makes a huge difference the better question could be "Why aren't you just using a database?"

So when running benchmarks it can pay to keep the big picture in mind, considering the context in which your algorithms will be used.

Effects on programmer efficiency

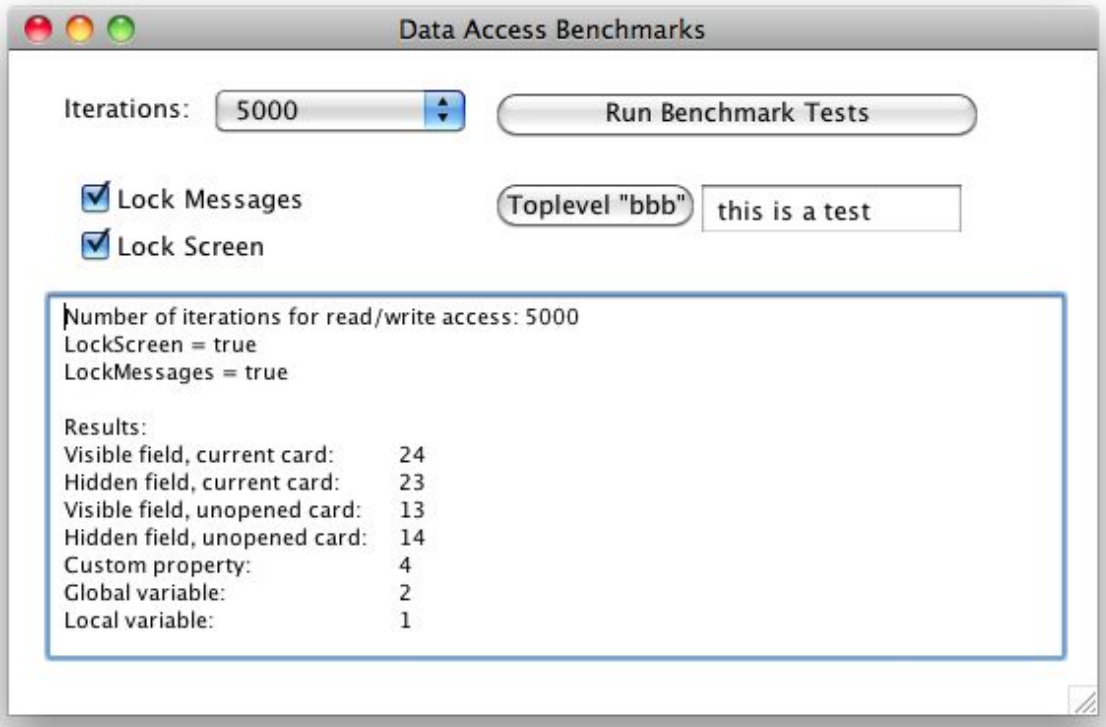
When considering which method to use to solve a problem, the value of your own time as the developer should also be taken into consideration as

well as that of the user. I tend to favor the user experience over my own time, but programming convenience makes for shorter delivery times which also favor the user, so it's a fine balancing act.
For example, regular expressions (such as can be used in Rev in the matchText and replaceText functions) are usually much slower than using chunk expressions to accomplish the same task, because the regex subsystem is broadly generalized in a way that optimizes programmer efficiency over performance efficiency. But given that you can often use a single line of regex to do what would take a dozen or more lines of chunk expressions, if the task isn't called frequently from other parts of your code you may benefit the end user more by just using regex and getting the feature into their hands that much more quickly.

For all of the reasons noted above, benchmarking results should be viewed in light of your specific circumstances with the project at hand. Ideally one would be able to afford a minute or two to independently verify a "comon knowledge" performance guideline before using it to guide your programming habits, with your tests honed in ways that reflect the specific context in which you'll be using your algorithms.

Example Benchmark Test: Data Access Methods

Here's a benchmark stack I set up some time ago to test the various ways to store and retrieve data:



Sure, it's ugly, but that's part of its charm: it took only a minute to make.
You can download it here: DataAccessBenchmarks.rev.gz
The script is here, with some of the results listed below:

```
--
-- Requires:
--
-- mainstack with
--    field named "f1" (visible)
--    field named "f2" (invisible)
--
-- substack named "bbb" with
--    field named "f1" (visible)
--    field named "f2" (invisible)
--
-- The string "this is a test" is put into all fields.
-- Also, a user prop named "u1" has been defined in the
-- mainstack which also contains that string.
--
-- An option control button to conveniently change the
```

```
-- number of increments during testing is also on the
-- main stack.  Curious about other options, I added
-- checkboxes for LockScreen and LockMessages.
--
on mouseUp
  set cursor to watch
  put empty into fld "results"
  wait 0 with messages -- show that field is cleared
  --
  -- Setup vars:
  global g1
  local t1
  put "this is a test" into g1
  put "this is a test" into t1
  --
  -- Setup options:
  set the lockMessages to ( the hilite of btn "lock
messages")
  set the lockScreen to ( the hilite of btn "lock
screen")
  --
  -- Number of iterations:
  put the label of btn "iterations" into n
  --
  -- TEST #1: Visible field, current card:
  put the millisecs into t
  repeat n
    get fld "f1"
    put it into fld "f1"
  end repeat
  put the millisecs - t into t1
  --
  -- TEST #2: Invisible field, current card:
  put the millisecs into t
  repeat n
    get fld "f2"
    put it into fld "f2"
  end repeat
```

```
put the millisecs - t into t2
--
-- TEST #3: Visible field, closed card
put the millisecs into t
repeat n
  get fld "f1" of stack "bbb"
  put it into fld "f1" of stack "bbb"
end repeat
put the millisecs - t into t3
--
-- TEST #4: Invisible field, closed card:
put the millisecs into t
repeat n
  get fld "f2" of stack "bbb"
  put it into fld "f2" of stack "bbb"
end repeat
put the millisecs - t into t4
--
-- TEST #5: Custom prop:
put the millisecs into t
repeat n
  get the u1 of this stack
  set the u1 of this stack to it
end repeat
put the millisecs - t into t5
--
-- TEST #6: Local var:
put the millisecs into t
repeat n
  get g1
  put it into g1
end repeat
put the millisecs - t into t6
--
-- TEST #7: Global var:
put the millisecs into t
repeat n
  get t1
```

```
    put it into t1
  end repeat
  put the millisecs - t into t7
  --
  -- Display results:
  put "Number of iterations for read/write access: "& n
&cr \
      &"LockScreen = "& the hilite of btn "Lock Screen"
&cr \
      &"LockMessages = "& the hilite of btn "Lock
Messages" &cr&cr \
      &"Results:  "&cr\
      &"Visible field, current card: "&tab& t1 &cr \
      &"Hidden field, current card: "&tab& t2 &cr \
      &"Visible field, unopened card: " &tab& t3 &cr \
      &"Hidden field, unopened card: "&tab& t4 &cr \
      &"Custom property: "&tab& t5  &cr \
      &"Global variable: "&tab& t6 &cr \
      &"Local variable: "&tab& t7 &cr into fld
"Results"
end mouseUp
```

Here are some of the test results:

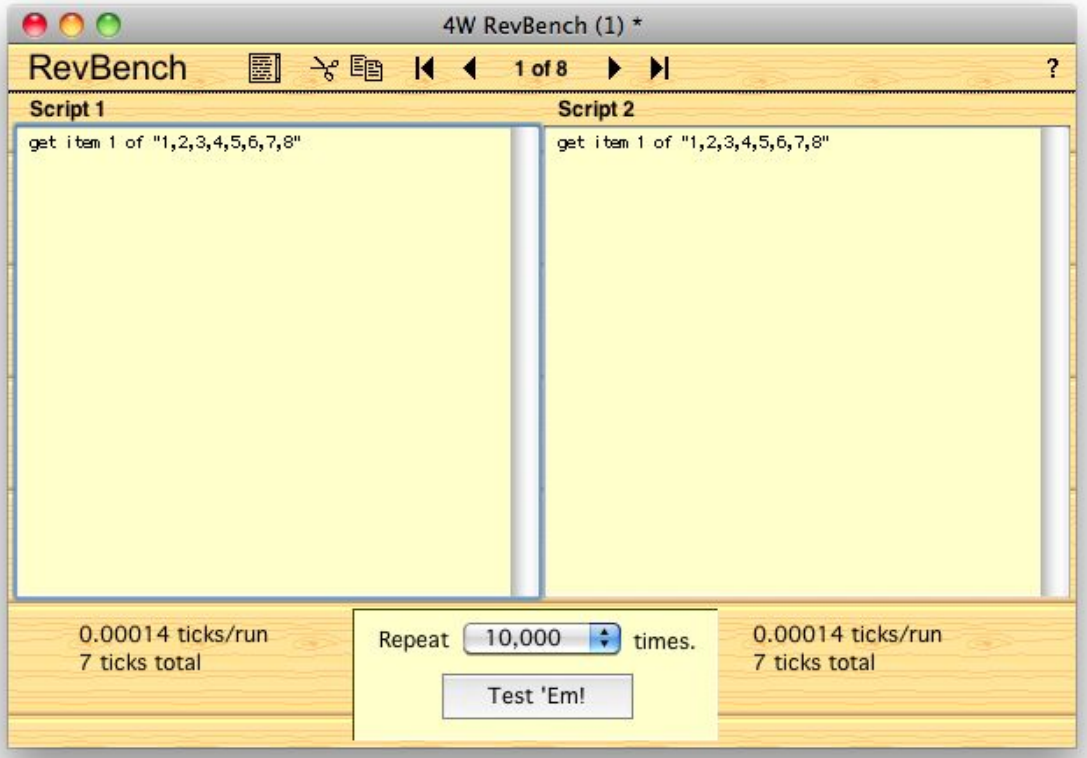
```
Number of iterations for read/write access: 5000
LockScreen = true
LockMessages = true
Results:
Visible field, current card:  22
Hidden field, current card:  23
Visible field, unopened card:  13
Hidden field, unopened card:  14
Custom property:  5
Global variable:  1
Local variable:  2

Number of iterations for read/write access: 5000
LockScreen = false
LockMessages = false
```

```
Results:
Visible field, current card:  3353
Hidden field, current card:  4823
Visible field, unopened card:  13
Hidden field, unopened card:  14
Custom property:  13
Global variable:  2
Local variable:  1
```

The results are not all that surprising if you keep in mind that in general the more complex a data structure is the more steps will be needed to traverse it.
I once wrote an analogy on the use-revolution list which describes the difference in the number of steps needed to get data in and our of a field object relative to a custom property. While tongue-in-cheek, those who've worked with text in low level languages will appreciate the humor there.
More interesting is the difference between fields on the current card and fields on unopened cards, but still not surprising if you imagine what's going on under the hood: When a field is on the current card it's expected to handle a lot more work, initialized into the event loop such that even when the lockMessages global property is turned on the mechanisms needed to hande interactivity and messaging must still exist. On an unopened card, a field can be safely tucked away in a simpler state.
This can be handy if you need to use a hidden field for text processing, like stripping HTML tags by first setting the htmlText of the field and then gettings its text. Doing this on an unopened card will save you significant time.
Least surprising is that variables are faster than anything else. This has been consistently true with all xTalk implementations I've worked with, including HyperCard, SuperCard, OMO, Gain Momentim, Toolbook, and of course LiveCode.
The reasoning follows the Golden Rule of data structure complexity: variables have the least overhead of any data access option, so traversing them will require the fewest steps.

Benchmarking Tool: RevBench

Some years ago I put together a simple stack called 4W RevBench to simplify some of the benchmarking I was doing.



You can download RevBench in the Stacks section of RevNet: In LiveCode, see **Development->Plugins->GoRevNet**

Next Steps

In future LiveCode Journal articles we'll cover some of the benchmarks noted above in greater detail, such as the unexpected performance of traversing nested arrays relative to chunks, and explore why these results came about and the implications for your projects.

We'll also provide a summary or architectural and coding tip to help you optimize your app's performance and your workflow in making them.