Optimisation - is it possible?

One of the first thoughts that many people have when they first look at Offsider technology is "*Can the framework be made faster*"?

This rant considers that question.

Why is it so slow anyway?

Offsiders were not designed for speed, they were designed for persistence, flexibility and ease of use. The were designed for extreme late binding of everything. They were designed for simultaneous concurrent access by multiple processes.

The way offsiders work is very unusual, and at first glance is a really really bad idea. They thrash the hard-drive, are i/o bound, spawn hundreds of processes - and for what? Well, the *for what* are all the points mentioned in the previous paragraph. If you think about it long and hard, you will find that the unusual (and "inefficient") design of the framework is a straight-forward and simple solution to the design criteria. In other words, I chose my requirements first, and then looked for the easiest (not the fastest) way to implement those requirements. The real question is: "*Is there a better way*".

My suspicion is that there is not.

How I approach software design

Everybody probably knows that "premature optimisation is the root of all evil", or something similar. Whether everybody truly believes that is another thing altogether. Whether everybody works that way is something else again.

Fortunately for me, some of my earliest experiences as a computer programmer enforced in me a healthy suspicion of optimisation in general. One was where I was required to maintain a piece of software that had been optimised to death (unneccessarily), the other was where I was writing a programmable keyboard driver on the Commodore C64 computer. I was using an interactive, interpreted language yet was shocked to discover that the software **ran too fast** and **needed to be slowed down!**. I soon discovered that firstly *optimisation is sometimes a really bad idea*, and secondly that *you can't second-guess the speed of the software*.

I guess the other thing that influenced by thinking was that my university degree was in pure-mathematics. I valued the abstract beauty of a good design much more than the ugly messiness of a clever coding hack.

So my approach is

1. Come up with a good, clean design that encapsulates the problem domain in an intuitive way.

2. Implement that design in the most straight-forward and transparent way possible

3. Run the resulting software

4. If it is too slow, then find out why and concentrate there, otherwise do nothing.

How I wrote the offsider framework

..That is the approach I took in building the offsider framework.

At first, I must admit that even I was worried about the volume of file system access, and was tempted to sneak in some speed concessions here and there.

I decided to do all my initial development and testing on my oldest and slowest machines, figuring that any real speed problems would show up sooner rather than later. Gradually I came to realise that - yes, the software was relatively slow but - it was fast enough to do real work, even on a slow machine.

That was all I needed. I gradually became more comfortable in implementing the "correct" design rather than agonising over whether I should maybe cheat here and there. As I gained more experience with the framework, and extended it, I came to trust the object-oriented messaging paradigm more and more.

Moreover, as I used the framework in more complex settings, I came to demand power and flexibility. As a result, the framework was rebuilt several times. Each time I rebuilt the framework, I became more committed to ignoring opimisation short-cuts.

By the time the Offsider project was released onto SourceForge, I knew from experience that the framework was both powerful and fun to use. I also knew from experience that it was fast enough for practical use.

It's too early to optimise yet

I have described the overall direction for the Offsider project in my project roadmap. Basically, I have said that from now until the 1.0 release, I am focussed on getting the API right, and that I am not interested in optimisation.

There are a few reasons for that decision.

1. Until the API is fixed, any optimisation may well be wasted effort, if you are optimising something that won't make it into the final release.

2. Until we have a formal API, and a comprehensive test suite, it is hard for me to be confident that any optimised solution works correctly under all circumstances.

3. Until we have a final API, and some standard usage scenarios, it is hard to demonstrate that any given optimisation is truly an improvement, or that it is even solving an identified bottleneck.

4. I want to avoid the situation where an existing optimisation hack becomes a road-block to changing the design of the API.

Points 2. and 3. probably need some extra explanation. Because offsiders use extreme late binding everywhere, you need to consider what happens if a user over-rides any of the methods that interact with, or are called by, the code you have changed. It might make your optimisation completely redundant or ineffective. In the worst case scenario, you might even break the API by attempting to short-circuit the message parsing logic (or to put it another way, by breaking the information-hiding rule of object-oriented programming).

"*But I'm really keen, and I have some great ideas!*"

I don't mean to put anyone completely off the idea of optimisation. If someone can come up with a solution that works, and is effective, then I encourage you to pursue that idea, and let me know about it. I just don't want be side-tracked at this stage. After the release of version 1.0, I will concentrate on speed. Just not now.

What I will do, however, is to include optimisation work in a separate part of the release, so people who want to work there, or to try out the optimised versions, have access to the code.

I think it is important to do this work anyway, because (as I hinted earlier), I don't think optimisation at the framework is easy. In fact I suspect that there is almost nothing that can be done. So in a way, I am encouraging you to find that out for yourself. If nothing else, you will come to a much better understanding of how the framework really works, and what it is really trying to do. In fact, I would value that as a sign of someone who has mastered the framework. On the other hand, if your idea really does work, then that is excellent and can be an important contribution.

Why it can't be done

I would like offsiders to be accepted far and wide. I would like offsiders to be used in all sorts of different applications and systems. Unfortunately, the speed and resource-usage problems will inevitably limit its acceptance and use.

Therefor, I keep trying to think of ways that I can make the framework run faster.

I have considered:

- trying to encode common methods into the `offsider` executable,

- writing everything in C,

- systems that automatically re-write methods so they replace messages with hard-coded early binding (but in a way that preserves the appearance of late-binding, and keeps the late-binding code available for the user to edit).

- putting the filesystem into memory,

- bypassing the filesystem altogether and doing raw block i/o,

- writing a dedicated filesystem,

- using special purpose operating systems,

- hacking the kernal

- AND MORE!

Unfortunately, everything I have looked at has problems of one sort or another. Usually it simply doesn't work because it breaks one of the design criteria. Sometimes it simply doesn't run faster.

For example, I spent several days exploring the possibility of a system that would rewrite shell script methods. It would scan the method, looking for method calls. It would then work out exactly what the `offsider` executable would do for that method call, and insert a the end result into the code at that point. This is quite a tricky thing to do, but to my great dissappointment and surprise, the code didn't run particularly fast, and sometimes ran slower. I still can't quite understand why, but quite frankly I have better things to do with my time.

What I am trying to say here, is that I have actually thought long and hard about this issue, and I have already come to the conclusion that the framework is already running as fast as it can.

The real bottleneck is in the filesystem and the operating system.

The good news is that you can get your code to run faster, without optimising the framework at all!

It's the application, stupid!

No, I'm not suggesting that anyone is stupid. It's just a catchy title. :)

One of the reasons it is so difficult to optimise at the framework level is that the offsider framework is so over-the-top flexible. It is so flexible that there is almost nothing you can say for sure about anything. That means you can make no assumptions. It means you don't know ahead of time what anything is doing, or how it works or what it is supposed to do. You can't make any assumptions about the grammer or syntax of a message, about the internal structure of the offsider, or about the run-time behaviour of a method.

You can't optimise something unless you have some concrete knowledge to work with.

On the other hand, why do you want to optimise anyway. Obviously, because you want YOUR application to run faster. There is the solution.

Don't try optimising the framework. Concentrate on optimising your application. It turns out that it is easy to optimise at the application level, and the gains are often immediate, obvious and measurable. The difference is that when you come to consider a specific application, you can suddenly start making assumptions. In fact, they aren't even assumptions. You now KNOW for certain what the messages are, what the internal structure of your offsider is and what the run-time behavious of your methods are.

You now have lots of useful information that you can use to make the application work as required. You can determine which parts are not running fast enough. You have lots of scope for design changes (large or small). You can replace object-oriented message passing with hard-wired file I/O, or calls to bare executables. You can cut back on using offsiders altogether. You can rewrite your entire applicaiton in machine code.

A word of warning, however. I still believe strongly that "premature optimisation is the root of ALL evil (in software development)". In other words, don't throw away the object-oriented message paradigm for no good reason. Only optimise if you have run the software and know for a fact that it runs too slow, and also you have profiled the software and know exactly where the bottleneck is.

The offsider framework was designed for power and flexibility. If you turn you back the object-oriented message passing, you turn your back on that power and that flexibility. That will have implications on your maintenance effort into the future. Make sure you get value for your effort.