Exupery's Direction
The long term design

Bryce Kampjes

Contents

1 Introduction

Exupery is a native code compiler for Squeak Smalltalk. The goal is to be nearly as fast as C, to provide a practical performance improvement and to be a widely used, it tries to produce as good code as possible for frequently executed methods where the extra effort spent compiling will pay off. It doesn't try to quickly compile the majority of code that is hardly executed. Exupery is written in normal unprivileged Smalltalk. The basic design principles are:

1. Dynamic inlining as the key send optimisation
2. A heavy duty optimiser (SSA)
3. The optimiser will make compilation too slow for stop and compiler operation
4. Combining an SSA optimiser and dynamic inlining should allow near C performance
5. Development costs must be kept down
6. Written in Smalltalk
7. Test driven development and incremental development

By combining dynamic inlining with a classical optimiser it should be possible to remove most of the overhead from Smalltalk's dynamic semantics. Dynamic inlining and type feedback provides a robust way to discover what types are actually being used and to remove the overhead from using clean well factored small methods. The classical optimiser will then be able to remove the exposed redundancy. Such a system should be able to get most arithmetic instructions inside loops down to 2 instructions (one for the overflow check). This should provide a system that delivers roughly C level performance for regular code on modern hardware. [1]

Dynamic compilers are built around a compromise. They require both fast run time and fast compile time. Doing both at the same time is hard. Exupery steps around this problem by relying on the interpreter to execute methods that are not compiled. This also avoids the issues created by writing the compiler in normal unprivileged Smalltalk.[2] Exupery is designed to compile only the methods that really need to be fast, not the code that's barely run because it's designed to work with the current interpreter rather than replacing it.

Adding a proper optimiser to a Self style compiler will make compilation much slower and the compiler more complex which changes the design parameters. Self stopped execution while compiling a method which makes the risk of long pauses unacceptable. To be faster than Self the system must tolerate slower compile times. The better optimiser forces the compiler to deal with slow compiles.

If Smalltalk was twice as fast as Self[3] then there would be no performance benefit to writing the compiler in C/C++. The compiler that produced faster code would also be slower and more complex than Self's due to the extra optimisation.

2 The Problem

The problem is trying to make Smalltalk as fast as possible. Smalltalk is slow because of frequent message sends, common higher order programming (do: loops etc), dynamic typing, and late binding. These are the same things that provide Smalltalk's power and flexibility.

Modern CPUs provide exceptional but not simple performance. There's often one or two orders of magnitude difference between the fast, hopefully common case, and the slow worst case. The problem is to eliminate the overhead of Smalltalk's flexibility when the flexibility is not being used while running on normal commodity hardware.

[1] Self published numbers claiming 1/2 the speed of C. Being sufficiently better to be interesting means 2-4 times faster. Speed differences will be different for other benchmarks though, chances are Self's floating point performance was much worse. Also C often executes at 1 instruction per clock so there's room for out of order machinery to hide a little safety checking overhead.

[2] There is a problem with breaking even on the time spent compiling, this can be worked around by saving compiled code

[3] roughly as fast as C

2.1 Smalltalk and Squeak

Smalltalk provides many powerful high-level facilities from using dynamically typed message sends to higher order functions to reflective access to the entire system.

Being able to build the development environment inside the developed system is wonderful but it does limit what optimisations can be performed because anything could be changed at any time. Luckily most code isn't changed frequently and uses the same types.

Squeak's object memory wasn't designed for exceptional speed. It had been designed for space efficiency in the mid '90s. The two key problems are using 1 as the integer tag and using a remembered set rather than card marking as the write barrier.

Having the integer tag as 1 rather than 0 means that integers must be detagged before use then retagged after use. This takes three instructions and adds two clock cycles of latency. Luckily in many cases the overhead can be optimised away.

The write barrier is used to record all pointers from old space into new space. This is used so that the generational garbage collector can collect new space without needing to scan all of old space. Squeak uses a remembered set write barrier which is an array which contains pointers to all objects in old space that contain pointers into new space. The problem is that only objects in old space with pointers into new space need be added so the write barrier code needs to check that the object being written to isn't a root, that it is in old space, that the object being written isn't a SmallInteger, and is in new space.

2.2 Modern Hardware

Modern CPUs are very fast but making efficient use of the speed isn't simple. A modern x86 CPU can execute three instructions per clock however optimised C only executes one instruction per clock. Memory access times vary from around 2-3 clocks from first level cache, to 10 clocks for second level cache, to about 100 clocks for RAM. Write bandwidth is also limited, a dual channel Athlon 64 can write 1 word every 4 clock cycles. A taken jump[4] can cost either a single clock if it's predicted correctly or 10-30 clocks [5] if it isn't.

A very big benefit of out of order CPUs is they allow some waste instructions to be hidden while the CPU is waiting for another resource. C programs execute about one instruction per clock cycle while the CPU can execute 3 instructions per clock (more with some RISCs). That leaves some spare room to hide, say, the jump on overflow check required for SmallIntegers to overflow.

Latencies still matter, an instruction sequence isn't going to execute in less cycles than it takes data to flow through the longest instruction sequence. The cost of detagging and retagging is two clocks of latency [6] while hopefully the 4 instructions to tag check can be hidden behind some other delay by the out of order machinery.

2.3 Sources of inefficiency

Smalltalk's inefficiency comes from the same features that makes it such a nice environment to develop in. Higher order programming and complete object orientation mean that methods are very small and a loop is often scattered across several methods and blocks. Because everything is an object all arithmetic and object accesses must tag check their arguments first. To provide both safe array access and efficient garbage collection all array accesses are range checked and all writes need to go through a write barrier.

1. Sends and Blocks
2. Tagging and detagging including type checks
3. Range checking and the write barrier

[4] A conditional jump which is taken. So execution continues at the jump target not the next instruction

[5] Athlon XP at 10 to a recent Pentium 4 at 30 with most other modern CPUs taking about 15 clocks

[6] This can be removed in many cases for addition and subtraction. If both arguments are variables then tagging and detagging can be reduced to a single instruction and clock cycle of latency

These three causes are the major reasons why Smalltalk isn't as efficient as carefully crafted C. Some of these costs are shared with other languages and environments which trade raw speed for flexibility. Smalltalk chooses flexibility, Exupery tries to avoid paying for it when it isn't being used.

When choosing optimisations there is often the choice between one optimisation which will be fastest for common cases and another that will speed up all cases. Ideally, the system should have both but that's costly in both development time and compile time.

3 Background

Exupery's design is heavily based on the first half of Appel's *Modern Compiler Implementation in C* combined with the Urs Holzle's work in self. The first half of Appel's book is a good current introduction to optimising compiler implementation and provides a nice minimal optimising compiler design. Urs Holzle's thesis covers how to use dynamic type feedback or inlining to convert a Smalltalk like language to something close to Pascal.

Self demonstrates how to compile a Smalltalk like language into something like Pascal. The key things are inlining driven by dynamic type feedback and careful choices for the object memory. For Squeak we can't change the object memory without breaking image compatibility.

Compilation pauses where a key issue dealt with in Holzle's thesis. The traditional way to avoid them is to use a fast compiling compiler which limits the amount of optimisation that can be done. Exupery relies on the interpreter to execute non-optimised code and doesn't stop execution while it's compiling which avoids introducing pauses.

4 Types of compiler

Exupery's design draws of three different schools of compiler design, Self and Smalltalk's dynamic compilers, mainstream optimisers, and Lisp compilers. The main influence of the Lisp compilers was that building heavily optimising compilers that are self hosting in an exploratory programming environment is possible and has been since the '70s. Self's work demonstrates how to inline even with Smalltalk's potentially very high levels of polymorphism. Mainstream optimisation techniques haven't been used with Smalltalk because without inlining there's very little to optimise and stop and compile compilers can't afford the potential pauses.

Self introduced dynamic inlining using a compiler that was in between the fast compile everything compilers and the slow compile well compilers.

Most of the algorithms in a slow compiler are $n * log(n)$ average case while some may be n^2 or n^3 worst case. The worst case normally only shows up for funky control flow graphs that are not reducible (means needs some hairy gotos). As Smalltalk lacks gotos it's likely to avoid even the theoretical possibility of the worst case.

The Deutsche/Shiffman compiler from '83 provided the first high performance commodity hardware Smalltalk engine. It was a dynamic compiler that was fast enough to compile everything as required. It used a code cache and discarded code that hadn't been used recently.

In '96 Self enhanced the dynamic compiler approach by adding dynamic inlining. They stayed with the stop and compile approach but needed to add a second faster compiler as their inlining compiler was too slow for infrequently used methods. Before Self, inlining was nearly impossible in Smalltalk as there wasn't a reliable way to figure out what senders to inline a send for because every thing is dynamically dispatched.

Inlining does two key things, first it removes most of the send overhead. It's common for Smalltalk to execute many sends for a small amount of work, second it exposes code to other optimisations. It's difficult to apply traditional optimisations to Smalltalk as most methods are too small to expose many optimisation opportunities.

Lisp compilers tend to be too slow to risk stopping execution. They don't tend to run in the background but can be used to compile at any time.

Dynamic compilers range from ones that compile quickly enough to compile everything before execution to compilers that are too slow to compile while stopping execution. VisualWorks and Mimic are fast compilers. Most Lisp compilers are slow compilers.

Aggressive optimisation is slow. Most optimisations are $n * log(n)$ however many optimisations can theoretically balloon to n^2 or worse. This is where Self's approach of stopping execution for compilation breaks down, dynamic inlining provides the opportunity to use an aggressive optimiser on Smalltalk but the architecture they've chosen stops them doing this because the optimiser would be too slow.

Lisp has had aggressively optimising compilers since at least the 70's but Lisp compilers don't rely on dynamic type feedback to inline.

The big question in Smalltalk language implementation is how to combine dynamic type feedback with an aggressive optimiser.

There's also a large body of work on writing optimisers for "traditional" languages. After inlining Smalltalk looks fairly similar to a GCed Pascal (C with safe arrays and pointers).

Dynamic compilers vary based on how quickly they compile:

1. So fast they can compile everything executed
2. Fast enough that they can stop execution to compile
3. Too slow to risk stopping execution

Self's optimiser is in the middle fast enough to stop execution but not fast enough to compile everything executed. VisualWorks is fast enough to compile everything executed.

5 Send and Block Optimisation

If it's possible to eliminate the cost of blocks and message sends then Smalltalk's performance will be close to that of Pascal (C with range checks) or low level Java (C with range checks and garbage collection). There are two sources of overhead from small methods, the sends can be expensive and frequent sends prevents traditional optimisation.

The main send optimisation planned for Exupery is dynamic method inlining. Send optimisation is critical for Smalltalk as the send count is so high. Inlining is required to create methods big enough to benefit from optimisation. It's very common in Smalltalk for a single loop to be spread across several different blocks and methods which makes it hard to optimise.

Inlining statically is hard in Smalltalk because it's difficult to know which implementor of a message to inline. Most static type inferencers haven't provided good accurate results for general cases even if they've done well on small benchmarks.

5.1 Full method inlining

Full method inlining is the planned way of speeding up sends and blocks. Full method inlining driven by type feedback as pioneered by Self is the best way to eliminate common sends and block calls. The advantage is only a type check is left for a general send and nothing needs to be left for a type check.

Full method inlining is the ideal solution for do: loops as it can collapse the entire loop into a single native method which can then be optimised by classical (SSA) techniques. Once a block is created and used inside the same inlined method it can be completely removed. Dynamic inlining is a fairly simple optimisation but it complicates debugging as it multiplies the number of states the system can get into. All a dynamic inliner does is profile then inline the most common sends. Inlining just involves compiling both methods then joining the two with a type check, jumps, and moves rather than a message send.

Inlining is so important because it enables other optimisations and it's hard because it makes debugging more complex. Even though inlining was part of the original vision for Exupery it will not be in the 1.0 release. It's not needed to provide a very useful compiler for Squeak and adding it later will be much easier once the rest of the compiler is well debugged.

Exupery uses a profiler based on Squeak's statistical profiler. This would have been good enough to drive dynamic inlining without PICs except for primitives. Primitives are not seen by the profiler so wouldn't be inlined.

Exupery already inlines primitives. This is driven by reading PICs. Primitives are easier to inline as they don't create their own context so don't require context remapping and deoptimisation. Primitive inlining and PICs were introduced to provide fast sends and #at:.

Dynamic inlining is likely to increase the opportunity to optimise because the inlined methods will be written to run in the general case rather than in their specifically inlined case. e.g. if a method is inlined into a loop then a calculation that is done once can be hoisted out of the loop.

5.2 Tuning the current send system

Sends in Exupery currently take about 120 clocks. The context creation and argument copying is done by a Slang helper method. By compiling the common case where the new context is recycled it should hopefully be possible to double the send speed.

Tuning the current send system has the advantage that it'll speed up all sends, not just sends that are inlined or stay inside a context cache. Major tuning will probably be left until after dynamic inlining is added.

5.3 Using a context cache

Other Smalltalks use a context cache which stores some contexts as native stack frames. If the context remains inside the cache then it will execute quickly however the context may need to be flushed into object memory if either it's referenced or if the cache overflows.

Context caches could be very interesting when used with method inlining. Inlining will produce optimised contexts that contain several source contexts, one for each of the inlined methods. These must be converted back to separate contexts if either method is changed or if the code cache is cleared. If optimised contexts only exist in the context cache then it will be quick to deoptimise them. The disadvantage is contexts will have to be inlined every time they are loaded then deoptimised every time they are spilled from the cache.

Dynamic method inlining should remove the overhead of common sends almost completely. It may also make optimisations like VisualWork's context cache unnecessary. The context cache speeds up sends while dynamic inlining removes them so if enough are removed then the performance of the remaining sends may not matter.

6 SSA - A Proper Optimiser

To compete with C Exupery needs a proper optimiser to remove unnecessary type checks and range checks and for standard low-level optimisations. C doesn't need type checking or range checking as it's unsafe. A proper optimiser will also be needed because all C compilers will be using one. Introducing a decent optimiser should allow Exupery to execute bytecodes 2-4 times faster than it does now which is enough to compete with C.

The two primary approaches available to increase Exupery's speed are either changing the object memory to be more efficient or by using a optimiser. Using an optimiser allows more general optimisations. An optimiser will allow all overhead to be removed for some important cases including counters in loops or for floating point operations on floating point arrays. Adding an optimiser will also change the gains available from changing the object memory.

To gain in performance over simple compilers for executing bytecode [7] a proper optimiser will be needed. Choosing an optimisation framework involves trading compilation time and complexity against run time performance. The plan has always been to use an heavy duty optimiser based on SSA

SSA adds a extra complexity initially as the intermediate will need to be transformed into SSA then back to regular intermediate afterwards. The main advantage is that many classical optimisations are trivial in SSA, so after adding only a few optimisations the compiler should be both faster and simpler than it would be without SSA.

[7] Bytecode performance is just the performance inside a single method. If a method has inlined methods in it then the entire inlined set of methods is what's optimised here.

SSA is a representation where every variable is assigned to once. This makes many optimisations trivial. To do common sub-expression elimination, just combine identical expressions with the same arguments. In a classical optimiser first a data flow analyser would need be run to find out which versions of each register are the same.

The first use for an SSA optimiser would be to move most of the #at:put: bytecode out of the loop. In Smalltalk the implementation must perform a type check, then range check the arguments. This is just a simple code motion optimisation. All that the optimiser needs to know is where the loops are and what values are constants during the loop.Fully optimising #at: or #at:put: would require:

1. removing the type check
2. moving the size look up out of the loop. (easier than the entire range check)
3. moving the range check out of the loop

Removing type-checks and range checks completely from the loop is a little more complex. Removing the type check just involves induction variable analysis but if the hoisted check (for the entire loop now) fails then we want the range check to fail for the same iteration as it would have for an unoptimised loop.

Removing type checks is simpler so long as the remaining type check dominates the removed ones. It isn't safe to remove a type check unless it's dominated by other type checks. The only trick with removing type checks is to push them into the re-entry code after an infrequent send [8]

Removing the tagging and detagging overhead across several statements would also be easy. The trick is to make sure that if an argument isn't of the expected type then the optimised code will still execute correctly. This should always be possible by splitting loops into a fast version and a slow version. The fast version can jump into the slow version if it encounters something it doesn't expect such as an object of an unexpected class or a SmallInteger operation overflowing.

Induction variable analysis should make it possible to move all of the range checking out of loops. Induction variable analysis just figures out how expressions like "count := count + 1" work. This then allows the optimiser to optimise the uses of count so it could range check an "#at: count" once for the entire loop.

An optimiser that could optimise away all the costs of #at: inside loops would have all the information available to begin vectorisation. Building such an optimiser would be a lot of work but with it, for optimisable loops, Smalltalk would be able to compete with any other language for raw speed.

Currently, the plan is to introduce SSA to the high level intermediate for #at: and tag checking optimisations. Eventually, Exupery may convert all intermediate up to the register allocator to SSA.

7 Dealing with a slow compiler

Exupery is designed to be slower than many competing approaches. That's unavoidable if you want to fully take advantage of the opportunities to optimise that dynamic inlining exposes. Is this a serious problem? Being slower does make recompiling because of changed use is more expensive but it may be possible to save compiled code to provide more time to amortise the compilation costs.

Exupery shouldn't be that slow, the algorithms are only $n * log(n)$ average case however they can blow out. Exupery compiles in a background thread rather than stopping execution. This means that only a later execution of the method will run the compiled code but execution is never stopped. It's still important to compile quickly to reduce the break even time but there is no risk in introducing very infrequent long pauses.

[8] A send to handle the unexpected case after an inlined operation. The most common case would be the send in an arithmetic operation if the arguments are not integers. Often the compile will have good reason based on the PICs to believe that the uncommon send is never performed.

It should also be possible to save compiled code with the image. By saving the compiled code or sharing it between images the chances of using it often enough to break even and gain time is greatly increased.

Another option would be to write a faster compiler to go beside the main slow but high quality compiler. This is what Self ended up doing with a much faster designed compiler. Without the colouring register allocator Exupery 1.0 should be fast enough especially if it's compiled by a decent compiler.

It's still important to keep compilation fast as the more time spent compiling code the longer it will take to break even due to time saved from the compiled code running faster. Most of the algorithms are $n * log(n)$ normally but some may be n^2 or higher worst case (which may never happen for Smalltalk).

For now, it seems reasonable to assume that if it's worth compiling it's worth compiling well. Exupery runs with an interpreter, it can not take over all execution so the interpreter can always cover for methods that are not executed enough to compile. It is possible to build a compiler than can compete (equal not better) with an interpreter for a single execution but the interpreter will be simpler. It may be that a fast compiler is still beneficial, but that decision can wait until after Exupery is running well with an interpreter for real measurements to be made.

8 Written in Smalltalk

Smalltalk is a nice language to write complex software in. It's powerful, expressive, and simple. If there's already enough complexity in the problem domain it's great not to have to deal with unnecessary extra complexity from your language.

Exupery is a hobby open source project, time is limited. Writing a decent compiler in any language is a large enough project even for a full time team. For Exupery to be practical, the software development effort must be kept to a minimum.

There are two strong reasons why Self and VisualWorks are not written in Smalltalk, compilation needs to be quick and the compiler must always be available.

If you're going to compile on demand then the compiler needs to be available at all times, it can't stop half way through compiling a method to compile itself especially if the method it was compiling is needed to compile. AOsTA works around this by disabling Smalltalk level compilation while compiling itself. Exupery compiles in the background, this both avoids circularity problems and pauses.

The code cache is managed from Smalltalk. This should make it easy to allow code to be saved then reloaded rather than recompiled. It should also allow easy experimentation with having a cluster of machines sharing compiled code say in a shared Magma database or via Spoon's replication.

The only code that isn't in Smalltalk is the interfacing to the interpreter and PIC population. PICs need to be filled with short sequences of machine code that compare the class then jump to the next compiled method, these sequences of machine code are generated by Slang methods but are very simple.

9 Incremental Development

You don't need to be right if you can change your mind. Any big problem can be solved if it can be decomposed into a sequence of smaller solvable problems. It is much easier to debug a single component when you trust the rest of the system, so much so that it can be easier to build a series of simpler components then throw them away than to build the entire system and have to debug it in one big go.

The major releases planned are:

1. The basic compiler
2. Dynamic inlining
3. SSA based optimiser

The basic compiler is something that's useful enough to build up a solid user community. It's not what Exupery is designed to be, merely a useful stepping stone to get there. It's the first point where rising to production quality is useful. By taking 1.0 up to production quality it will be possible to make many decisions about working with a slow dynamic compiler based on actual measurements.

Dynamic inlining doesn't look like a very large feature in isolation. The problem is it adds an extra set of variables to debugging. What's inlined now into what? This shouldn't be too bad to debug as Exupery already logs what it compiles and what it inlines. So, after a bug it's possible to recreate the code cache then debug with exactly the same compiled code.

The SSA optimiser will be much more useful after dynamic inlining as Smalltalk tends to have small methods with very little to optimise. Inlining is often required to create a big enough method to allow a classical optimiser to work with.

There is no implementation reason why SSA couldn't be implemented before dynamic inlining. This would be useful if the goal became to reduce the need for C by allowing hand inlined Smalltalk to run at near C speeds before adding dynamic inlining. It seems that optimising high level code is more useful especially as many of the promising low level methods have been replaced by hand written primitives already.

Writing a compiler is a tough engineering challenge, trying to write one that's significantly faster than all previous compilers in it's space is harder. I don't know how to do a detailed design for such a system but a high level overview isn't hard.

Many decisions interact. For instance the register allocator is used to remove the move instructions inserted to deal with two operand instructions, if the register allocator was changed to a different one that couldn't remove redundant moves another way then another phase would need to deal with 2 operand addressing. By working to make it easy to change decisions later it's possible to work out such integration issues when the implementation is developed enough to provide accurate measurements.

10 Test Driven Development

Test driven development provides three things, a design discipline, the ability to work in small chunks of time, and regression testing to keep bugs out when code is later modified. Compiler bugs are a pain and can consume a lot of time.

Tests have several uses in Exupery:

1. Allowing part time development
2. Preventing regressions
3. Decomposing and driving the design

One of the harder parts of writing a compiler as a hobby is developing a complex piece of software with short pieces of time. For a hobby project it must be possible to make progress in a few hours. Having 2 days in a row when you can focus on it is a rare luxury. Tests are great at getting back up to speed in minutes.

Bugs are a pain. Having solid unit and system test suites helps reduce the chances of introducing bugs. The unit tests are small and quick to debug. The system tests compile a method or two then run them checking their correctness.

It's possible for a change to break most of the system test suite (say it broke the method prolog which all methods use) but rare for a change to break many unit tests. Sometimes it's necessary to leave many system tests broken while relying on the unit tests and vice versa. If a design change is made that breaks many unit tests then the system tests can be used to make sure that it produces correct if different code before fixing the broken unit tests.

Besides the unit and system test suites there is also pre-release testing. The pre-release tests currently consist of the stress test and running the background compiler. The stress test runs almost all the test classes in the system, compiles the top methods found by the profile, runs the test class again, compiles any methods that use an inline-able primitive, then runs the test suite one more time. The goal is to run a variety of code, not written by me, exercising common

inner loops. Exupery relies the stress test and running the background compiler for "real world" testing.

The stress test runs almost all the test classes in the system three times. It first runs it profiling, then it runs them after compiling the most frequently used methods, then it runs them after also inlining any primitives called by a native method. This provides a reasonably though test of the system.

Running the background compiler tests Exupery as it is intended to be used. It'll cover cases where almost all the frequently used methods will be compiled and here there's thousands of compiled methods.

The advantage of both of these forms of testing is they are good at flushing out bugs and deciding on whether Exupery is currently releasable and how usable it currently is. They are good at guiding decisions about whether stability, compilation speed, or run time speed are currently the most pressing.

10.1 Debugging to Tests

The problem with both of these approaches is that the bugs they find can be hard to debug.

To make it easier to debug real bugs Exupery can log every method compiled including what's inlined and selectively replay these logs. This makes it possible to reproduce the code cache at the point where the bug occurs then to slim down the number of methods that need to be compiled to produce a bug.

The first stage in debugging is to get a reproducible error. Normally, this just involves replaying the logs which will work if the bug was triggered by the stress test or during compilation. Once there is a reproducible bug it's time to reduce the reproduction down to something that can be added to the system tests.

Reducing the bug reproduction begins by removing compiled methods until the bug disappears or becomes intermittent. Often bugs caught at this stage are intermittent. Once the bug's become intermittent it's time to find a good way to reproduce it reliably. Often this will just involve running something that reliably reproduces the bug many times such as compiling a method but not registering the code (so it's not compiled in the final reproduction). With a reliable way to reproduce the bug it's possible to further reduce the size of the reproduction down to hopefully one or two compiled methods. This small reproduction can then be turned into a test which will be used to debug the problem.

11 Where Exupery is now

The current compiler can run in the background, it can dynamically inline primitives but not full methods. It has generated and should generate bytecode that's faster than VisualWorks [9]. It's about 7 times slower than VisualWorks for sends but over 2 times faster than Squeak.

Exupery currently runs in the background but not well enough to be used. It doesn't have either SSA or dynamic inlining yet. My goal is getting Exupery to 1.0 as quickly as possible. There are three areas where Exupery needs improvement before a 1.0:

1. Reliability - bug fixing
2. Compile time performance
3. Run time performance

The current compiler relies on simple tree rewriting except for the register allocator which is a complex colouring coalescing allocator. Tree rewriting can do a surprisingly good job quickly and simply. It's good enough to get to about 1/2 C's speed on modern hardware for the bytecode benchmark.

Exupery already has PICs and can dynamically inline primitives. PICs are required for primitive inlining as the Smalltalk profilers can not see primitives. PICs also allow the 1.0

[9] It's got a better but slower compiling back end. The register allocator gives it the edge

release to comfortably beat Squeak's send performance. Being useful enough to be widely used is the goal of 1.0.

12 The Register Allocator - The first slow stage

The register allocator is the first, and currently only, slow compiling stage. It's probably too slow and complex for the 1.0 compiler which otherwise only uses fast simple optimisations, but it's the right choice for a highly optimising compiler. Replacing the register allocator with a simpler faster design would cause the generated code to slow down but would also speed up compilation. The register allocator will not look slow in an optimising compiler as the optimiser will be at least as slow.

The register allocator provides two key benefits even in a system without a strong optimiser. First it generates faster code by providing better register allocation and removing redundant moves. Second it hides more of the x86's details from the front end by removing moves used to simulate three operand instructions. Hiding the complexity of the x86 from the rest of the system covers the complexity of the allocator even in the simple 1.0 system.

Register allocation works like this:

1. Perform liveness analysis
2. create interference graph
3. colour interference graph
4. either add spill code and go back to liveness analysis or convert registers to coloured values

This is basically a single iterated optimisation. Liveness analysis is a data flow analysis which is typical of classical optimisations [10]. Colouring the interference graph should be efficient. Adding spill code or changing the registers to their coloured values is linear.

The iteration should only happen once or twice. If the iteration fails then the registers that failed to find a colour are spilled into memory so will not cause problems next time. If the allocator iterates many times then there's a problem with the spill heuristics as it's spilling registers that are not helping.

13 The Future

The next big feature after 1.0 is likely to be either full method inlining to eliminate the costs of common sends or an SSA optimiser which should allow bytecode performance to increase 2-4 times. The choice is between optimising high level code that's send heavy or optimising hand inlined Smalltalk inner loops.

Exupery is written in normal Smalltalk so it can be changed or extended by anyone just like any other part of a Smalltalk system. This allows a high performance library writer to modify any part of the compilation process or add specialised optimisations anywhere.

While running normal Smalltalk at or close to the speed of C should be possible for most programs would be a major achievement it should be possible to beat C's performance in many applications where speed really counts by having high performance libraries that extend Exupery.

Another area where Exupery has potential is for floating point calculations. By combining dynamic type feedback with static program analysis it should be possible to eliminate floating point boxing and deboxing from calculations. If the floats are stored in float arrays then they may never need to be boxed and so run at full hardware speed.

Once Exupery has the machinery needed to fully eliminate the overhead of array access inside loops then it's got most of the machinery needed for vectorisation. Once a compiler is vectorising it will be transforming enough that performance is compiler verses compiler rather than language verses language.

[10] Modern optimisers often convert to SSA which doesn't require data flow analysis as it's represented in the graph

14 Conclusion

It should be possible to have Smalltalk run at nearly the speed of C. Close enough to be as fast for most real programs. [11]. Almost all of the overhead can be removed or hidden behind unavoidable hardware delays.

This is not true for programs with heavy array access outside of loops or that execute a very large volume of code without any real hot-spots to inline. However such programs are likely to perform badly on current hardware.

Smalltalk can not be as efficient in all cases, but it can be very close to the speed of C often and when it can't there's a good chance that the CPU's optimisations are breaking down and slowing things down enough to hide the remaining costs from the language.

[11] There's reason to believe it already is and has been since the 80's for many real programs. Real programs are rarely fully optimised