Exupery
A Native Code Compiler for Squeak

Bryce Kampjes

Contents

1 Introduction **2**

2 The Problem **2**
2.1 Smalltalk and Squeak . . . 2
2.2 Modern Hardware . . . 3

3 Background **3**

4 The Current System **3**
4.1 Overview . . . 3
4.2 Phases . . . 4
4.3 Intermediate Languages . . . 4
4.4 Intermediate Generation . . . 4
4.5 High level intermediate . . . 5
4.6 Intermediate simplification . . . 5
4.7 The Instruction Selector . . . 6
4.8 Register allocation . . . 6
4.9 Assembler . . . 6
4.10 Code Loading . . . 7

5 The Big Design **7**
5.1 Sources of inefficiency . . . 7
5.2 Send and Block Optimisation . . . 8
5.2.1 Full method inlining . . . 8
5.2.2 Tuning the current send system . . . 8
5.2.3 Using a context cache . . . 8
5.3 Bytecode optimisation . . . 8

6 The Future **9**

7 Summary **9**

8 Appendix - The bytecode benchmark **10**
8.1 The benchmark . . . 10
8.2 The code for the marking loop . . . 10
8.3 Analysis of the loop . . . 12

9 Acknowledgements **14**

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 have a widely used project. Exupery is written in normal unprivileged Smalltalk. The basic design principles are:

1. Write the hard stuff in Smalltalk
2. Don't try and compile everything
3. A slow running compiler producing fast code
4. Don't compile in-line(don't cause pauses)

Exupery is a hot-spot compiler, 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 quickly the majority of code that is hardly executed.

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 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).

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. 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.

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), and dynamic typing (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.

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[1] can cost either a single clock if it's predicted correctly or 10-30 clocks (Athlon XP at 10 to a recent Pentium 4 at 30) 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 [2] while hopefully the 4 instructions to tag check can be hidden behind some other delay by the out of order machinery.

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 The Current System

4.1 Overview

The current system is 15% faster than VisualWorks for bytecode performance and twice as fast as the Squeak interpreter for sends.

It relies on simple tree based optimisations and a colouring coalescing register allocator. The register allocator serves to hide much of the x86's differences from the rest of the compiler.

The compiler is currently a very simple optimising compiler. The only heavy optimisation is the colouring register allocator which involves an analysis phase.

The stack is translated into direct register access but temporaries are always accessed via the stack frame. This is inefficient but only the arithmetic loop benchmark would really gain by moving temps into registers, even the bytecode benchmark has enough overhead from #at: and #at:put: to hide the cost of memory access for temps.

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

[2] 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

4.2 Phases

Exupery has the following phases which are executed by Exupery»run:

```
run
        machine compiledMethodNumber: (Exupery registerCompiledMethod: method ).
        self generateIntermediate.
        self treeOptimiser.
        self findBlocks.
        self convertIntermediate.
        self lowLevelOptimiser.
        self selectInstructions.
        self allocateRegisters.
        self removeJumps.
        self assemble.
```

4.3 Intermediate Languages

Exupery's intermediate languages are tree and register based. Optimisations are carried out by tree rewriting. This makes them simple but limits their power. However besides the register allocator all other optimisations are fast due to their simplicity.

An advantage of using simple optimisations is they rely on getting the representations right. For instance the high level intermediate was introduced to allow a simple tree traversal optimiser to easily remove Boolean boxing and unboxing in conditional expressions and branches.

4.4 Intermediate Generation

```
add: a to: b
        ^ a + b
```

The method above is represented by the bytecodes below

```
5 <10> pushTemp: 0
6 <11> pushTemp: 1
7 <B0> send: +
8 <7C> returnTop
```

Which are converted into the following high level intermediate:

```
(MedMethod
(block1
  (createContext)
  (return
    (label
      (or 1
        (jo block2
          (add
            (integerValueOf t1 block2)
            (integerValueOf t2 block2))))
      (send
        (mem (add (mem (add (mem specialObjectsOop) 96)) 4))
        (t1 t2)
        7)
     (block2))))
```

Intermediate generation is probably the largest phase [3] but luckily it's logically simple.

Intermediate generation is a big tree expansion. It starts with the **ByteCodeReader** which is responsible for converting the bytecodes back into something like an abstract syntax tree. It first finds all jump targets and the stack heights at each target with the **JumpTargetRecorder**. The **ByteCodeReader** then rereads the method's bytecodes using the **IntermediateGenerator** and **IntermediateEmitter** to generate high level intermediate.

4.5 High level intermediate

The high level intermediate is used both to do a little semantic analysis and to remove common forms of duplication between bytecodes. Most action bytecodes first detag or debox their arguments, operate on them, then retag or rebox the arguments. For integer operations this isn't too bad as tagging and detagging only take a few instructions each but the operation itself is usually a single instruction so it is wasteful. But for Booleans [4] the overhead is much heavier involving several basic blocks and branches. The high level intermediate makes it easy to walk over the tree removing boxing and deboxing.

The high-level intermediate is also used for some semantic analysis. Currently it's the level where basic blocks are divided into those that are used for Smalltalk blocks and those that are executed as part of the method itself. The distinction is critical because blocks need to access their home context to get their temporary variables. The block finding code can't deal with jumps so they are removed by the tree optimiser.

4.6 Intermediate simplification

```
(or 1
   (jo block2
      (add
         (integerValueOf t1 block2)
         (integerValueOf t2 block2))))
```

The high level intermediate fragment above is converted into:

```
  (mov (mem (add currentContext 28)) t1)
  (mov (mem (add currentContext 32)) t2)
  (mov t1 t6)
  (jnc block2 (bitTest t6 0))
  (mov t2 t7)
  (jnc block2 (bitTest t7 0))
  (mov
    (or 1
      (jo block2
        (add (sub t6 1)
        (sub t7 1))))
    t5)
  (jmp block5)
```

Intermediate simplification translates high-level intermediate into low-level intermediate. The low level intermediate is just below assembly with each operation representing a basic operation. Assembly instructions are made up of several low level intermediate operations. The low level intermediate is used for both optimisation and also to feed into the instruction selector.

[3] Except for possibly register allocation

[4] Boxed floats are much worse as the actual operation is a single floating point instruction without an overflow check because Squeak uses NaNs rather than floating point exceptions for overflows. Boxing floats is a very expensive operation because a new object needs to be allocated.

The low level optimisations remove common redundant sequences around SmallInteger addition and subtraction and in compare instruction sequences. These sequences depend on what the arguments are, if either argument is a constant then constant removal will remove more instructions.

4.7 The Instruction Selector

```
(mov (28 currentContext) t1)
(mov (32 currentContext) t2)
(mov t1 t6)
(bitTest 0 t6)
(jnc block2)
(mov t2 t7)
(bitTest 0 t7)
(jnc block2)
(mov t7 t29)
(sub 1 t29)
(mov t29 t30)
(add t6 t30)
(jo block2)
(mov t30 t5)
(jmp block5)
```

The instruction selector algorithm is a simple maximal munch algorithm as described in Appel's book. Currently, the maximal munch algorithm leads to reasonable code quality. There is still plenty of room to improve the instruction selection by using more complex addressing modes without needing to change the algorithm to a more complex one.

4.8 Register allocation

```
(mov (28 ecx) eax)
(mov (32 ecx) edx)
(bitTest 0 eax)
(jnc block2)
(mov edx ebx)
(bitTest 0 ebx)
(jnc block2)
(sub 1 ebx)
(add eax ebx)
(jo block2)
(jmp block5)
```

The register allocator is primarily responsible for assigning machine registers to the temporary registers used so far. It allows the front end of the compiler to pretend there are an infinite number of registers.

The register allocator reduces the instruction size by removing unnecessary moves. It may also add spill code that shuffles values out of registers and into memory if there are not enough hardware registers.

4.9 Assembler

The assembler is again simple. It just applies similar pattern matching to the instruction selector [5] to produce machine code from the assembly instructions. The output is a byte array containing

[5] It would be nice to have a single definition for instruction formats which could be used to drive both the instruction selector and the assembler. The current system isn't causing enough problems to justify changing it.

the machine code representation of the method.

One consideration is how to arrange the rules for instruction selection. This is probably less of an issue on a RISC. The key to having tolerable code for x86 assembling was to add encoder objects. They just represent the different encodings which x86 instructions can have.

The x86 instruction encodings are surprisingly regular with only a few different encoding patterns. The problem is every instruction supports a different set of encoding patterns. The **AssemblerExupery** class is the core of the assembler and the encodings have been factored out into separate **Encoding...** classes. Assembling an instruction is done by trying each encoding available until one knows how to assemble it.

For a RISC, having a simple tree traversal/visitor assembler would probably be adequate.

4.10 Code Loading

The generated machine code is then copied into a code cache and prepared for execution. Some references are relocated because their address isn't known until after the method's location is known.

Once the code is executable, it's then registered with the support machinery so it'll get executed. The method is added to a dictionary that maps classes and selectors to natively compiled methods. The dictionary is used to execute natively compiled methods instead of interpreted methods.

5 The Big Design

Exupery has always been intended to be nearly as fast as C. This, I believe, is achievable by combining dynamic type feedback[6] with a global SSA [7] based optimiser.

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.

5.1 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

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.

[6] full method and block inlining

[7] SSA is an intermediate form that makes many optimisations very simple. Every value is statically assigned only once. With SSA common sub-expression elimination just involves combining identical expressions. Code motion would just involve moving expressions out of a loop when all the arguments came from outside the loop.

5.2 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.

5.2.1 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 now be optimised by classical (SSA) techniques. Once a block is created and used inside the same inlined method it can be completely removed.

5.2.2 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.

5.2.3 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.

5.3 Bytecode optimisation

To compete with C Exupery needs a 2-4 times speed increase in bytecode performance. To compete for all cases it will need to eliminate all the overhead; however for most programs there are enough delays to hide some waste from type checking while still executing as fast as 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.

The first use for an SSA optimiser would be to move most of the cost of #at:put: out of the loop. 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.

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 vectorising. 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.

6 The Future

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.

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.

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.

7 Summary

It should be possible to have Smalltalk run at nearly the speed of C. Close enough to be as fast for most real programs. [8]. 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.

[8] 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

8 Appendix - The bytecode benchmark

This appendix analyses an inner loop from the bytecode benchmark. The bytecode benchmark is a prime number sieve that's been used since the 70s to measure the raw bytecode performance of Smalltalk engines.

8.1 The benchmark

```
1  benchmark  "Handy bytecode-heavy benchmark"
2          "(500000 // time to run) = approx bytecodes per second"
3          "5000000 // (Time millisecondsToRun: [10 benchmark]) * 1000"
4          "3059000 on a Mac 8100/100"
5      | size flags prime k count |
6      size := 8190.
7      1 to: self do:
8          [:iter |
9          count := 0.
10         flags := (Array new: size) atAllPut: true.
11         1 to: size do:
12             [:i | (flags at: i) ifTrue:
13                 [prime := i+1.
14                 k := i + prime.
15                 [k <= size] whileTrue:
16                     [flags at: k put: false.
17                     k := k + prime].
18                 count := count + 1]]].
19     ^ count
```

8.2 The code for the marking loop

The code below is just for lines 15 to 17 in the benchmark. It only includes the integer loop, it does not show the send code if a type check or overflow check fails.

The loop from 11 to 18 is very similar. The major differences are that loop uses #at: rather than #at:put: so doesn't have the write barrier code and it has more integer arithmetic.

Useful Values	
(-28 ebp)	the active context
(28 activeContext)	size
(32 activeContext)	flags
(36 activeContext)	prime
(40 activeContext)	k
(-16 ebp)	a temp containing the stored value (false)

block6

	[k <= size] whileTrue:
(mov (-28 ebp) ebx)	
(mov (40 ebx) eax)	Load k
(mov (28 ebx) ecx)	Load size
(bitTest 0 eax)	Type check k
(jnc block20)	
(bitTest 0 ecx)	Type check size
(jnc block20)	
(cmp ecx eax)	Compare and
(jg block5)	conditionally jump back to the main loop

block65

	[flags at: k put: false.
(mov (-28 ebp) ebx)	Load activeContext
(mov (32 ebx) eax)	Load flags
(mov (40 ebx) ebx)	Load k
(mov falseObj ecx)	
(mov (ecx) ecx)	
(mov ecx (-16 ebp))	
	The #at:put:
(bitTest 0 eax)	Type check the receiver
(jc block22)	
(mov 126976 ecx)	
(and (eax) ecx)	
(mov 12288 edx)	
(cmp ecx edx)	
(jne block22)	
(mov (-16 ebp) ecx)	Check the index is an integer
(mov ecx (-32 ebp))	
(bitTest 0 ebx)	
(jnc block22)	
(mov 3 ecx)	and at least one
(cmp ebx ecx)	
(jg block22)	
(mov (eax) ecx)	Find the size of the array
(mov ecx (-24 ebp))	
(mov 3 ecx)	
(mov (-24 ebp) edx)	
(and edx ecx)	
(jz block79)	
(mov 252 ecx)	
(mov (-24 ebp) edx)	
(and edx ecx)	
(sar 2 ecx)	
(jmp block80)	

block79 The array was big so the size is in a header word

(mov eax ecx)	
(sub 8 ecx)	
(mov (ecx) ecx)	
(sar 2 ecx)	

block80

(mov ebx edx)	
(mov edx (-20 ebp))	
(mov (-20 ebp) edx)	
(sar 1 edx)	
(mov edx (-20 ebp))	
(cmp ecx edx)	The range check itself
(jge block22)	
(sub 1 ebx)	Calculate the address to store into
(sal 1 ebx)	Better addressing mode support will allow this to be done in the store
(add eax ebx)	
(mov (-32 ebp) ecx)	
(mov ecx (ebx))	The store for the #at:put into the array The write barrier

	(bitTest 0 ecx)	
	(jc block81)	
	(mov youngStart ecx)	
	(mov (-32 ebp) ebx)	
	(cmp (ecx) ebx)	
	(jl block81)	
	(mov youngStart ebx)	This move is here because Exupery doesn't yet support SIB addressing modes.
	(cmp (ebx) eax)	
	(jge block81)	
	(mov (eax) ecx)	
	(mov 1073741824 ebx)	
	(and ecx ebx)	
	(jnz block81)	
	(mov rootTableCount ebx)	
	(mov (ebx) ebx)	
	(add 1 ebx)	
	(mov rootTableCount edx)	
	(mov ebx (edx))	
	(sal 2 ebx)	
	(add rootTable ebx)	
	(mov eax (ebx))	
	(mov 1073741824 ebx)	
	(or ecx ebx)	
	(mov ebx (eax))	
block81		
	(mov (-32 ebp) eax)	Move the stored value into a register in case the answer of the #at:put: is used. In this case it's not.
	(jmp block78)	
block78		
		k := k + prime
	(mov (-28 ebp) ebx)	
	(mov (40 ebx) eax)	
	(mov (36 ebx) ecx)	
	(bitTest 0 eax)	This add sequence is near optimal
	(jnc block23)	
	(bitTest 0 ecx)	
	(jnc block23)	
	(mov ecx ebx)	This move is redundant here
	(sub 1 ebx)	removing a tag
	(add eax ebx)	
	(jo block23)	
block87		
	(mov (-28 ebp) eax)	Store k
	(mov ebx (40 eax))	
	(jmp block6)	

8.3 Analysis of the loop

The code above definitely has plenty of room for improvement. VisualWorks is only 2.4 times slower than C for this benchmark and Exupery is faster than VisualWorks. If Exupery was three times as fast as it currently is then it will be as fast as C.

It's interesting that C is so slow. Without measuring, I'd guess that C is slow due to either a

memory write bandwidth bottleneck or branch mispredicts.

Instruction counts for the whole loop	
Instructions	Description
42	Executed #at:put: instructions
23	Unexecuted #at:put: instructions
11	Loading and storing temporaries
8	SmallInteger type checks
3	Loading false
3	Unconditional jumps
3	Real work instructions. The cmp and jg from "k <= size" and the add from "k + prime"
1	detagging for the add
1	overflow checking the add
1	move because the x86 only has 2 operands
66	**total executed**
96	**total instructions**

Two thirds of the executed instructions are going into #at:put:. Out of the 66 instructions executed only 5 are absolutely necessary. How easily can this be sped up and what techniques will be necessary?

The two different operations in this loop are the #at:put: and the integer arithmetic on the loop counter. Most of the overhead of both does not need to be performed for each iteration.

Instruction counts for the #at:put:	
Instructions	Description
6 (+ 18)	write barrier
16 (+ 5)	getting the array's size (5 in optional branch)
2	the actual range check
7	type checking the array
4	address calculation (to store into)
1	the move
6	instructions checking the index is an integer greater than 0.
42	**total executed**
65	**total instructions**

Code motion would allow us to move the write barrier out of the loop. It would also allow the array's size to be calculated before the loop. The address calculation could be reduced to 1 instruction by using more complex addressing modes. The code to figure out the array's type could also be moved out of the loop just leaving a compare and jump. The compare and jump needs to stay unless the loop is split so that it still behaves the same even if the receiver isn't an Array. Combined this removes 30 instructions from the 42 executed reducing it to 12 instructions relying on just code motion and minor tuning.

If we split the loop into a fast version and a slow version then we can remove the remaining 2 instructions checking the array. By using induction variable analysis as well it will be possible to remove the 6 instructions that check the index is a SmallInteger greater than 0. This will leave only 4 remaining instructions.

The integer operation in blocks 78 and 87 breaks down into 4 instructions integer type checking, 5 instructions moving temporaries into and out of registers, and 4 doing the addition including tag checking. Moving temporaries into registers will remove the 5 loading and storing instructions. The type check and tagging instructions should be able to be removed from the fast version of the loop. The redundant move should go with a bit more register allocator tuning. That just leaves two instructions, the add itself and an overflow check. Induction variable analysis should allow the overflow check to be removed as well.

A basic SSA based optimiser that just removes type checks from integer statements in loops and moves the loop constant parts of a #at:put out of the loop may be enough to compete with C for this benchmark. Such a optimiser will allow many more optimisations to be easily developed if the two basic optimisations are not enough. The Smalltalk code however would be executing many more instructions to do the same basic work. More optimisations will eliminate most of

this extra work and allow more programs to run at near C speeds.

References

[1] May, C. *Mimic: a fast System/370 simulator.*, 1987 Symposium on Interpreters and Interpretive techniques.

[2] Deutsch, L. Peter, Schiffman, Allan M. *Efficient Implementation of the Smalltalk-80 System*, Principles of Programming Languages 1983

[3] Appel, Andrew, *Modern Compiler Implementation in C*, 1998, Cambridge University Press

[4] Holzle, Urs, *Adaptive Optimization for Self: Reconciling high performance with Exploratory Programming*, 1994, PhD Thesis

[5] Chaitin, M.A., *Register allocation via coloring*, 1981, Computer Languages 6, pp. 47-57

[6] Briggs, P., *Register Allocation via Graph Coloring*, 1992, PhD from Rice University

[7] George, L. and Appel, A. W., *Iterated register coalescing*, 1996, ACM Trans. on Programming Languages and Systems 18(3), pp. 300-324

[8] Ingalls, D., Kaehler T., Maloney, J., Wallace, Kay, A., *Back to the Future, The Story of Squeak, A Practical Smalltalk Written in Itself*, Squeak Open Personal Computing and Multimedia

9 Acknowledgements

Many thanks to the useful criticism of the people at the 2006 Brussels Smalltalk Party especially Peter van Rooijen, Goran Krampe, and Cees de Groot. Thanks also to Ken Causey for proof reading the draft.