A Sketch for an Adaptive Optimizer for Smalltalk written in Smalltalk

Eliot Miranda, Paolo Bonzini, Steve Dahl, David Griswold, Urs Hölzle, Ian Piumarta, David Simmons

A Sketch for an Adaptive Optimizer for Smalltalk written in Smalltalk

Eliot Miranda
with contributions from Paolo Bonzini, Steve Dahl, David Griswold, Urs Hölzle, Ian Piumarta and David Simmons.
October 2002

Change History:
[1] 9 Nov 2002 eem
Added description of intercession of sends during development phase.
Added handling of doesNotUnderstand: entries in send data.
[2] 3 Dec 2002 eem
Extended discussion of profiling alternatives. Changed spec of send data primitive to include loop and/or basic block counts.
Added no-check inst var store bytecode.
[3] 27 Jan 2003 eem
Changed callbacks to eliminate pc parameter. Clarified performance of conditional counters. Noted that the sketch is indeed real and partially implemeted for HPS. Described the VW registration facility. Added a mod bytecode. Added ikp's dependency management suggestion.

1. Adaptive Optimization In Smalltalk

This is a design sketch for an adaptive optimizer implemented in Smalltalk above a conventional "jitting" virtual machine with "minor" extensions. Adaptive optimization as we are discussing it here was invented by Urs Hölzle and is described in [Hölzle94]. The sketch is far from complete, focussing instead on the interface between the optimizer and the virtual machine, hence outlining a potentially portable architecture where an optimizer written in Smalltalk can be hosted above a range of specific virtual machines. This portability is intended to allow us to collaborate on the project without having to define and implement a common vm, with all the difficulties of migrating current systems to a new VM, allowing us to apply the optimizer within our current systems. Of course, this architecture still requires significant extensions to the execution machinery of extant VMs but these extensions amount to something far from a rewrite.

To ground the sketch in an extant VM the sketch will be specific to HPS, the VisualWorks virtual machine, which is a second generation implementation of Peter Deutsch's classic PS Smalltalk described in the Deutsch-Schiffman paper "Efficient Implementation of the Smalltalk-80 System". I'm only intimately familiar with this VM and the BrouHaHa bytecode and threaded code VMs, and I need to base the sketch in HPS to make it as real as possible. I do expect the sketch to apply more broadly but it may not. I also assume that you all won't be bound by the sketch and will freely propose alternatives as well as extensions. The sketch is current;ly functioning as a specification for the HPS implementation. [3]

My terminology will be erratic. I prefer the term "dynamic translation" for the compilation of bytecode to executable machine code to "jitting", but am too lazy to type it. I shall use terminology from HPS design documents where V-methods (V for "virtual") are image-level bytecoded methods, and the code contained in them is called V-code. The executable form of a method is called an N-method (N for "native"), and the code in an N-method (which is assumed to be native code, but might conceivably be threaded code) is called N-code. Spelling will reflect my increasingly

schizophrenic confusion of U.S. and English norms. (* N) marks a footnote.

OK then.

2. Introduction

Adaptive optimization (AO) aims to apply a number of profitable optimizations dynamically as the program runs, guided by type information gathered by the system as it runs. Optimization is performed when execution profiling detects an expensive computation and causes the optimizer to be invoked. To preserve accurate debugging, and to assist subsequent re- or de-optimization, mapping information is included that permits optimized activations to be dynamically deoptimized into unoptimized ones. Let's describe these four components briefly.

2.1 Execution Profiling

The running program is instrumented with some form of execution profiling. It may be based on periodic sampling of the program counter, or it may be based upon counters planted in the code. The thesis describes invocation counters incremented in the prolog of a method (termed node counters since they count the nodes in the call graph), which we shall assume are used and are augmented with loop counters. On every method invocation or backward branch a counter is incremented and if it reaches some settable limit the optimizer is invoked.

The assumptions are that a counter will either trip long before a computation invoking it completes, or soon enough before it is repeated that optimization will pay back more than it takes to apply, and that the computation is actually optimizable. The thesis presents algorithms for ameliorating situations when these assumptions are invalid.

Explicit invocation, conditional and loop counters turn out to be expensive but affordable in HPS, where Smalltalk-intensive computational benchmarks slow down by as much as 25% on contemporary hardware, depending on the counting scheme. This is unsurprising; adding a read-modify-write to each procedure call is a large overhead. But the assumptions are that the program will spend the majority of its time running optimized code and that this optimized code will spend much less time tripping counters because, for example, optimized code will eliminate loop counters and only count on invocation. Further, research in Jikes [Arnold01][Arnold02] shows how hybrid sampling and explicit profiling approaches can provide accurate data more cheaply. [2][3]

The advantages that explicit counters have over statistical profiling is that they are predictable, making the optimizer synchronous and hence more easily debuggable, and that they are reasonably fine grain. Send-site counters (termed edge counters because they count the edges in the call graph) would be more accurate since two different send sites that invoked the same target method would have distinct counts, but code density would be far worse. Experience with the Jalapeño VM shows that edge counters can significantly out-perform node counters [Arnold00]. [Arnold02] points out that accurate edge counts can be inferred from conditional branch counters. [2]

2.2 Type Information

The type information available to the optimizer is that present (or absent) in the in-line caches of sends in the executable code for the language. Excuse me getting concrete here but we have gone a little further than the original PIC paper and I think its worth describing.

In the HPS implementation when a method is first jitted the code generated for a send is the load of a register with the message's selector followed by a call on some run-time lookup routine.

When first executed the lookup routine will be invoked, and will lookup the message selector in the receiver's class and rewrite the send site to load the register with the receiver's class and to call the cache checking entry-point of the method found in the lookup. The lookup routine will then jump to the prolog of the method proper.

When subsequently executed the send will directly call the cache checking entry-point which will extract the current receiver's class and compare it against that loaded at the send site. If they match control continues to the method's prolog. If they don't match the send site will be rewritten to call a polymorphic inline cache (PIC).

A PIC is a jump table that extracts the current receiver's class and does a series of compare-with-constant, conditional jump, where the constants are classes and the conditionals jump to the prologs of the relevant methods on a match. Typically PICs are fixed size, initially having two cases, the original target of the send and the new method that was found in the new receiver's class when the monomorphic send failed. If control falls through the type cases a call is made to code that adds a new type case to the PIC if there is room. In HPS PICs have up to 8 type cases. These I'll call "Closed" PICs because they support up to some closed set of classes.

In the original PIC paper [Hölzle91] once a PIC was full falling through resulted in a conventional message lookup (e.g. hash into the first-level method lookup cache and then walk the class hierarchy). In HPS falling-through instead causes the run-time to replace the send site's call on of the closed PIC with a call to an "Open" PIC, which is a code sequence that does a hash table lookup of the first-level method lookup cache, calling the run-time if it fails. "Open" PICs then handle "megamorphic" send sites. (* 1). During normal execution our system typically has about 40% of sends un-taken, and of the remaining 60%, 90% are monomorphic send sites, 9% are polymorphic and 1% are megamorphic.

The insight that concludes the original PIC paper is that the system collects precise concrete type information in the inline caches at send sites. This information can be used by an optimizer to generate more efficient code for that which execution profiling guesses is worth optimizing.

2.3 Adaptive Optimization

2.3.1 What scope to Optimize?

Using invocation counters to choose when to optimize and in-line cache data to know what's what the system can then apply whatever profitable optimizations apply to the code in question. "What code to optimize?" is an interesting question. Invocation counters trip at some point in the execution of a program and the optimizer is then able to examine the stack of activations below this point and the methods invoked from these activations' methods.

In Urs' thesis there is discussion of the difficulty in defining heuristics that chose suitable units for optimization. Invocation counters directly identify high-frequency callers and long loops. Not looking further up the stack, and simply optimizing up to the callee might waste time in the long run by requiring the system to reoptimize subsequently when the initially optimized code is invoked frequently. Looking further up the stack at their callers may enable further optimization within the callers and loops, e.g. by inlining a block invoked in a do: method within the caller of the do: when the block value: method trips its invocation counter.

But looking too far up the stack might result in the optimizer trying to optimize a collection of method activations so large that the compilation pause becomes noticeable, or might attempt to optimize too much, spending effort optimizing outer loops that only contribute small fractions to overall execution time. Urs' thesis describes heuristics such as following the static chain of a block to select the activation to attempt to optimize. But David Griswold informs us that experience with the Animorphic VM is that always choosing just the caller of the invocation count results in better overall performance than using heuristics. So the KISS principle wins and the decision is simple; simply look one frame down. If the optimized code is itself invoked frequently enough the optimizer will be invoked again and the optimized method's caller will be optimized, and so on, for as long as the code in question continues to trip counters. (* 2)

2.3.2 What optimizations to apply?

Essentially any profitable optimization one can think of that can be applied without high computational effort can be applied. Pauses due to optimizing compilation must be short enough (* 3) that for the given application the system's responsiveness is adequate. Different applications will differ markedly in what is acceptable, animation having vastly different requirements from batch calculation. I am assuming that some degree of tailorability will be required for the optimizer to be effective in a broad range of contexts, but I won't speculate on any such scheme at this point.

The need to provide good interactive response, the desire to make incremental progress on the system, the desire to provide an architecture portable across typical Smalltalk VM designs and the many obvious and simple opportunities for performance improvement in typical Smalltalk programs on conventional VMs all lead me to suggest we concentrate on some obvious targets:

- "splitting" at polymorphic dispatch points; i.e. creating multiple copies of a polymorphic code sequence such that the polymorphism is reduced or eliminated within the individual copies. e.g. inlining a send

at a point with two receiver classes might produce code that branches along two different paths depending on the actual receiver class.

- inlining blocks used in control structures to eliminate block invocation overhead and enable further optimizations

- inlining establishing exception-handlers (eliminating activations of ensure: and ifCurtailed:) for the same reasons

- eliminating interrupt points to allow other optimizations (such as type propagation, eliminating store checks, batching allocations, ...)

- combining instance creation and instance initialization to eliminate store checks

- type and range analysis on variables, especially loop variables, to eliminate bounds checks, tag tests and overflow tests in indexing and arithmetic

- inlining instance variable accessor methods direct instance variable access of objects other than the receiver

- unboxing floating-point values on the stack (and possibly in pointer-object instances) to increase floating-point performance

Inlining per-se isn't necessarily a profitable optimization on current processors where branch prediction hardware can very effectively optimize-away call/return overhead, unless performed a number of times when benefits are cumulative. But inlining enables many other optimizations and so is central to the optimizer. Inlining short methods (such as instance variable fetches) can also generate significantly shorter code sequences, with indirect benefits in I-cache usage.

2.3.4 What to optimize to?

Urs' thesis describes an optimizing compiler that generates native code, but this has to be excluded on portability grounds. Instead we posit an extended bytecode set that contains a number of specialised operations (for example, an at: that does no tag and bounds checking on its index argument) from which an underlying native code generator in the VM can generate native code that is much more efficient than the code the VM is usually able to generate directly from unoptimized bytecode. Some of these bytecodes will be like an extended set of special selector bytecodes, representing more efficient versions of commonly executed primitives, while some will be more exotic control-flow bytecodes (such as "test type cases and trap"). The Strongtalk bytecode set http://www.cs.ucsb.edu/projects/strongtalk/big/bctable.pdf certainly looks like part of such a set (it omits the set of primitives). Peter Deutsch added bytecodes that referenced a register allocator and a subset of go-faster "in-line primitives", primitives that are included in bytecode rather than invoked through message sends, to HPS in the late 80's. Peter's problem was how to generate such bytecodes, given that it appeared one needed a typed language. Adaptive optimization solves that problem nicely. Peter's bytecodes were one of the inspirations behind this design.

The "test type cases and trap" approach to ensuring optimization invariants allows the optimizer to omit code for the general case, where for example some new receiver class is encountered in the execution of optimized code. Within any scope in which particular receiver classes are assumed to be from a known set (because inline cache data predicts the set) but where the optimizer cannot prove that the receiver classes are restricted to this set (for example because the potential receiver is a method argument), the optimizer simply has to output the relevant "test type cases and trap" code at the beginning of the scope, avoiding wasting time and space generating general code that is unlikely to be executed, and allowing type information to propagate when the results of message sends to such a restricted set of receiver classes return a known set of receiver types.

2.4 Dynamic Deoptimization

A method expressed in the go-faster bytecode set is typically the combination of a number of distinct unoptimized methods. To preserve source-level debugging we need some way of mapping back from an activation of the optimized method to activations of the unoptimized methods. [Hölzle92] We can also use the same facilities to map optimized activations back into a format suitable for reoptimization when optimized code is invoked frequently enough to be reconsidered or deoptimization when constraints in the optimized code (as checked by "test type cases and trap") are violated.

In this architecture all such mapping is removed from the VM and moved up to the image level. Given that it is reported that dynamic deoptimization is the trickiest and most bug-prone part of Self-93 [Wolczko96] this is fortunate indeed. Smalltalk code maps activations of optimized methods back into activations of unoptimized methods which can then be used by the debugger, the optimizer or any other client that must reflect on execution but only understands the unoptimized format.

3. A Portable Interface for Adaptive Optimization.

3.1 Overview

Invocation counters send back into Smalltalk when they trip. The optimizer uses extended execution reflection facilities (Contexts and Methods) to analyse execution, generates a new method using an extended bytecode set, installs the optimized method in the relevant method dictionary and constructs a suitable optimized context that resumes execution at the interrupted point. The optimizer then replaces the relevant activations with the new one and allows execution to resume.

To be able to implement adaptive optimization portably above conventional Smalltalk VMs the optimizer can use an extended form of the standard execution reflection facilities to analyse execution and can express its output as bytecoded methods that can use an extended set of bytecodes to express more efficient operations. There needs to be some means of rendering the execution state stable when the optimizer is invoked so that the execution of the optimizer itself does not affect that which it is analyzing. The next four sections will present invocation counters, execution reflection facilities, the extended bytecode set and stabilizing execution state respectively. Much of the interface will be

expressed as a set of primitive methods, either on standard classes like CompiledCode, an abstract superclass for all bytecoded method classes, and Context, an abstract superclass for all execution classes.

I'm assuming that there is a one-to-one mapping from compiled methods in the image to natively-compiled methods in the VM. Some VMs (e.g. the BrouHaHa threaded code VMs) do specialize, a la Self, producing different executable versions of methods tailored to specific classes of receiver, but I'm not aware of any VMs other than the Self VMs (and BHH). If any of you do have such a VM then that specificity can probably be accommodated by defining the affected parts of interface on Context instead of CompiledCode. In this design it is the optimizer that produces more specific code, ensuring its suitability either by placement in the class hierarchy or by inserting the requisite type tests, and so I'm also assuming that any such specialization done by the base VM will have little impact and its complication can be discarded without difficulty or penalty.

A fundamental assumption in this design is that a stack-based bytecode set is perfectly adequate as the source language of a native code generator on a register machine. The HPS translator is an good existence proof. HPS generates register-based code using a stack simulation to collect operand descriptors during the "interpretation" of source bytecodes, emitting register-based code when other bytecodes consume operands on the stack simulation. So when you read about stacks and bytecodes you should the VM will actually generate register code. The Intel VTune Java JIT described in PLDI '98 uses a scheme very similar to that of Peter Deutsch for HPS [Tabatabai98].

3.2 Profiling Counters [2]

3.2.1 Choosing a Profiling Counter Mechanism (new for [2], rewritten for [3])

Profiling counters serve two basic functions. Profiling counters serve as trigger points at which the normal execution of the VM is interrupted and the call-back into Smalltalk to invoke the optimizer is made. Profiling counters also inform the optimizer as to the dynamic execution frequency of code, and hence guide the choice of what code to optimize, avoiding wasting time optimizing infrequently executed code. But profiling counters are expensive (increasingly expensive the more efficient the base Smalltalk VM is) and so there are interesting trade-offs between different profiling schemes.

In all schemes, choosing the value at which to trigger is important. The value determines the rate at which the VM calls-back to the optimizer. There are various events one can count. The ones considered here are message sends, method invocations, backward branches and conditional branches.

The most accurate form of counter is an "call graph edge counter" that counts specific sends, but these are the least affordable. Ideally the code to increment a send count is places in the callee, using the return address to locate the actual counter. For various reasons the return address may not always point to the callee (* 7), and so care has to be taken to locate a valid counter. In fact, explicit edge counters seem so expensive that I've not yet implemented them. So this could be hot air.

A simpler form of counter is a "node counter", that counts invocations of a method or block. The code can be included in the prolog. This provides ambiguous data because a given method may be shared between send sites with widely different invocation frequencies. So the optimizer can be misled by the shared invocation count into considering a rarely used send site to be of high dynamic frequency. This kind of count was used in the Self-3 an dSelf-4 compilers.

Arnold et al [Arnold02] in the Java Jikes RVM point out that accurate edge counters can be inferred from basic block counters. If counters are placed at conditional branches that count the taken and untaken frequencies (e.g. as one executed counter and either the taken or untaken branch counter) the invocation counts in a conditionally executed basic block are the same as the untaken frequency of the conditional branch that precedes it. If the optimizer is invoked when a conditional branch counter trips then the counts for all sends from that basic block can be found by following the call graph through sends in the basic block and propagating the basic block's count. So from conditional counters one can derive accurate edge counters, with the proviso that non-local returns (^-returns in blocks) take a short cut return path that can leave sends unexecuted. Since studies have shown that for Smalltalk conditional branches are much less frequent than sends [Krasner83][Ungar83] it would seem that conditional branch counters would be less expensive than send counters.

Loop counters are the least expensive explicit counting mechanism I've looked at. Loops are not nearly as frequent as conditional branches, and yet loops are reliable indications of potential hot spots for all but purely recursive algorithms, and Smalltalk makes much heavier use of looping than it does of recursion for iteration. Because backward branches also have to poll for events the relative cost of loop counts at backward branches are also lowered. But relatively few methods contain loops, so using only loop counters provides very poor execution coverage information.

Dan Ingalls invented receiver-type prediction in Smalltalk-76, implementing certain of the special selector bytecodes as open-coded implementations that test for SmallInteger receiver,argument pairs and perform the relevant operation (arithmetic or comparison), invoking a normal send on non-SmallInteger arguments and overflow. This is a significant optimization but its use in this context would mean that many send sites would not contain accurate receiver-class data, because SmallInteger receiver,argument pairs would not be recorded. So we must also consider disabling the optimization in unoptimized methods so as to provide more accurate send-site data.

To help choose between these different schemes I benchmarked HPS with various combinations, using the computational subset of the macro benchmarks. The first implementation placed counters in-line in the generated machine code. This produced very poor performance and David Simmons explained the issue is to do with the flushing of the instruction cache caused by the write to data space that is aliased with inmstructions in the processors cache. He pointed out that counters must be kept out of line at some distabnce away form teh code. I've used malloced space and the out-of-line counters perform makedly better (* 11). Here are results for a 600 Pentium III, and a 500 MHz PowerPC 750. 'Count Invocations' means an invocation counter in the prolog of methods

and blocks. 'NoInlineSS' means that all special selector inlining (receiver type prediction) has been disabled and that hence all sends are real sends. 'Count Conditionals' means that conditional branches are counted for their executed frequency and their taken frequency (untaken frequency being the difference between the two). 'Count Primitives' means that only activations of methods containing primitives are counted. 'Count Blocks' means that only block activations are counted. 'Backward Branches' means that the backward branches in loops are counted.

600 MHz PIII (Linux)
'Normal'->0.0->1.0
'No InlineSS'->-12.1976->0.878024
'Count Invocations'->-12.7016->0.872984
'Count Invocations No InlineSS'->-24.7984->0.752016
'Count Invocations & Backward Branches'->-12.0968->0.879032
'Count Invocations & Backward Branches No InlineSS'->-26.1593->0.738407
'Count Conditionals'->-9.77823->0.902218
'Count Conditionals No InlineSS'->-16.8851->0.831149
'Count Primitives'->-3.22581->0.967742
'Count Primitives No InlineSS'->-19.6069->0.803931
'Count Blocks'->-0.806452->0.991935
'Count Blocks No InlineSS'->-13.6593->0.863407
'Backward Branches'->-1.41129->0.985887
'Backward Branches No InlineSS'->-13.8609->0.861391
'Count Primitives & Blocks'->-4.48589->0.955141
'Count Primitives & Blocks No InlineSS'->-18.246->0.81754

500 MHz PPC 750CX (Linux)
'Normal'->0.0->1.0
'No InlineSS'->-8.56463->0.914354
'Count Invocations'->-11.9746->0.880254
'Count Invocations No InlineSS'->-21.6495->0.783505
'Count Invocations & Backward Branches'->-15.3053->0.846947
'Count Invocations & Backward Branches No InlineSS'->-23.0769->0.769231
'Count Conditionals'->-11.8953->0.881047
'Count Conditionals No InlineSS'->-15.3846->0.846154
'Count Primitives'->-3.8065->0.961935
'Count Primitives No InlineSS'->-13.64->0.8636
'Count Blocks'->-0.396511->0.996035
'Count Blocks No InlineSS'->-8.96114->0.910389
'Backward Branches'->-1.34814->0.986519
'Backward Branches No InlineSS'->-12.4504->0.875496
'Count Primitives & Blocks'->-2.85488->0.971451
'Count Primitives & Blocks No InlineSS'->-14.1951->0.858049

The coice I'm investigating in the HPS implementation is to count conditionals without inlining the special selectors. This seems a good trade-off between performance and coverage. Note that counting conditionals also counts finite loops, since all finite loops begin or end with a conditional. The cost for this is quite acceptable at about 15% total slowdown. This contrasts with about a 22% to 25% slowdown for counting invocations without inlining the special selectors.

Reducing the set of methods counted to just primitives and blocks is unlikely to be a wise choice, because certain codes would not be counted. However, it would be interesting to investigate not inlining the "macro selector" control structures and counting block invocations.

I now take the position that the Jikes RVM hybid scheme of statistical sampling and explicit profiling is complex, introduces difficulties with reproducibility, and would only inctease base unoptimized performance by 10% to 20%. Since the system should spend most of its time in optimized code I strongly suspect the gain ain't worth the pain.

3.2.2 The Hybrid Approach (new for [2])

In the hybrid approach described in [Arnold01] and [Arnold02] the pc is sampled (e.g. at 100Hz) and methods that show up above some threshold are instrumented. [Arnold02] describes this being implemented by producing an instrumented copy of the uninstrumented method, modifying the uninstrumented method to jump to the instrumented method at various points (* 8). I haven't investigated this scheme, and while it is tempting as the cost of sampling the PC is typically below 1% it is elaborate and introduces complications (* 12) to provide reproducible runs. In any case, the presence of statistical sampling doesn't remove the presence of profiling counters, it merely restricts the set of native methods that contain them. So its presence or absence shouldn't affect the discussion in the next section.

3.2.3 Interfacing to the Plethora of Potential Profiling Counter Mechanisms [2]

Its too early to commit to a profiling scheme and so the interface provides for most of the above. The base VM's N-code generator is extended to generate invocation counters, either edge counters or node counters, and/or loop counters or basic-block counters. [2] The interface makes it appear that counters count up towards the relevant limit, but in the VM it will be more efficient to count down towards zero, initializing the counters from the limits. The counters shall be at least sixteen bit unsigned integers, initialized from defaults held in variables in the VM and accessible from the image.

```
CompiledCode class>>loopCountLimit
    <primitive>
    Answer the value used to initialize loop counters

CompiledCode class>>loopCountLimit: anInteger
    <primitive>
    Set the value used to initialize loop counters. Fail if the
argument is not a SmallInteger between 0 and some platform-specific limit
of at least 65535. Setting the limit has no effect on currently cached
VM code. Only code executed for the first time after the setting of the
limit will be affected by the new limit.

CompiledCode class>>sendCountLimit
    <primitive>
    Answer the value used to initialize invocation counters

CompiledCode class>>sendCountLimit: anInteger
    <primitive>
    Set the value used to initialize invocation counters. Fail if
the argument is not a SmallInteger between 0 and some platform-specific
limit of at least 65535. Setting the limit has no effect on currently
cached VM code. Only code executed for the first time after the setting
of the limit will be affected by the new limit.
```

```
    CompiledCode class>>conditionalCountLimits [2]
        <primitive>
        Answer the value used to initialize conditional branch counters.
The VM is free to arrange whether the conditional branch counters trip on
the taken, untaken or executed frequencies, but it must provide two out
of the three.

    CompiledCode class>>conditionalCountLimit: anInteger
        <primitive>
        Set the value used to initialize conditional branch counters.
Fail if the argument is not a SmallInteger between 0 and some platform-
specific limit of at least 65535.  Setting the limit has no effect on
currently cached VM code.  Only code executed for the first time after
the setting of the limit will be affected by the new limit.
```

[3] In the HPS implementation as implemented so far we use a trio of primitives to communicate information such as the counter limits above. This scheme is simple and extensible, and can serve for more than just AOStA. The three primitives are

```
    ObjectMemory>>registrationNames
        "Answer an Array of the registration purposes supported by this
VM."
        <primitive>

    ObjectMemory>>objectRegisteredWithEngineFor: purposeString
        "Answer the object registered with the engine for the purpose
specified by
         purposeString, or nil if none.  The set of purpose names is
answered by
         registrationNames."
        <primitive>

    ObjectMemory>>registerObject: anObject withEngineFor: purposeString
        "Register the first argument, an arbitrary object, to be used for
a purpose
         specified by the second argument, a ByteString or ByteArray in
ASCII
         encoding."
        <primitive>
```

So, for example, to set the conditional count limit and callback selector in the HPS implementation one can evaluate

```
    ObjectMemory
        registerObject: 6000 withEngineFor: 'conditionalCountLimit';
         registerObject: #conditionalCountReached withEngineFor:
'conditionalCountReachedSelector'.
```

To allow the system to work with these various counts we do not specify access to explicit per-method invocation counts. Instead, access to counts is included with in-line cache information (see below). [2]

Experience with the Animorphic VM suggests that 30,000 is a good initial value for loop and invocation counters. So we infer that 16 bit counters give ample range. Initializing a counter to non-zero and decrementing it, tripping when it reaches zero, is just as functional as counting up but allows much simpler code to be generated. Keeping the count to 16

bits allows platforms with good 16-bit support to save space storing counts but doesn't prevent other platforms from using a 32-bit variable to hold the count. (* 4) Defining these via a primitive interface allows the VM to put the actual variables only in generated native code, rather than paying the overhead on every v-method.

When a loop count trips execution of the current context stops with the pc at the backward branch, which is typically already a suspension point in conventional VMs since they must check for interrupts in potentially infinite loops, and the message loopCountReached is sent to the current context. If the image-level optimizer is interested it can find the pc by querying the context. When a method invocation count trips execution of the method is suspended at the method's first pc, which is typically already a suspension point because VMs must check for interrupts at frequent intervals in execution and typically sends (or rather the activations of sends) are the only reliable choice (* 5). These two messages are the entry-point into the optimization system. As discussed below, on sending these messages the base VM may take whatever measures necessary to "stabilize" execution so that the optimizer observes a frozen execution state.

Context>>loopCountReached [3]
This message is sent by the VM when the receiver's method's loop count at pcInteger reaches the loopCountLimit.
Context>>conditionalCountReached [2] [3]
This message is sent by the VM when one of the receiver's method's conditional branch counts at pcInteger reaches the conditionalCountLimit.
Context>>sendCountReached
This message is sent by the VM when either the receiver's method's invocation count reaches the sendCountLimit or when a send resolving to the receiver's method reaches the sendCountLimit, depending on whether the system uses node counts or edge counts.

3.3 Execution Reflection Facilities

The optimizer needs to determine type information at send sites, which is that present in in-line caches and send counters. Hence the available type information for a specific send site is that it may be un-taken (have not been executed since the last time the send cache was voided (* 6)), may be linked to a single target method for a single receiver class (monomorphic send) with a given send count, may link up to n target methods for m receiver classes, n <= m, (polymorphic send) each with a given send count, or may link to an unknown set of target methods for an unknown set of target classes (polymorphic send), with send count information unavailable (and irrelevant since inlining is infeasible given the set of target methods is indeterminate. In the abstract a suitable definition would be

CompiledMethod>>classesTargetsAndCountsForSendAt: pcInteger
<primitive>
Answer the class, target method, send count triples for the send bytecode at pcInteger as a trio-wise array of class, method, count pairs. Fail if pcInteger is not an integer or is not the pc of a send bytecode, or if the result cannot be allocated. If the send has not been taken answer nil. If the send is a megamorphic send where the set of classes and target methods cannot be determined, answer an empty array. For

example, at a send of #+ where both SmallInteger and Float receivers have been encountered the answer might be equal to

```
Array
    with: SmallInteger
    with: (SmallInteger compiledMethodAt: #+)
    with: 1234
    with: Float
    with: (Float compiledMethodAt: #+)
    with: 30000
```

But because of the implementation of the mapping from N-code PCs to V-code PCs, such an interface may turn out to be O(N^2) when querying a sequence of send sites. Instead the interface method answers information for all sends and for conditional and loop counts, if the VM is instrumenting them:

```
CompiledMethod>>sendAndBranchData
    <primitive>
```

Answer information describing sends, and conditional and backward branch counts as an Array. The top level Array is organized as a sequence of pairs of bytecode pc followed by data for the particular send, conditional or loop. Sends are represented as an Array of the class, target method, send count triples for a given send. Backward branches are represented as their invocation count. Conditional branches are represented as an Array of the taken and untaken branch counts. [2] Fail if the result cannot be allocated. For each send in the receiver, if the send has not been taken the result omits the entry for that send; if the send is a megamorphic send where the set of classes and target methods cannot be determined, the entry is an empty array. If any entry for a class is a doesNotUnderstand: case the method is nil [1]. For example, in the compiled method

```
Number>>ceiling
    self <= 0
        ifTrue: [^self truncated]
        ifFalse: [^self negated floor negated]
```

where the method has been used on positive Float and Double receivers in the SmallInteger range so that only the second arm of the ifTrue:ifFalse: has been taken the result might be equal to

```
Array
    with: 3 with: (Array
        with: Double
        with: (Double compiledMethodAt: #<=)
        with: 1234
        with: Float
        with: (Float compiledMethodAt: #<=)
        with: 2345)
    with: 4 with: (Array with: 0 with: 3579)
    with: 8 with: (Array
        with: Double
        with: (ArithmeticValue compiledMethodAt: #negated)
        with: 1234
        with: Float
        with: (ArithmeticValue compiledMethodAt: #negated)
        with: 2345)
    with: 10 with: (Array
        with: Double
        with: (Number compiledMethodAt: #floor)
        with: 1234
```

```
                    with: Float
                    with: (Number compiledMethodAt: #floor)
                    with: 2345)
                with: 11 with: (Array
                    with: SmallInteger
                    with: (Integer compiledMethodAt: #negated)
                    with: 3579)
```

The result of CompiledMethod>>sendAndBranchData should be computable by the VM in a single pass, and can easily be parsed into something the optimizer would find more useful such as a Dictionary from PCs to objects representing send caches.

[1] One or more entries in a PIC for some class can be for MessageNotUnderstood errors. In this specification we require the method entry to be nil. This is potentially wasteful as one could imagine wanting to in-line doesNotUnderstand:, and, since the underlying PIC might record what doesNotUnderstand: method to activate for the case, not passing the method discards that information. But the optimizer does need to know if a case is a MessageNotUnderstood case, and the optimizer can easily locate the doesNotUnderstand: method at the image level.

The optimizer also needs to know the maximum size of a closed PIC that the VM supports. This can be either a manifest constant or a primitive. if a primitive then e.g.

```
        CompiledCode class>>vmClosedPICSize
            <primitive>
            Answer the maximum number of type-cases in a closed PIC on
the current VM
```

3.4 Extended Bytecode Set

The extended bytecode set needs to address a few different requirements. One is to generate "linked sends", sends with an in-line cache, so that optimized code avoids lookups and runs with "hot" send caches immediately. Another is to implement efficient type-case and trap tests used to ensure the invariants expected by optimized code and to invoke the optimizer when these invariants are violated. Another is to express efficient forms of operations such as indexing and arithmetic. I expect this last set to evolve and grow as we gather experience with the system, and so the list here should not be considered at all exhaustive.

This design does not determine how optimized v-methods that use the extended bytecode set reference that bytecode set. In the context of HPS there are probably (hopefully?) sufficient unused bytecodes that the extended bytecode set can be expressed using them, given that the set already includes a few codes for expressing in-line primitives. Claus Gittinger's eXcept VM supports four distinct bytecode sets, selected by a two-bit field in the method header, one of which is used for Smalltalk, and one of which is the Java bytecode set. So in Claus' case it might make much more sense to define a new set for compactly and conveniently expressing optimized methods [provided he hasn't used all four already ;)].

I am assuming the optimizer-VM interface will be designed to abstract away from specific bytecode set encodings, there being dialect-specific

front-ends mapping the execution state into suitable input for the optimizer, and dialect-specific back-ends generating actual bytecode from the parse-tree(s) built by the optimizer. This degree of freedom will help us migrate the optimizer to any new bytecode set(s) that evolve out of this project.

Given the extended bytecode set is portable (and potentially interpretable) it should not express platform-specific issues such as register allocation. However, allocating variables to registers is a very important process in generating efficient code on most of today's processors. So while the extended bytecode set is still a stack-based one, a simple scheme informs the VM as to register allocation, prioritising temporaries according to the likely profitability of housing them in registers. The following sections flesh out the bytecode set and the register allocation scheme.

Some of these bytecodes specify that under certain circumstances the system's behaviour is undefined. This means that the circumstances are an error that should not normally occur and that behaviour may vary from an ungraceful crash through to an orderly VM exit with error message to a send-back into the optimizer.

3.4.1 Extended Bytecodes for Linked Sends

Conceptually only one bytecode is required whose operands are the number of arguments, the literal index of the selector and the literal index of an Array of class, target-V-method pairs that represent the in-line cache data for the send. An empty Array represents a megamorphic send. The behaviour of the bytecode with an Array larger than the maximum closed PIC size is undefined. A more compact representation would combine the selector and the Array into a single literal Array whose first element is the message selector. Multiple versions of the bytecode may be present to provide compact encodings.

It may prove useful to have a set of these bytecodes that parallel the special selector bytecodes that predict SmallInteger as receiver types (special selector bytecodes for #+ #- #* #// #< #> #<= #>= #= #~=). These would generate code that predicts SmallInteger operands, using tag tests that branching to in-line code or to a linked send for the types in the Array. But I would expect the optimizer to use in-line primitives for many cases. This reveals a problem with predicting SmallInteger receivers for these special selectors. If SmallInteger are predicted (i.e. if the VM generates code that tag tests and does the operation in-line for SmallInteger operands) and only SmallInteger operands occur without overflow for #+ #- #* or divide-by-zero for #//, then the send looks just like an un-taken send. Experience will show if it is better to not predict SmallInteger receivers and for the base VM to simply generate normal sends, relying on the optimizer to recover performance given the more accurate type information it can gather if ordinary sends are used.

3.4.2 Extended Bytecodes for Type-Cases

There are bytecodes that take as their operand the literal index of an Array of classes and perform some action depending on whether the class of the top of stack is a member of the Array.

Object trapIfClassNotMemberOf: anArrayLiteral
If the class of the object on top of stack is not a member of anArrayLiteral then send the message #typeCaseTrap to the current context [3]

OptimizedContext>>typeCaseTrap

Object isTopOfStackAMemberOf: anArrayLiteral
The top of stack is replaced by a boolean that is the value of whether the class of the object on top of stack is a member of anArrayLiteral.

It is assumed that the VM generates code that tests the type cases in the order they appear in the Array operand to permit the optimizer to benefit from dynamic frequency information it may be able to derive from send counts.

It may be more convenient to include multiple versions of the bytecodes that take as an additional operand how far down the stack to look for the object to type case. This would allow arguments to be marshaled before a type case is applied, which might be convenient if the optimizer wanted to branch to two different implementations of an inlined send, depending on a type case.

3.4.3 Extended Bytecodes for In-line Primitives

The set presented here is only an initial suggestion. Experience with the system and with specific applications may suggest new primitives. Extending the set should only be problematic if there turn out to be many (e.g. > 256) primitives, which shouldn't happen too soon. The set here is largely derived from Peter Deutsch's set for HPS. Specific encoding of primitives is left to the encoding of each VM's particular bytecode set. Most primitives have no additional parameters beyond their arguments on the stack. But some primitives take an additional operand encoded in the bytecode (e.g. the number of slots for instance creation primitives).

Peter's scheme also includes a bytecode that invokes an arbitrary primitive that answers two results, the top of stack being a boolean indicating whether the primitive succeeded or not. I'm not sure yet if this is useful; after all a primitive can be invoked just as well by an ordinary send, and if so, the failure case is handled by the failure code in the method. But the general idea that there may be more primitives than described below which are derived from the base set of primitives might be useful.

Some in-line primitives may fail (e.g. unchecked integer addition that checks for overflow) and some may not (e.g. unchecked integer addition that wraps on overflow). Those that can fail must be followed by a conditional branch (to test whether the primitive succeeded) or a pop (indicating that compiler knows that the primitive will always succeed). Unless otherwise noted below, the classes of the receiver and arguments are ASSUMED to be as documented, and are not checked: if an object of the wrong class is used, the results are undefined and almost certainly very harmful!

3.4.3.1 SmallInteger Arithmetic and Bit Manipulation

SmallInteger ilpSiAddFailing: aSmallInteger => aSmallInteger
Add the receiver and argument. Fail if the result overflows the SmallInteger range.
SmallInteger ilpSiAddNoFail: aSmallInteger => aSmallInteger
Add the receiver and argument, modulo the SmallInteger range.
SmallInteger ilpSiSubtractFailing: aSmallInteger => aSmallInteger
Subtract the argument from the receiver. Fail if the result overflows the SmallInteger range.
SmallInteger ilpSiSubtractNoFail: aSmallInteger => aSmallInteger
Subtract the argument from the receiver, modulo the SmallInteger range.
SmallInteger ilpSiMultiplyFailing: aSmallInteger => aSmallInteger
Multiply the receiver and argument. Fail if the result overflows the SmallInteger range.
SmallInteger ilpSiMultiplyNoFail: aSmallInteger => aSmallInteger
Multiply the receiver and argument, modulo the SmallInteger range.
SmallInteger ilpSiDivFailing: aSmallInteger => aSmallInteger
Divide the receiver by the argument rounding towards -oo. Fail if the argument is zero.
SmallInteger ilpSiDivNoFail: aSmallInteger => aSmallInteger
Divide the receiver and argument rounding towards -oo. The behavior is undefined if the argument is zero.
SmallInteger ilpSiQuoFailing: aSmallInteger => aSmallInteger
Divide the receiver by the argument rounding towards 0. Fail if the argument is zero.
SmallInteger ilpSiQuoNoFail: aSmallInteger => aSmallInteger
Divide the receiver and argument rounding towards 0. The behavior is undefined if the argument is zero.
SmallInteger ilpSiModFailing: aSmallInteger => aSmallInteger
Take the receiver modulo the argument. Fail if the argument is zero.
SmallInteger ilpSiModNoFail: aSmallInteger => aSmallInteger
Take the receiver modulo the argument. The behavior is undefined if the argument is zero. [3]

SmallInteger ilpSiBitAnd: aSmallInteger
Bitwise-and the receiver and argument. Cannot fail.
SmallInteger ilpSiBitOr: aSmallInteger
Bitwise-or the receiver and argument. Cannot fail.
SmallInteger ilpSiBitXor: aSmallInteger
Bitwise-exclusive-or the receiver and argument. Cannot fail.

SmallInteger ilpSiBitShiftLeftFailing: aSmallInteger
Bit-shift the receiver left argument number of positions. Fail if the result overflows the SmallInteger range. The behavior is undefined if the argument is negative or zero.
SmallInteger ilpSiBitShiftLeftNoFail: aSmallInteger
Bit-shift the receiver left argument number of positions. Discard bits outside the SmallInteger range. The behavior is undefined if the argument is negative or zero.
SmallInteger ilpSiBitShiftRightNoFail: aSmallInteger
Bit-shift the receiver left argument number of positions. Discard bits outside the SmallInteger range. The behavior is undefined if the argument is negative or zero.

3.4.3.2 Integer Comparisons

```
SmallInteger ilpSiLessThanNoFail: aSmallInteger
SmallInteger ilpSiGreaterThanNoFail: aSmallInteger
SmallInteger ilpSiLessEqualNoFail: aSmallInteger
SmallInteger ilpSiGreaterEqualNoFail: aSmallInteger
    Compare two SmallIntegers and answer true or false.
```

3.4.3.3 Indexing

```
anObject ilpAtNoFail: aSmallInteger
    Answer the slot at the one-relative index aSmallInteger in
anObject.  Perform no bounds or indexability checks.  Do not account for
any named instance variables in anObject.  The behavior is undefined if
anObject is not a pointer object or if aSmallInteger is not in the range
1 to the number of slots in anObject.

anObject ilpAtNoFail: slotIndex
    Answer the slot at the zero-relative index slotIndex in anObject.
slotIndex is an operand encoded in the bytecode (suggested range 0 to
255).  Perform no bounds or indexability checks.  Do not account for any
named instance variables in anObject.  The behavior is undefined if
anObject is not a pointer object or if aSmallInteger is not in the range
1 to the number of slots in anObject.

anObject ilpAtNoFail: aSmallInteger put: aValue
    Assign to the slot at the one-relative index aSmallInteger in
anObject with aValue.  Perform no bounds, indexability or immutability
checks.  Do not account for any named instance variables in anObject.
The behavior is undefined if anObject is not a pointer object, is
immutable, or if aSmallInteger is not in the range 1 to the number of
slots in anObject.  The assignment is store-checked.

anObject ilpAtNoFail: slotIndex put: aValue
    Assign to the slot at the zero-relative index slotIndex in
anObject, which is below aValue, the top of stack.  slotIndex is an
operand encoded in the bytecode (suggested range 0 to 255).  Perform no
bounds, indexability or immutability checks.  Do not account for any
named instance variables in anObject.  The behavior is undefined if
anObject is not a pointer object, is immutable, or if aSmallInteger is
not in the range 1 to the number of slots in anObject.  The assignment is
store-checked.
```

3.4.3.4 No-Check Inst Var Stores [2]

The optimizer may be able to identify stores of immediate data (typically SmallIntegers and Characters) to instance variables in sufficient cases that it may be profitable to provide a no-check inst var store bytecode that eliminates the unnecessary store check for an immediate.

3.4.3.5 Instance Creation

These bytecodes enable more efficient instance creation by avoiding checks on the class receiver and more efficient initialization by eliminating store checks. A bytecode that ensures a given amount of new space is available is provided to ensure that the instance creation bytecodes cannot fail.

aSmallInteger ilpEnsureNewSpaceSlots

If the amount of eden newSpace from which to allocate is less than aSmallInteger provoke a scavenge. This bytecode is a suspension point, or rather, the preceding bytecode, such that when execution resumes after the current process is suspended it does so at the ensureNewSpaceSlots bytecode. i.e. execution only proceeds from the bytecode when sufficient newSpace is available. If insufficient newSpace exists after a scavenge (for example, because newSpace is smaller than aSmallInteger) behaviour is undefined. It is assumed that the optimizer will be informed as to the capacity of newSpace and to the per-instance overhead in the underlying VM and will hence generate viable values of the operand aSmallInteger.

aBehavior ilpPointerNew: numFields

numFields is an operand encoded in the bytecode itself (suggested range 0 to 255). Replace the object on top of stack with a pointer instance having numFields slots initialized with nil and a class field of aBehavior. The behaviour is undefined if insufficient allocatable newSpace is available, or if aBehavior is not a valid pointer behavior.

aBehavior {numFields objects} pointerNewInitializing: numFields

numFields is an operand encoded in the bytecode itself (suggested range 0 to 255). Replace the object numFields down the stack (aBehavior) with a pointer instance having numFields slots and a class field of aBehavior. The fields of the instance are initialized with the numFields objects on the stack, the top-most being assigned to the last slot in the new instance. This bytecode enables the elimination of store-checks on instance initialization. The behaviour is undefined if insufficient allocatable newSpace is available, or if aBehavior is not a valid pointer behavior.

aBehavior ilpByteNew: numBytes

numFields is an operand encoded in the bytecode itself (suggested range 0 to 255). Replace the object on top of stack with a byte instance having numFields bytes initialized with zero and a class field of aBehavior. The behaviour is undefined if insufficient allocatable newSpace is available, or if aBehavior is not a valid non-pointer behavior.

3.4.4 Register Allocation in N-code

While the extended bytecode set is still a stack-based one, temporary variables on the stack have a priority, implicit in their position on the stack, that should be used as a hint by the VM's code generator as to which temporaries should be allocated to registers. The highest priority temporary is the last temporary variable on the stack, and the lowest priority is the first temporary (also the first argument). The optimizer is then free to put temporaries used with the highest dynamic frequency in the last temporaries, and the VM is free to house as many high-priority temporaries in registers as it can fit in the available registers of the underlying processor. The optimizer will emit the temporary initialization code before any other to simplify enabling the VM's code generator to avoid initializing temporaries with nil when other initial values are available.

It is still required that temps are initialized to permit easy conversion from stack activations to context objects within the VM. The optimizer has to be able to determine which temporaries are live at each point in execution so that it can define suitable variable descriptors for dynamic deoptimization. But the VM is being kept unaware of the dynamic deoptimization representations so that this burden may be carried entirely at the image level. Hence when a stack activation is reified as an OptimizedContext there can not be any undefined slots below the top of stack so as to protect the garbage collector from bogus pointers.

Experimental versions of HPS include bytecodes for explicitly allocating and deallocating registers. The temporary priority policy above is more easily portable than these explicit bytecodes. After all, how many registers should the optimizer try and allocate given that it does not know how many are available? The Animorphic bytecode set appears to include bytecodes that allocate or deallocate a given number of temporaries, presumably to be used to reduce temporary initialization overhead at the start of a method by restricting the use of a set of temporaries to a subset of the method. I'm not at all sure if this is better or worse than the temporary register priorities scheme.

3.5 Initial Floating-Point Unboxing Scheme

While it should be a goal to unbox floats in pointer instances this sketch ignores that possibility for now. Smalltalk imposes no restriction on the type of object stored an a pointer instance variable. Therefore any unboxing scheme needs to be per-instance, not just per-class (although one could imagine a scheme that used anonymous behaviors to distinguish instances of a class that contained unboxed data from instances of the same class that did not). At least in the HPS memory manager such flexibility poses a problem and I would like to make immediate progress. So this unboxing scheme only handles unboxing within an OptimizedContext, being rather analogous to a floating-point co-processor unit.

An OptimizedContext has two stacks, one for normal objects, and one for raw data. The raw stack is organized as a number of slots large enough to hold the largest floating-point format supported by the Smalltalk VM. The size of an OptimizedContext's stack is zero by default. If non-zero its size is defined by information in the context's OptimizedMethod, e.g. either a field in the header, or some initial bytecode (analogous to pushCopiedValues at the start of a copying block) that specifies the number of slots. The stack can be implemented as a pair instance variables in OptimizedContext that are normally nil, but otherwise contain a suitably large ByteArray and a raw stack pointer. Whenever an OptimizedMethod that specifies a non-empty raw stack is activated the initial contents are undefined and the stack pointer is 0 (1 relative), i.e. there is no support for floating-point arguments. It is assumed that in-lining will reduce the demand for floating-point parameter passing enough for it to be lived without.

A set of in-line primitives can access the raw stack as IEEE floating-point data, moving values between the raw stack and the pointer stack or object fields. The primitive set would be extended to support unboxed access to fields in pointer instances if and when required. On Smalltalks with different sized floating-point classes (VisualWorks supports 32-bit Float and 64-bit Double) the primitive set may provide

access to each float size. Here we sketch only a set for 64-bit Double floating-point values. If the set handles multiple sizes of data, each slot can hold only one instance of a small value.

The set of raw stack primitives are stack based because it is much easier to map a stack-based addressing scheme with a finite sized stack onto a register set than it is to map a register-based scheme onto a stack, and the infamous x86 floating-point processor, which is stack-based, is likely to remain an important target for users of this system.

The raw stack could also be used to optimize integer arithmetic, supporting arithmetic on untagged 8, 16, 32 and 64-bit widths as in Java. Rather than waste time specifying this I'll leave open the possibility of adding a set of bytecodes to allow 64-bit arithmetic and conversion to and from tagged and boxed SmallIntegers and LargeIntegers on the normal stack.

3.5.1 Raw Stack Data Movement

rawSetStackPointer: n
: increase the raw stack pointer by n slots. May only be used once before any other raw bytecodes. The values of the slots between 1 and n remain undefined.

rawDup
: push the top element of the raw stack onto the raw stack. The results are undefined if the value on top of stack is not a Double.

rawPushTempDouble: n
: push the n'th double in the raw stack on the top of the raw stack, 0-relative. The results are undefined if the value in the slot is not a Double.

rawPopTempDouble: n
: pop the top of the raw stack into the n'th slot on the raw stack. The results are undefined if the value on top of stack is not a Double.

rawPushLitDouble: n
: push the value of the n'th method literal, which must be an instance of Double, on the raw stack.

rawPushPopDouble
: pop the Double off the top of the normal stack and push its value on top of the raw stack

rawPopPushDouble
: pop the double value on top of the raw stack and push a new instance of Double with this value onto the normal stack. (* 9)

aByteObject rawPushDoubleAtNoFail: aSmallInteger
: index the ByteArray below top of stack on the normal stack with the SmallInteger on top of the normal stack taken as a one-relative byte index and push the double value on the raw stack. The results are undefined if the object below top of stack is not bytes and suitably large or if the index is not a SmallInteger or out of bounds, or if the index is not one plus a multiple of the size of a Double.

aByteObject rawPopDoubleAtPutNoFail: aSmallInteger
: pop the double value on top the raw stack into the slot in the ByteArray below top of stack on the normal stack indexed by the SmallInteger on top of the normal stack taken as a one-relative byte index. The results are undefined if the object below top of stack is not bytes and suitably large or if the index is not a SmallInteger or out of bounds, or if the index is not one plus a multiple of the size of a Double.

3.5.1 Raw Stack Arithmetic and Comparison

rawAddNoFail
rawSubtractNoFail
rawMultiplyNoFail
rawDivideNoFail
Perform the arithmetic operation on the top two elements on the stack (last argument on top), popping the elements and pushing the result. Use IEEE semantics, where overflows produce infinities, and NaNs are contagious. The results are undefined if the top two elements are not Doubles.

rawLessThanNoFail
rawGreaterThanNoFail
rawLessThanOrEqualNoFail
rawGreaterThanOrEqualNoFail
Perform the comparison operation on the top two elements on the stack (last argument on top), popping the elements and pushing the boolean result on the normal stack. Use IEEE semantics, where comparison with NaNs answer false. The results are undefined if the top two elements are not Doubles.

rawDoubleIsNaN
Pop the top element off the raw stack and push a boolean on the normal stack, true if the element is an IEEE NaN, false otherwise.
rawDoubleIsInf
Pop the top element off the raw stack and push a boolean on the normal stack, true if the element is an IEEE Inf, false otherwise.
rawDoubleIsInfOrNaN
Pop the top element off the raw stack and push a boolean on the normal stack, true if the element is either an IEEE NaN or an IEEE Inf, false otherwise.

The above set should allow reasonably efficient floating-point code to be generated for floating-point operations on floating-point data in byte objects. So provided the Smalltalk programmer is willing to use byte objects to hold floating-point values boxing overhead can be eliminated. To provide the same benefits for pointer-objects, and in particular heterogeneous pointer objects (where not all fields may be floats) is left for future work (e.g. any immediate response you all might have).

4. Coordinating Optimization and Normal Execution.

The system so far described executes a mixture of unoptimized and optimized ncode until a counter trips, causing a send-back that invokes the optimizer, whereupon the optimizer performs some analysis, possibly computing some new optimized code, and returns back to normal mixed-mode execution. The optimizer uses standard reflective facilities to analyse the state of mixed-mode execution, implying that state remains stable during analysis. Since the optimizer is running on the same VM some mechanism must be provided to ensure that stability. Since the optimizer is a complex piece of code some form of interactive development is essential if it is to be implemented quickly. This section discusses these two aspects.

4.1 Stabilizing the Underlying Execution State

As mentioned above, it is assumed that the underlying VM maintains a set of executable native code methods. It is further assumed that the amount of space the VM is willing to allocate to these nmethods is finite and that occasionally normal execution is paused while a subset of nmethods are discarded to make room for new ones as execution requires. Once the optimizer is invoked when a counter trips, the optimizer may have to examine an arbitrarily large call graph which could include the callees of the current method's caller, the callee's callees and so on to an arbitrary depth. For the optimizer to be able to analyse the current execution state this subset of the call graph must remain fixed and undisturbed so that the optimizer can extract valid send type information. The system must somehow arrange to leave this subset undisturbed while the optimizer is running extracting this information.

If it was the VM's responsibility to mark the set of nmethods the VM might mark too large a set, leaving insufficient room for the optimizer's own code to run efficiently. Were it the optimizer's responsibility there could be a race condition between the compilation to ncode of the optimizer's vcode as it traverses the call-graph and the call graph itself, with ncode reclamation being provoked before the optimizer finished its traversal.

To avoid either of these pathologies the system needs to provide some headroom for the execution of the optimizer, such that while the optimizer is extracting information the set of nmethods being examined does not change. A simple model for such a scheme is that there are two nmethod arenas, one used for normal code and the other solely for the optimizer. When a counter trips the VM arranges to flip arenas, executing the optimizer's code only within the second arena, and arranging that no loop counters are compiled into nmethod in the optimizer arena. When the optimizer is done and returns control back to the VM, the VM flips back to the normal arena. This flipping should be extremely cheap to implement, being only the swapping of pointers to the ncode allocation pool, and the test during every vmethod compilation of whether to include counters or not.

We assume that in production the optimizer's vcode exists in the form of OptimizedMethods, that will be compiled to ncode on first use into the optimizer's arena, devoid of loop counters. The optimizer's arena will not be disturbed during execution of code in the normal arena. So on subsequent use the optimizer will run at full speed. The optimizer's arena needs only to be large enough to run the optimizer, which may be less than that of the normal arena. In a production system one could even imagine the optimizer's code being pre-compiled and pre-optimized, hidden within the VM.

Another issue is invoking the optimizer in a natively multithreaded environment. How should the optimizer be made reentrant? How can the set of nmethods remain stable while other threads are potentially executing the same code? This I'm leaving to future work. For the moment I'm presuming a single-threaded VM and that on invoking the optimizer the VM shuts out all other threads by raising the priority of the optimizer to the highest priority, guaranteeing stability while it runs.

4.2. Development and Bootstrapping

A crucial objective of this design is to support rapid implementation by facilitating implementation in Smalltalk using the normal interactive programming environment. To facilitate this the architecture supports running code to be optimized in a sand box that provides the optimizer with input execution state to optimize and allows the Smalltalk level to intercede in method binding in the sand boxed code to simulate binding to optimized methods without affecting the code for the development system [1]. During interactive development of the optimizer the functions of the two nmethod arenas are different. Normal execution begins in the initial arena without including loop counters in nmethods. Instrumented code can only be produced using a special primitive that runs a block within the second arena. As the block runs any vmethods bound to are compiled to distinct nmethods in the instrumented arena that include invocation and loop counting code. For example the primitive's signature might be

```
CompiledBlock executeInstrumented
```

When the counters in this arena trip execution resumes in the normal arena
with a send-back of loopCountReached, conditionalCountReached or sendCountReached as appropriate. [2] [3]

The VM arranges that sends in the sand boxed code call-back into Smalltalk a little like a doesNotUnderstand: that specifies what message is being sent. The Smalltalk code then binds the send to a suitable vmethod and returns. The method bound to could be an optimized method created previously during the running of the sand box. The Smalltalk level maintains a faked set of method dictionaries that provide a simulation of the binding behaviour in the deployment context where as the system optimizes normal methods are replaced by optimized methods over time.

Now the normal Smalltalk programming environment can be used to interactively develop the optimizer, to the degree that the VM can insulate the system from the effects of bugs in the code produced by incorrect versions of the optimizer. For example, the VM could send-back on detecting an invalid sequence of bytecodes in an OptimizedMethod, or could trap crashes in the VM's code generator but would not be able to prevent a crash if incorrect optimized code corrupted the heap.

To bootstrap the system the development time scheme can be used to provide the optimizer with a training set, running the optimizer on itself. The optimizer is run in the instrumented arena on some training sets. The optimizer runs in the normal arena, optimizing itself based on the execution state in the instrumented arena. The optimizer is bootstrapped only to gain performance. The optimizer could function unoptimized, but if it is worth its salt, optimizing itself will mean it can work at least three to five times harder, or rather shorter.

In production much instantiation in the optimizer's arena could be done via stack allocation, eliminating all store checks and garbage collection, simply throwing away the allocation zone after optimization. The optimizer's code can be checked statically to ensure it makes no assignments to objects outside its allocation domain, and the result OptimizedMethod could quickly be copied into the normal heap.

5. Image-level Facilities Not Involving the VM

In this design a number of facilities that would have to be implemented in the VM have been moved up entirely to the image level. The two important ones are the deoptimization of optimized activations into deoptimized optimizations and the maintenance of dependency information so that optimized code can be discarded when the unoptimized code upon which it depends is changed.

5.1 Deoptimization

There needs to be a way to map an activation of an optimized method into an equivalent set of unoptimized activations at each suspension point in an optimized method. This supports both interactive debugging and reoptimization when e.g. a typeCaseAndTrap traps with a previously unencountered class. In conversations with David Griswold I think I'm right in thinking that the compact encoding of the necessary scope descriptors which map back variable locations in optimized code into variable locations in unoptimized code is an important space saving issue. In any case that's all I need say about it here, except that as part of the development methodology we can do a lot to ensure its correctness.

Recall that Mario Wolczko has indicated deoptimization a bug-prone area. During development we can test that each piece of optimized code the optimizer produces correctly deoptimizes at all suspension points by simulating execution of the optimized code with a single pass analogous to the single pass that the VM's code generator makes to produce ncode As a single pass is made through the optimized method a simulated stack is filled with markers representing the results of evaluating the method. Every time a suspension point is encountered deoptimization is attempted and each deoptimized method is similarly symbolically executed, and the corresponding activations compared. If they differ there is a bug in deoptimization. If they agree there may still be bugs in the deoptimizer, but not with the case in question.

5.2 Maintaining Dependencies

Last but not least we must consider how optimized code is maintained in the running system. When the optimizer constructs an activation of an optimized method with which to resume execution it will also put the optimized method in a method dictionary or block closure somewhere so that on a subsequent send or block activation the optimized code will be used in place of the optimized code.

Smalltalk environments typically do not replace activations of methods that are redefined with their redefinitions, because it is in general an unsolvable problem to map the execution state of the previous version into an equivalent activation of the new method. Instead these activations remain and in the debugger show up as unbound methods. But optimized code is different. It includes inlined copies of code that, were the code unoptimized, would not be executed when redefined because sends would bind to the fresh definition of a method. Luckily this process is trivial to implement given deoptimization because when a new

method is installed in a method dictionary any activations of optimized code will be at suspension points (in a single threaded system (* 10)).

So the system needs to maintain a map from unoptimized methods to optimized methods and must be able to enumerate activations of optimized methods when unoptimized methods are redefined. Again David Griswold has indicated that the efficient encoding of this dependency information is an important consideration. The map from unoptimized to dependent optimized methods can be a single system-wide global optimized for compactness. Presumably the enumeration of optimized activations can be done with the usual allInstances and allOwners facilities. If this proves to be too slow then we'll need to think of something cleverer.

Ian Piumarta points out that the obvious way to represent this dependency is with the unoptimized methods. For example, when an unoptimized method is inlined into some optimized method the unoptimized method could be wrapped by an object that references the unoptimized method and each of the optimized methods into which it is inlined. The wrapper could replace the method in the method dictinary holding the method. The VM would be modified to indirect through such wrappers. The set of wrappers could be maintained ina global, supporting fast enumeration of inlined and inlinees. The image-level method dictionary maintennance code could then easily check for a method having been inlined by searching the method dictionaries from the point of definition up the class hierarchy. [3]

A. Footnotes

(* 1) (this changes style half way through where I pasted in a mail message I had handy)

Because an Open PIC is specific to a particular selector both the selector and the part of the hash function derived from it are constants, yielding slightly faster code and less register pressure. Further, Open PICs conveniently disambiguate full PICs. Without Open PICs a full Closed PIC might just invoke all N cases or might invoke more than N, we can't tell. But these two facets are dull.

The interesting fact is that once execution has settled down Open PICs are the only source of lookups that remain in the program, and that these megamorphic sends tend to be sends in abstract superclasses high in the hierarchy, because of course these sends are the ones inherited by lots of classes.

Simmons, Gittinger and myself all realised that this has implications for the first-level method lookup cache policy. Consider he open PIC sends

A classic is Object copy:

```
    copy
        ^self shallowCopy postCopy
```

a send of copy somewhere will probably be specific, but within the single copy method the send of shallowCopy and of postCopy will likely end up being open because it is inherited by most classes in the system.

So in a stable state where all caches are "hot" the only class hierarchy lookups being done are from open PICs when the method cache hash table

lookup fails. But tragically (because this is a character flaw of open PICs) these lookups are the most expensive, because they tend to be resolved towards the roots of the class hierarchy, meaning that many classes get traversed in each lookup.

The blue-book method lookup algorithm inserts the result of the lookup under the message's selector and the receiver's class. So imagine a series of lookups to nil, true and false. For the first send the method dictionary search in UndefinedObject misses, the method dictionary search in Object succeeds, and a cache entry gets added for UndefinedObject. For the send to true, the method dictionary search in True misses, the method dictionary search in Boolean misses, the method dictionary search in Object succeeds, and a cache entry gets added for True. For the send to false, the method dictionary search in False misses, the method dictionary search in Boolean misses, the method dictionary search in Object succeeds, and a cache entry gets added for False.

Now you will have realized the obvious optimization. On finding a method, enter the hit in the cache for each class on the receiver's class's superclass chain. On each step up the class hierarchy check the method lookup cache for the current class.

Now the above reduces. For the first send the method dictionary search in UndefinedObject misses, the cache lookup for Object fails, the method dictionary search in Object succeeds, and cache entries gets added for UndefinedObject, and Object. For the send to true, the cache lookup and method dictionary search in True and in Boolean miss, the cache lookup in Object succeeds, and cache entries get added for True and Boolean. For the send to false, the method dictionary search in False misses, the cache lookup in Boolean succeeds, and a cache entry gets added for False. That's four method dictionary searches down from eight.

In fact, this one change in the first-level method lookup cache policy doubled the performance improvement we saw from implementing PICs. In the non-I/O subset of the standard Smalltalk macro benchmarks the performance improvement due to PICs increased from about 15% to about 30%.

(* 2) To ensure that infrequently executed code is not optimized eventually by invocation counters eventually tripping, counters can decay, e.g. exponentially. The thesis discusses policies for when and how to decay.

(* 3) the thesis discusses pause clustering as a metric that computes perceived pauses rather than real-time pauses. The user may perceive a burst of several short pauses as a single longer distracting pause. Pause clustering gives us means to deal with perceived pauses and hence better engineer the system for interactive use.

(* 4) In initial tests of counts on x86 and PPC there was no significant performance difference between 16-bit and 32-bit counts, even on PPC, so saving the space seems fine.

(* 5) HPS folds stack overflow checking into event checking by having a method check for interrupts implicitly in its stack overflow check. By setting the value in the variable or register holding the stack limit to a value that always causes the stack overflow check to fail allows stack overflow processing to check for events.

(* 6) in-line caches are voided when the method cache is cleared (Blue Book primitive 89) or the method cache entries for a specific selector are cleared (in HPS this is Behavior>>#flushVMmethodCacheEntriesFor:) or when reclamation of native methods causes native methods to be discarded, clearing check entries at sends that link to them.

(* 7) e.g. in HPS, after a stack segment overflow, the return address points to a trampoline that causes a return to cross stack segment boundaries. [2]

(* 8) At least in HPS the existence of two copies of the method with different code raises a problem. HPS avoids mapping native PCs to bytecode PCs in stable contexts by allowing stable contexts to represent native PCs as negative integers, positive ones indicating bytecode PCs. To map a stable context's native PC to a bytecode PC (e.g. at snapshot or debug time) the system re-translates its bytecoded method. But with the hybrid scheme we potentially have two different forms of a native method, uninstrumented and instrumented, but the stable context only refers to the bytecoded method. A simple fix would be to tag the negative native pc, e.g. doubling it and setting it to odd to mark the PC of an intrumented method. Such a scheme can easily provide a few tag bits. [2]

(* 9) This could be replaced by a byte instance creation primitive followed by rawPopDoubleAtPutNoFail:. But this feels ugly.

(* 10) and in a multi-threaded system it would be simple to block until all threads executing Smalltalk code advanced to a suspension point.

(* 11) On a 600MHz PIII with conditional counters inlined into the code, the computational subset of the macro benchmarks slowed down by 53%, but with out-of-line counters the slowdown was only -10%.

(* 12) One would have to provide some means of both recording the choice of which methods to instrument and "replaying" this choice.

B. References

[Tabatabai98] Ali-Reza Adl-Tabatabai, Michael Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, James M. Stichnoth, "Fast, Effective Code Generation in a Just-In-Time Java Compiler", In Proceedings of the SIGPLAN `98 Conference on Programming Language Design and Implementation, p. 280-290. Published as SIGPLAN Notices 33(5), May 1998.

[Arnold00] Matthew Arnold, Stephen Fink, Vivek Sarkar, Peter F. Sweeney, "A Comparative Study of Static and Profile-Based Heuristics for Inlining", in Proceedings of ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo '00), published as SIGPLAN Notices, 35(7), July 2000.

[Arnold01] Matthew Arnold and Barbara Ryder, "A framework for reducing the cost of instrumented code.", in SIGPLAN 2001 Conf. on Programming Language Design and Implementation, SIGPLAN Notices, ?(?), pp168-179, June 2001. [2]

[Arnold02] Matthew Arnold, Michael Hind, Barbara G. Ryder, "Online Feedback-Directed Optimization of Java, in SIGPLAN 2002 Conf. on Object-

Oriented Programming Languages, Systems and Applications, SIGPLAN Notices, ?(?), pp111-129, Nov 2002. [2]

[Hölzle91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In ECOOP'91 Conference Proceedings, Geneva, 1991. Published as Springer Verlag Lecture Notes in Computer Science 512, Springer Verlag, Berlin, 1991.

[Hölzle92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In Proceedings of the SIGPLAN `92 Conference on Programming Language Design and Implementation, p. 32-43. Published as SIGPLAN Notices 27(7), July 1992.

[Hölzle94] Urs Hölzle, "Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming", available on the web as
http://www.cs.ucsb.edu/labs/oocsb/papers/urs-thesis.html
http://www.cs.ucsb.edu/labs/oocsb/papers/hoelzle-thesis.pdf
http://www.cs.ucsb.edu/labs/oocsb/papers/hoelzle-thesis.ps.Z

[Krasner83] Smalltalk-80 Bits of History, Words of Advice, Glenn Krasner, Ed. Addison Wesley, 1983. [2]

[Ungar83] David M. Ungar. "Berkeley Smalltalk: Who Knows Where the Time Goes?" in [Krasner83], pp 189-206 (in particular p191) [2]

[Wolczko96] Mario Wolczko. Private Communication.