Context Management in VisualWorks 5i

Eliot Miranda
ParcPlace Division
CINCOM, Inc.
eliot@parcplace.com

Abstract

Smalltalk-80 provides a reification of execution state in the form of *context* objects which represent procedure activation records. Smalltalk-80 also provides full closures with indefinite extent. These features pose interesting implementation challenges because a naïve implementation entails instantiating context objects on every method activation, but typical Smalltalk-80 programs obey stack discipline for the vast majority of activations. Both software and hardware implementations of Smalltalk-80 have mapped contexts and closure activations to stack frames but not without overhead when compared to traditional stack-based activation and return in "conventional" languages. We present a new design for contexts and closures that significantly reduces the overall overhead of these features and imposes overhead only in code that actually manipulates execution state in the form of contexts.

Keywords:

Representation, Optimization, Activation Records, Stack Frames, Closures, Contexts, Smalltalk-80, VisualWorks

1 Introduction

Smalltalk-80 provides a reification of execution state in the form of *context* objects which represent procedure activation records [Ingalls76]. This feature provides a portable abstraction of execution state which has several advantages, including

- writing the system's debugger and the majority of the exception-handling system entirely in Smalltalk
- making processes persistent, typically as snapshots of a running system
- adding new computational mechanisms not directly supported by the underlying execution semantics, such as backtracking and dynamic (non-lexical) variable binding [Deutsch81] [LaLonde88]

A naïve implementation of contexts, as specified by the "blue book" [Goldberg83] definition, implies creating a context object for every method and closure activation, and eventually reclaiming a context for every return. It also implies copying arguments from the caller context to the callee context on every activation, and some form of interaction with the garbage collector to manage references from contexts to arguments and temporary values. These basic operations can be the source of considerable overhead and can dominate execution costs. One hardware implementation managed to achieve good performance with such an implementation [Deutsch83] but software implementations have been markedly less efficient [Ungar83] [Ingalls97]. Consequently, much work has been done to implement contexts more efficiently, both in hardware [Lewis86] [Samples86] and software [Deutsch84] [Caudill86] [Moss87] [Miranda87]. The earliest and highest performance software approach is presented in [Deutsch84] which describes the PS virtual machine[1]. The same techniques were evolved to implement a more portable machine, HPS[2], the virtual machine used in VisualWorks Smalltalk [CINCOM99]. The work in this paper is a modification of the HPS virtual machine.

[1] PS stands for either Peter's Smalltalk, or Portable Smalltalk

[2] HPS stands for High Performance Smalltalk

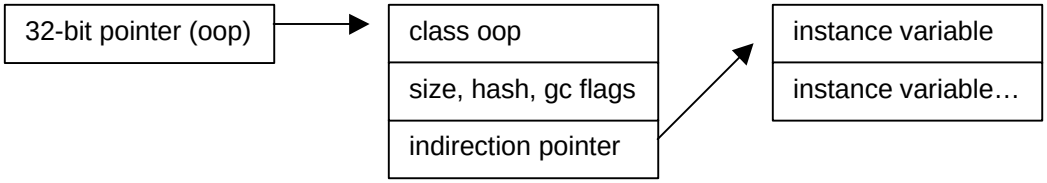
2 Context-Stack Mapping in HPS

Smalltalk-80 is a late-bound object-oriented programming language with full closures. Message sends create method activations which return to their senders. Conceptually closures are created within method activations or nested within closure activations within method activations. Closures can close-over the receiver and local variables of their lexically-enclosing-scoped closure and method activations.[3] The vast majority of Smalltalk activations obey stack discipline, and the only common exception from this is "non-local return" from a closure activation where a closure activation returns to the caller of its enclosing method activation. Effectively, then Smalltalk-80 obeys stack discipline for procedure (method and closure) activation. Hence efficient implementation of contexts can most obviously be achieved by stack allocation of contexts.

In HPS the key technique for efficient implementation of contexts is to map activations to stack frames during normal execution and only reify execution state when required. This, along with other significant optimizations outside the scope of this paper, is facilitated by translating Smalltalk bytecode into native machine code which is cached and used to execute all Smalltalk code. By mapping contexts to conventional stack frames HPS is able to use the host machine's native call and return instructions, and to pass arguments directly on the stack, avoiding copying arguments from caller to callee contexts.

In HPS there are three context representations. *Volatile* contexts are procedure activations which have yet to be accessed as context objects. These are conventional stack frames, created using native procedure call and prologue instructions, and reclaimed by stack discipline on executing native procedure epilogue and return instructions. *Stable* contexts are the standard object form of procedure activations, contexts proper. *Hybrid* contexts are a pair of a context object and its associated procedure activation. The context object acts as a proxy for the activation.

Objects in HPS are composed of a header and a body. The header holds an object's class, a flag word encoding the object's size, hash and garbage collector flags, and an indirection pointer which points to the body which holds the object's instance variables:



Hybrid contexts adapt this representation such that the class field holds a special class object used to mark the object as hybrid, the indirection pointer holds the frame pointer, and the actual indirection pointer is copied to a slot in the frame.

The stack frames of hybrid contexts need to be marked to distinguish them from volatile frames and need to refer to their hybrid context object. This requires three extra slots in the frame:

- The *saved pc slot*. A hybrid frame is flagged as such by using its return pc which has the advantage that it is set for free on procedure call, when the return pc is always pushed. To mark a frame hybrid its return pc is written into the saved pc slot and the return pc is set to zero. Consequently the return code sequence must be modified to check for a zero return pc and a *hybrid return* executed, of which more below.
- The *my data slot* which holds the indirection pointer referencing the context's body.
- The *my oop slot* which holds the pointer to the context object itself.

These three extra slots are only considered valid if the return pc is zero so the normal procedure prolog needs to make room for the extra slots but does not need to initialize them.

---

[3] This is a departure from the original blue book specification where anonymous functions were implemented as partially-initialized activation records called BlockContexts. Evaluation of an anonymous function involved making its activation record current, hence breaking stack discipline.

Volatile contexts are converted to hybrid contexts when a volatile context must be referenced as an object. This can happen in a number of situations (listed approximately from high to low frequency):

- A block is created which references its enclosing activation.
- A hybrid context is converted to stable, in which case its caller, if volatile, must be made hybrid. Thus stack walks, e.g. to deliver an exception or construct a stack trace for the debugger, convert volatile frames to stable. To avoid this when delivering exceptions, primitives are used to search the stack for frames marked as handler contexts, without hybridizing intervening volatile contexts.
- A process switch is made. Processes refer to their chain of activations via the top context. Thus operations like a blocking wait on a Semaphore entail making the top context hybrid.
- Volatile and hybrid contexts are housed in a stack zone composed of small stack segments usually 2k bytes in size. When a deep call chain overflows a stack segment a new segment is allocated, the top frame in the previous segment is made hybrid. Each segment starts with a dummy frame at its base, the *root frame*, uses this context to chain back to the stack segment. A *root return* is a return from a root frame, which calls a run-time routine to return to the context.
- Smalltalk code references the thisContext pseudo-variable which refers to the current context.

Volatile or hybrid contexts are converted to stable contexts whenever the full object representation of a context is required. This also occurs in a number of circumstances:

- When executing a return, if the current context is hybrid it must be converted to stable, since the context might be referenced by a block which has outlived its activation. This is a *hybrid return*.
- When executing a non-local return all hybrid contexts between the returning context and the home context from which it returns must be converted to stable. This is simply a compound form of the preceding single hybrid return case.
- When allocating a new stack segment if no empty stack segment exists, the least recently used segment is evacuated by converting all the frames in the segment, volatile or hybrid, into stable contexts.
- Whenever a message is sent to a hybrid context it is converted to its stable form.
- When a snapshot is made all volatile or hybrid contexts are converted to stable so that only objects are written to the snapshot file. Since the snapshot may be resumed on a different machine only the machine-independent stable representation is relevant.

Stable contexts are converted to hybrid in the following circumstances

- A non-local return or a root return returns to a stable context, which must be made hybrid before execution can resume
- A block activation references a temporary variable on the stack of a stable context, which is hybridized before the reference is made.
- A process switch switches to a process whose top context is stable which again must be made hybrid before execution can resume
- A snapshot is resumed and the snapshotting context must be hybridized before execution can continue

Note that whenever a stable context is hybridized a new stack segment must be allocated, potentially evacuating an entire stack segment to make room for the single context. However root returns are implemented such that a long chain of root returns recycles a single stack segment

This representation successfully eliminates the use of contexts for code that does not use blocks, and allows stack discipline to reclaim volatile contexts. However the representation still imposes considerable overhead for a number of high-frequency operations. The problem is not the creation of the hybrid context; which merely involves a normal object creation, and swapping the indirection pointer with the frame pointer. The significant problem is the introduction of hybrid returns.

3 Problems with Hybrid Returns

Since a hybrid context has an unusual object representation the incremental garbage collector is potentially affected by their introduction. To avoid impacting the garbage collector the header of a hybrid context is marked as an object containing raw bits rather than object pointers. Hence the incremental garbage collector never follows the indirection pointer, which is in fact a frame pointer, when encountering a hybrid context. When the scavenger runs all hybrid contexts have their indirection pointer reset, an insignificant cost. But there is one very serious side-effect. The HPS context representation stores the context's stack in a separate Array object. Because a hybrid context is marked as a bit object it cannot prevent the stack array from being garbage collected. Hence the stack array is not instantiated with the hybrid context and is instead deferred until the hybrid return, with the significant disadvantage that long return sequences may perform many object allocations. For example, the return from the top of a call chain composed of largely hybrid contexts, where each activation is poised to return to its caller results in a spate of allocations for each context's stack array. If memory is low this event can exhaust memory and crash the system.[4] A simple fix would be to add an extra slot to the frame which references the stack array. However, this would add yet another slot to the frame and would necessitate instantiating the stack array when the context is allocated, slowing down hybridization.

Since the return pc of a frame is used to mark a hybrid context, returns from hybrid contexts must check the return pc and perform a hybrid return if it is zero. Consequently any hybrid context must execute a special return sequence that performs this check. Just performing the check is expensive, slowing down simple activation-return code such as nfib[5] by as much as 35%. To avoid the check in the general case the native machine code generated for a return comprises a return instruction followed by sufficient no-ops to be able to replace the sequence with a hybrid return sequence. Each native method has a flag signifying if the return sequence is normal or hybrid. If a volatile frame must be made hybrid and its method's return sequences are normal then the sequences are rewritten as hybrid sequences.

On machines where the instruction cache must be flushed rewriting return sequences can be a significant issue. As described in [Deutsch84] an inline method cache is used to speed-up message sends. This technique involves rewriting the sequence of instructions for a send, a register load followed by a call, every time a send site is used on a different class of receiver, a process called relinking. On machines where instruction cache flushes are sufficiently expensive an alternative representation is used where the value to be loaded and the address of the call are held in data space and loaded indirectly. Hence relinking involves updating locations in data space and avoids the instruction cache flush. This scheme can be adapted to hybrid returns. For example, instead of a saved pc slot a frame could have a slot referencing the required return sequence, normal or hybrid. For a return the code generator would generate an indirect branch through the return sequence pointer. But this would add considerable overhead to returns, indirect jumps being relatively expensive on contemporary pipelined microprocessors.

To avoid the cost of the instruction cache flush when creating hybrid frames the Smalltalk bytecode compiler adds a hint bit to any method that explicitly creates a context, either by creating a block containing an up-arrow return or by using the thisContext pseudo-variable. The code generator generates hybrid return sequences for methods with the hint set. However, frames whose methods are not hinted can still become hybrid on a process switch where the current method is non-hybrid. Even though context switches are typically far less frequent than hybrid frame creations due to block creation, on some machines

---

[4] HPS's memory manager, apart from the scavenger, is designed to be controlled entirely from the image level. Hence it is designed to cope with allocation failure only where allocation occurs through an instance creation primitive, since the failure code can take appropriate action such as running a full garbage collection or growing memory, under full control of the image. Providing a means of transferring control to the image when return sequences exhaust memory is judged too complex to be worth while.

[5] nfib can be defined in class Integer as follows:

```
nfib self < 2 ifTrue: [^1] ifFalse: [^(self – 1) nfib + (self – 2) nfib + 1]
```

It has the property that the result returned is the count of the number of activations to compute it (hence the + 1 in the recursive case). Thus activations per second can be derived by dividing the result by the time taken to compute it. Suitable values are those that take around 30 seconds to compute.

the cost of the instruction cache flush can be sufficiently high that the system is modified to generate all return sequences as hybrid, incurring the performance overhead of the check on all returns in return for a considerably reduced context switch time.

If a hybrid return check indicates that the frame is hybrid then the context must be converted to stable and the frame's contents copied to the stable context. This is too complex to perform in-line and hence a hybrid return involves calling a runtime routine that actually does the work before returning to the caller. A plausible alternative would appear to be one where hybrid frames are given a special return address so that returning from a hybrid frame returns to the runtime routine without the need for a check. However, if interrupts are delivered on the run-time stack, then an interrupt delivered once the epilog has taken down the frame will overwrite the very state which needs to be written-back to the context. Hence the check (or an expensive indirect branch) and the run-time call are in general unavoidable.

The overhead of a full hybrid return is considerable. Adding a hybrid return to nfib slows execution by a factor of 10. On a machine where the average execution time of a single invocation nfib is 0.27 μseconds a hybrid return from nfib takes about 3 μseconds, of which around 0.1 μseconds is incurred by the return check.

In summary, the main problems posed by hybrid returns are

- Hybrid returns entail an object allocation for a context's stack array at a point where allocation failure is fatal.
- Hybrid return checks are relatively expensive, and hybrid returns are extremely expensive.
- Converting non-hybrid methods to methods with hybrid return sequences can be extremely expensive due to cache flushing costs; sufficiently expensive for it to be preferable to make all return sequences hybrid.

For these reasons we would like to eliminate hybrid returns. Our new design eliminates all copying of state from the stack to stable contexts at return time by stealing an implementation technique used by Lisp implementations, and by redesigning the cross-referencing between hybrid contexts and their stack frames.

4 Eliminating Hybrid Returns

Hybrid returns have three effects, the first two required by the implementation, the last required by the language. The first effect is to convert the unusual object representation of a hybrid context into the conventional object representation of a stable context. This is required by the garbage collector which cannot cope with this anomalous representation. The second effect is to free the hybrid context's frame for use in a subsequent call. This is required to enable the use of conventional procedure call and prolog code to create volatile contexts. Essentially, access to a hybrid context after its frame has been accessed is not possible, because the frame to which the hybrid context refers is either dead or in use by some other frame. The last effect is to copy the state of the hybrid frame into the stable context, which comprises the temporary variables of the frame and the pc. This is required by the implementation of non-local variable access in block closures.

Consider the inject:into: method from Collection, which may be familiar as a functional language's fold:

```
Collection methods for enumerating
inject: thisValue into: binaryBlock
    "Accumulate a running value associated with evaluating the argument, binaryBlock, with the
     current value and the receiver as block arguments. The initial value is the value of the
     argument, thisValue.  For example, to sum a collection, use:
         collection inject: 0 into: [:subTotal :next | subTotal + next]."

     | nextValue |
     nextValue := thisValue.
     self do: [:each | nextValue := binaryBlock value: nextValue value: each].
     ^nextValue
```

The block [:each | nextValue := binaryBlock value: nextValue value: each] has two free variables, nextValue and binaryBlock. The nextValue variable is assigned-to within the block and accessed from the enclosing context as the value returned. Hence both the block and its enclosing activation must share an lvalue[6] used to represent the variable. Since binaryBlock does not change value after the block's creation its value could be copied into an rvalue private to the block.

HPS distinguishes three types of blocks. Block closures that make no reference to variables in their enclosing environment are called *clean blocks*, and can be created at compile time. Blocks that only make read-only references to variables in their enclosing environment which cannot change value after the block is created are called *copying blocks*. The values of all so-referenced variables are copied into rvalues private to the block when it is created, allowing the block to dispense with the reference to its enclosing environment and avoiding hybridizing the frames comprising that environment. Blocks that either assign-to outer variables in the enclosing environment, or read variables whose value might change after block creation, are called *full blocks*. These must share the lvalues of these variables with their declaring environment. Historically Smalltalk-80 has implemented access to non-local variables as direct access to the locations on the stack of the context declaring the variable, and HPS followed suit. The significant consequence of this representation is that the stack state of a hybrid context must be copied into its stable form on returning from the hybrid context so that subsequent non-local access to temporaries on its stack from blocks see up-to-date values[7]. Hence the major reason for hybrid returns is to support non-local access to temporaries from within blocks.

If instead the lvalues for non-local temporaries were not on a context's stack, but in some heap object, as is the case in many Lisp implementations [Bartley86] the copying would be unnecessary. The remaining problem is to allow access to a hybrid context after its frame had been exited, so as to avoid the need for a hybrid return to convert a hybrid context to a stable context.

4.1 New Block Representation

To avoid non-local access to temporaries, the compiler now generates code to place all such shared temporaries in heap objects, using the Smalltalk Array class for convenience. Each scope level that introduces one or more shared temporaries allocates an Array with enough slots to hold all the shared variables. This array we call an *indirection vector*. With this modification a block can copy all required closed-over state into private rvalues, since though the contents of an indirection vector may change, the temporary location containing the vector itself is never assigned-to after initialization. To illustrate, this modification effectively re-writes the above inject:into: method as follows:

```
inject: thisValue into: binaryBlock
    | indVec |
    indVec := Array new: 1.
    indVec at: 1 put: thisValue.
    self do: [:each | indVec at: 1 put: (binaryBlock value: (indVec at: 1) value: each)].
    ^indVec at: 1
```

Three new bytecodes are added; one to allocate an Array of N slots, and a pair to do an indirect temporary fetch and store through a temporary holding an indirection vector. The allocation bytecode avoids having to store the binding for class Array as a literal in the method, which is unnecessary since class Array is well-known the virtual machine, and avoids having to send #new: to Array. The fetch and store bytecodes

---

[6] An *lvalue* is a memory *location* holding a *value* which may change. An *rvalue* is a *read-only value* which does not change.

[7] Another consequence of this is that non-local temp access must check for the outer context being hybrid or stable. In most cases the outer context would be hybrid, so for simplicity the code generated to access outer temps converts stable contexts to hybrid if required rather than coping with both representations. Hence in rare circumstances contexts can swap between stable and hybrid representations of they both are accessed as objects, which causes them to stabilize, and have their temporaries accessed by blocks. More importantly, hybridizing a stable context may cause an entire stack segment to be stabilized if the stack zone is full.

avoid having to send #at: or #at:put: to the indirection vector, avoiding unnecessary lookups and bounds checks.

This technique is common in the Lisp world [Bartley86] and requires no extra analysis in the compiler. More interesting is avoiding the conversion of hybrid contexts to stable on return.

4.2 Avoiding Premature Conversion from Hybrid to Stable

To avoid converting hybrid contexts to stable contexts on return the implementation must be able to convert to stable after a hybrid context's frame has been exited. We'll term hybrid contexts whose frame has been exited *dead hybrids*, and call those which still have a frame *live hybrids*. To do this requires three things:

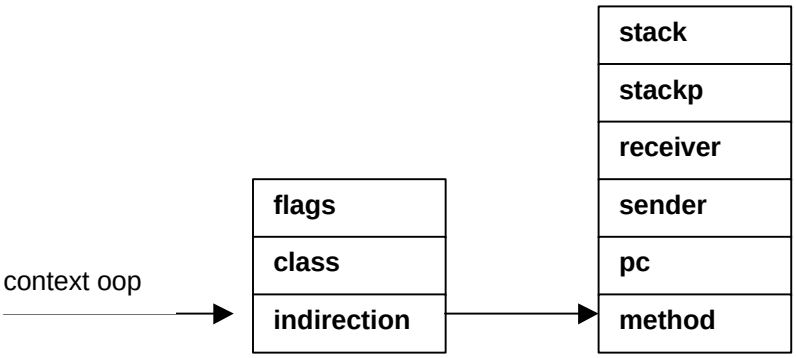
- that dead hybrids have sufficient state from which to construct a valid returned-from stable context
- that dead hybrids can be distinguished from live hybrids
- that live hybrids can be accessed as normal objects

To avoid converting live hybrid contexts to stable contexts when they are accessed for whatever purpose the implementation cannot use an anomalous object representation for hybrid contexts and must be able to derive from the hybrid context the equivalent behaviour obtained from a stable context. In general avoiding conversion on assignment isn't possible, since certain assignments to fields of a hybrid context may involve major manipulations of the stack, for example assigning to a hybrid context's sender may involve splitting a stack segment in two. But it is possible to avoid conversion for the majority of accesses.

Before delving into the details it helps to understand HPS's representation of objects, SmallIntegers and stable contexts.

Objects in HPS are either normal objects where a 32-bit object pointer points directly to a 32-bit aligned object header containing an object's class, size, identity hash and so on, or *immediate objects* where an object's class and state are encoded directly in the 32-bit object pointer. Since normal object pointers point to a 32-bit aligned header their least significant two bits are zero. Pointers with non-zero tags are used to encode 30-bit 2's complement SmallIntegers and 16-bit unsigned Characters.

Stable contexts in VisualWorks have the following shape:



The method field refers to the context's Smalltalk method object. The pc field holds an integer representing the program counter, and is the index of the current bytecode within the method. According to the blue-book specification the pc of a returned-from context should be the index of the last bytecode plus one. The sender field holds the context's caller, i.e. the context to return-to when the current context executes a normal return[8]. According to the specification a returned-from context's sender is nil. The receiver field either holds the method's receiver, or the BlockClosure from which a block context was

[8] When a block context executes a non-local return (an up-arrow return from within a block) it returns to the sender of its home context. There is a conventional static chain through the block context's closure which is stored in its receiver slot.

created, but this distinction need not trouble us, since the receiver field merely holds an arbitrary Smalltalk object. The stackp and stack fields represent the context's stack, the stack field holding an Array large enough to hold the context's arguments, temporaries and intermediate results, and the stackp holding an integer that is the one-relative index of the top of stack in the stack array.

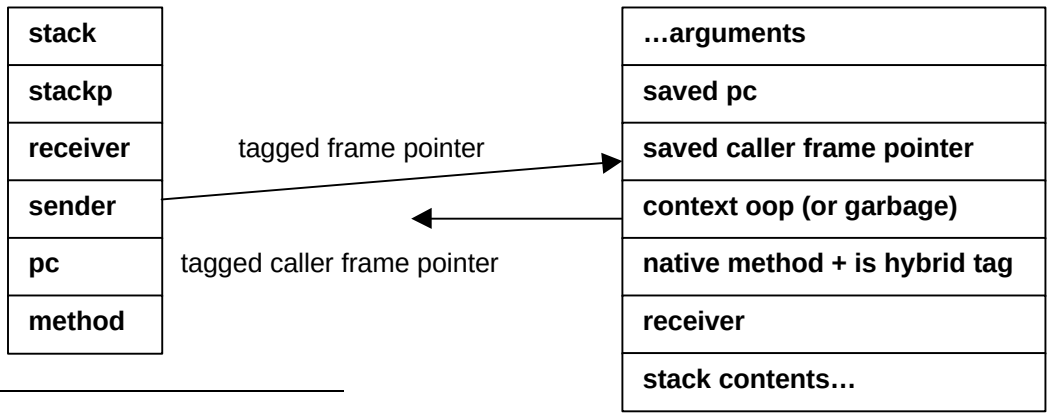
The following table lists how values are obtained for live and dead hybrids respectively.

field	value in live hybrid	value in dead hybrid
method	Initialized when created, accessed directly	Initialized when created, accessed directly
pc	Obtained from frame	Set to index of last bytecode + 1
sender	Obtained from frame caller	Set to nil
receiver	Initialized when created, accessed directly	Initialized when created, accessed directly
stackp	Obtained from frame	Initialized when created to SmallInteger 0
stack	Obtained from frame	Initialized when created to nil

When the implementation creates a hybrid context it

1. Instantiates a context object, setting the class field to one of a pair of constants, HybridMethodContext or HybridBlockContext, represented as SmallIntegers, which distinguish hybrid contexts from all other objects and block from method hybrids.
2. Sets the method and receiver fields to the frame's receiver and Smalltalk method.
3. Sets the sender field to the frame's frame pointer, represented as a SmallInteger. Since frame pointers are aligned on word boundaries this can be done by setting the least significant two bits to the SmallInteger tag bits without restricting the range of frame pointers.
4. Sets the pc field to the frame's saved frame pointer, also setting the SmallInteger tags. This will be explained below.
5. Sets the stackp to the SmallInteger 0 and stack to nil.

The frame must also be marked as hybrid and referring to the newly-created hybrid context. Since a hybrid is just a normal object the frame needs just one slot to hold the hybrid context, and we could use this slot as the marker, initializing it to zero when creating volatile frames. But writing zero into a slot on every frame build is extremely expensive. You will recall that the advantage of using the return pc to distinguish between volatile and hybrid frames is that the pc "flag" gets set "for free" when the pc is saved on call. The existing HPS frame format includes a pointer to the translated method,[9] and translated methods are also at least 32-bit aligned so we use the frame method's least significant bit to indicate a hybrid frame and leave the context slot uninitialized. This gives us the following stack format



[9] Although not strictly necessary, including the method makes it much easier to parse the stack

So setting the frame to hybrid entails setting the least significant bit in the frame native method slot and initializing the frame context oop slot with the newly-created hybrid context.

To distinguish dead hybrids from live we need to determine if a hybrid points to a live frame, and if so, that the frame is hybrid and its context is the hybrid context in question. It is straight-forward to check that a hybrid context's frame lies in the live portion of a stack segment, but since frames are of differing sizes, there is no guarantee that a dead hybrid points to a frame. In fact it may point to an arbitrary location on the stack. The implementation must therefore ensure that it is impossible to choose a frame pointer at random and accidentally find a match with any dead hybrid. This is easy to achieve since the frame format is sufficiently constrained.

The above diagram shows a single stack frame which contains some object pointers as arguments, receiver and stack contents, an address in the stack zone as caller frame pointer, the saved pc which is an address in the native code segment, and one possibly uninitialized slot, the context oop slot, which may contain an arbitrary value (since interrupt handlers may run on the stack and could write any value to the stack). Because the stack zone, the heap, and the native code zone are all in different parts of the address space they form disjoint sets of addresses. In particular, the only places in the stack where two adjacent slots can hold addresses in the stack zone are saved caller frame pointer/context oop pairs where the context oop contains a garbage frame pointer value. On either side of these slots lie a saved pc and a frame native method, whose addresses are in native code.

Whenever a hybrid context is accessed the implementation derives the context's frame pointer and copy of the frame caller by stripping the SmallInteger tags from the sender and pc fields. It then checks that the derived frame pointer lies below the base above the top of stack[10] of an allocated stack segment. It then compares the derived copy of the saved caller frame pointer with the slot pointed-to by the derived frame pointer, compares the context oop slot against the hybrid in question, and checks the is hybrid tag in the frame method slot. If the saved caller frame pointer matches then either the derived frame pointer is pointing to a frame, or is pointing to a stale value in an uninitialized context oop slot. But if it were pointing to a stale value then the comparison of the hybrid in question against the context oop slot would fail since this would be a native method slot which can't be confused with a non-immediate context object. Hence if the saved caller frame pointer matches the derived frame pointer is pointing to a valid frame, and hence the native method slot is also valid. If the frame's native method slot has the is hybrid tag set the frame is hybrid and the context oop slot is also valid. The garbage collector guarantees that object pointers are not recycled until the objects pointed-to are unreachable, so if the context oop slot matches the hybrid in question the hybrid is unambiguously pointing to its frame.

If any of these tests fail the implementation can infer that the hybrid has been returned-from, and can convert it to a returned-from stable. To do so it sets the object's class to either MethodContext or BlockContext, based on the hybrid's class, sets the sender field to nil, and the pc field to the returned-from value.[11] Clearly the correct value of the stack and stackp can only be obtained from the frame, but we depart from the blue book specification and consider it legitimate for a returned-from stable context to have an empty stack, leaving behind no record of the context's arguments or temporaries,[12] excepting closed-over variables in block contexts.

---

[10] In HPS stacks grow down.

[11] In HPS the pc field of a context can hold a native pc, an offset from the start of a native method, stored as a negative integer. Because of this one possible returned-from pc value is a negative constant, rather than the number of bytecodes plus one.

[12] However, there is one situation in which the stack contents are useful. According to the blue book specification if a context cannot be returned-from the virtual machine sends the #cannotReturn message to that context. A context will only get into a connot-be-returned-from state if assigned-to, so whenever a hybrid context is assigned to we also initialize the context's stack and stackp fields, copying the context's arguments to the stack array. Again we consider it legitimate to discard the temporary variables, and only

Finally the implementation must ensure that the test for live verses dead hybrids be made whenever necessary. For the run-time this means checking at points such as root returns where validity checks have to be made anyway. But the implementation must also ensure that any access from the Smalltalk level is also mediated, which it does using the translator and a few new primitives. The primitives are special versions of the basic access primitives defined in Object, #basicAt:, #basicAt:put:, #basicSize, #instVarAt:, and #instVarAt:put:[13], and a pair of primitives to mediate access to the stack, Context>>#localAt: and Context>>#localAt:put:. As before, for methods on Context, its superclasses and subclasses, the code generator translates access to instance variables into calls on run-time routines to mediate access. Any access of a dead hybrid first converts the dead hybrid to a returned-from stable as a side-effect. No reads of hybrid context state convert live hybrids to stable, so values such as the pc or stack contents are computed from the context's frame. Any assignment to a context or a context's stack locations does cause the context to be stabilized, except for the sender field. In general, assignments to hybrid contexts only happen in the debugger, by which time it is no longer important to preserve the hybridity of a context. But assigning to the sender of a context is done in some cases where performance is an issue, for example when performing co-routine-style computing with contexts [Deutsch81] [LaLonde88]. If a hybrid is already at the base of a stack segment then the sender can be assigned to the root frame beneath the frame. If not, the hybrid's frame and the frames "above" it (its callees) are moved to another stack segment, and the assignment made to the new root frame.

5 Results

The new scheme imposes no overhead on methods which obey conventional stack discipline and which do not reify execution state as contexts, except for the one word write required to initialize a frame's native method slot. Instead, overhead is shifted to handling of contexts where determining if a hybrid context is live or dead is more complex. However, as performance measurements show, this overhead is more than repaid by the elimination of hybrid returns.

The redesign has improved performance in a number of areas and has not reduced performance elsewhere. Exception –handling and block non-local return are doubled in speed. Assigning to non-local temporaries is also faster, simple #inject:into: examples being some 30% faster. Process blocking via Semaphore>>#wait, Process>>#suspend or threaded C call-outs [Miranda 97] are all improved since these operations create hybrid contexts and used to do hybrid returns. Surprisingly, Semaphore>>#wait and threaded call-out benchmarks are improved by about 33%!.

The new block representation is particularly successful at reducing the number of hybrid context creations. For VisualWorks 3.0, which uses the old representation, in an image containing 3878 block creation bytecodes, 30.5% are full block creations which cause a hybrid context to be created in block creation. In a VisualWorks 5i image containing 3912 block creation bytecodes only 9.3% are full blocks, 2.3% being full and 7% being full copying blocks. This image also contains 435 create array bytecodes (the bytecode used to create assignable non-local temps) showing that statically, only 11% of block creations assign to non-local temporaries. So 30.5% of block creations require an additional allocation in the old representation compared to 20.3% (9.3% + 11%) in 5i. However, create array bytecodes are considerably faster than hybrid context creation since they typically only allocate a single element array as opposed to a full hybrid context.

The new implementation also successfully increases the residency of frames in the stack zone. The old implementation would convert hybrids to stable contexts whenever any message was sent to a context, and convert a stable context to hybrid whenever a full block accessed a non-local variable in a stable context.

---

preserve the arguments. Smalltalk enforces constant arguments, so these cannot change after initialization of the stack.

[13] Since in HPS context objects are not indexable #basicAt:, #basicAt:put: and #basicSize are not strictly necessary, but we define them for completeness. BlueBook contexts are indexable, and the indexable fields are used to hold a context's stack. In such an implementation these primitives would be necessary. HPS's arrangement is more efficient since the stack does not have to be initialized on hybrid creation.

The new implementation also has two less fields, so it can fit more frames into the same stack space.

The time profiler is now more accurate since interrupting a computation does not change its code to use hybrid returns. But most importantly the system can no longer exhaust memory and crash executing long sequences of hybrid returns, since no allocation is done at return time.

6 Conclusions

Smalltalk-80's contexts are an elegant design of the reification of execution state, and the ability to reify execution state provides much power to the Smalltalk-80 system. We have redesigned an implementation of contexts and closures upon conventional hardware which imposes no overhead on code which does not reify execution state when compared to traditional stack-based activation and return mechanisms. This implementation preserves all useful features of the original Smalltalk-80 context specification. Associated overhead is imposed only on code which actually uses the reification facilities, and the overhead is significantly reduced compared to the preceding implementation. The implementation is suitable for both software and hardware implementations. This implementation demonstrates that contexts are an extremely affordable feature of efficient Smalltalk implementations, and that there is no reason to exclude them on performance terms.

Acknowledgements

My colleagues Steve Dahl and Diane Severeide were both instrumental in this work. Steve did the compiler work and Diane helped design the VM implementation. Dan Ingalls is due a huge debt of thanks for creating the challenge in the first place. This work wouldn't have been possible without the pioneering and unsurpassed work by L. Peter Deutsch and Allan Schiffman. Peter also kindly reviewed the paper.

References

[Bartley86] David H. Bartley and John C. Jensen, "*The Implementation of PC Scheme*", proceedings of the ACM Conference on Lisp and Functional Programming, Page 86, ACM, 1986.

[Caudill86] Patrick J. Caudill, and Allen Wirfs-Brock, "*A third generation Smalltalk-80 implementation*", Conference proceedings on Object-oriented programming systems, languages and applications , Page 119, ACM, 1986.

[CINCOM99] VisualWorks 5i Application Developer's Guide, CINCOM, Inc., 1999, DS601-0499,

[Deusch81] L. Peter Deutsch, "*Building Control Structures in the Smalltalk-80 System*", in BYTE Magazine Special Issue on Smalltalk, pp 322-346, Vol. 6, #8, August 1981, McGraw-Hill.

[Deutsch83] L. Peter Deutsch, "*The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture*", in Smalltalk-80, Bits of History, Words of Advice, G. Krasner, Ed, Addison-Wesley, 1983.

[Deutsch84] L. Peter Deutsch, Allan M. Schiffman, "*Efficient Implementation of the Smalltalk-80 System*", 11th Annual Symposium on Principles of Programming Languages, pp. 297-302, January 1984, ACM.

[Goldberg83] Adele Goldberg, David Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983, ISBN 0.201.11371.6.
Now out of print but available by combining
Adele Goldberg, David Robson, Smalltalk-80: The Language, Addison-Wesley, 1989, 0.201.13688.0 with
http://users.ipa.net/~dwighth/smalltalk/bluebook/bluebook_imp_toc.html

[Ingalls76] Daniel H. H. Ingalls, "*The Smalltalk-76 Programming System: Design and Implementation*", Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages, ACM, 1978.

[Ingalls97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay , "*Back to the future: the story of Squeak, a practical Smalltalk written in itself*", Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications , Pages 318 – 326, ACM, 1997

[LaLonde88] Wilf R. LaLonde, Mark Van Gulik, "*Building a Backtracking Facility in Smalltalk Without kernel Support*", Conference proceedings on Object-oriented programming systems, languages and applications , Page 105, ACM, 1988

[Lewis86] David M. Lewis, David R. Galloway, Robert J. Francis, Brian W. Thomson, "*Swamp: a fast processor for Smalltalk-80*", Conference proceedings on Object-oriented programming systems, languages and applications, Page 131, ACM , 1986

[Moss87] J. Eliot B. Moss, "*Managing stack frames in Smalltalk*", Papers of the Symposium on Interpreters and interpretive techniques , 1987, Page 229, ACM

[Miranda87] Eliot Miranda, "*BrouHaHa- A portable Smalltalk interpreter*", Conference proceedings on Object-oriented programming systems, languages and applications , Pages 354 – 365, ACM, 1987.

[Miranda97] Eliot Miranda, "VisualWorks Threaded Interconnect", http://www.parcplace.com/products/thapi/, ParcPlace Division, CINCOM, Inc.

[Samples86] ; A. Dain Samples, David Ungar, and Paul Hilfinger, "*SOAR: Smalltalk without bytecodes*", Conference proceedings on Object-oriented programming systems, languages and applications , Page 107, ACM, 1986

[Ungar83] David M. Ungar, David A. Patterson, "*Berkely Smalltalk: Who Knows Where the Time Goes?*", in Smalltalk-80, Bits of History, Words of Advice, G. Krasner, Ed, Addison-Wesley, 1983.

7 Appendix

Code for inject:into: in the old and new bytecode sets:

inject:into:

```
                                    1 <D4 00> create array size 1
                                    3 <4E> store local 2; pop
 1 <10> push local 0                4 <10> push local 0
 2 <4E> store local 2; pop          5 <DC 20> store local 2 at 0; pop
 3 <44> push self                   7 <44> push self
                                    8 <11> push local 1
                                    9 <12> push local 2
 4 <CF 00> make full block         10 <FA 00 02> make copying block (2)
 6 <CE 1B> non-immediate send do:  13 <CE 20> non-immediate send do:
 8 <66> pop                        15 <66> pop
 9 <12> push local 2               16 <D9 20> push local 2 at 0
10 <65> return                     18 <65> return
```

block within inject:into:

```
                                    1 <CB 02> push 2 copied values
 1 <2D> outer(1) push local 1       3 <11> push local 1
 2 <2E> outer(1) push local 2       4 <D9 20> push local 2 at 0
 3 <10> push local 0                6 <10> push local 0
 4 <90> send value:value:           7 <F0 94> send value:value:
 5 <DD 22> outer(1) store local 2; pop  9 <DC 20> store local 2 at 0; pop
 7 <2E> outer(1) push local 2      11 <D9 20> push local 2 at 0
 8 <65> return                     13 <65> return
```

The new code is more verbose, since it requires the two byte CreateArray bytecode and an extra store bytecode, and the indirect fetches typically take 2 bytecodes instead of 1 for the old code. But this adds an insignificant amount of code to the system since these facilities are used so infrequently. For example in a system with 25000 methods there are only 435 uses of the CreateArray bytecode.