The Interpreter is Dead (slow). Isn't it?

Position Paper for OOPSLA 99 Workshop:
"Simplicity, Performance and Portability
in Virtual Machine Design"

Blair McGlashan, Andy Bower.

Object Arts Ltd

"Smalltalk is not interpreted. It's been using a JIT for years. There hasn't been a commercial Smalltalk with an interpreter for at least a decade..."[1]

This is not the only statement we've seen recently from a well-known industry figure along similar lines. Not only are these statements incorrect - we know of at least one commercial Smalltalk system employing a pure bytecode interpreter, our own - but we take issue with the implication that the use of an interpreter is some how an outmoded concept. Our position is that the decision between an interpreting or dynamic translating VM is not so quite so clear cut as some would have us believe, even on platforms and in application areas where the traditional strengths of the interpreter (simplicity, low memory usage) are perhaps less of an issue. In particular we think that modern CPU's may change the parameters back in favour of interpreters for many applications.

Even if one doesn't accept that interpreters still have a place on general computers such as the PC for running traditional applications, there are classes of device and application where the use of interpreters is more compelling. Interpreted code requires very little start-up time, and is up and running "out of the blocks" almost immediately without the need for expensive dynamic analysis or translation. This is a useful characteristic for short-lived applications, such as applets embedded in browser pages.

The new class of consumer devices offering games playing capabilities combined with internet connectivity (as exemplified by the Sega Dreamcast) have enormous CPU power, but relatively limited memory. We think interpreters are likely to be a better choice for these machines as they can offer entirely adequate performance, faster start-up and lower memory usage.

Our experience stems from development of the Dolphin Smalltalk system. Dolphin is just about the fastest Smalltalk interpreter for the Win32 environment, although it remains noticeably slower than many other (dynamically-translated, or native code compiling) ST systems. However, the performance gap is not consistent across all measures

Dolphin-VisualWorks Performance Comparison

(266Mz P1, 96M, Win98)

STIC Benchmark	**Dolphin 2.1**	**Dolphin 3.01**	**VWNC 3.0**	**D2.1/VW**	**D3.0/VW**
Towers Of Hanoi	4715	4615	678	6.95	6.81
Alloc.	10990	11973	8936	1.23	1.34
Array Write	72791	70180	20520	3.55	3.42
Dictionary Write	7980	8074	3153	2.53	2.56
Floating Math	75870	84630	90170	0.84	0.94
Integer Math	14639	14501	14720	0.99	0.99
OrderedCollection Iterate	12617	8709	5090	2.48	1.71
OrderedCollection Write	21946	20262	8694	2.52	2.33
String Compare	3981	10964	18659	0.21	0.59
String Compare 2	7738	18355	18653	0.41	0.98

[1] From posting to comp.lang.smalltalk, 7th October 1999

As can be seen from the table[2], performance on some bytecode heavy benchmarks (integer and FP arithmetic) is actually better than a notable dynamic translator, Visual Works 3.0 (NC). There are some who probably don't believe that it is possible for a pure interpreter to compete with a dynamic translator on any measure, so they most likely won't believe these figures. However, they can take heart from the markedly worse performance on the two benchmarks, which are dominated by message sending. The Hanoi benchmark, in particular, is very "send heavy".

Of course, message send performance is critical in OO systems coded in "good" style, which emphasizes the use of many short methods. Smalltalk systems typically send a message for every few bytecodes interpreted. Thus the overall performance of Dolphin is two or three times behind VW, and in fact this is the kind of difference one observes in more macro benchmarks. At the application level, however, Dolphin applications typically offer a more responsive UI than VW, principally because Dolphin employs native widgets and is specifically designed to offer very high performance call-in and call-out, sometimes at the expense of interpreter performance.

So why do people mistake Dolphin's interpreter for a JIT compiler, albeit a slow one? Conversely, why don't they mistake it for a fast one? Let's consider the last question first. The design principles behind Dolphin emphasize integration rather than isolation.

External Interfacing

A significant factor in VM performance, which is often ignored, and certainly not measured by commonly employed benchmarks, is the speed of the external interfacing facilities. There are systems which attempt to provide a complete computing platform, for reasons of portability, marketing, or perhaps doctrine, but in reality the efficiency with which the VM can interface with external systems is important to many applications and application developers. This is particularly so for shrink-wrapped applications, and may be even more important for consumer appliances where specialist hardware is provided, high performance access to which is possible via operating system services. Duplicating these services may not be making most efficient use of available resources (development or machine), and so we believe that fast, generic, external interfacing capabilities are an important VM facility. There are design tensions here, particularly for the garbage collector, as many external / OS APIs expect fixed memory addresses to be maintained.

Memory Architecture

High performance and easy external interfacing is a particular strength of Dolphin, and contributes to its tight platform integration, but unfortunately this has not been achieved without significant cost. The memory architecture is the particular problem. Dolphin uses a two-level allocator that does not move memory blocks; furthermore it imposes an overhead for every memory store including stack operations[3]. The latter is the principal reason for poor message send performance - witness Dolphin's uncharacteristically poor performance on the message-send dominated Hanoi benchmark.

With the benefit of hindsight, we can see that we got the emphasis wrong here. The use of fixed memory blocks for external interfacing purposes is relatively infrequent, and it would have been better to have the image explicitly allocate from a fixed heap where possible and have the VM copy to a fixed heap in other cases.

We are currently in the process of implementing a compacting, generational GC, which we believe will dramatically improve Dolphin's performance. It is, however, too early at this stage to claim that our interpreter will be able to beat the VW translator on most benchmarks, much as we'd like to! We'd be happy to report results back later when we have them.

[2] An independent third party provided these timings of the Smalltalk Industry Council Benchmarks.

[3] We can elaborate further as to the nature of the memory store overhead at the workshop if desired

We have just seen how the external interfacing requirements of a modern VM can have a significant impact on the design of the memory architecture. However, the object representation within this architecture is also of great relevance.

In Dolphin, we still employ an Object Table and this has some benefits in rapid implementation of pointer switching (become:) and makes certain memory scans easy and fast. However, the cost of the indirection is very high on Intel CPUs. Our object pointers are object table entry addresses, rather than indices, but even so accessing an object often requires that a double –de-reference be performed in close proximity. This frequently forces the dreaded Address Generation Interlock (AGI) penalty[4], as well as reduced locality of reference. Putting the class in the object table entry might help to reduce the overhead imposed by an OT (since it is needed for message send purposes), but so far we haven't tried it.

A further disadvantage of the use of an OT is the increased per-object overhead to hold the pointer to the object body. An extra word per object is arguably too much for low memory devices.

In the near future we will be moving to a direct pointer memory as a prerequisite for the implementation of a new GC, and should thus be able to collect some figures as to the true cost of this indirection on the Pentium.

Optimization

When we started developing Dolphin at Intuitive Systems in early 1995, we were total Smalltalk neophytes, although we had some years experience building compilers and development environments, and our previous systems IS/1 and IS/2 employed bytecode interpreters. Even with this experience our initial attempt at a Smalltalk VM was so slow that we christened it "Intuitive Snailtalk". Over the following months we employed various techniques to speed up Dolphin about 200 times. Here are some of the most important:

- Stack-based activation records
- Assembler bytecode dispatch loop, with "threaded" dispatch[5]
- Selective implementation of critical primitives in assembler
- Global CPU register allocation for principal interpreter registers.
- Macro instructions combine certain common operations, e.g. increment, jump if nil.
- Detailed tuning at the assembler level using Intel's VTUNE tool to get optimal instruction sequences and ordering for the Pentium.

Of these the most dramatic improvements were achieved from the use of stack-based activation records, assembler bytecode interpreter with threaded dispatch, global register allocation, and optimization with VTUNE. The last may surprise, but it is our experience that optimization at this level can result in dramatic performance improvements on processors such as the Pentium. It is quite easy to double the speed of a routine, and interpreters can be very sensitive to even quite small differences in the performance of common instructions. We found, for example, that saving a single cycle in the *Push Self* instruction resulted in a 5% performance improvement overall!

Instruction optimization on machines such as the Pentium is a complex task[6], and indeed it is this which leads us to question whether a translator can necessarily do a good enough job in a reasonable amount of time. A bytecode interpreter is a relatively small and fixed set of code, and with tools like VTUNE it is possible to hand tune each bytecode instruction to a level far beyond even an optimizing C compiler. The move toward hybrid interpreter/optimizing translators which is now happening seems far more likely to narrow the gap on compiled native code, but such hybrid machines need fast interpreters too!

[4]For those not familiar with the Pentium, 3 instructions must typically separate the calculation of an address (for example loading the address of an object from an object table) from its use, otherwise an AGI penalty is incurred.

[5] Each instruction includes dispatch logic at its end, rather than returning to a dispatch loop.

[6] "How to optimize for the Pentium Processor", Agner Fog, 1996

Smalltalk’s block closure semantics and reflective capabilities make it rather stack hostile - it cannot be implemented using only a stack. We employ a hybrid design with contexts where needed for block closures[7]. All methods have activation record, whether or not a context is needed. The contexts no longer contain activation details. This was one of our most successful performance modifications - at least an order of magnitude difference was measured at the time. The effect is so dramatic because it virtually removes context allocation (though one can use a free-list to ameliorate this, it’s still slower than a stack) and the better locality of reference makes it much more cache friendly. Similar approaches have been used in other systems, for example Visual Age and Visual Smalltalk, but in those systems the traditional ease of examination of the run-time state is impaired by the poor abstractions provided in the image. This is not a fundamental problem, however, as we provide a StackFrame abstraction that has no less reflective capability than full object contexts.

Another of our most dramatic performance improvements was achieved by dispatching independently from each instruction rather than from a main interpreter loop (threaded dispatch). This actually has very little effect on the code size, as at most four instructions are required. The implementation of many bytecodes is very short, and the dispatch instructions can often be intermixed with the bytecode’s instructions in such a way that the overhead is reduced to the cost of the jump. Ironically it is often the case on the Pentium that having a few extra instructions to play with can help one to achieve an optimal instruction mix and avoid execution penalties.

Global register allocation is essential to minimize the load / store overhead of the interpreter registers. We found we had to employ assembler to completely achieve it because of the miserable number of registers available on the Intel architecture, and because Wintel C compilers typically don’t offer anything like the level of control needed.

Writing the core of the interpreter in assembler clearly reduces its overall portability, but the amount of code we are talking about is relatively small. The Dolphin VM consists of about 4000 lines[8] of assembler in the core interpreter. In addition there are 5200 lines of assembler primitives, of which about 2200 are for external interfacing purposes. Many of the primitives need not be in assembler at all.

On balance we think it is almost essential to employ at least some assembler if one wants to achieve optimal performance of a pure interpreter since C compilers simply aren’t designed for this task. For example, most C compilers cannot be persuaded to omit the range check on a typical interpreter case statement of 256 branches, even when the switch condition is a single byte. This check must then be manually patched out to avoid losing 15-30% on speed. However only a relatively small core of the interpreter need really be written in assembler and, with an appropriate architecture, it is possible to get up and running with a pure C interpreter, especially when porting to new hardware. Subsequently one can substitute assembler routines if and where necessary.

A common approach to speeding up bytecode interpreters is to open code every instruction including, for example, every instance of instructions such as *Push Temporary N* where *N* is encoded in the instruction. We have found that on the Pentium this can actually reduce performance since the flexible indexed addressing of the Intel instruction set can perform the memory access in a single instruction anyway, and the increase in core interpreter size has a negative impact on cache utilization.

Instruction Set

We’ve already briefly covered the GC above, and expect dramatic results from this. Perhaps more interesting, however, is the design of a new instruction set.

Standardizing the instruction set, á là Java, brings certain undeniable benefits, but it does compromise one’s ability to innovate in the area of instruction set design[9]. We hope that the Smalltalk community will allow

---

[7] Design of Dolphin VM allows for full lexical block-closures, but they are not currently implemented

[8] Includes comments, but not expanded macros.

[9]One might be able to translate to a different instruction set on load of course, particularly from a neutral representation.

things to mature a while longer (preferably another twenty years) before canonizing the bytecode instruction set.

There are essentially two areas we are interested in:

- CISC vs RISC
- Stack machine vs hybrid Register+Stack machine

Our early interpreters (IS/1 and IS/2) employed a RISC design, having about 7 instructions in all, and this did make the interpreter very small and simple, but the dispatch cost was overwhelming. RISC is appropriate for hardware and (and perhaps translators), but is absolutely the wrong approach for a software interpreter. Dispatch costs are high, particularly on modern machines, because the CPU cannot predict an indirect jump through a table, or to a calculated address in a register and it inevitably suffers a branch mis-prediction penalty, which can run to many cycles. Simple bytecodes can be implemented in a very few machine instructions, and thus the dispatch cost can easily dominate.

Apart from reducing the number of dispatches overall, a further advantage of the CISC design to an interpreter is to increase the number of instructions executed per bytecode, which opens up more opportunities for optimal instruction scheduling. Translators have the advantage of being able to optimize (assuming they have the time) at a more global level than a single bytecode, and a CISC design could offer similar opportunities to an interpreter. We have some macro instructions in our set and they certainly consume fewer cycles than the aggregate equivalent, but it is difficult to draw any conclusions about how much better a full CISC interpreter might perform overall.

We think then, that when designing an instruction set, there is a tension between the needs of translators, where a RISC design that is simpler to translate might be favoured, versus interpreters, where employing a CISC design to reduce dispatch overhead might be desirable.

It is remarkable that most VMs are Stack machines (including ours), yet a Register & Stack architecture seems likely to offer considerably better performance, especially for Smalltalk which consists mainly of sends to self, or a temporary, of zero or one arguments. Only two registers are thus needed in the common case, and memory writes on call/return can be significantly reduced. Allen Wirfs-Brock and Pat Caudill[10] describe an experimental VM of such an architecture, and its instruction set looks remarkably similar to that of Visual Smalltalk, a translator which certainly offers very high performance. A point of note is the authors' statement that the instruction set is "…not intended for direct interpretation and certain features, … would be inefficient if used by an interpreter."

We think this is a productive area for research (for example, in determining ideal instruction sets for translation versus interpretation), and would like to see more work done in this arena.

Conclusion

Ultimately we don't really question that an optimizing translator can most likely offer the best performance, but we are far from convinced that simple, faster, translators are necessarily capable of substantially better performance than a tightly coded interpreter. Even if the performance is better, on modern machines (which frankly have a surplus of CPU power for most current applications) the compactness, ability to "hit the ground running", and ease of construction of interpreters weighs significantly in their favour.

We think these arguments are particularly strong in the case of the consumer appliance marketplace, which is ultimately much bigger than that of the PC. In this arena, memory and external interfacing constraints may push interpreters back to forefront of virtual machine technology.

[10] "A Smalltalk Virtual Machine Architectural Model", Allen Wirfs-Brock, Pat Caudill, 1994. http://www.smalltalksystems.com/publications/avmarch.pdf