When are Bytecodes Faster than Direct Execution? *

James K. Doyle
J. Eliot B. Moss †
Object Systems Laboratory
Department of Computer Science
University of Massachusetts
Amherst, MA 01003, USA
{jdoyle,moss}@cs.umass.edu

Antony L. Hosking
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, USA
hosking@cs.purdue.edu

Research Paper submitted to OOPSLA '97
Subject area: Language design and implementation

Abstract

We demonstrate the tradeoffs between interpreting code in compact forms (such as the bytecoded method formats used in Smalltalk and Java), and direct execution of code (as in dynamic-translation Smalltalk and just-in-time Java compilers). Experiments with the interpreted Smalltalk language show that the widely held belief that direct execution is always faster than compact interpretation is not correct. Specifically, it is untrue for memory architectures with high cache-miss penalties, where the larger volume of less frequently repeated instructions inherent in direct execution outweighs the reduction in the number of instructions executed. The experiments presented demonstrate that the space of cycle costs for different types of cache misses can be divided into regions where a particular option is faster, and that these regions differ for different cache configurations. This suggests that interpreter designers must consider memory architecture and language characteristics when deciding whether to use direct execution.

*This work has been supported in part by gifts from Sun Microsystems Laboratories.
†Submission contact: Eliot Moss, Phone: (413) 545-4206, Fax: (413) 545-1249

1 Introduction

The hypothesis investigated in this paper is that for some memory systems of the future, interpreting compact forms will be faster than direct execution. In an interpreter, one can take a compact form of code to be executed, such as *bytecodes*, and interpret it with some number of instructions that perform the operations the bytecodes signify, and some number of instructions that represent the overhead of looping through each bytecode, determining which operation to perform, and branching to that operation. One can also take this compact form and generate native code, where once the native code is produced, it consists only of instructions performing the operations signified by the bytecodes. The elimination of overhead in terms of executed instructions has led many interpreter-writers to choose direct execution as necessarily faster [DS84].

This view of relative performance leaves out one critical issue: that of memory subsystem overhead. When using bytecodes or other compact forms to execute code, the interpreter must repeatedly execute a small pool of pre-compiled instructions, those implementing the bytecodes. When using direct execution, the interpreter repeatedly executes a larger pool of dynamically generated instructions. In order for an interpreter to avoid excessive memory subsystem overhead, it must be able to maintain its pool of instructions in cache. Interpreters that execute more instructions need larger caches to run with minimal memory subsystem overhead.

This difference in memory behavior has significant impact on performance because of trends in memory architectures. The higher clock rates being used in most new processors discourage designers from increasing primary cache size, because larger caches may require more than one clock cycle for an access. Because these processors are likely to keep using small on-chip primary caches (256KB or smaller), large pools of native code would swamp the caches of most modern and future architectures. This causes a large proportion of instruction fetches to be cache misses.

This would not have such significant consequences were it not for the fact that as processor speeds increase and memory system speeds lag behind, relative cache-miss penalties will increase dramatically [PH96]. These two factors in future memory architectures (small primary cache, fast processors relative to memory) mean that large pools of native code cause the speed of direct execution to be limited by memory speed. Bytecoded interpreters, with instruction pools that are more likely to fit in the cache, are more often free of this constraint and will therefore outperform direct execution under many architectures.

This tradeoff is tempered by the fact that data accesses are also a factor in memory subsystem overhead. In programs where the pool of accessed data is as large as the pool of executed instructions, the cache behavior of data accesses may be as significant to overall performance as the cache behavior of instruction fetches. The style of execution may have a less drastic effect upon interpreter implementations for which this is the case. However, this does not mean that the tradeoff in

execution style is reversed; it simply means that it causes a smaller performance difference.

An example illustrates the magnitude of the problem that direct execution causes: a benchmark used in the database community, TPC-B, is estimated to require 1 million instruction executions, using 800,000 distinct instructions [Nei]. In order to interpret such a benchmark quickly with direct execution, the instruction cache would have to be about 4MB in size! On the other hand, the bytecodes that could represent this benchmark might fit into caches of no more than 16KB, possibly even smaller.

Assume a processor that runs at 200 MHz and takes 1 cycle per instruction (cpi), with a memory bandwidth of 7.5 cpi. A program fitting in cache, such as a bytecoded interpreter, would run at processor speed, or 200 million instructions per second (mips), while one that is too large for the cache, such as a direct-execution interpreter, would run at memory speed, or 26.7 mips. Suppose that the overhead for a bytecoded interpreter and a direct-execution interpreter is such that the bytecoded interpreter takes 5 times as many instruction executions as the other. Because the direct-execution interpreter is running at memory speed, however, this improvement of a factor of 5 is mitigated by a factor of 7.5, resulting in the direct-execution interpreter performing slower by a factor of 1.5. This relationship would exist for any cache between 16KB and 4MB in size!

If the relative overhead between bytecoding and direct execution in terms of the number of instructions executed is x, and the relative overhead between memory bandwidth and processor bandwidth is y, the factor of difference in execution speed is x/y. As long as x is larger than y, direct execution is the right choice, but once y becomes larger than x, the cache behavior of a bytecoded interpreter makes it the more efficient. Circumstances like these will become more common in the future, making interpreters that use bytecodes or other compact forms worthy of consideration.

The following sections describe the Smalltalk interpreters with which these claims were examined, the relationship between these different Smalltalk interpreters and the hypothesis above, the experiments themselves, and the resulting performance analysis. The paper concludes with an analysis of how these results might be extrapolated to other languages and architectures.

2 Smalltalk interpreters

The different interpreting schemes possible for Smalltalk, referred to as Bytecoded, Threaded, and Translated, are described here and compared in terms of their time and space overheads. Bytecoded is the simplest, most natural scheme, but incurs a high overhead in dispatching virtual machine instructions because it interprets them at execution time and uses a naive dispatch technique. Threaded has some overhead because it, too, interprets compact forms at execution time; however, it has lower overhead than Bytecoded because of its more sophisticated dispatch technique. Trans-

lated avoids all overhead of dispatch by dynamically translating virtual machine instructions into native code and performing direct execution. Each of the following sections details how a different interpreter scheme executes Smalltalk bytecodes, the intermediate representation of compiled methods in Smalltalk. An overview of Smalltalk concepts is included in Appendix A.

2.1 Bytecoded

The Bytecoded interpreter scheme is the simplest one. A method being executed contains the bytecode numbers that represent its internal form, and the bytecode numbers are examined one at a time to determine which bytecode implementation must be executed. The logical program counter thus points to a bytecode number in the method object. The implementation of each bytecode is present as a section of pre-compiled C code, and the addresses of all bytecode implementations are stored in an array indexed by bytecode number.

Table 1 presents the compiled code that implements a **push-temporary-variable-1** bytecode on a SPARCstation 2. This sequence is used by a Bytecoded interpreter; similar code sequences will be shown for the other interpreters, to contrast the overhead required for each. Provided with each instruction is a fragment of C code that represents the action of the machine code. The table indicates with emphasized text those instructions that perform the essential work of the bytecode.

The bytecode is meant to load a value from the home frame's stack of temporary variables and store this value into the stack of the currently executing frame. This work is performed by the 3 instructions at addresses 0x5e50, 0x5e54, and 0x5e5c, which decrement the stack pointer, load the temporary variable value, and store the value in the stack. The overhead imposed by the Bytecoded interpreter scheme requires the other 7 instructions: one to load the bytecode number that the program counter references (0x5e60); two to form the base address of the array of bytecode addresses (0x5e58 and 0x5e64); two to index the bytecode array with the bytecode number and load the indexed address (0x5e68 and 0x5e6c); one to jump to the loaded address (0x5e70); and one to increment the program counter (0x5e74).

The overhead of 7 instructions, large compared to the average bytecode size of 3 instructions, gives the Bytecoded interpreter the worst execution handicap of any of the interpretation schemes. This handicap is worsened by the fact that the branch and the two loads have a dependency relationship among themselves and with the other instructions; this makes instruction scheduling across their delay slots more difficult, as demonstrated by the load-dependent jump, which will cause load stalls. On the other hand, this scheme is also the most compact, requiring only 1 byte of space per bytecode.

An interpreter could keep the base address of the bytecode array in a register, in an attempt to reduce the overhead of bytecode transition. However, the instruction sequence would only be

smaller, not faster, because of load delays. This effect might vary by bytecode, but the savings would likely be one cycle at most.

2.2 Threaded

The Threaded interpreter scheme is an attempt to improve upon the indexing necessary in the Bytecoded scheme. Essentially, the work of finding bytecode-implementation addresses is moved out of the critical path by doing it in advance. An auxiliary sequence of addresses is associated with each executed method object. Its addresses represent the addresses of the code sections implementing the corresponding bytecodes numbered in the compiled method. The logical program counter thus points to an address in this *threaded code*. Before execution, the threaded code must be generated and then stored along with the method object.

Table 2 presents the Threaded-interpreter implementation of the **push-temporary-variable-1** bytecode. As in the Bytecoded interpreter, the essential work of the bytecode requires 3 instructions, those at addresses 0x5410, 0x5414, and 0x5418. The overhead, here, though, requires only 3 additional instructions: one to load the bytecode implementation address that the program counter references (0x541c); one to jump to this address, the address of the next bytecode in the method (0x5420); and one to increment the program counter (0x5424).

The cost of indexing the bytecodes array to get the address of the correct code section is absent, at least on a per-bytecode basis. The load, jump, and increment, however, remain a significant overhead, which is worsened by the load stall introduced by the load-dependent jump. This scheme also introduces additional space overhead, since an address needs to be stored for each bytecode in each method, requiring 4 bytes of space per bytecode.

2.3 Translated

The Translated interpreter scheme is meant to improve upon the bytecode transitions required in both Bytecoded and Threaded schemes, by using *direct execution*. Each executed method is dynamically translated, generating a predetermined sequence of native-code instructions for each bytecode specified in the compiled method's bytecode-number sequence. The sequences generated for a given method are stored into a continuous array of translated code. The logical program counter points to an instruction in the translated code.

There are occasions when a bytecode is not translated into machine instructions, or is only partially translated, because of its complexity. Some bytecodes, for instance, involve branches that prohibit straight-line code sequences; for others, the expense of the bytecode is large compared to the cost of jumping to an out-of-line version, so that bringing it inline is not advantageous. In these cases, a jump out of the translated code and into a pre-compiled bytecode implementation

0x5e50	*add %l0, -4, %l0*	*sp_reg - -*
0x5e54	*ld [%l2 + -4], %g2*	*home_reg->stack[-1]*
0x5e58	sethi %hi(0x86c00), %g3	&bytecodes[0]
0x5e5c	*st %g2, [%l0]*	**sp_reg = home_reg->stack[-1]*
0x5e60	ldub [%l1], %g2	(int)*pc_reg
0x5e64	or %g3, 0x3c8, %g3	&bytecodes[0]
0x5e68	sll %g2, 2, %g2	bytecodes[(int)*pc_reg]
0x5e6c	ld [%g2 + %g3], %g2	bytecodes[(int)*pc_reg]
0x5e70	jmp %g2	goto bytecodes[(int)*pc_reg]
0x5e74	inc %l1	pc_reg++

Table 1: Push temporary variable 1, Bytecoded interpreter

0x5410	*ld [%l2 + -4], %g2*	*home_reg->stack[-1]*
0x5414	*add %l0, -4, %l0*	*sp_reg - -*
0x5418	*st %g2, [%l0]*	**sp_reg = home_reg->stack[-1]*
0x541c	ld [%l1], %g2	*pc_reg
0x5420	jmp %g2	goto *pc_reg
0x5424	add %l1, 4, %l1	pc_reg++

Table 2: Push temporary variable 1, Threaded interpreter

0x3e09c4	*ld [%l2 + -4], %o0*	*home_reg->stack[-1]*
0x3e09c8	*add %l0, -4, %l0*	*sp_reg - -*
0x3e09cc	*st %o0, [%l0]*	**sp_reg = home_reg->stack[-1]*

Table 3: Push temporary variable 1, Translated interpreter

0x3e200c	call 0x3e61f8	goto AddrOf_send[0]
0x3e2010	*ld [%l4], %l3*	*literal_reg[0]*

Table 4: Send selector 1 with 0 arguments, Translated interpreter

is made, and the logical program counter is updated with the current hardware program counter value. When the bytecode implementation completes, a jump is made back to the address in the program counter.

Table 3 presents the Translated-interpreter implementation of the **push-temporary-variable-1** bytecode. As in the other two interpreters, 3 instructions implement the work of the bytecode, but here they are the only instructions needed. Rather than being followed by bytecode-transition code, these 3 instructions are simply followed by the implementation of the next bytecode in the method.

Table 4 presents an example of an out-of-line call within translated code. The bytecode shown is a **send-0-arguments-selector-1** bytecode. The first instruction executes a call instruction to the address of a send implementation, which first stores the return address in the logical program counter and then dispatches the send. The second instruction is in the call's delay slot, and loads the literal variable value representing the selector to be sent; only one cycle, then, is used to transfer control from translated code to the pre-compiled implementation. One more cycle, that of a jump to the logical program counter value, will be required to transfer control back to the translated code.

The cost of accessing the program counter, jumping indirectly to the address it points to, and advancing the program counter is altogether absent in the Translated interpreter. The jumps into pre-compiled code sections introduce some additional cost, but this cost is not large relative to the cost of the bytecodes that require them. The overhead of translation is minimized by keeping the translated code object in a permanent code table. The space overhead introduced, however, is significant: each bytecode requires an average of about 3 native code instructions, yielding about 12 bytes per bytecode.

2.4 Time and space considerations

The two primary factors of interest in the performance of these three schemes are time and space. As stated above, the Translated interpreter has no inter-bytecode execution overhead, while the Threaded interpreter has 3 instructions and 1 stall in overhead, and the Bytecoded interpreter has 7 instructions and possibly 2 stalls in overhead. Space concerns vary in a reverse way: the Bytecoded interpreter uses 1 byte per bytecode, while the Threaded interpreter uses 4 bytes per bytecode, and the Translated interpreter 12 bytes per bytecode.

Of course, different compilations and instruction scheduling may result in different values for instruction overhead, and auxiliary data structures may affect the values for space overhead. These figures, though, give a rough idea of the magnitude of the difference. Actual statistics give a more concrete idea of the time/space tradeoff of the three schemes.

	Raw number	Cycles incurred
Bytecoded Inst	583694445	841168221
Bytecoded Stall	67699214	67699214
Bytecoded Annul	12119055	12119055
Bytecoded Sum		920986490
Threaded Inst	489613663	726093646
Threaded Stall	73074891	73074891
Threaded Annul	12468770	12468770
Threaded Sum		811637307
Translated Inst	456704950	656133418
Translated Stall	42816425	42816425
Translated Annul	11425249	11425249
Translated Sum		710375092

Table 5: Time Overhead

	Total bytes in 1029 methods	Average per method
BC numbers	24080	23.40
BC numbers + auxiliary	64691	62.87
Thread addrs	96320	93.61
Thread addrs + auxiliary	145712	141.61
Translation	300608	292.14
Translation + auxiliary	388168	377.23

Table 6: Space Overhead

Measurements of the tradeoffs between the UMass Smalltalk interpreters, using the benchmarks detailed later in this paper, follow. In terms of time overhead, one can look at number of instruction executions, number of stalls, number of annulments, and number of cycles due to these factors (equivalent to assuming a perfect memory subsystem with no cache misses). In terms of space overhead, one can look at bytes allocated to compiled methods, both as totals and as an average across the methods, both with and without auxiliary data structure and headers; the bytes used to record bytecode numbers, to record threaded addresses, and to record translated instructions, can be compared.

Tables 5 and 6 present these statistics. The statistics support the theoretical fi gures given above. The space ratio for raw bytecodes, threaded addresses, and translated instructions is about 1:4:12. Auxiliary structures, particularly the literals contained along with the method bytecodes, make the ratio less extreme, closer to 1:2:6. The overall time ratio is about 9:8:7, which indicates that the smaller overhead for Threaded and Translated interpreters does indeed produce an overall savings. The ratio of inter-bytecode overhead between the three interpreters does not correspond exactly to the ratio of execution time, of course, because the interpretation of bytecodes accounts for only part of the overall system. A surprising aspect of the time statistics is that the Threaded interpreter has more stalls than the Bytecoded interpreter; the Bytecoded interpreter, however, might allow better scheduling of its inter-bytecode instruction sequence because it has 7 instructions to schedule instead of only 3.

3 Tradeoff between cache and execution

Both the time *and* space tradeoffs described above determine the overall speed of a particular interpreter, since the effects of cache behavior translate spatial characteristics into a time cost. Spatial characteristics and their relationship to cache behavior are often ignored in assessing the speed of programs, but to do so is a short-sighted omission. The model of interpreter performance considered here includes processor *and* memory subsystem performance.

The performance model used in this paper is a linear weighted model of different execution statistics, referred to here as *cost components*. Cost components relating to the processor include the following:

E_t number of executions of each machine instruction type

S_i number of processor stalls following integer load

S_f number of processor stalls following floating-point load

A number of annulments of instructions

Cost components relating to the cache include the following:

I number of cache misses in fetching instructions

D_r number of cache misses in reading data

D_w number of cache misses in writing data

For a given interpreter scheme, cache configuration, and benchmark, a simulation of an execution can be performed, yielding values for each of the above cost components. The simulated total cost is a measure of time in units of processor cycles, calculated by weighting each of the cost-component values by some number of cycles that this component incurs, called a *cost factor*. Cost factors relating to the processor include the following:

e_t number of cycles processor takes to execute an instruction of type t

s_i number of cycles spent stalling for an integer-load

s_f number of cycles spent stalling for a floating-point

a number of cycles used by annulling an instruction

Cost factors relating to the cache include the following:

i number of cycles to fetch an instruction from memory following cache miss

d_r number of cycles to read data from memory following cache miss

d_w number of cycles to write data to memory following cache miss

The formula for the simulated total cost, then, is:

$$T_s = \left(\sum_{t \in INSTRS} e_t E_t \right) + s_i S_i + s_f S_f + aA + iI + d_r D_r + d_w D_w \quad (1)$$

The cost factors e_t, s_i, s_f, and a are dependent on the processor, and the cost factors i, d_r, and d_w are dependent on the memory subsystem. Thus, even assuming the same processor (and the same processor-dependent cost factors), there is still a large range of possible cost factors, and thus a large range of possible total costs.

In fact, given a particular processor and cache configuration to be modeled, and a particular execution (the sequence of instructions generated by one interpreter for one benchmark), the simulated total cost T_s can be stated as a *function* of the cost factors i, d_r, and d_w. The processor-dependent cost factors become a constant ($P_s = (\sum e_t E_t) + s_i S_i + s_f S_f + aA$), and the cost components I, D_r, and D_w become coefficients for the independent variables.

Since there are three different interpreter schemes being compared in this paper, for a given processor, cache configuration, and benchmark, there are three distinct $T_s(i, d_r, d_w)$ functions. Each of the interpreters yields a different set of values for P_s, I, D_r, and D_w, and thus has a T_s with a different shape in the cost-factor space formed by the three independent variables.

As shown in the section above, the Translated interpreter has the lowest time overhead, the Threaded interpreter slightly higher time overhead, and the Bytecoded interpreter the highest time overhead. This results in the P_s value being lowest for Translated and highest for Bytecoded. Similarly, because Bytecoded has the lowest space overhead and Translated the highest, the values I, D_r, and D_w are in general lowest for Bytecoded and highest for Translated.

This implies that part of the cost-factor space, particularly near the origin where P_s dominates, represents a region where the Translated interpreter is fastest and the Bytecoded interpreter slowest. However, it also implies that further out in the cost-factor space, where i, d_r, and d_w grow large enough to allow I, D_r, and D_w to dominate, the Bytecoded interpreter will be fastest and the Translated interpreter slowest. (The Threaded interpreter is expected to be sandwiched in the middle for the majority of the cost-factor space.)

This means that one can find curves in the cost-factor space that represent "crossover boundaries", where the disparity in memory-subsystem costs and the disparity in processor costs are balanced, yielding the same overall performance for the different interpreters. At these crossover boundaries, the ratio of memory bandwidth to processor bandwidth more or less matches the ratio of interpreter execution overheads. Being aware of the crossover boundaries is critical to the effective design of an interpreter, because the existence of these crossovers in a region of positive cycle costs indicates that the tradeoffs need to be considered to decide between implementations.

4 Experiments

Exploring the cost-factor space required first verifying our simulator and the performance model described in Equation 1, then investigating the variance in cost component values provided by the simulator. With these two steps accomplished, the simulator could be used to produce cost component values under different cache configurations, and the performance model used to predict the shapes of the three T_s functions under these different cache configurations.

The verification step compared simulated time cost to real time cost; a linear-regression fit applied to real elapsed times as a function of simulated times yielded a function with extremely low error, verifying the model and simulator as useful. The variance step compared the component costs under different stack allocation patterns (a source of variance even when otherwise repeatable benchmarks are used). Standard deviations in cache miss rates, as a ratio to the miss rate values themselves, proved to be very low, showing that the miss rate figures are reliable up to at least 3 digits in most cases. The prediction step showed that many cache configurations yielded cost-factor spaces where the Bytecoded interpreter dominates. Although the crossover point varied greatly for the different caches, in general it occurred as the i and d_r cost factors were increased.

4.1 Experiment specifics

The experiments were run on a SPARCstation 2 running SunOS 4.1.4. This machine has a 64KB unified, direct-mapped cache with 32-byte lines and write-invalidate policy.

The cache simulator used was a version of Sun Microsystem's Shade tools modified in our laboratory. This cache simulator uses Shade primitives to step through instruction execution, noting memory accesses and delivering them to a simulated cache structure, also counting load stalls, annulled instructions, and the execution frequency of each instruction type. It thus provides all of the cost components that are used in Equation 1. The simulator does not take into account such things as context switches and more complicated stall patterns.

The Smalltalk version used was the April 1996 UMass Smalltalk system, supporting all three interpreter types and instrumented with timing mechanisms and stack-allocation-adjustment mechanisms. The mechanism for stack adjustment essentially determines the height of the stack at the initial frame of execution and pads the stack with allocation up to a certain alignment and an offset from that alignment. The alignment and the offset are run-time parameters. This mechanism ensures that stack allocation patterns can be controlled and made reproducible, and was used in the variance study.

The chief benchmark used is a variant of the Interactive benchmark originally used in our laboratory [HMS92]. This benchmark is composed of benchmarking operations included in the

standard Smalltalk environment [Kra83]. Interactive includes all of the "macro" methods, those that reproduce what users do in a typical interactive session. Not only are these the most realistic benchmarks, but they also use the largest pool of methods, thus generating a large pool of native methods in the Translated interpreter case. The hypothesized cost-factor crossovers in time cost depend on the context of a large pool of native methods that fill the cache, and this benchmark provides that context. The benchmark can be repeated any number of times without leaving the Smalltalk environment, providing a tunable knob to extend the benchmark length. In addition, the benchmark produces completely reproducible instruction traces, because of the absence of references to the machine clock or external files.

Additional benchmarks were used to extend the results derived from the chief benchmark. These included the following Smalltalk programs: Richards, Lambda, Swap, and Destroy [HMS92]. The first is an operating system simulator, the second a pure lambda-calculus interpreter, and the last two variations on tree-modification activity. All of these use a much smaller pool of methods than the Interactive benchmark, and are thus less likely to allow the hypothesized performance crossover between Bytecoded and Translated interpreters.

4.2 Validation

The methodology and results of the verification step are given in more detail here. One can measure real elapsed times that relate in a linear way to the simulated times that are *predicted* by the simulator and model used in this study. An accurate simulator and model will produce a linear fit with very little error. The parameters of the fit, the slope and the intercept, take into account details that cannot be captured by the simulator and model, such as different initial cache states and the interference of other processes.

The real elapsed times were taken by using a timing-instrumented version of the Smalltalk interpreters, where the benchmark code is bracketed by primitives that cause the implementation to record the machine-clock time. The system calls to obtain the machine-clock time come as close as possible to the beginning and ending of the actual benchmark code, to make sure the elapsed time reflects only the benchmark. The three interpreters were run on the SPARCstation 2 in single-user mode, to keep the interference of extra processes to a minimum. Each interpreter was run on the Interactive benchmark, once with the benchmark alone, and once with the benchmark repeated 5 times. Each of these runs was performed 20 times, to get a large number of samples for elapsed time. All of the interpreters and benchmarks were executed in one run before proceeding to the next identical run, in order to keep the initial state of the cache relatively well-shuffled.

For each interpreter version and number of benchmark repetitions, the mean of the 20 elapsed time samples is listed Table 7, along with the minimum and maximum samples. The Translated

interpreter is fastest, and the Bytecoded interpreter slowest, with the difference becoming greater when the benchmark is repeated more times. The difference between the various times is much greater than the error involved, indicating that this ordering is consistent.

Simulated times were generated by first running the simulator for each interpreter and the two different-length benchmark runs. The cache configuration used in the simulator was the known configuration for the SPARCstation 2 cache, detailed earlier in this paper. The executables on which the simulator was run were the same as those used for the real timings, to ensure that the instruction sequences analyzed were the same. The simulator yielded all of the cost components needed for the performance model in Equation 1.

In addition to the cache configuration, the cost factors for the SPARCstation 2 needed specification. Individual e_t values were determined, from 1 cycle for simple operations like **or** to upwards of 100 cycles for complex operations like **fdiv**, floating-point division [ROS90]. The value for s_i and s_f were set to a stall of 1 cycle. The value for a was set to 1 cycle, the cost of skipping any annulled instruction. i and d_r were set to 24.5 cycles, an average of 24 cycles and 25 cycles, which happen with fairly equal frequency. d_w was set to 4.5 cycles, a similar average of 4 cycles and 5 cycles [Irl91].

Given these cost factors, the performance model yielded simulated cycle costs for each of the interpreters and benchmark runs. The cycle costs were converted to times using the processor speed: 40 MHz. Regression analysis was then performed, using the simulated time values as samples for the independent variable, and using the corresponding minimum-sample real times as samples for the dependent variable. The minimum samples were used for the real times because any deviation in real times is due to *external* factors like other processes. The minimum sample is the sample perturbed by the least amount of this interference, the sample closest to the actual benchmark time needed.

The regression fit yielded the following linear function predicting machine elapsed times from simulated times: $T_{me}(T_s) = 1.0413 \times T_s + .0632$ seconds. The regression F-statistic, which measures the statistical difference between the data and a random distribution, has a value of 29307.45 and a significance of 6.98×10^{-9}; such a high value and low significance indicates that applying a linear fit to the data is valid. Among the residual errors between the linear function and the sampled times, the largest residual error (as a percentage of the sampled value itself) was 1.3%, indicating that the fit was very exact. The graph in Figure 1 plots the elapsed vs. simulated function points predicted by the above function, with error bars indicating the values of the actual samples. Given such a good fit to real elapsed times, the simulation can be judged as usefully predictive.

Reps	Version	Mean	Min	Max	Max-Min	(Max-Min)/Mean
1	Bytecoded	6.4341838	6.4301640	6.4514560	0.021292	0.003309
	Threaded	6.2022816	6.1992060	6.2078830	0.008677	0.001399
	Translated	5.9788356	5.9759360	5.9839650	0.008029	0.001343
5	Bytecoded	32.0152433	32.0068660	32.0310150	0.024149	0.000754
	Threaded	31.1846563	31.1791100	31.1947410	0.015631	0.000501
	Translated	28.0134462	28.0074990	28.0228530	0.015354	0.000548

Table 7: Measured execution timings (seconds) on a SPARCstation 2

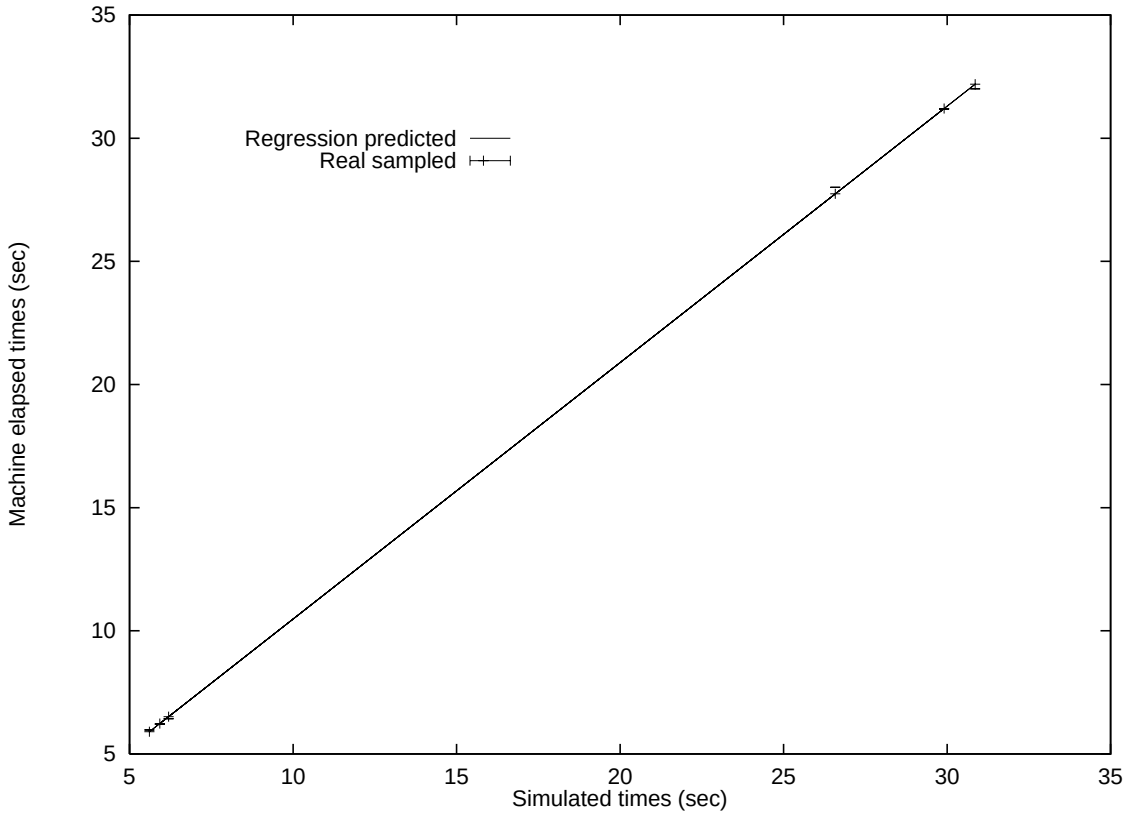

Figure 1: Linear fi t plot of elapsed time vs. simulated time

4.3 Variance

Even if the simulator and model are adequate, the usefulness of results still depends on the reproducibility of the analyzed instruction and data-access sequence. For the most part, this depends on whether the benchmark generates a consistent instruction trace. Benchmarks that access external files or access the machine clock, for instance, run the risk of executing different branches of conditionals that depend on these values. Even if such problems are eliminated, and a consistent instruction trace generated every time, there may remain an aspect that is not reproducible: the data-access pattern.

Accessing data on the heap produces repeatable data-access patterns, since the sequence of allocation locations depends only on how the heap segment is set up in memory. Accessing data on the stack, however, can produce non-repeatable data-access patterns, because the sequence of allocation locations depends on the configuration of the stack when the base frame of the program begins execution [Goo96]. This configuration can differ between executions of the same benchmark with the same executable. The contents of the stack are initialized to the environment variables of the parent process and the arguments passed to the executable, which may well differ slightly and yet have no effect on the instruction sequence experienced.

The UMass Smalltalk system has options for explicitly aligning the stack to a certain power. Aligning the stack to power n ensures that all allocations on the stack begin at an address $i \times 2^n$, where $i \in \{0, 1, 2, \ldots\}$. Thus, if a stack allocation occurs at address x in one execution of a benchmark, and occurs at address y in another execution of that benchmark, the two addresses are related by $y = x + c \times 2^n, c \in Z$. If the cache size is 2^k bytes, $k < n$, then the locations of x and y in the cache will be equal. Thus, the data-access pattern from the cache's point of view will be consistent, and cache behavior will be consistent.

In addition, the UMass Smalltalk system allows the stack to be offset some number of words from its alignment. Thus, a given benchmark can be run with different initial stack offsets, to compare the effects of these different data-access patterns. Examining how the resulting cache cost components vary with different stack offsets indicates how accurate the cache results are, and whether the results might simply be an artifact of a *particular* stack configuration.

This part of the study was designed to examine variance for the different cost components (instruction-miss rate and data-miss rate, corresponding to I, D_r, and D_w), under each interpreter, and for different caches. The expectation was that the variance contributed by the stack deviation would be low, and that the distributions of the miss rate values would be fairly uniform, regardless of the interpreter or cache size. An odd distribution or significantly different distributions for different interpreters or caches would have indicated an unexpected bias in the cache analysis.

Each interpreter was run under the simulator using cache sizes of 8KB, 64KB, and 512KB,

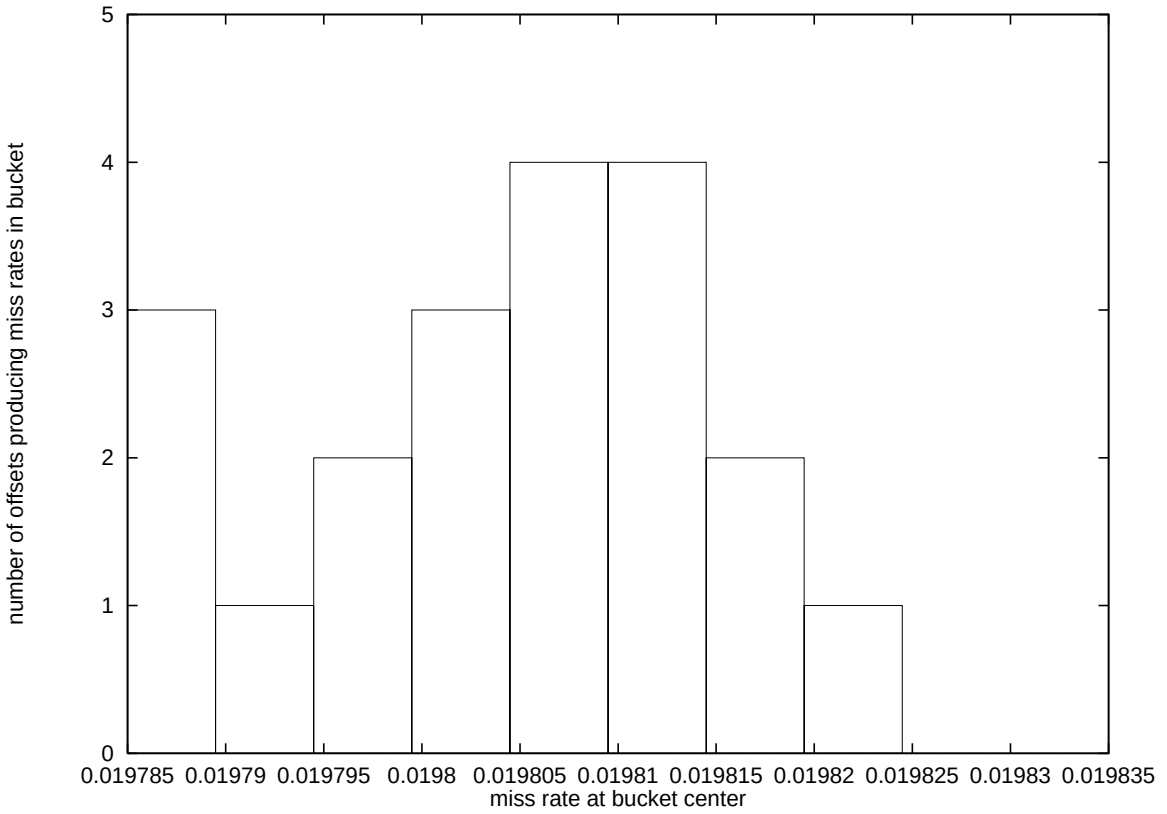

Figure 2: Distribution of data miss rates for 512KB unified cache, Bytecoded

both split and unified caches, with the benchmark running only 1 repetition. For each of these runs, 20 different stack offsets were used, chosen from a uniformly random distribution, to avoid choosing biased offsets that corresponded in some way to the cache structure. Distributions for each run were examined, and they were all relatively uniform, for both the instruction-miss and data-miss rates. Figure 2 shows one such data-miss rate distribution, for the Bytecoded interpreter, with a 64KB unified cache. The results were consistent across the different interpreters and cache sizes.

The distributions can also be analyzed to produce standard deviations, which indicate how accurate the results are, with respect to the magnitude of the results. Taking the ratio of the standard deviation over the mean indicates how large a fraction of the results can be affected by the deviation. Table 8, for instance, shows these ratios for each of the tested cache sizes, using the Bytecoded interpreter and a unified cache.

These ratios are very similar for the different cache sizes; all are less than $1/1000$ and greater than $1/10000$, indicating that the first 3 significant digits of data-miss rate values can be considered valid. This affords much confidence to the simulation results, since 3 significant digits is generally all that is required for miss rates. The deviation ratios for other interpreters and caches are all of this magnitude, except for a few that are between $1/100$ and $1/1000$. Given these consistently low variance results, one can be certain that simulation results are justifiably a result of the benchmark/executable characteristics, and *not* of a particular stack configuration.

4.4 Predicted cycle costs

With a suitably predictive simulator and model, one can examine how the different interpreters would compare for different memory subsystems. This part of the study compares the simulated performance results for each interpreter, run on the 1-repetition benchmark and the 5-repetition benchmark. Each of these runs was performed under a set of cache configurations that spans the space of likely caches.

All simulated configurations were single-level caches. The default configuration used was that of the SPARCstation 2, and various configuration parameters were varied from this default. Unified caches of sizes 8KB to 1MB, by powers of 2, were examined, and split caches of sizes 8KB, 64KB, and 512KB were examined. Also, associativity was varied from direct-mapped to 2-way to 4-way, and line size was varied from 4 bytes to 32 bytes, with 4-byte sub-blocking turned off and on.

The resulting cost component figures for each interpreter, benchmark, and configuration were then substituted into the simulated-cost equation (Equation 1), using cost factors for a familiar architecture, the SPARCstation 2. Thus, this experiment essentially probed results that would occur if the cache on a SPARCstation 2 were changed in different ways.

Size	Mean	Stddev	Stddev/Mean
8	2.1419e-01	9.5201e-05	4.4447e-04
64	9.6512e-02	5.2148e-05	5.4033e-04
512	1.9807e-02	1.0411e-05	5.2562e-04

Table 8: Standard deviation in data miss rates for unified caches, Bytecoded

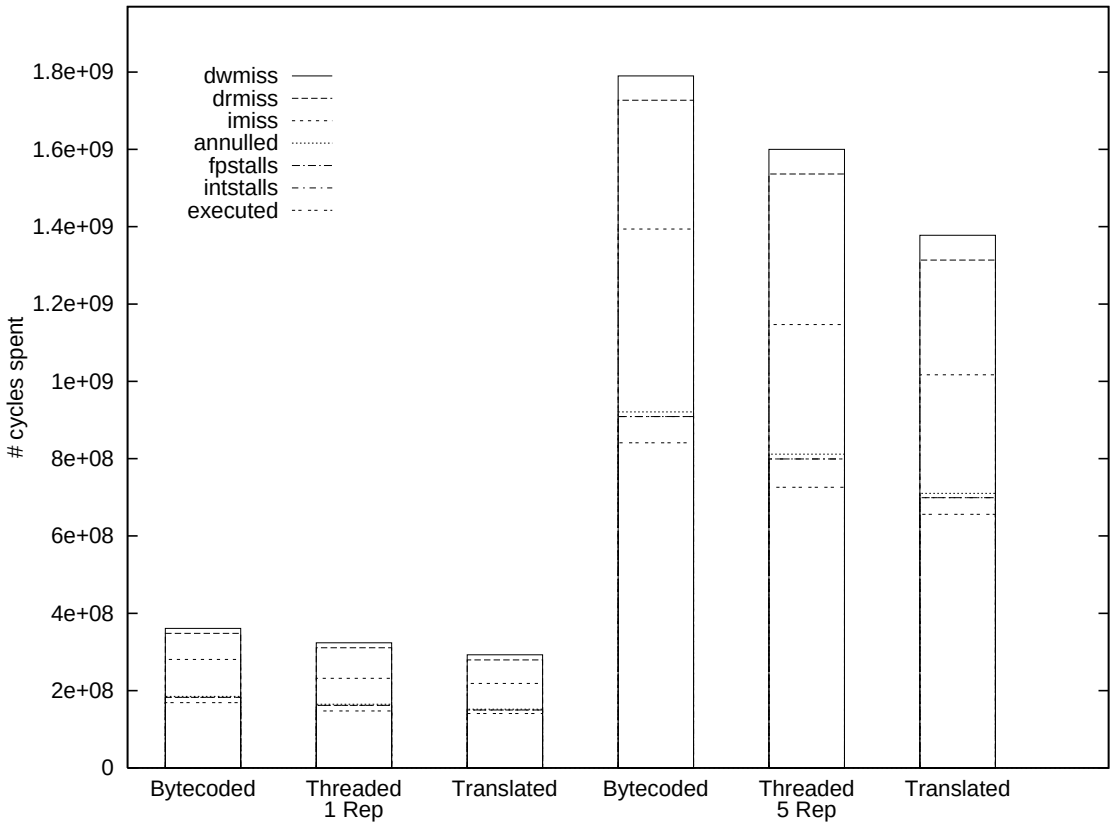

Figure 3: Translated dominates, 16KB unified cache

The results indicated that for most cache configurations, the processor-based costs far outweighed the memory-based costs, and thus the Translated interpreter fared better in terms of performance. Figure 3 shows the cycle-cost breakdown for a unified 16KB cache, an example of Translated performing best even with a small cache.

There were a few instances where Translated was not the best interpreter, particularly those involving sub-blocking, as in Figure 4. Still, it should be noted that these results are only valid given the cost factors of the SPARCstation 2, and even limited to these factors, there are nonetheless some configurations producing results that contradict popular belief in the superiority of direct execution.

The results can also be organized to demonstrate the change in various cost components as cache parameters are varied. Of particular interest is the change in instruction misses as cache size changes. These results are capable of verifying the hypothesis that the greater volume of code in Translated will cause it to have higher instruction miss rates than Bytecoded, for a range of moderate-size caches. If this range of fewer instruction misses for Bytecoded produces a similar range of lower Memory Subsystem costs, then the cache sizes in this range possess a processor/memory tradeoff for the different interpreters.

The graph in Figure 5 plots instruction miss data for unified caches, showing that Bytecoded has better instruction miss rates for cache sizes between 64KB and 512KB. The graph in Figure 6 presents a similar plot of Memory Subsystem cost, which demonstrates the same range of crossover cache sizes. Thus, caches between 64KB and 512KB in size are likely to produce interesting total-cost curves in cost-factor space. The cost-factor space under these and other cache configurations is explored in the following section.

4.5 Cycle cost crossover

Given "interesting" cache configurations, where the Translated interpreter incurs higher memory subsystem cost than the other interpreters, examining cost-factor spaces is straightforward. The cost components are recorded for the three interpreters under the specified set of cache configurations and a single benchmark. Here, the benchmark used is the 5-repetition benchmark, and the cache configurations used are the 128KB and 256KB unified cache. With these cost components, processor-based cost factors can be set according to the SPARCstation 2, and the cache-based cost factors remain independent variables.

The cost factors can be varied independently across the space of likely values, yielding a series of simulated cost values. This computation produces the $i, d_r, d_w, T_s(i, d_r, d_w)$ data points that make up the cost-factor space. The space was analyzed here in two different ways. The first analysis sets the i, d_r, d_w values to be different multiples of their values on a SPARCstation 2, so that a single

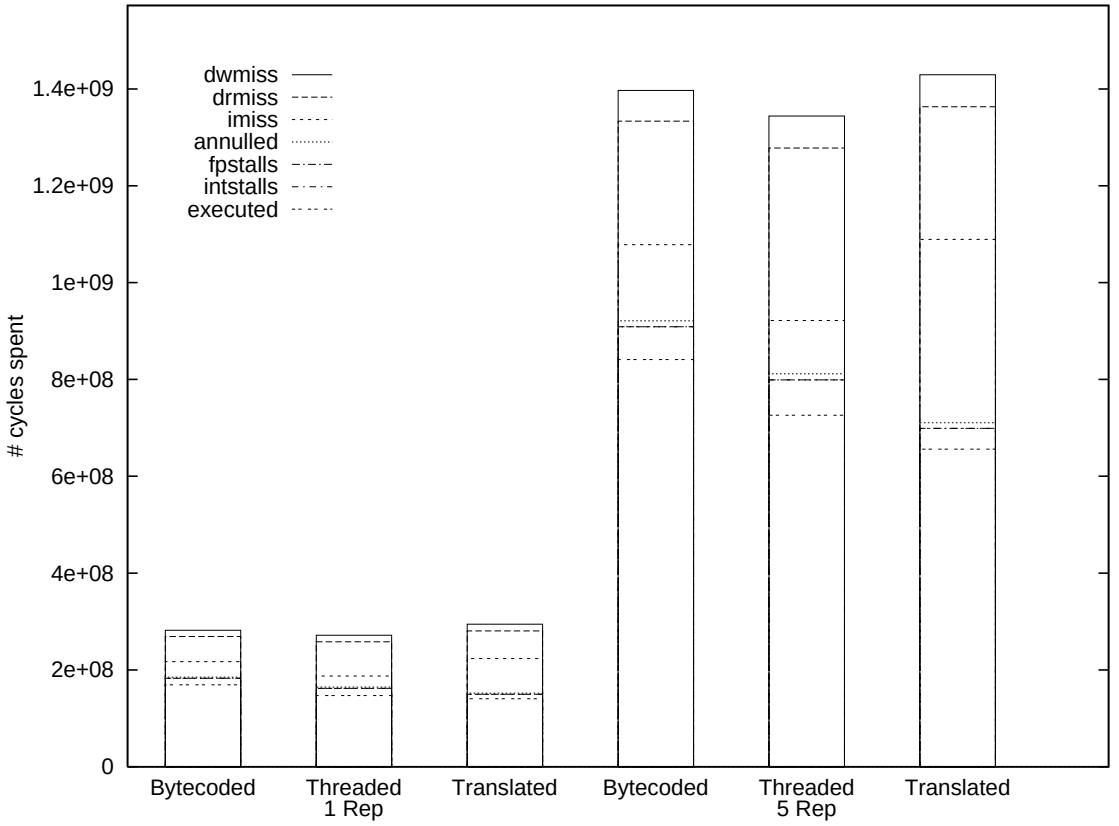

Figure 4: Threaded dominates, 64KB split cache, 16-byte lines, 4-byte sub-blocks

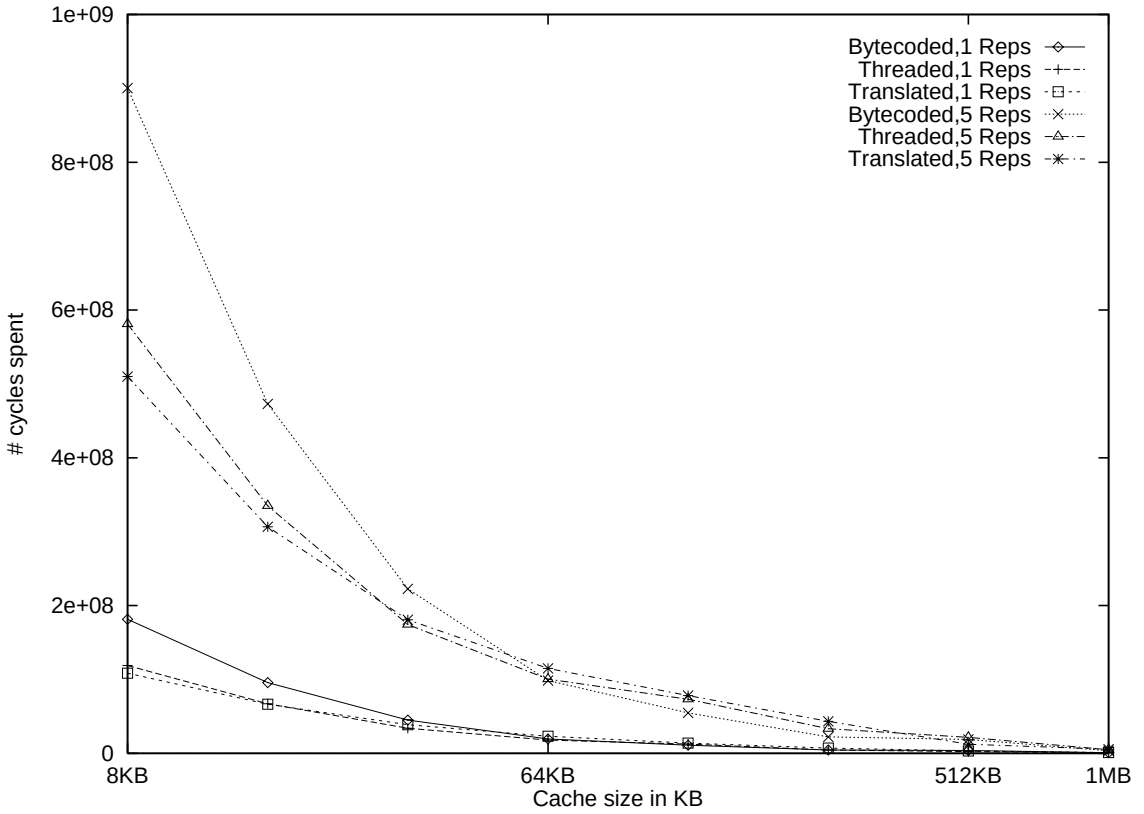

Figure 5: Simulated imiss cycles for unifi ed caches

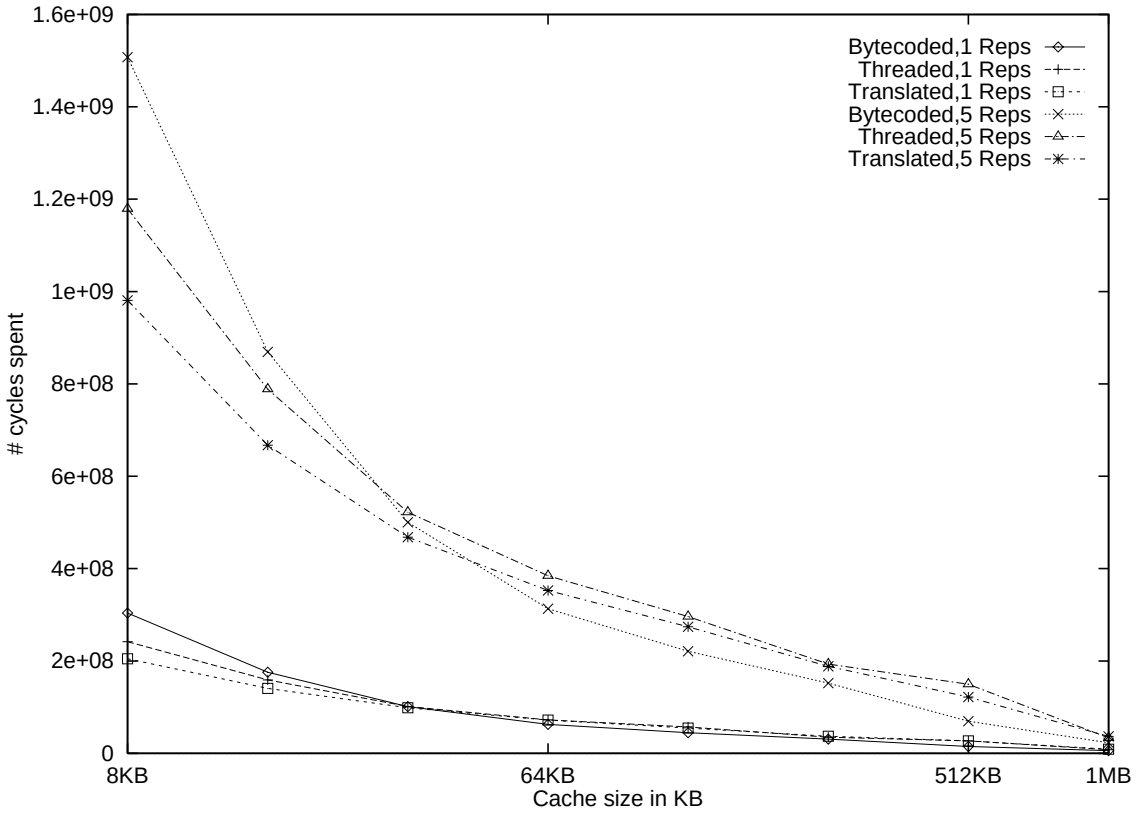

Figure 6: Simulated memory-subsystem cycles for unified caches

multiplicative factor adjusts the relative importance of the memory subsystem and the processor. The multiples examined were 0.5, 1, 2, 3, 4, 5, 6, 8, and 10. The second analysis of this space varies the i and d_r factors as the same value, but varies d_w independently. Rather than setting these factors to explicit values, however, this step determined their values at boundaries where interpreters perform equally well, and used these to describe the space.

Figures 7 and 8 plot the first analysis of cost-factors for the two cache configurations mentioned, graphing the simulated total cost for each interpreter as a function of the cost factor multiple. For the 128KB cache, the Bytecoded interpreter can perform faster than the Translated interpreter with reasonable cost-factors (100 cycles for an instruction or data-read miss). For the larger 256KB cache, higher cost factors (150 cycles for an instruction or data-read miss) are required before the Bytecoded interpreter can perform faster. Clearly, the combination of the two factors—small cache size and long memory access times—exploits the compactness of the Bytecoded interpreter and makes direct execution the less attractive option.

Figures 9 and 10 plot the second analysis of cost-factor space, graphing the curves on which each pair of interpreters have the same exact performance cost. In this way, these graphs present cost-factor space as regions where different interpreters dominate.

In the case of the 128KB unified cache, the space is clearly divided into two regions, one where the Translated interpreter dominates, surrounding the origin (where memory costs are nil), and one where the Bytecoded interpreter dominates, further from the origin where memory costs put more weight on cache behavior than processor behavior. The boundary between the two regions is nearly parallel to the d_w axis and is at an i value of about 75 cycles.

No regions of dominance appear for the Threaded interpreter, apparently because as soon as the cost factors are large enough to punish the Translated interpreter for its instruction misses, they are also large enough to punish the Threaded interpreter for its data misses. The larger 256KB cache follows a similar pattern, only with different boundaries between the two regions; the region of Translated-interpreter dominance is the same size, but the boundary extends more diagonally, depending equally on i and d_w. Presumably, with a larger cache like the 256KB, the cache misses due to accessing translated code become less significant, compared to the always-present cost of writing out the code in the first place.

The other four benchmarks, used to confirm the results found here, were run under a 64KB unified cache simulation for each Smalltalk interpreter. Boundaries of equal performance were calculated for each benchmark and used to determine regions of dominance in cost-space, as was done with the Interactive benchmark that is the focus of this paper. Both Lambda and Destroy had a region where Bytecoded is the dominant interpreter; Lambda's, though, was very far from the origin, at an instruction-miss cost of about 1000 cycles. Also, Richards and Swap had no crossover

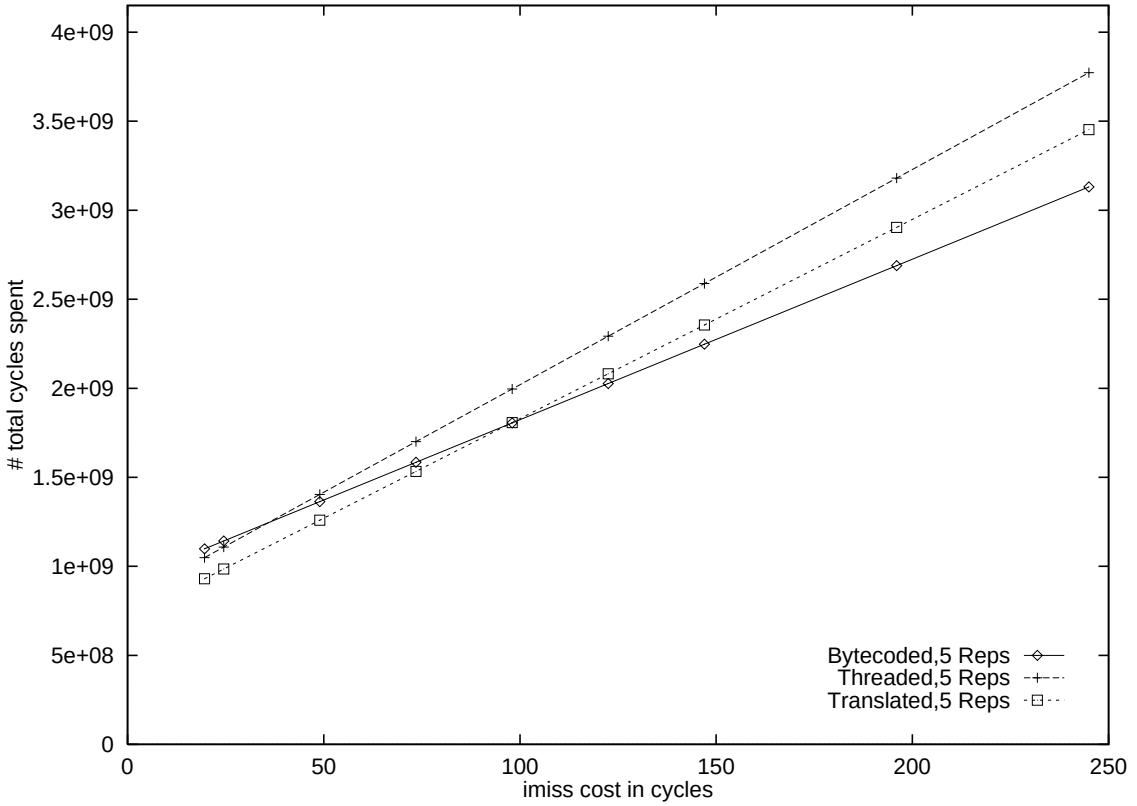

Figure 7: Effect of miss penalty on total cost, 128KB unifi ed cache

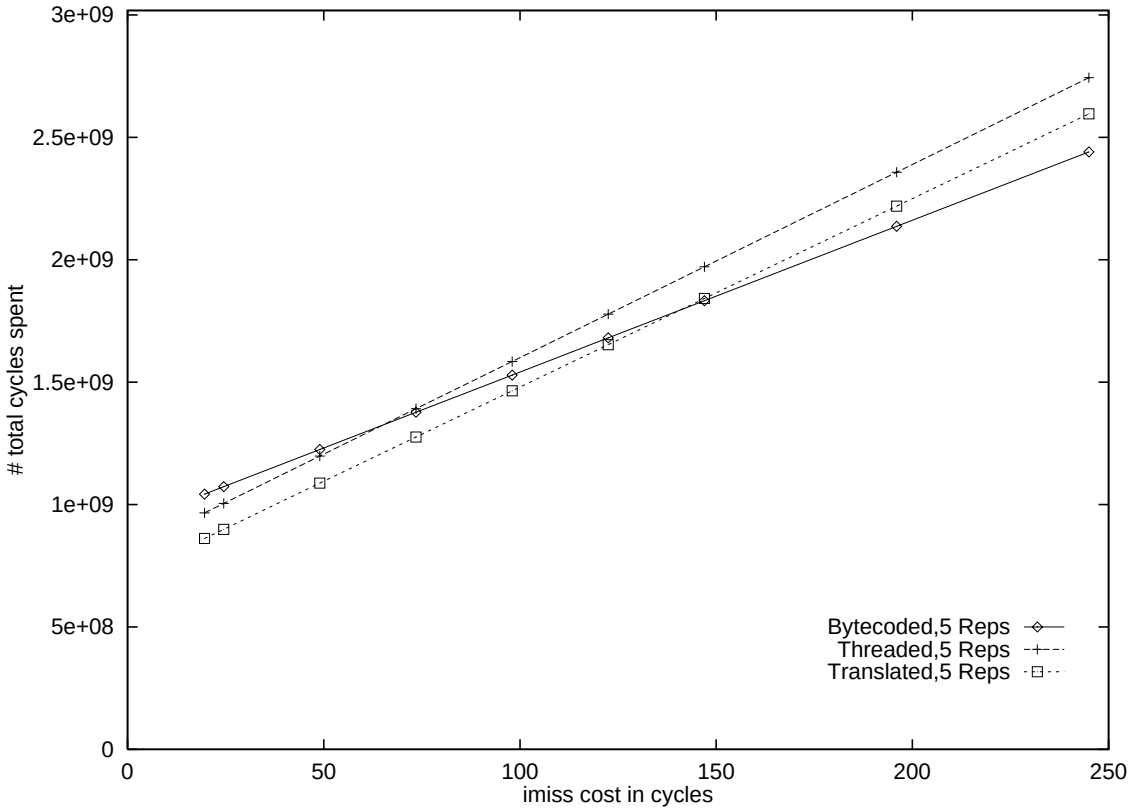

Figure 8: Effect of miss penalty on total cost, 256KB unifi ed cache

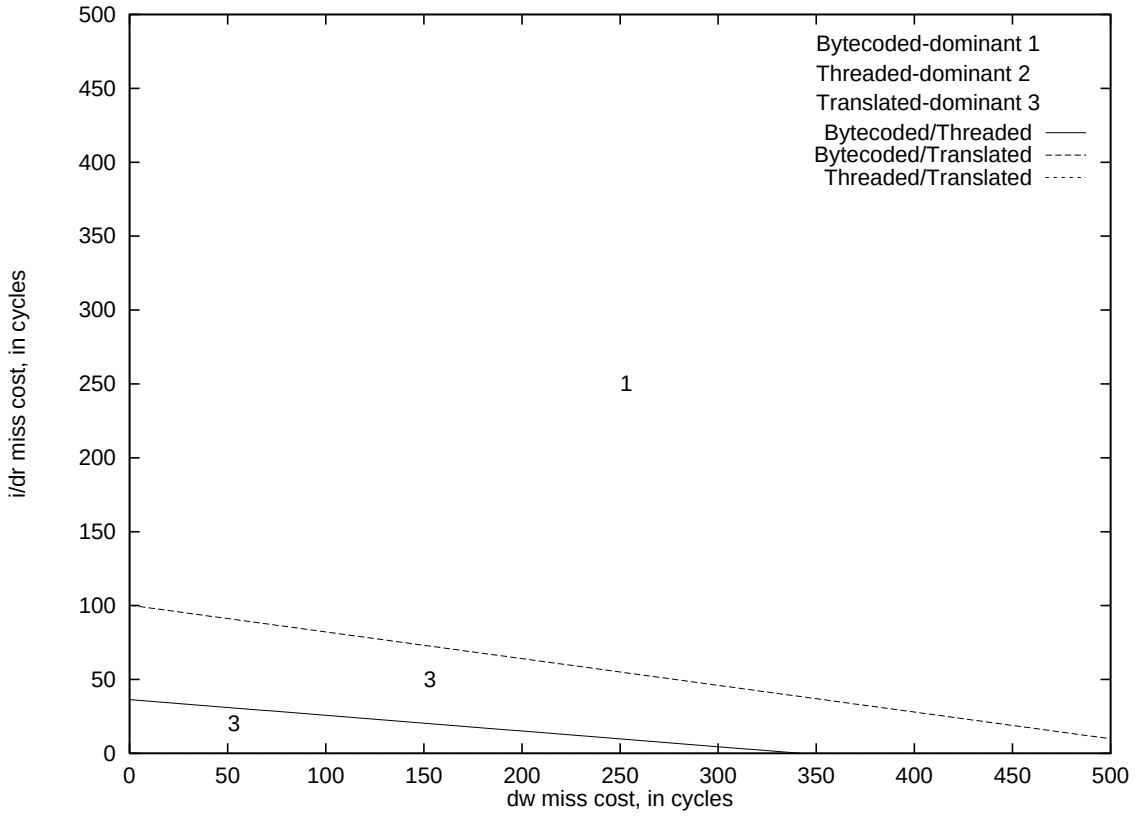

Figure 9: Dominance regions in cost-space, 128KB unifi ed cache

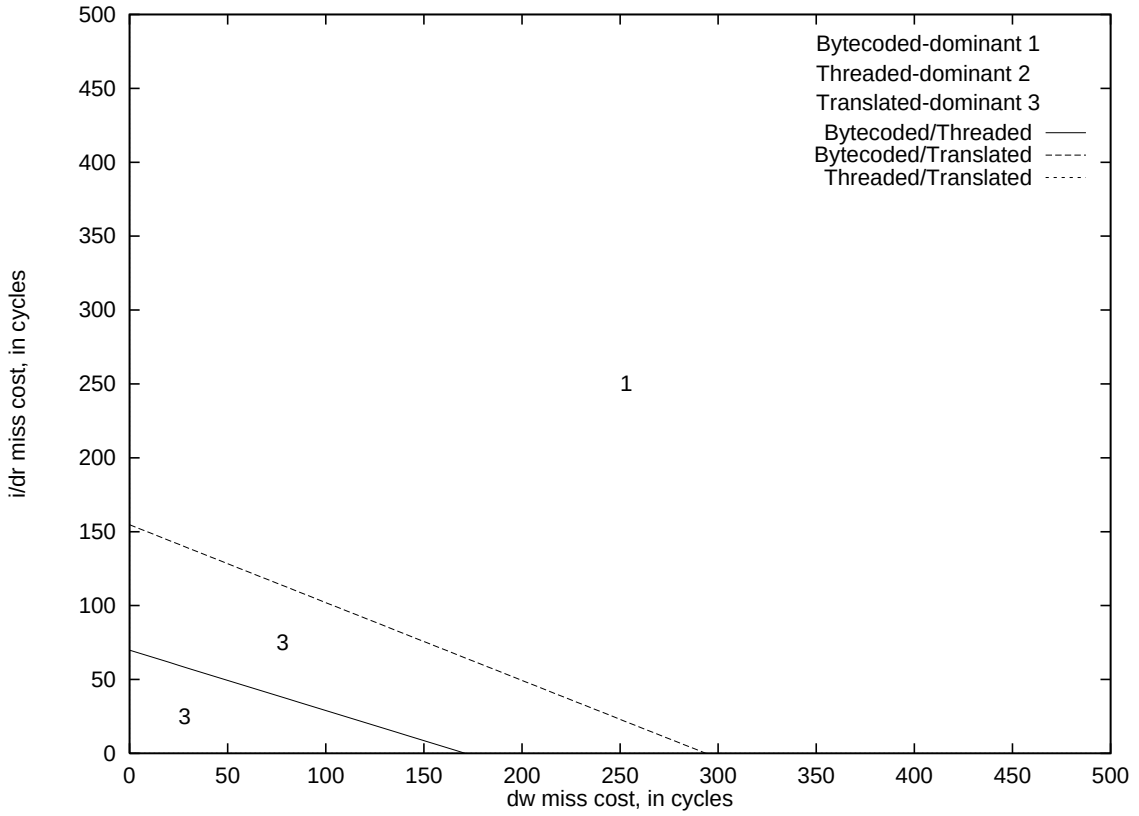

Figure 10: Dominance regions in cost-space, 256KB unifi ed cache

at all between Translated and the other interpreters, though they did have a crossover between Bytecoded and Threaded. Even though a small cache size of 64KB was used, the volume of code used by the Translated interpreter was generally too small to flood the cache. This demonstrates the dependence of the hypothesis on the volume of code being executed.

5 Conclusion

The experimental results in the previous sections confirm the hypothesized advantage for a bytecoded interpreter in instruction misses, and the consequent advantage in overall memory costs. The results further confirm that this advantage exists only for a range of moderate-sized caches, where the interpreter's instructions fit in the cache while dynamically-translated instructions do not. Finally, the results confirm that for these cache sizes, increased memory penalties inflate this advantage to the point where a bytecoded interpreter is less expensive than direct execution.

The results of the described experiments could be augmented by investigating other benchmarks that vary the size of the native-code pool, for instance, with benchmarks that generate any number of distinct methods and then invoke them. This would point out the code pool size needed to make a bytecoded interpreter the better-performing option.

The effects of optimizing dynamically translated code before executing it were not included in this study, because the issue of optimization is orthogonal to the processor/memory tradeoffs investigated here. Optimization involves tradeoffs between the amount of time spent optimizing and the amount of time spent executing methods. It generally does not increase overall code size dramatically, and thus would not have interesting interactions with the memory architecture.

There are, however, some uninvestigated issues that may affect the results presented here. The cycle figures in the results depend on the processor-based cost factors: the cycle costs of various instruction types, the cost of a stall, and so forth. They also depend on the machine instruction sequences generated for the given executables and benchmarks. The cost factors used in the experiments, and the executables used, were those for a SPARCstation 2. Many processors differ from the SPARCstation 2 in significant ways. For example, there are CISC processors like those in VAX machines, there are superscalar processors like the SPARCstation 10, and there are 64-bit processors like the Digital Alpha. These issues may have a bearing on the results presented here.

With non-RISC processors, instructions are capable of doing more work, and thus native translations of interpreted code would be shorter. With smaller pools of native code, translated interpreters would be at a smaller disadvantage, and would be able to perform better than bytecoded interpreters for a wider range of caches.

With superscalar processors, the average number of cycles per instruction would go down, since multiple instructions may be executed at the same time. If this affects all of the interpreters

equally, then the difference between them in terms of processor-based costs will be smaller and more easily offset by memory-based costs, meaning that a bytecoded interpreter will win out over a wider range of cost-factors.

With 64-bit processors, instructions are generally still 32-bit, so that multiple instructions can be packed into the 64-bit data path. The memory bandwidth with respect to instruction fetching would thus be greater, aiding translated interpreters. That is, the ratio between processor and memory bandwidth would be lessened, restricting a bytecoded interpreter to a smaller window of overhead where it could outperform a translated interpreter.

In addition, the experiments in this paper were limited to the Smalltalk language. Other interpreted languages have their own characteristics that may have an effect on the results. For instance, the Java language was designed with an intermediate bytecode representation where each bytecode performs the minimum amount of work possible, once various constants and invocations have been resolved, and where there are fewer bytecodes that can be considered complicated [LY96]. In general, the average Java bytecode is likely to take half the native instructions a Smalltalk bytecode takes. With the actual work of the bytecodes requiring fewer instructions, a translated interpreter stands to gain much more in removing the overhead of bytecoded and threaded interpreters. Also, the overall volume of native code required would be smaller, allowing the translated interpreter to outperform the others under a wider range of architectures.

Another language issue is the amount of code that is required to be native. There must be some part of any interpreted system that executes native code, a basis of primitives on which to build the methods or functions of the language. In Smalltalk, there are a few substantial primitives that serve this purpose, for storage, arithmetic, and display. In Java, however, there is a substantial set of base classes that have native-code implementations for their methods. That is, Java brings with it a much larger pool of native code. Thus, the difference between a bytecoded or threaded interpreter and translated interpreter may be less significant; Java interpreters are executing much more native code to begin with.

The results, however, confirm the hypothesis for such a wide range of cache configurations and cost factors that they should apply to many other processors and languages as well. At the very least, these experiments demonstrate that there are cases in which the variety in memory architectures makes compact-form interpretation more reasonable than direct execution.

6 Acknowledgments

Cindy Stein shared much of the work of implementing the experimental tools, including the modifications to the Shade cache simulator and modifications to the Smalltalk interpreter and benchmarks. We also thank Darko Stefanovi´c for providing comments on drafts of the paper.

References

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, Utah, January 1984.

[Goo96] James Goodman. Computer Sciences Department, University of Wisconsin-Madison, Personal Communication, 1996.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[HMS92] Anthony L. Hosking, J. Eliot B. Moss, and Darko Stefanovi´c. A comparative performance evaluation of write barrier implementations. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, October 1992.

[Irl91] Gordon Irlam. The low level computational performance of SPARCstations. Document made available via Internet, July 1991.

[Kra83] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, MA, 1983.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 1996.

[Mos87] J. Eliot B. Moss. Managing stack frames in Smalltalk. In *SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 229–240, St. Paul, MN, June 1987.

[Nei] Marie-Anne Neimat. Hewlett-Packard Laboratories, Personal Communication.

[PH96] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[ROS90] ROSS Technology, Inc., Austin, TX. *SPARC RISC User's Guide*, second edition, February 1990.

A Smalltalk

Smalltalk is a language that has its own interactive programming environment, which is essentially written in the language itself. This gives the user the ability to modify much of the environment

by recompiling the Smalltalk classes that implement it. Visible to the user—and mutable by the user—are such things as methods, stacks, frames, and program counters. These are all Smalltalk objects in themselves, e.g., objects of the method class have methods that manipulate their data, so that one can invoke a method on a method object [GR83].

Underneath the Smalltalk objects that the user can see, however, are the objects in the language implementation, which may differ in signifi cant ways from what the user sees. The UMass Smalltalk system, for example, is implemented in C, and many of the Smalltalk objects seen by the user, such as frames and program counters, have their implementation counterparts manipulated by the system in subtle ways to improve performance [Mos87]. Also, the implementation changes from one version to the next without disturbing the Smalltalk world that the user interacts with.

A.1 Smalltalk environment and implementation

Understanding how the objects in the Smalltalk world relate to the structures in the implementation world aids in understanding how different interpreters achieve the same result with different speed and memory behaviors. Figure 11 depicts the division of the Smalltalk system. The objects shown are described in the following text.

The half of the Smalltalk system comprising the Smalltalk environment consists of Smalltalk objects. Each object is a piece of data that can be manipulated by the user. It is an instance of a given class in an inheritance hierarchy. The class specifi es how its instances may be manipulated by providing a set of *methods* on its instances. Each method is named by a string called a *selector*, is given some number of *arguments*, and is provided a *body* specifying its actions, as determined by its class. A compiled method has its body represented as a sequence of numbers, or *bytecodes*, indicating specifi c instructions. A Smalltalk object is associated with the method itself, and this object contains the compiled sequence of bytecodes. The object representing the class contains a *Method Dictionary* listing these method objects by selector.

The bytecodes that represent method bodies are a set of 256 instructions that form an intermediate representation. Each is specifi ed by a single byte in the method object, the byte storing the number of the intended bytecode; some bytecodes take arguments that are stored in trailing bytes. Possible bytecode meanings include **push**, **pop**, **jump**, **send** (call), and **return**. These are complemented by the *primitives*, which are a set of low-level operations, such as mathematical functions, that are each indicated by some primitive-number, similar to bytecodes. These may be invoked in place of true methods.

The Smalltalk objects, including the compiled methods, are passive entities; the active entity in the Smalltalk environment is the *Virtual Machine*, which continually interprets bytecodes, calls primitives, and dispatches methods. It dispatches methods by maintaining *method contexts*,

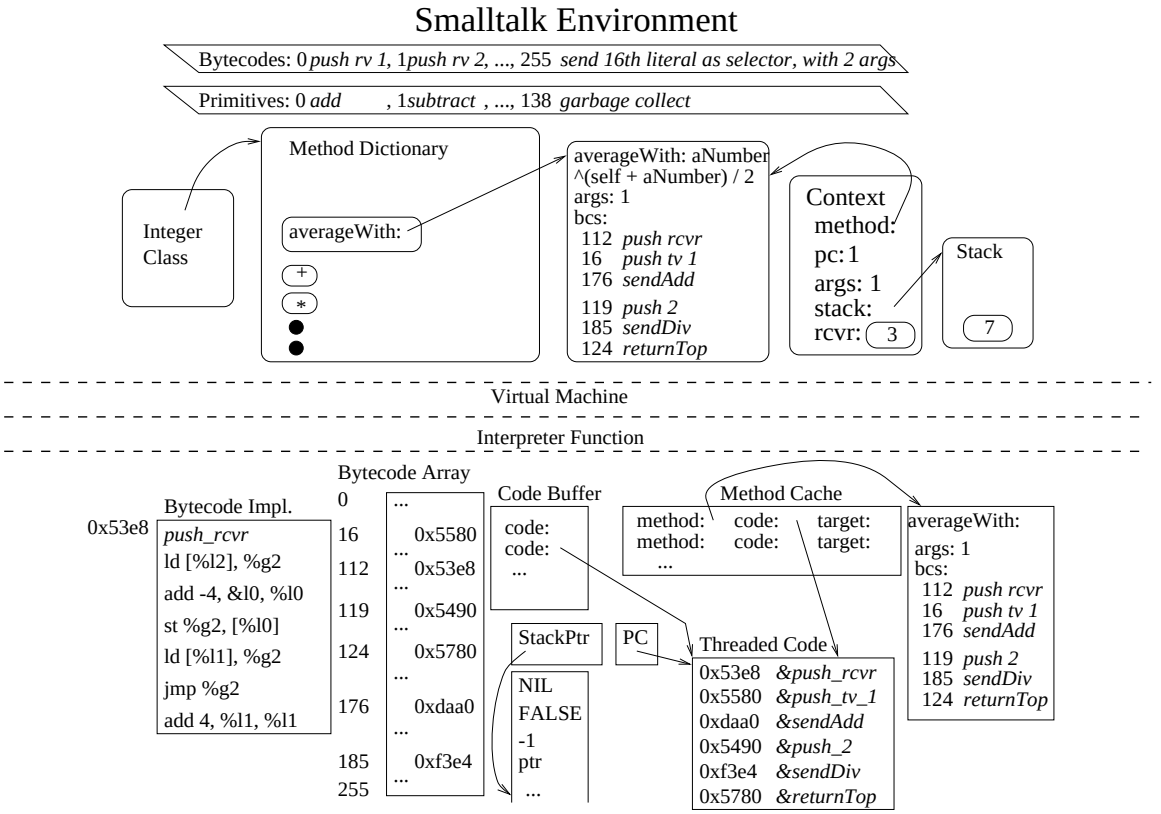

Figure 11: Smalltalk Execution Model

which are equivalent to the stack frames and closures of other languages. The context contains information about the method's invocation, including the sender, receiver, method object, stack for temporaries and arguments, and a *program counter* value. The program counter is an index into the method's bytecodes, indicating the most recent point of execution. The Virtual Machine maintains a unique program counter that refers to the currently executing method and indicates the next bytecode to execute.

The half of the Smalltalk system comprising the Smalltalk implementation consists of C structures and functions. The Virtual Machine is implemented by an *interpreter function* that jumps between different bytecode and primitive implementations and orchestrates the interactions between the different implementation objects. The work signified by bytecodes is performed in *bytecode implementations*, which are sections of C code compiled along with the rest of the Smalltalk implementation. The purpose of each is similar to that of a function (say, "**void bytecode_0()**") that would be called during bytecode interpretation to execute the action of the individual bytecode. Using only a target label at the beginning and a computed goto at the end avoids function call and return overhead. The bytecodes perform most of their work by interacting with the *implementation stack*, which stores receivers and arguments of primitives and methods. The stack directly contains simple objects like integers, and contains pointers to more complex objects.

The computed gotos that transfer control between bytecode implementations use the *implementation program counter* and a *bytecode array* to determine the correct addresses. The bytecode array, initialized at system startup, maps bytecode numbers to the addresses of the corresponding implementations. The program counter depends on the interpretation scheme being used, and may point to the number of the next bytecode to be executed, or to a pre-computed address of the next bytecode implementation. In the case of dynamically-translating interpretation, the program counter is used as a return address when calls out of native code must be made, and the bytecode implementations are not used at all. In all cases, the program counter is preferably stored in a hardware register, but is nonetheless distinct from the hardware program counter. A sequence of pre-computed implementation addresses is referred to as a *threaded code object*. Both these and the sequences of native instructions used in dynamic-translation are called *auxiliary code objects*, as they are associated with specific method objects.

There are also extra mechanisms in the implementation that speed up method invocation. One, the *method cache*, retains the most-recently executed methods, in most cases removing from dispatch the work of looking up methods in Method Dictionaries. It also keeps certain information associated with the method, such as any auxiliary code object, and information used to speed the critical path of invocation. The other mechanism, the *code buffer*, stores auxiliary code objects on a permanent basis, avoiding the work of threading or translation that would be incurred in the case

of many method cache misses.

A.2 Smalltalk method life cycle

The process of method compilation and execution in the Smalltalk model follows the following pattern. The case given here, that of the user directly invoking the method, can be generalized to any Smalltalk object invoking a method on another object.

Compilation:

1. The user wishes to create a new method for an existing class, so that data contained in objects of this class can be manipulated in a different way.

2. The user enters and edits Smalltalk code through the interactive environment, associates the code with a method name, and associates the method with a class. The user requests compilation of the source.

3. Requesting compilation invokes a compilation method. The compilation method takes the textual form of the user's code and compiles it into bytecodes; the actions signified by the bytecodes will supply the intended effect of the method.

4. The compilation method creates a method object and stores the sequence of bytecode numbers in it. The compilation method stores the new method object in the Method Dictionary of its associated class, making it available for execution. These actions are reflected in the implementation: the Method Dictionary of the class contains the method, and the method contains the bytecodes.

Execution:

1. The user enters an invocation of the method on an object of the associated class, supplying the selector (name) of the method, the name of the object to which it must be applied, and any arguments. The user requests evaluation of this method invocation.

2. Requesting evaluation invokes the method. The Virtual Machine creates a method context object with appropriate sender, receiver, and stack values; it searches the Method Dictionaries of the receiver's class hierarchy for the requested method; it sets the method object into the method context; it sets the context's program counter to the first bytecode index in the method and begins executing each bytecode in sequence.

3. At this time, things occurring in the implementation are subtly different from those in the Smalltalk environment. Rather than actually accessing the Method Dictionaries, the interpreter function first searches the method cache. This particular method has not been placed

in the method cache yet, so this search fails. The interpreter then resorts to scanning the Method Dictionary of the receiver's class and obtains the method object. Depending on what kind of interpretation scheme is used, auxiliary code may be produced and stored in the code buffer. Then the interpreter places the method object, any auxiliary code, and associated invocation information into the method cache. The interpreter sets the implementation program counter according to the interpretation scheme, using the counter to jump into the code that executes the first bytecode's actions.

This general case holds no matter what interpretation scheme is used. The specific actions taken by each interpreter to execute a method are explained in Section 2.