Stress-testing Control Structures for Dynamic Dispatch in Java

Olivier Zendra

INRIA Lorraine - LORIA / McGill University
615 Rue du Jardin Botanique, BP 101
54602 Villers-Les-Nancy Cedex, France

http://www.loria.fr/~zendra

Olivier.Zendra@loria.fr

Karel Driesen

McGill University
School of Computer Science - ACL Group
Montreal, Quebec, Canada H3A 2A7

http://www.cs.mcgill.ca/~karel

karel@cs.mcgill.ca

ABSTRACT

Dynamic dispatch, or late binding of function calls, is a salient feature of object-oriented programming languages like C++ and Java. It can be costly on deeply pipelined processors, because dynamic calls translate to hard-to-predict indirect branch instructions, which are prone to causing pipeline bubbles. Several alternative implementation techniques have been designed in the past in order to perform dynamic dispatch without relying on these expensive branch instructions. Unfortunately it is difficult to compare the performance of these competing techniques, and the issue of which technique is best under what conditions still has no clear answer. In this study we aim to answer this question, by measuring the performance of four alternative *control structures* for dynamic dispatch on several *execution environments*, under a variety of precisely controlled execution conditions. We stress test these control structures using micro-benchmarks, emphasizing their strenghts and weaknesses, in order to determine the precise execution circumstances under which a particular technique performs best.

Keywords

Java, dynamic dispatch, control structure, optimization, JVM, binary tree dispatch, virtual function call

1. INTRODUCTION

Object-oriented message dispatch is a language concept that enables data (objects) to provide a functionality (message) by relying on a type-specific implementation, or method. At run time, the object that receives a message, or virtual method call, retrieves the corresponding class-specific method and invokes it. This late binding of dispatch targets allows any object to play the role of the receiver object, as long as the new object implements the expected interface (is substitutable à la Liskov [Lis88]). Such type-substitutability enables better code abstraction and higher code re-use, and is therefore one of the main advantages of object-oriented languages.

As a consequence, dynamic dispatch occurs frequently in object-oriented programs. For instance, virtual method invocations in Java [GJSB00] occur every 12 to 40 byte codes [DLM$^+$00] in SPEC JVM98. Such late-bound calls are typically expensive on modern deeply pipelined processors, because they translate to hard-to-predict indirect branch instructions that are a cause for long pipeline bubbles [DHV95].

It is nonetheless exceedingly difficult to precisely measure the time spent on dynamic dispatch itself by real object-oriented programs. Indeed, virtual function calls occur frequently, which makes it difficult to isolate dispatch time from the runtime of regular code. Furthermore, call frequency and amount of runtime polymorphism strongly depend on coding style as well as runtime parameters. Finally, on modern superscalar processors, call code sequences can be co-scheduled with regular code, which further blurs the picture. Dispatch overhead therefore depends not only on the dispatch code sequence itself, but also on the code surrounding the call and on the processor ability to detect and take advantage of instruction level parallelism.

An estimate of dispatch overhead, based on real programs and relying on super scalar processor simulation, can be found in [DH96]. The authors measure a median dispatch overhead of 5.2% in C++ programs and 13.7% in C++ programs with all member functions declared virtual (as is the default in Java). For one program, the overhead was as high as 47% of the total execution time. While evidence from practice suggests that most Java programs exhibit little polymorphism at run time, it is true that for some programs the optimizations tested in this study can make as much as a 50% difference in execution time, as demonstrated by the micro-benchmarks. It thus appears very sensitive to optimize dynamic dispatch, in order to avoid incurring a significant performance penalty when relying on the object-oriented design style.

Alternative implementation techniques are available to perform dispatch to multiple targets without using expensive branch instructions. Unfortunately, comparing the performance of these competitive techniques is hard, and the literature typically reports measurements of few alternatives, on only one execution environment.

In this study, we propose and report on the results of a proof-of-concept methodology to measure the performance of several control structures for dynamic dispatch on a *variety of Java Virtual Machines and hardware platforms*. In this first-step study, we rely on micro-kernel benchmarking to determine and magnify the relative performance of control instructions under a large number of *varying execution conditions*.

The results show, among other things, that:

- Virtual method call performance is highly *dependent on the execution pattern* at a particular call site
- When the call site has a low (2-3 target types) to medium (6-8 target types) degree of polymorphism, optimizations are possible that *improve performance across JVMs and hardware platforms* (that is, platform independent optimization)
- *Processor architecture shines through*, especially on high-performance JVMs: the performance profile from different VMs executing on the same hardware look similar, those from the same VM executing on different hardware look different.

This paper is organized as follows. Section 2 reviews dynamic dispatch implementations and related work at software, run-time system and hardware level. Section 3 presents our methodology and the experimental setup. Section 4 presents some of our results and discusses them. Finally, section 5 concludes and points at future research directions.

2. BACKGROUND

2.1 Monomorphism vs. Polymorphism

Dynamic dispatch is expensive because the target method depends on the run-time type of the receiver, which generally cannot be determined until actual execution.

Many different optimization techniques have thus been proposed, which can be seen as falling into two broad categories:

Optimizing monomorphic calls Since dynamic dispatch is expensive, the fastest way to do it is to avoid it altogether.

Various kinds of *program type analysis* (such as [DGC95, SLCM99, SHR$^+$00]) enable the devirtualization of provably monomorphic calls (calls with only one target type), replacing the expensive late-bound call by by a direct, cheaper, early-bound call. This technique has the added advantage of allowing inlining of target methods, thus stripping away all of the call overhead and enabling a more radical optimization of the inlined code by classical methods.

Dynamic optimization (e.g [DDG$^+$96, HU94]) such as employed by the SUN HotSpot™ Server JVM allows method inlining at run time, which permits further optimization of calls that are monomorphic in only a particular run of the program, even though multiple target types are possible after compile time.

Optimizing polymorphic calls In spite of all efforts, some calls cannot be resolved as monomorphic. Optimizing the remaining polymorphic ones (calls with more than one target type) is crucial.

Program type analysis can also optimize these polymorphic calls, especially when the number of possible types is very low. For example, a compiler can replace a late-bound call with two possible target types by a conditional branch and two static, direct, early-bound calls. At run time, a cheap conditional branch and cheap static call are executed instead of one expensive late-bound call (strength reduction). Such a strength reduction operation is usually a win on current processors, since prediction of conditional branches is easier than prediction of indirect branches. Furthermore, the dominant (most common) call direction can be inlined, leading to similar optimization opportunities as for monomorphic calls, with the guard of a cheap conditional branch [AH96].

Dynamic optimization can also replace a call that is dominated by one target type at run time, enabling the same operation as above with increased type precision.

These solutions to optimize dynamically dispatched calls are amenable to two approaches: hardware and software.

2.2 Hardware Solutions

Virtual method invocations in Java translate, in the native machine code, into two dependent loads followed by an *indirect branch* (or indirect jump). This indirect branch is responsible for most of the call overhead [Dri01]. Branches are expensive on modern, deeply pipelined processors because the next instruction cannot be fetched with certainty until the branch is resolved, typically at a late stage in the pipeline (e.g., after 10-20 cycles on a Pentium III).

Most processors try to avoid these pipeline bubbles by speculatively executing instructions of the most likely execution path, as predicted by separate *branch prediction* micro-architectures. For example, a Branch Target Buffer (BTB) stores one target for each indirect, multi-way branch and can predict monomorphic branches with close to 100% accuracy, which removes the branch misprediction overhead in the processor.

Unfortunately, polymorphic calls are harder to predict. Sophisticated two-level indirect branch hardware predictors [CHP98] can provide a similar advantage as a BTB for multi-target indirect branches that are "regular" and whose target correlates with the past history of executed branches.

Unfortunately, indirect branches are more difficult to predict than conditional branches. A conditional branch has only one target, encoded in the instruction itself as an offset, so a processor only needs to predict whether the conditional branch is taken or not (one bit). Indirect branches can have many different targets and therefore require prediction of the complete target address (32 or 64 bits). Sophisticated predictors [DH98a, DH98b] can reach high prediction rates, but generally require large on-chip structures. Indirect branch predictors thus tend to be more costly and in practice less accurate than conditional branch predictors (Branch History Buffers, BHTs), even in modern processors.

Therefore, replacing at the code level a rather unpredictable indirect, multi-way branch by one or several more predictable conditional branches followed by a static call seems a likely optimization, helping the processor. This strength reduction of control structures is exploited by several of the techniques in the next section.

2.3 Software Solutions

Most JVMs include some way to de-virtualize method invocations that are actually monomorphic, by replacing the costly polymorphic call sequence by a direct jump. For example, various forms of whole program analysis (e.g., [BS96, SHR+00]) show that most invocations in object-oriented languages are monomorphic.

Some JVMs use a dynamic approach. For example, HotSpot relies on a form of inline caching [DS84, UP87]. The first time a virtual method invocation is executed, it is replaced by a direct call preceded by a type check. Subsequent executions with the same target are thus direct, whereas executions with a different target fall back to a standard virtual function call.

Actual run-time polymorphism can also be optimized in software, for example by using Binary Tree Dispatch (BTD), as implemented in the SmallEiffel compiler [ZCC97]. BTD replaces a sequence of powerful dispatch instructions using an indirect branch by a sequence of simpler instructions (conditional branches and direct calls). When the sequence of simple instructions remains small, it can be more efficient than a call through a virtual function table, and should perform particularly well on processors with accurate conditional branch prediction and large BHT. A BTD is a static version of what is commonly known as a Polymorphic Inline Cache [HCU91]. A PIC collects targets dynamically at run time (it is a restricted form of self-modifying code), effectively translating a lengthy method lookup process into a sequential search through a small number of targets. The *if sequence* control structure exercised in our micro benchmark suite (see section 3.3) is akin to the implementation of a PIC described in [HCU91]. As in the latter paper, we found that megamorphic [DHV95] call sites (more than 10 possible target types) are too large for a sequential *if* to be cost-effective.

Chambers and Chen also proposed a hybrid implementation mechanism [CC99] for dynamic dispatch that can choose between alternative implementations of virtual calls based on various heuristics. The experiments in our study complement their approach, since we aim to more precisely define the gains and cutoff points reachable with each technique on multiple platforms.

3. METHODOLOGY

3.1 Overview

We started this work in order to find out whether control structure strength reduction could be used to optimize dynamic dispatch under specific execution conditions and across different hardware platforms, i.e. to find out whether platform independent optimization is feasible.

In order to allow platform independent optimization to be effective, two conditions must hold. First, strength-reducing operations must be guided by platform independent information; the analysis may include profile data if it is not platform-specific. Second, the performance of control structures must be consistent across platforms.

The first condition is fulfilled by various forms of static program analysis and program-level profiling, and many studies show that optimizable call sites are common.

The second condition needs to be verified. Even the reasonable assumption that direct static calls are faster than monomorphic virtual ones may not always hold in practice due to implementation features, at the virtual machine level or at the processor micro-architecture level. For instance, a Pentium III stores the most recent target of indirect branches, which can make monomorphic virtual calls as efficient as static calls.

In the next section we discuss our experimental framework to measure performance of control structures across different JVM and hardware platforms.

3.2 Experimental Setup

Since we focus on polymorphic calls, a large variety of execution behaviors and control structures has to be measured on several platforms. Therefore, we design a comprehensive suite of Java micro benchmarks as a proof-of-concept simulation of various implentations of dynamic dispatch in various JVMs. This allows us to test the performance of control structures under controlled execution conditions, leveraging the wide availability of the Java VM to measure on different execution environments.

All benchmarks use the same superstructure: a long-running loop that calls a static routine which performs the measured dispatch. The receiver object (actually, its type ID) is retrieved from a large array, which is initialized from a file that stores a particular execution pattern as a sequence of type IDs. This initialization process ensures that compile-time prediction of the type pattern is impossible. Different files store a variety of type ID sequences, representing different patterns and degrees of polymorphism.

The experimental parameter space thus varies along three dimensions:

Control structures How do different different control structures for dynamic dispatch perform?

Execution patterns This dimension has three related sub-dimensions. First, the *static number* of possible receiver types at the dispatch site, which influences the program code and can be determined by program analysis before execution. Second, the *dynamic number* of receiver types at the dispatch site, that is the range of types occurring in a particular program run. Third, the *pattern* of receiver type IDs, that is the order and variability of receiver types at run time.

Execution environments This dimension has two related sub-dimensions: the *virtual machine* used and the *processor* it is run on.

Each data point (timing) within this parameter space is measured as follows. First, the benchmark is run 5 times over a long (10 million) loop, which gives a “long run average” running time. This average comprises only the loop part (not the initialization). When executed on dynamically optimizing JVMs such as HotSpot, this execution time comprises both the execution as “cold code” and the execution as optimized once the optimizer has determined the loop is a “hot” one. The JVM is thus given ample opportunity to fully optimize control structures. Then, the benchmark is re-run 5 times over a very long (60 million) loop, which provides a “very long run average”. The difference between these two averages, “long” and “very long”, represents only “hot”, optimized loops, and gives us our final result, after normalization to 10 million loops.

The three dimensions of the parameter space are detailed in sections 3.3, 3.4 and 3.5.

3.3 Various control structures

We measure a variety of control structures for dynamic dispatch implementation. Although it is not comprehensive, we believe it covers the main possibilities available to optimizing compilers at the bytecode and native code level.

Virtual calls At the Java source code level, a dispatch site is a simple method call: `x.foo()`. At the Java bytecode level, a special instruction, `invokevirtual`, is provided to implement virtual calls. The dynamic dispatch instruction uses the message signature (argument to the `invokevirtual` bytecode) and the dynamic type of the receiver object (atop the stack) to determine the actual target method. Generally, this translates at the hardware level into a table-based indirect call [ES90]. This constitutes the first implementation of dynamic dispatch we tested, in our “Virtual” series of micro-benchmarks.

It is however possible to use other control structures, based on simpler bytecode instructions, such as type equality tests followed by static calls. These control structures can take at least three forms:

If sequence First, a sequence of 2-way conditional type checks can be used. For example, let's assume a polymorphic site `x.foo()` where global, system-wide analysis detected that the receiver could only have four possible concrete types at runtime: T_A, T_B, T_C and T_D. The corresponding pseudo-code is shown in figure 1, where the tests discriminate between all the known possible types and lead to the appropriate leaf static call. This implementation of dynamic dispatch is tested in our "IfSequence" series of micro-benchmarks, where the type ID is an integer stored in an extra field of every object.

```
xTypeID = x.typeID;
if (xTypeID == ID_FOR_TYPE_A) then
    A.static_foo(x);
else if (xTypeID == ID_FOR_TYPE_B) then
    B.static_foo(x);
else if (xTypeID == ID_FOR_TYPE_C) then
    C.static_foo(x);
else if (xTypeID == ID_FOR_TYPE_D) then
    D.static_foo(x);
endif
```

Figure 1: P-code for if-sequence dispatch

A variant of this would not test against a type ID field added to the objects, but use a series of `instanceof`s expressions. This technique would avoid the need for the type ID field, and the associated space and initialization overhead. Its performance compared to the *if sequence* above would mostly be related to the relative performances of the `instanceof` and `getfield` bytecodes instructions. We did not include this variant in our benchmarks.

Another variant consists in accessing the type descriptor (`Class` object) of the receiver, using the `getClass()` method, instead of the type ID field. Since `getClass()` is a final native function of `Object`, with JVM support, it is likely to be quite fast. This technique also avoids the need for the type ID field and the associated costs. However, the type test would have to be done against `CLASS_FOR_TYPE_A`, `CLASS_FOR_TYPE_B`, etc. instead of `ID_FOR_TYPE_A`, `ID_FOR_TYPE_B`, etc. Retrieving each of these class descriptors incurs a cost as well, using either a static method for each class, or (in the context of our proof-of-concept Java micro-benchmarks) the more general function `forName (String className)` in class `Class`, whereas the type IDs are constants. This variant thus also seems to be potentially slower. We did not include it in our benchmarks.

Binary Tree Such 2-way conditional tests can be organized more efficiently, as a binary decision tree [ZCC97]. Let's assume the type IDs corresponding to the types T_A, T_B, T_C and T_D, are, respectively, 19, 12, 27 and 15. Then, the pseudo-code generated for `x.foo()` looks like the one in figure 2. We test this implementation of dynamic dispatch in our "BinaryTree" series of micro-benchmarks.

```
xTypeID = x.typeID;
if (xTypeID <= 15) then
    if (xTypeID <= 12) then
        B.static_foo(x);
    else
        D.static_foo(x);
    endif
else
    if (xTypeID <= 19) then
        A.static_foo(x);
    else
        C.static_foo(x);
    endif
endif
```

Figure 2: P-code for binary tree dispatch

Note that a BTD using the `getClass()` variant described above for *if sequence* would be slower than the one we present, since the tests in the dispatch tree would not be done with constants anymore. We thus did not include this variant in our benchmark suite.

Switch Finally, a multi-way conditional instruction can be used, namely a Java dense `switch`, translated into a `tableswitch` bytecode instruction, whose suggested implementation [LY99] by the JVM is an indirection in a table. The corresponding pseudo-code, tested in our "Switch" series of micro-benchmarks, is shown in figure 3.

For the sake of simplicity, we only test dense switches. Indeed techniques exist (global analysis, coloring,...) to have a compact allocation of type IDs. In case of sparse type IDs, sparse switches should be translated into a standard `lookupswitch` [LY99] bytecode instruction, that can be implemented by the JVM as a series of *ifs* or a binary search, thus falling back to one of the techniques we already present in this study.

```
xTypeID = x.typeID;
switch (xTypeID)
    case ID_FOR_TYPE_A then
        A.static_foo(x);
    case ID_FOR_TYPE_B then
        B.static_foo(x);
    case ID_FOR_TYPE_C then
        C.static_foo(x);
    else ID_FOR_TYPE_D then
        D.static_foo(x);
endswitch
```

Figure 3: P-code for tableswitch dispatch

The general idea behind strength reduction for dynamic dispatch is that simpler instructions, although more numerous, should be more predictable and executed faster than complex instructions.

All these control structures, except the plain `invoke-virtual`, have a size that is proportional to the number of tested types. When used to implement dynamic dispatch, without any fall-back technique, all possible types have to be tested; this set of possible types thus has to be determined by a global analysis. This is accounted for in our benchmark suite, by creating, for each distinct dispatch technique, several benchmarks differing only by the number of types they can handle.

In the last three control structures, the leaf calls are purely monomorphic. They are thus implemented as Java static calls `X.static_foo(x)`, with the original receiver object being passed as the first argument (instead of being the implicit `this` argument in the virtual call). We thus gave a "StaticThisarg" suffix to these benchmarks. We also benchmark leaves implemented as monomorphic virtual calls which, as we expected, turn out to be generally slower than the static leaves. As a consequence, we do not detail those results in this paper.

Note that the last three techniques may also be used to serve as run-time adaptive caches catching the most frequent or more recent types, preceding a more general fall-back technique. In this case, they would be akin to PICs, or more accurately, as different alternative control structures which can be used to implement various sizes of PICs.

Figure 4 shows a synthetic comparison of these four control structures.

3.4 Various type patterns

The runtime behavior of the program is another crucial factor in the performance of a given dynamic dispatch site. In order to simulate varying behaviors while keeping precise control, we timed our benchmarks by generating various type ID patterns. Each micro benchmark reads a particular pattern from file at run time to initialize a 10K int array holding type IDs, which is then iterated over a large number of times.

For this study, we used synthetic patterns which represent extremes in program behavior. We plan to use real applications or real application traces in future work. We decided to design patterns comprising between one and 20 possible receiver types, in order to cover a wide range of cases. In most real applications though, the degree of polymorphism remains typically much smaller (3 to 5). The low degrees of polymorphism in our tests thus have a lot of importance for most cases, while higher degrees tend to show how a specific technique scales up. The following four patterns are presented below and in figure 5: the constant pattern, the random pattern, the cyclic pattern and the stepped pattern.

Constant This pattern is the simple 100% monomorphic case, where the receiver type is always the same and is thus perfectly predictable. This is a very common case. Various techniques detect such monomorphic dispatch sites and get rid of them by replacing them with direct calls (de-virtualization). However, these techniques may not always be applied, do not detect all monomorphic call sites and do not handle call sites that are in principle polymorphic but never change targets within any single run. It is thus worth testing the behavior of dynamic dispatch techniques on this best-case constant pattern. Since the value of the constant type ID influences performance, we have to test various IDs within the static range.

Random This pattern is the exact opposite of the previous one: it can't be predicted, features high polymorphism (many receiver types) and high variability (many changes during execution). As such, it represents a worst-case scenario likely to be rare in object-oriented programs.

Cyclic The cyclic pattern features a regular variation of the type ID, each ID being the previous one incremented by 1 up to maxID and back to 1, and so on. This pattern is thus highly polymorphic and has a very high variability (the type changes at every call), like the random pattern, but is still very regular. Advanced micro-architecture such as two-level branch predictors are capable of detecting some

Structure	Pros	Cons
Virtual	Short code sequence. Size independent of # of static types.	Translates to expensive indirect branch. Generally slow.
If Sequence	Uses an inexpensive static call. Fast for types at beginning of sequence. Translates to conditional branches better predicted by hardware than indirect ones.	Long code sequence. Slow for types at end of sequence. Size depends on # of static types.
Binary Tree	Uses an inexpensive static call. Equally fast for all types (distance to leaf). Translates to conditional branches better predicted by hardware than indirect ones.	Long code sequence. Speed depends on # of static types (log2). Size depends on # of static types.
Switch	Uses an inexpensive static call.	Long code sequence. Size depends on # of static types. Unreliable speed: depends on JVM.

Figure 4: Comparison summary of various control structures

cyclic branch behavior and therefore should predict this pattern accurately, especially for small cycles. As such, and even though it is probably fairly uncommon in OO programs, this pattern represents a kind of intermediate point between constant and random.

Stepped This pattern is a regular variation of the cyclic pattern, close to the constant pattern in behavior. It features a variation of the type ID from 1 to maxID, with increments of 1, but with as few changes as possible within a single run. It thus exhibits long, constant steps, whereas the cyclic pattern has a step length of 1. The stepped pattern has the same degree of polymorphism as the cyclic one (same number of types), but much lower variability. It should thus be highly predictable, even by simple predictors such as a Branch Target Buffer. This stepped pattern is probably quite common in object-oriented programs, for example when iterating over containers of objects, which often contain instances of a single type.

3.5 Various execution environments

The execution environment is the last varying dimension in our study and consists of two parts: the hardware platform and the virtual machine used to execute the benchmarks. Running different virtual machines is similar to testing a particular program using different compilers. The addition of an extra execution layer, the JVM, makes execution more complex and makes it significantly harder to interpret performance results, but it provides platform-independence and is thus essential to our approach.

The benchmark suite was run on three hardware platforms:

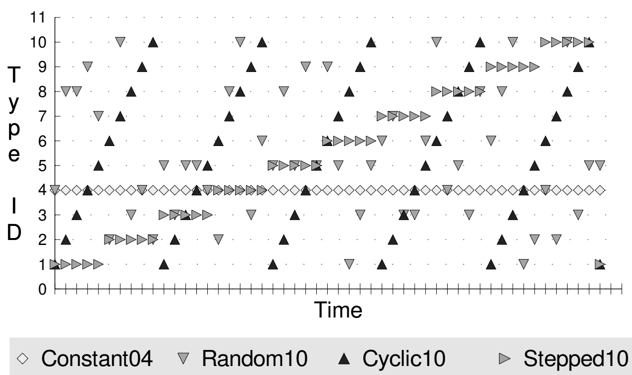

Figure 5: Patterns dynamic behavior

SUN UltraSparc III This machine is based on one 750 MHz processor and 1 GB of RAM, with SunOS 5.8.

Intel Pentium III This machine has dual 733 MHz processors, with 512 MB of RAM, running Linux Mandrake with kernel 2.2.19. Note that for our benchmarks, dual processor capability should have little if no impact.

Intel Celeron This lower-end machine comprises one 466 MHz Celeron with 192 MB of RAM and Linux Mandrake with kernel 2.2.17.

Of course, not all JVMs are available on all hardware platforms. Furthermore, the fact that a JVM is available under the same name on several different OS and hardware platforms is no guarantee at all they are indeed the same JVM: their back-ends for instance must be different. The JVMs tested during this study are generally in their 1.3.1 version. We show the IBM JVM (known as "the Tokyo JIT") and the SUN HotSpot Server as examples of high-performance JVMs and the SUN HotSpot Client, which is the most widely available JVM, and runs on many different hardware platforms.

The following result section shows the essence of the large amount of data gathered.

4. RESULTS AND DISCUSSION

As explained in the previous section, we measure the performance of different control structures in a number of varying dimensions: hardware, JVM, number of possible types (static) and type pattern (dynamic). This leads to a vast parameter space, in which we gather a very large number of data points (more than 21,000).

For space constraints reasons, we cannot show all the data and therefore we pick a representative sample: the dual Pentium III and the UltraSparc III, two hardware platforms described in section 3.5. The Celeron provides results very similar to the Pentium III, which is consistent with the fact both processors share the same architectural core; we thus did not include Celeron figures in this paper.

We also focus on a maximum number of possible types (static) of 20, which allows testing both low and high degrees of polymorphism, with patterns featuring as low as 1 actual live type at runtime (monomorphic) and as many as 20 (megamorphic [AH96]). Overall, this maximum degree of polymorphism of 20 is representative of behaviors and data we gathered at various sizes (we actually tested all maximum sizes from 1 to 10, then 20, 30, 50 and 90). Shorter static type sizes, which are the most common in real applications, typically lead to more efficient *if sequences* and binary search trees.

Results are presented in figures 6 and 7, that show two different JVMs on the same Pentium platform, as well as in figures 8 and 9 that show the HotSpot client JVM on two different hardware platforms.

On all these graphs, the same 5 benchmarks are tested, resulting in the 5 curves on each graph:

Virtual20 A plain virtual call, implemented with the `invokevirtual` bytecode, that can cope with any number of possible receiver types[1].

BinaryTreeStaticThisarg20 This is a binary tree dispatch, with 20 leaves that are static calls, the receiver object being passed as an explicit argument.

IfSequenceStaticThisarg20 A sequence of ifs containing 20 static leaf calls.

SwitchStaticThisarg20 A Java switch, translated into a `tableswitch` bytecode with 20 cases, each being a static call.

NoCall20 This benchmark contains no call at all, it shows the base cost of the benchmark mechanism (loop and static method call).

The different control structures are tested against 41 execution patterns of the four kinds presented in section 3.4, **c**on**st**ant, **cycl**ic, **r**a**nd**om and **step**ped, that compose the x axis. The numbers appearing in the pattern name indicate the active range of type IDs for each pattern. Thus `rnd-01-07` is a pattern made of random type IDs between 1 and 7, `step-01-09` is a type ID pattern with 9 steps, from 1 to 9, and `cst-04` is a pattern with constant type ID 4, and so on.

4.1 Observations

Figure 6 shows performance in milliseconds of execution time for the IBM JIT on a Pentium III. Plain *virtual calls* (`invokevirtual`, shown as continuous black curve) appear to be sensitive to the dynamic execution patterns tested. Virtual calls executing constant patterns and stepped patterns take about 700 ms, compared to 1000 ms for cyclic and random patterns. The NoCall20 micro-benchmark executes in 600 ms. Therefore the overhead of virtual calls varies between 100 and 400 ms, a factor of four due only to differences in type patterns. Other JVMs on the Pentium platform show similar ratios (figures 6 and 7). On an UltraSparc III (figure 9), virtual calls appear less sensitive to execution patterns. The constant pattern is executed slightly more efficiently, but a stepped pattern shows the same performance as a random or cyclic pattern. In contrast, stepped patterns with low variability behave well on all Pentium JVMs (figures 6, 7 and 8), with a cost close to that of the constant pattern. Overall, virtual calls tend to be more expensive than other structures especially when the number of different types is small and when the type pattern is cyclic. These results indicate optimization opportunities for JVM implementors.

The performance of *if sequences* depends on the size of the sequence and the rank of ifs exercised, shorter sequences being faster. Short *if sequences* are the most efficient way to implement dynamic dispatch among the tested control structures across all platforms, all JVMs and all execution patterns. Although the precise cutoff point varies, it is safe to consider that *if sequences* up to 4 are a sure win over current implementations of virtual calls. The actual gain in performance varies but can be as high as 52% (including benchmark overhead) on the duomorphic

[1]For Virtual and NoCall, the "20" in the name is only kept for consistency with other benchmark names.

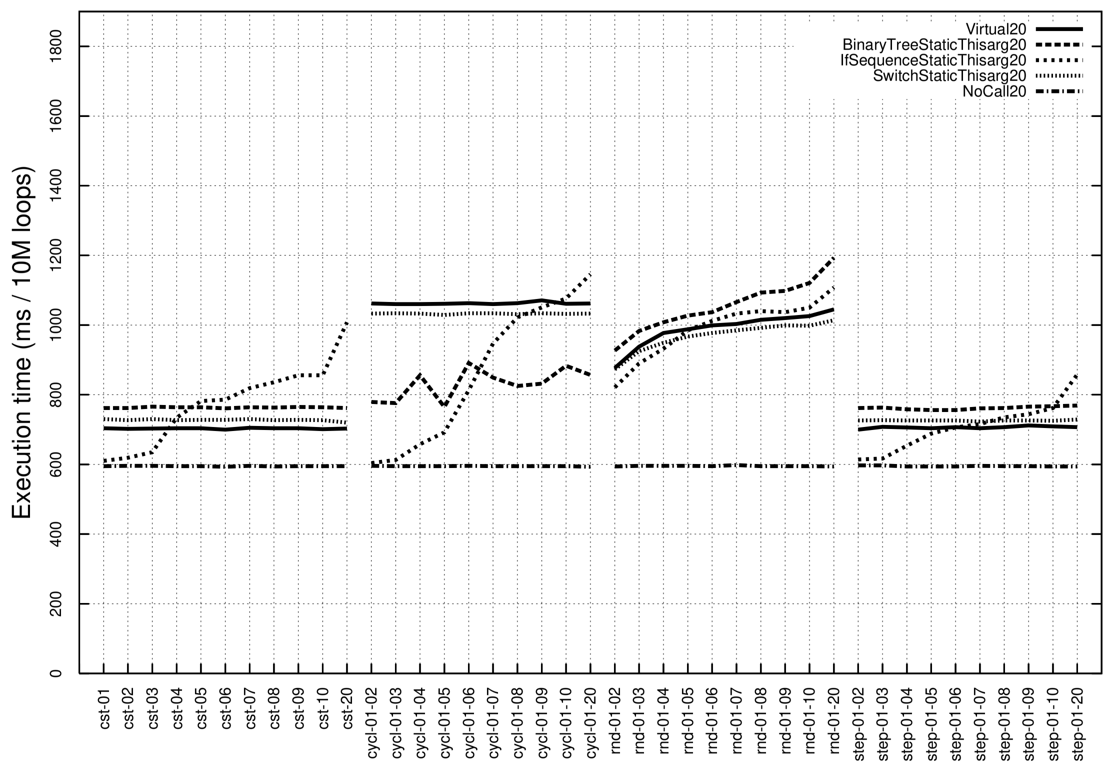

Figure 6: IBM cx130-20010502 on a dual Pentium III

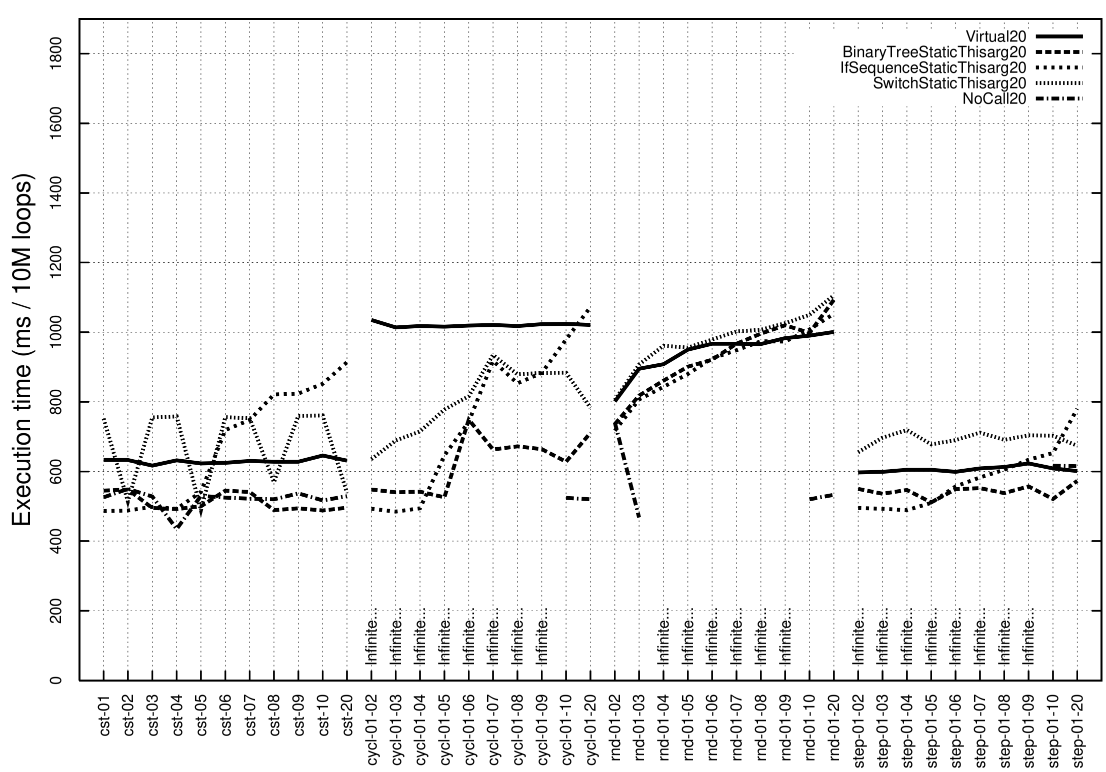

Figure 7: SUN HotSpot Server 1.3.1-b24 on a dual Pentium III

`cycl-01-02` pattern on HotSpot Server on Pentium III (figure 7) or 24% on the `step-01-02` pattern on HotSpot Client on UltraSparc III (figure 9). Therefore one can significantly optimize the implementation of dynamic dispatch in current JVMs when the number of possible types is known (by static analysis or dynamic sampling) to be small.

Binary tree dispatch (BTD) provides another way to perform strength reduction of dynamic dispatch sites. Binary trees appear to be significantly faster than virtual calls in most cases (all figures, particularly figures 7 and 9). When BTDs are slower than virtual calls, it is generally by a small margin, as figures 6 and 8 show. Since the cost of BTD grows as the logarithm of the number of branches, whereas sequences of *ifs* have a linear cost, BTD is more scalable. This makes BTD a good implementation for dynamic dispatch when the number of types is too large to use simple *if sequences* (above 4 or 8, depending on the JVM and platform), but small enough to prevent extensive code expansion. The cutoff point where BTD become faster than *if sequences* is clearly visible for cyclic patterns on all JVMs and platform, and for constant and stepped patterns in the SUN HotSpot JVMs on both platforms (figures 7, 8 and 9).

Figure 6 shows that Java *dense switches* (bytecodes `tableswitch`), when used to implement dynamic dispatch, result in performance very similar to that of virtual calls on the IBM JVM, revealing an implementation based on jump tables. In the HotSpot Client JVM however, both on Pentium III and UltraSparc III (figures 8 and 9), `tableswitch`es behave exactly like *if sequences*, which indicates an actual implementation based on sequences of conditional branches. Table switches are therefore unreliable in terms of performance across JVMs.

The "Infinite..." results in figure 7 correspond to executions of NoCall20 that were running forever[2]. We think that this behavior indicates an optimization bug on this particular JVM and platform, since the call of an empty method is an unlikely (but legal) occurrence, and all other JVMs dealt with it correctly. Indeed, the exact same bytecode for NoCall20 is correctly executed on all other JVM-platform combinations, that is with all other JVMs on the same platform and with all JVMs on all other platforms (we also checked on Athlon and Celeron). The same problem happens under the exact same conditions for other NoCall benchmarks with other sizes, but is much less frequent.

Since all our benchmarks are very small and simple and share most of their code, we are confident their Java source code (including the one for NoCall20) is correct. Furthermore, since all the benchmarks are executed correctly on all JVM-platform combinations but one, we trust the javac compiler generated a correct bytecode. We thus suspect some aggressive, non-systematic optimizations by the JVM might be the cause of this issue.

4.2 Discussion

Obviously, using micro-benchmarks focused on dynamic dispatch magnifies the impact of the various dispatch techniques in terms of performance. Although the actual impact on real programs is likely to be smaller, since programs generally do other things than dispatch, our study makes it possible to get a clearer view of what is actually happening. We thus believe that the previous results are an important first step and can already be widely used.

First, these results are important to Java compiler and Java VM designers, when implementing multiple-target control structures such as dynamic dispatch. We show that the performance of dynamic dispatch varies a lot across JVMs, hardware and execution patterns. It is safe to say that dynamic dispatch implementation in current JVMs is not always optimal and can be significantly improved, using mostly known techniques. Direct implementation in the virtual machine is likely to provide the highest payoff.

Second, these results are also useful to Java developers, since they stress differences between the various JVMs, highlighting strengths to take advantage of and weaknesses to avoid, for instance large `tableswitch`es in the HotSpot Client.

Third, our results show that strength reduction of control structures is likely to be beneficial regardless of the hardware and JVM, when the number of possible receiver types can be determined to be small. For numbers of possible types up to 4, *if sequences* are most efficient. Between 4 and 10, binary tree dispatch is generally preferable. For more types, the best implementation is a classical table-based implementation such as currently provided by most JVMs for virtual calls. These are safe, conservative bets, that generally provide a significant improvement and, when not optimal, result only in a small decrease in performance.

Finally, these measurements expose architectural features (especially branch predictors) of the target hard-

[2] "Forever" means for example that such a program was still running after 18 *hours*, instead of a typical execution time below one minute.

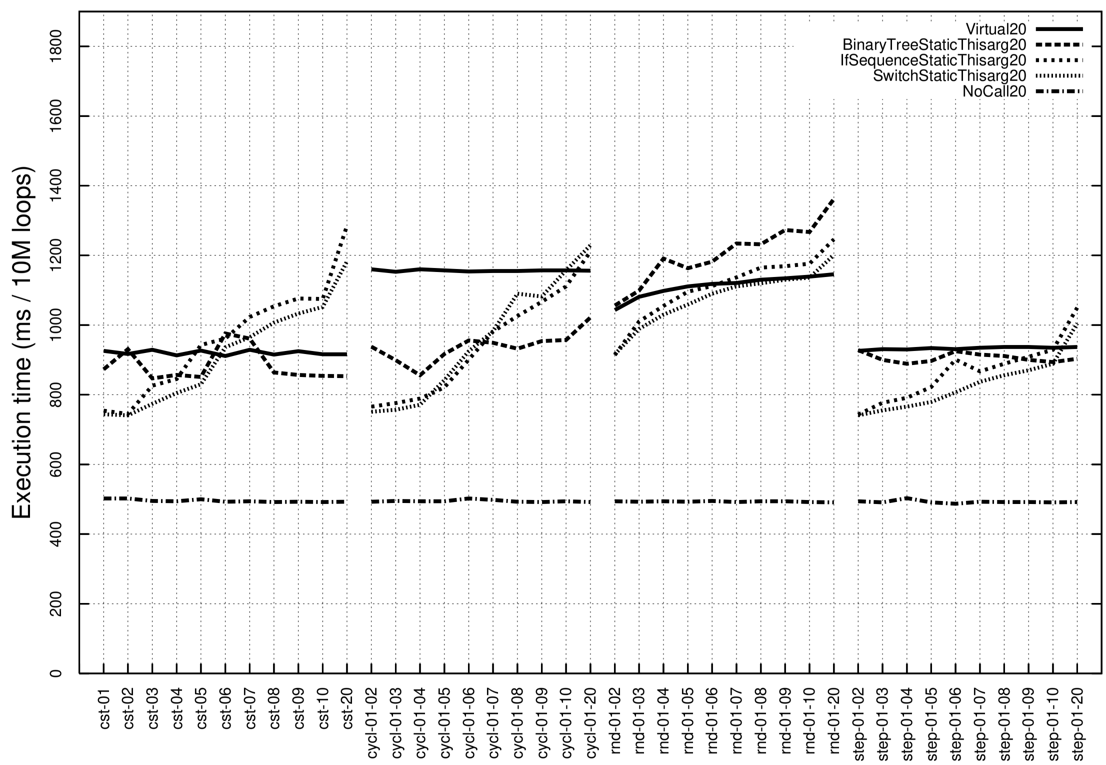

Figure 8: SUN HotSpot Client 1.3.1-b24 on a dual Pentium III

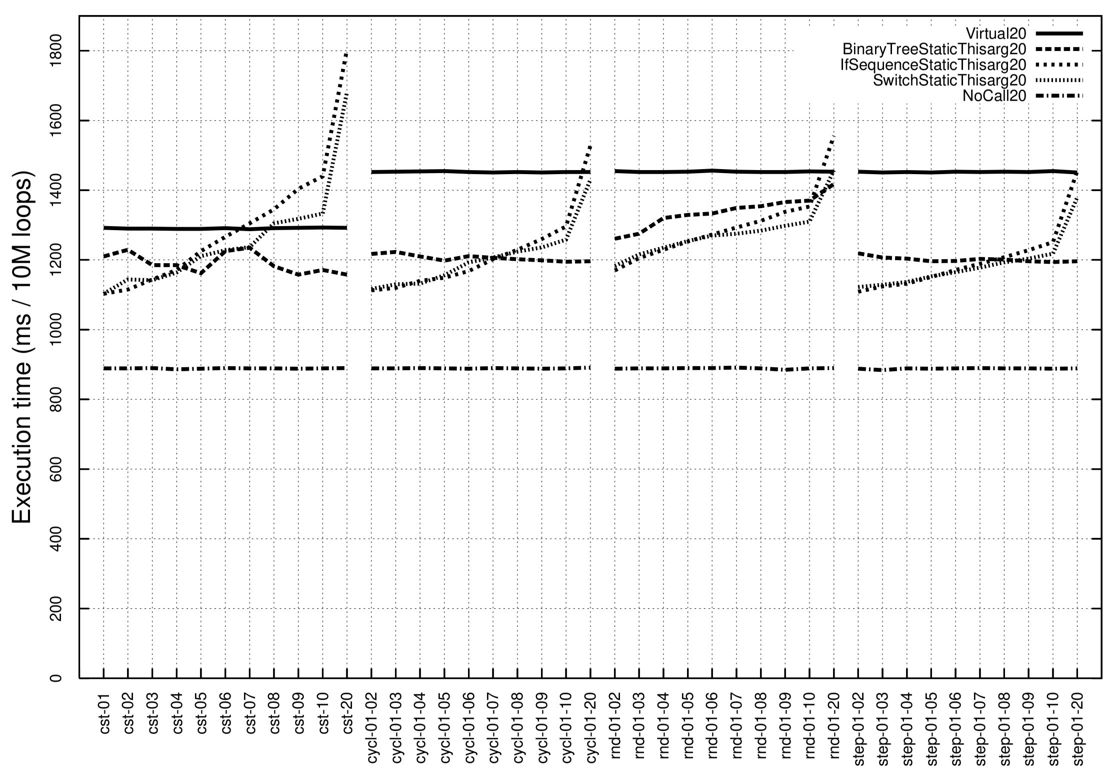

Figure 9: SUN HotSpot Client 1.3.1-b24 on UltraSparc III

ware. For instance, when executing virtual calls the Pentium III branch target buffer ensures that constant patterns have performance nearly identical to that of slowly changing stepped patterns, whereas this is not the case for the UltraSparc III. Similarly, when executing *if sequences*, small cyclic patterns are predicted accurately by the Pentium's conditional branch predictor, which, for all JVMs, results in better performance on small cyclic patterns than on random patterns.

Consequently, the results we provide in this paper can be applied at various levels.

The information we gathered can be used by a static compiler (e.g., javac) that performs a static analysis of the program to determine at compile time the number of possible types, and generate bytecode relying on the most appropriate implementations of dynamic dispatch for each call site, either aggressively targeting a particular platform or conservatively performing transformations for multiple platforms. An extra type ID field might have to be added to all objects, which would lead to per-object space overhead as well as initialization time overhead. However, smart implementations (see [CZ99] for an example in the context of Eiffel) can avoid the need for the type ID field for objects which are not subject to actual dynamic dispatch, as detected by global analysis. The type ID overhead can also be made smaller than an integer, for example when the number of types subject to dispatch fits in 16 or 8 bits, or by packing the type ID in available bits in the objets. Initialization overhead is dependant on the objets lifetime, creation rate, and call frequency, and thus varies between applications. Space and initialization overhead thus have to be better quantified to find the conditions under which each solution is the best.

JVM implementers can also make use of this information in a rather similar way, by dynamically compiling bytecode into the most suitable native code structures, based on program execution statistics. Dynamic optimizers could thus switch between several dynamic dispatch mechanisms, depending on context, execution environment and profiling information.

Finally, micro-architecture designers can use these measurements to determine how to better support the execution of JVMs and the programs that run on those JVMs, in particular with respect to dynamic dispatch, for instance by providing improved branch prediction mechanisms.

As mentioned in section 3.3, all the control structures we studied, except the plain `invokevirtual`, have a size that is proportional to the number of tested types. This number can become quite large, in real OO programs; for example, in the SmallEifel compiler, the maximum arity at a dispatch site is about 50 [ZCC97]. In such cases, an increase in code size could happen, with adverse effects on caches and performance, and thus would have to be mastered. We did not work on this aspect in the present study relying on micro-benchmarks. However, in the SmallEiffel projet, we tackled this issue and used a simple but efficient solution, which consists in factorizing all identical dispatch sites into one or a few dispatching routines ("switch" functions in [ZCC97]). Although we have obtained good results with this technique in Eiffel, we still have to mesure its feasabibility in Java.

5. CONCLUSIONS AND FUTURE WORK

The implementation of dynamic dispatch is important for object-oriented program performance. A number of optimization techniques exist, aimed at de-virtualizing polymorphic calls which can be determined, either at compile-time or runtime, to be actually monomorphic. Complementary techniques, either software- or hardware-based, seek to optimize actual run-time polymorphism as well.

We present a prototype study of various control flow structures for dynamic dispatch in Java, with varying hardware, virtual machine and execution patterns.

Our results clearly show that:

- Virtual call performance is highly dependent on the execution pattern at a particular call site.
- When the call site has a low or medium degree of polymorphism (2-3 target types up to about 10), strength reduction of control structures is likely to improve performance across platforms, using *if sequences* for up to 4 different target types and Binary Tree Dispatch between 4 and 10 different types.
- Processor architecture shines through, more especially on high-performance JVMs: virtual call performance of stepped patterns, for example, is markedly different on different platforms, but does not vary across different JVMs on the same platform.

In future work, we could experiment with more techniques or variants for dynamic dispatch, such as the ones we mentioned in section 3.3, and more platforms (JVM or hardware).

Another area we have to work on is interface dispatch in Java, which is more complex because of multiple interface inheritance, and where some of the techniques we described are not easily applied.

We also plan to more precisely assess the efficiency of the techniques we described by completing our micro-benchmarks suite with larger, real Java programs. This would give more applicable, although less precisely understandable, results.

We also intend to evaluate the impact of these various dispatch techniques with respect to code size and memory footprint, especially for techniques whose code size is proportional to the number of types (*if sequences* and BTD).

We can do so by applying our results either to open-source bytecode optimizers, such as Soot [VRHS+99], or directly to Java Virtual Machines, like the Open VM [Va01], the Jikes Research VM [IBM01] (formerly named Jalapeño) or the SableVM [GH01].

Acknowledgements

We thank Laurie Hendren and Feng Qian, who helped us in early stages of our experiments. We thank everyone who commented on the poster presentation of this work at OOPSLA 2001. We are also grateful to Wade Holst and Raimondas Lencevicius who commented on early versions of this paper, and Matthew Holly who proofread it. Finally, we thank the anonymous reviewers for their valuable comments and suggestions.

This research was supported by NSERC and FCAR (Canada), and INRIA (France).

6. REFERENCES

[AH96] Gerald Aigner and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs. In *10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166. Springer-Verlag, 1996.

[BS96] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 324–341. ACM Press, 1996.

[CC99] Craig Chambers and Weimin Chen. Efficient Multiple and Predicated Dispatching. In *14th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, pages 238–255. ACM Press, October 1999.

[CHP98] Po-Yung Chang, Eric Hao, and Yale N. Patt. Target Prediction for Indirect Jumps. In *1997 International Symposium on Computer Architecture (ISCA'97)*, July 1998.

[CZ99] Dominique Colnet and Olivier Zendra. Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler. In *29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99)*, pages 341–350. IEEE Computer Society, June 1999.

[DDG+96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 83–100. ACM Press, 1996.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer-Verlag, 1995.

[DH96] Karel Driesen and Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. In *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 306–323. ACM Press, 1996.

[DH98a] Karel Driesen and Urs Hölzle. Accurate Indirect Branch Prediction. In *1998 International Symposium on Computer Architecture (ISCA'98)*, July 1998.

[DH98b] Karel Driesen and Urs Hölzle. The Cascaded Predictor: Economical and Adaptive Branch Target Prediction. In *Micro'98 Conference*, pages 249–258, December 1998.

[DHV95] Karel Driesen, Urs Hölzle, and Jan Vitek. Message Dispatch on Pipelined Processors. In *9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Science*, pages 253–282. Springer-Verlag, 1995.

[DLM+00] Karel Driesen, Patrick Lam, Jerome Miecznikowski, Feng Qian, and Derek Rayside. On the Predictability of Invoke Targets in Java Byte Code. In *2nd Annual Workshop on Hardware Support for Objects and Microarchitectures for Java*, pages 6–10, September 2000.

[Dri01] Karel Driesen. *Efficient Polymorphic Calls*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 2001.

[DS84] Peter L. Deutsch and Alan Schiffman. Efficient Implementation of the Smalltalk-80 System. In *11th Annual ACM Symposium on the Principles of Programming Languages (POPL'84)*. ACM Press, 1984.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1990.

[GH01] Etienne Gagnon and Laurie Hendren. SableVM: A Research Framework for the Efficient Execution of Java Bytecode. In *1st Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 27–39. The USENIX Association, April 2001.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley, 2000. Second Edition.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *5th European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38. Springer-Verlag, 1991.

[HU94] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, volume 29 of *SIGPLAN Notices*, pages 326–336. ACM Press, 1994.

[IBM01] IBM Research - Jalapeño Project. The Jikes Research Virtual Machine. `http://www.ibm.com/developerworks/oss/jikesrvm`, 2001.

[Lis88] Barbara Liskov. Data Abstraction and Hierarchy. In *Special issue: Addendum to the proceedings of OOPSLA'87*, volume 23 of *SIGPLAN Notices*, pages 17–34. ACM Press, May 1988.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Language Specification*. The Java Series. Addison-Wesley, 1999. Second Edition.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical Virtual Method Call Resolution for Java. In *15th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*, volume 35, pages 264–280. ACM Press, October 2000.

[SLCM99] Ulrik Pagh Schultz, Julia L. Lawall, Charles Consel, and Gilles Muller. Towards Automatic Specialization of Java Programs. In *13th European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 367–390. Springer-Verlag, 1999.

[UP87] David M. Ungar and David A. Patterson. What Price Smalltalk ? *IEEE Computer Society*, 20(1), January 1987.

[Va01] Jan Vitel and al. The Open Virtual Machine Framework. `http://www.ovmj.org`, 2001.

[VRHS+99] Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java Optimization Framework. In *CASCON 1999*, pages 125–135, 1999.

[ZCC97] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, volume 32, pages 125–141. ACM Press, October 1997.