Trace-based Dynamic Compilation for Object-Oriented Programming Systems.

Mathew Zaleski[*]

26th April 2003

Contents

[*] This document in support of my Qualifying Oral examination for the Department of Computer Science , University of Toronto.
$Revision: 2.2 $

This draft not for public consumption.
If you are not a collaborator or advisory committee member then please ask for an up-to-date draft and/or permission to read further.
(That would be an appropriate moment to wish me luck.)

contact author at matz@cs.toronto.edu

List of Figures

1 Introduction

Purpose of this Document This document represents the written component of my preparation for the oral depth examination checkpoint of the PhD in Computer Science program at the University of Toronto. As such it should provide a reasonably deep and wide survey of my chosen field. In addition I have prepared a research proposal, available at `http://www.cs.toronto.edu/~matz/depth.pdf`, in which I propose to build a trace-based dynamic compiler for Java bytecode. My hope is that these two documents will together update my advisory committee (and other readers) as to the status of my research.

Selection of Topics A modern dynamic compiler for an object-oriented language represents a large amount of technology. In this paper I have tried to concentrate on those topics that are less commonly known. Thus, I freely use general terminology from the fields of computer architecture, compilation and optimization with little or no introduction. The largest section describes the trace-related work because it is central to my thesis and not particularly well-known. I give only a cursory overview of the Java virtual machine and no introduction to object-oriented programming in general or Java in particular because I believe these are well-known (and easy to research) fields. The field of dynamic optimization is rapidly expanding, so this paper will introduce several specific techniques in some detail with the intention of giving the reader a good feel of the aggressive approach currently being followed by many researchers.

Structure of this document The structure of this review is as follows:

- Informally define polymorphism, describing how it poses a challenge to a naïve compiler and how sophisticated caching techniques like those invented for Self [8, 24, 44] can help;
- Very briefly describe the working of the Java Virtual Machine. Sun's own description [10] is very good, so only an overview need be given here;
- Briefly describe manually directed dynamic optimization techniques like those offered by DyC [2, 18, 19]. The emphasis is why they are not, as yet, directly applicable to Java bytecode execution;
- Describe the trace-oriented dynamic optimization approach used by HP Dynamo and its successors as the departure point for my own proposal;
- Briefly describe a few key dynamic object-oriented optimization techniques invented for Self and Java, and;
- Relate the above-mentioned topics to the challenges faced by a method-based Just in Time compilers within a Java Virtual Machine.

2 Basic Terms – Traces and Dynamic Optimization

Trace A *trace* is a sequence of instructions that corresponds to a path taken through a program. The sequence may include conditional branch instructions and some mechanism for recording whether each branch was taken or not. When a trace is dispatched the remaining conditional branch instructions can be thought of as assertions that cause the trace to exit whenever a branch would take a different path than when the trace was recorded. Thus, traces are single-entry, multiple-exit paths through a program's control flow graph.

Traces are used by different research communities for different reasons.

The following list of research areas is in the order of least to most importance to my research.

- The Multiflow compiler [32, 15] performs instruction scheduling on traces of instructions.
- The Pentium 4 processor refers to its level 1 instruction cache as an "Execution Trace Cache"[23]. The concept of storing traces in a hardware instruction cache to maximize use of instruction fetch bandwidth is discussed by Rotenberg and Bennett in [41]. It turns out that optimization techniques such as the "Software Trace Cache" reorder code to achieve a similar result [40]. The impact of a trace cache on instruction prefetch bandwidth may help explain some performance results reported by dynamic optimizer research groups, notably [7].
- In the Dynamo [4, 3, 14, 5, 6] project, work that has had major impact on my approach, traces are used to identify, record and dynamically optimize heavily used paths within an program. This will be described in great detail.

Dynamic Optimization *Dynamic optimization* is a broad field of research wherein investigators look at ways programs can be improved as they run. There is a broad spectrum of approaches, from application-specific, special-purpose languages compiled at run-time, through manually written templates that are "filled in" dynamically at run-time. The most ambitious systems automatically select regions of code to optimize. Only automatic techniques can be applied within a Java Virtual Machine to speed up existing Java programs so these will be described in the most detail.

Trace oriented Dynamic Optimization One of the challenges with dynamic optimization is identifying which regions to optimize. One option, pioneered by Dynamo, is to heuristically select traces and dynamically optimize them.

```
void sample(Object[] otab){
   for(int i=0; i<otab.length; i++){
      otab[i].toString();  //polymorphic callsite
      }
   }
```

Figure 1: Example of Java method containing a polymorphic callsite

3 Polymorphism

The callsites of an object-oriented language are *polymorphic* as a consequence of methods being invoked on objects. Most object-oriented languages categorize objects into a hierarchy of *classes*. Each object is an *instance* of a class which means that the methods and data *fields* defined by that class are available for the object. Each class, except the root class, has a *super-class* or *base-class* from which it *inherits* many of its fields and methods. This organization is meant to reflect the pragmatic reality that real-world programs manipulate concepts that are specializations of more generic concepts.

The class of an object on which a method is invoked determines which implementation of the method is dispatched. The idea is that each type of object may override a method and so at runtime the system must dispatch the definition of the method corresponding to the type of object. In many cases it is not possible to deduce the exact type of the object at compile time. The invocation is called polymorphic because it might be invoked on multiple types of objects.

It is not our purpose here to discuss the software engineering benefits of polymorphism. Many enlightened introductions to object-oriented programming have been written. They range from the green fields approach taken by Goldberg and Robson in their famous Smalltalk-80 series [17] to the incremental approach taken by C++ designers like Stroustrup [42].

A simple example will make the above description concrete. When it is time to debug a program almost all programmers rely on facilities to view a textual description of a their data[1]. In an object-oriented environment this suggests that each object should define a method that returns a string description of itself. This need was recognized by the designers of Java[2] and consequently they defined a method in the root class `Object`:

```
public String toString()
```

to serve this purpose. The `toString` method can be invoked on every Java object. Consider an array of objects in Java. Suppose we code a loop that iterates over the array and invokes the `toString` method on each element as in Figure 1. There are literally hundreds of definitions of `toString` in a Java

[1] Programmers might view the text in an interactive debugger or a log file. In either case the text was probably produced similarly.

[2] The `printString` method plays the same role in Smalltalk.

system and in most cases the compiler cannot discern which one will be the destination of the callsite. Since it is not possible to determine the destination of the callsite at compile time it must be done when the program executes. Determining the destination at run-time incurs cost. The cost comes about in two main ways. First, locating the method to dispatch at run-time requires computation. Second, the inability to predict the destination of a callsite reduces the efficacy of inter-procedural optimizations and thus results in relatively slow systems. We discuss each of these issues in turn.

3.1 Finding the destination of a polymorphic callsite

Locating the definition of a method appropriate for a given object is a search problem. To search for a method definition corresponding to a given object the system must search the classes in the hierarchy. The search starts at the class of the object, proceeds to its super class, to its super class, and so on, until the root of the class hierarchy is reached. If each method invocation requires the search to be repeated, the process will be a significant tax on overall performance. Nevertheless, this is exactly what occurs in a naïve implementation of Smalltalk, Self or Java.

If the language permits early binding, the search may be converted to a table lookup at compile-time. For instance, in C++, all the possible destinations of a callsite are known when the program is loaded. As a result a C++ virtual callsite can be implemented as an indirect branch via a virtual table specific to the class of the object invoked on. This reduces the cost to little more than a function pointer call in C. In fact, early implementations of C++, which translated C++ source to C source, would translate each C++ virtual callsite to a call through an array of pointers to functions in C. The construction and performance of virtual function tables has been heavily studied, for instance by Driesen [13].

The search can very often be short-circuited by a cache. It turns out that real programs tend to have low *effective polymorphism*. This means that the average callsite has very few actual destinations. If fact, most callsites are *effectively monomorphic*, meaning they always call the same method. If this destination is cached then the search can be circumvented in most cases. Note that low effective polymorphism does not imply that a smart compiler should have been able to deduce the destination of the call. Rather, it is a property of real programs making use of polymorphism much less than they could have.

3.2 Inlined Caching and Polymorphic Inlined Caching

For late-binding languages it is seldom possible to generate efficient code for a callsite at compile time. In response, various researchers have investigated how it might be done at run-time. In general, it pays to cache the destination of a callsite when the callsite is commonly executed and its effective polymorphism is low. The *in-line cache*, as invented by Deutsch and Schiffman [12] for Smalltalk about 20 years ago, replaces the polymorphic callsite with the native instruction to call the cached method. The prologue of all methods is extended with fix-up code in case the cached destination is not correct. Deutsch and Shiffman reported hitting the in-line cache about 95% of the time for a set of Smalltalk programs.

Hölzle[24] elaborated the in-line cache to be a *polymorphic in-line cache* (PIC) by generating code that successively compares the class of the invoked object to a few possible destination types. The implementation is more difficult than an in-line cache because the dynamically generated native code sequence must sequentially compare and conditionally branch against several possible destinations. The performance of a PIC is good. A PIC extends the performance benefits of an in-line cache to effectively polymorphic callsites. For example, on a SPARCstation-2 Hölzle's lookup would cost only 8 + 2n cycles, where n is the actual polymorphism of the callsite. A PIC lookup costs little more than an in-line cache for effectively monomorphic callsites and much less for effectively polymorphic ones.

3.3 Impact on Inter-procedural Optimization

Inter-procedural optimization can be stymied by polymorphic callsites. At compile time, an optimizer cannot determine the destination of a call, so obviously the target cannot be inlined. In fact, standard inter-procedural optimization as carried out by an optimizing C or FORTRAN compiler [34] is simply not possible.

In the absence of inter-procedural information, an optimizer must make conservative assumptions about which values remain alive across a polymorphic callsite. This forces the compiler to reload values from memory, oftentimes unnecessarily, on the chance that code in the (unknown) callee has stored a new value. Knowledge of the destination of the callsite would permit a more precise inter-procedural analysis of the values killed by the call.

Given the tendency of modern object-oriented software to be factored into many small methods which are called throughout a program, even in its innermost loops, these optimization barriers prevent good performance. A typical example might be that common subexpression elimination cannot combine identical memory accesses separated by a polymorphic callsite because it cannot prove that all possible callees do not kill the memory location. To achieve performance comparable to procedural compiled languages, inter-procedural optimization techniques must somehow be applied to regions laced with polymorphic callsites.

In response, researchers have turned their attention to how optimizations across callsites can be delayed until run-time, when the destination of callsites becomes known. At run-time, a dynamic compiler can gamble that a callsite will continue to be effectively monomorphic and perform limited inter-procedural optimizations such as inlining. An excellent description of such techniques for the Self compiler appears in Urs Hölzle's dissertation. Hölzle records the type of object each method is invoked on as the program runs. To give an example of how this data can then be used, Hölzle writes, "Having obtained the program's type profile, this information is then fed back into the compiler so it can optimize dynamically-dispatched calls (if desired) by predicting likely receiver types and inline the call for these types"[3][24, pp 36].

We have described how polymorphic callsites pose a challenge to system performance. A more detailed discussion of how dynamic compilation techniques can be used to implement in-line caching as well as inter-procedural optimization appears in Section 8.

4 Overview of the Java Virtual Machine

A Java program is deployed as a set of classes. Methods are expressed in terms of a stack-oriented bytecode defined by the Java Virtual Machine Specification [31]. A Java Virtual Machine (JVM) is the core of the Java runtime system. It is a large computer program that loads and runs Java classes, manages threads and memory, and maintains security. The first publicly available implementation of the Java Virtual Machine was a bytecode interpreter shipped as part of Sun's Java Development Kit ("the JDK"). The JDK first appeared following the public debut of the World Wide Web. Very soon after its release, Java generated enormous interest by demonstrating portable Applets embedded in web pages running in early web browsers. Perhaps due to the excited mood of the period, many analysts overlooked the well-known fact that the performance of bytecode interpreters is generally unacceptably slow for many classes of application. Many application programmers, myself included, jumped on the bandwagon and wrote and deployed large Java applications. Several large technology companies, for instance IBM, anointed Java classes as the strategic deployment technology for enterprise applications. Ever since, JVM developers and researchers have been working hard to increase the performance of Java applications.

This section will introduce basic JVM concepts. Our discussion of techniques used to enhance Java performance will resume in Section **??** after we have reviewed earlier existing dynamic optimization research. We will revisit the Java example of Figure 1 in order to discuss the bytecode produced `javac`, the compiler packaged with Sun's JDK. Figure 2 shows output of the `javap` disassembler. I have manually commented the `javap` output (in italics in the figure) to make it easier for those who are unfamiliar with bytecode to understand.

4.1 Late binding

A basic design issue for any language is when external references are resolved. Java is forced to bind references very late in order to support downloadable code. The general idea is that a Java program may start running before all the classes that make it up are locally available. This means that there is no moment in the life of a Java program equivalent to the link-edit step of a traditional application; rather, classes are searched for and downloaded as they are needed.

Although the consequences of very late binding are far-reaching, of particular interest is the impact of late binding on the implementation of polymorphic

[3] The object a method is invoked on is called a "receiver" in Self and Smalltalk.

callsites. Late binding complicates using a table-driven approach in Java.

In a language such as C++, *virtual function tables* are used to dispatch methods. Each method of a class is associated, at compile time, with an offset which is used at run-time to index into a table of pointers to functions. Polymorphic dispatch is thus converted into an indirect branch at run-time. Since C++ programs must undergo an explicit link edit step before then can be run, the tables can be initialized then.

It is not clear-cut that virtual function tables are the right approach for Java. Sun's JVM specification [31] requires that the `invokevirtual` bytecode, which does the actual work of a virtual method invocation, must refer to the callee by name. For example, the `invokevirtual` bytecode appearing in Figure 2 refers to the `toString` method as `toString()Ljava/lang/String`[4]. Thus, the conversion of each callsite to use a virtual table, if it is to be done at all, must be done when the class is loaded, or even later, such as the first time the callsite is executed. However, it has already been shown that an in-line cache is hit much of the time, suggesting that the overhead of virtual table construction might not be worthwhile except for callsites known to have relatively high effective polymorphism for which an in-line cache or PIC is ineffective.

4.2 Stack oriented Bytecode and portability

Packaging Java classes as bytecode to be run by a virtual machine serves several purposes. First, stack oriented bytecode is very compact because the operands of instructions are implicit. Since Java classes might have to be fetched across the Internet this can save download time. Second, a stack oriented virtual machine is a traditional and very effective way of abstracting away the details of underlying hardware and thus enhancing portability. Java bytecode bears a striking resemblance to Pascal P-code, defined more than 20 years ago for the Pascal P4 interpreter [38]. Portability is, of course, desirable due to Java's "write once run anywhere" credo.

Java bytecode includes object-oriented instructions like method invocation and field getting and setting as well as standard fare such as arithmetic operations. Fortunately, the Java Virtual Machine Specification [31] manual gives a readable description. The flavor of bytecode can be quickly appreciated by quickly reading through the bytecode illustrated by Figure 2.

The bytecode in the figure depends on a stack organization that distinguishes between local variables and operand stack. *Local variable slots*, or *lva* slots, are used to store local variables and parameters. The simple function shown needs only three local variable slots (referred to as lva[0] through lva[2] in the figure). Lva[0] is used to store a hidden parameter, the object handle[5] to the invoked upon object and is never used in the example. Lva[1] stores the first and only real parameter, the `otab` array. Lva[2] stores `i`, an `int` local variable. The

[4] Actually the `invokevirtual` takes a parameter which is the index into the classes' constant table at which the string name of the method is stored. The `javap` disassembler, which produced the text reproduced in the figure, prints the value of the constant table entry.

[5] lva[0] stores the local variable known as `this` to Java (and C++) programmers.

```
//lva[N] refers to n'th
//Local Variable Array slot
//arguments: ''this''    in lva[0]
//            otab array in lva[1]

Method void sample(java.lang.Object[])
0 iconst_0     //push int constant 0 on stack
1 istore_2     //int store top of stack (zero) to lva[2]
2 goto 15      //jump into for loop
               //head of loop
5 aload_1      //push object lva[1] (otab)
6 iload_2      //push int lva[2] (i)
7 aaload       //load element from array

8 invokevirtual #2 <Method java.lang.String toString()>

11 pop         //discard return result
12 iinc 2 1    //bump int lva[2] by 1
15 iload_2     //push int lva[2] (i) on stack
16 aload_1     //push object in lva[1] on stack
17 arraylength //calculate length of array on stack
18 if_icmplt 5 //if int compare (i < otab.length) goto 5
21 return
```

Figure 2: Bytecode produced by Sun's `javac` compiler from `toString` example of Figure 1

operand stack is used to maintain the expression stack used for all calculations and parameter passing. In general "load" form bytecodes push values in lva slots onto the operand stack. Bytecodes with "store" in their mnemonic typically pop the value on top of the operand stack and store it in a named lva slot. For example, the first two bytecodes in the program (`iconst_0` and `istore_2`) push a literal zero onto the operand stack and then pop it off and store it in lva[2].

4.3 Speed of Interpretation and Dynamic Translation to Native Code

Although stack oriented bytecode is compact and machine-neutral, it is slow to interpret. Traditionally, interpreters were used primarily for application development and were complemented by true compilers which were used when applications were deployed. For instance, the Berkeley Pascal System[28] included `pi`, an interpreter, as well as `pc`, a compiler that generates executables that perform much better.

Java classes may be downloaded and run by different types of machines, so they must be deployed in a machine-neutral form. This eliminates the possibility of a complementary compiler. As a consequence, Java language developers turned their attention to dynamic compilation techniques to convert the bytecode into native code after it has been loaded by the JVM. The field has come to be called *Just In Time* compilation, or JIT compilation for short. This approach to speeding up interpretation long predates Java, perhaps first appearing for APL for the HP3000 [27] in about 1979. Deutsch and Schiffman [12] built an early JIT for Smalltalk that obtained a speedup of about two relative to interpretation.

Early systems were highly memory constrained by modern standards. It was of great concern, therefore, when translated native code was found to be about four[6] times larger than the originating bytecode. Lacking virtual memory, Deutsch and Schiffman took the view that dynamic translation of bytecode was a space time trade-off. If space was tight then native code (space) could be released at the expense of re-translation (time). Nevertheless, their approach was to execute only native code. Each method had to be fetched from a native code cache or else re-translated before execution. Today a similar attitude prevails except that it has also been recognized that some code is so infrequently executed that it need not be translated in the first place. Instead some heuristic should be used to identify *hot*, or frequently executed, regions to translate hoping to keep the size of the native code cache below some threshold. The bytecode of methods that are not hot can simply be interpreted.

A JIT can improve the performance of a JVM substantially. Relatively early Sun JIT compilers, as reported by the 1997 Sun development team in [11], improved the performance of the Java `raytrace` application by a factor of 2.2 and

[6] This is less than one might fear given that on a RISC machine one typical arithmetic bytecode will be naïvely translated into two loads (pops) from the operand stack, one register-to-register arithmetic instruction to do the real work and a store (push) back to the new top of the operand stack.

compress by 6.8. (These benchmarks are singled out because they eventually were adopted by the SPEC consortium to be part of the SPECjvm98 [10] benchmark suite. Today they are commonly reported by researchers.) More recent JIT compilers, for instance [33, 1, 43] have increased the performance further.

4.4 Other Challenges to Java Performance

There are many other issues that affect Java performance which this paper does not discuss. The most important amongst them are memory management and thread synchronization.

Garbage collection refers to a set of techniques used to manage memory in Java (as in Smalltalk and Self). In general the idea is that unused memory (garbage) is detected automatically by the system. (As a result the programmer is relieved of any responsibility for freeing memory that he or she has allocated.) Garbage collection techniques are somewhat independent of JIT techniques. The primary interaction requires that threads can be stopped in a well-defined state prior to garbage collection. So-called *safe points* must be defined at which a thread periodically drives its state to memory. Code generated by a JIT compiler must ensure that safe points occur frequently enough that garbage collection is not unduly delayed. Typically this means that each transit of a loop must contain at least one safe point.

Java supports explicit, built-in support for threads. *Thread synchronization* refers mostly to the functionality that allows one one thread at a time to access certain regions of code. Thread synchronization must be implemented at various points and the techniques for implementing it must be supported by code generated by the JIT compiler.

5 Manual Dynamic Optimization

Early experiments with dynamic optimization indicated that large performance pay backs are possible. Typical early systems were application-specific, in the sense that they compiled a language or data structure that described an algorithm specific to a particular purpose [29, 39]. Later, researchers built semi-automatic dynamic systems that would re-optimize regions of C programs at run-time [30, 2, 16, 19, 18].

Although the semi-automatic systems did not enable dramatic performance improvements across the board, I believe that this is partially because of the performance baseline they compared themselves to. The prevalent programming languages of the time were supported by static compilation and so it was natural to use the performance of highly optimized binaries as the baseline. Our situation in Java is somewhat different. I suspect that dynamic techniques which do not pay off relative to statically optimized C code will pay off when applied to code naïvely generated by a Java JIT. Consequently, a short description of a few early systems seems worthwhile.

5.1 Application specific dynamic compilation

In 1968 Ken Thompson built a dynamic compiler which accepted a textual description of a regular expression and dynamically translated it into machine code for a IBM 7094 computer [29]. The resulting code was dispatched to find matches quickly.

In 1985 Pike et al. invented an often-cited technique to generate good code for quickly copying regions of pixels onto a display [39]. They observed that there was a bewildering number of special cases (caused by various alignments of pixels in display memory) to consider when writing a good general purpose bitblit routine. Instead they wrote a dynamic code generator that could produce a good (near optimal) set of machine instructions for each specific blit. At worst their system required about 400 instructions to generate code for a bitblit.

5.2 Dynamic compilation of manually denoted static regions

In the mid-1990's Lee and Leone [30] built FABIUS, a dynamic optimization system for the research language ML [16]. FABIUS depends on a particular use of *curried functions.* Curried functions are those that take one or more functions as parameters and return a new function that is a composition of the parameters. FABIUS interprets the call of a function returned by a curried function as a clue from the programmer that dynamic re-optimization should be carried out. Their results, which they describe as preliminary, indicate that small, special purpose applications such as sparse matrix multiply or a network packet filter may benefit from their technique but the time and memory costs of re-optimization are difficult to recoup in general purpose code.

More recently it has been suggested that C and FORTRAN programs can benefit from dynamic optimization. Auslander [2], Grant [19, 18] and others have built semi-automatic systems to investigate this. Initially these systems required the user to identify regions of the program that should be dynamically re-optimized as well as the variables that are run-time constant. Later systems allowed the user to identify only the program variables that are run-time constants and could automatically identify which regions should be re-optimized at run-time.

Figure 3 reproduces an annotated C function given by Auslander in [2] showing how regions that can benefit from dynamic optimization are identified by the programmer. The dynamic region will be pre-compiled into template code. Then, at run time, the values of the run-time constant (the variable `cache` in Figure 3) will be substituted into the template and the dynamic region re-optimized. Auslander's system worked only on relatively small kernels like matrix multiply and quicksort. A good way to look at the results was in terms of *break even point.* In this view, the kernels reported by Auslander had to execute from about one thousand to a few tens of thousand of times before the improvement in execution time obtained by the dynamic optimization outweighed the time spent re-compiling and re-optimizing.

```
cacheResult cacheLookup(void *addr, Cache *cache) {
  dynamicRegion(cache) { /*cache is run-time constant*/
    unsigned blockSize = cache->blockSize;
    unsigned numLines = cache->numLines;
    unsigned tag =
      (unsigned) addr / (blockSize * numLines);
    unsigned line =
      ((unsigned) addr / blockSize) % numLines;
    setStructure **setArray =
      cache->lines[line]->sets;
    int assoc = cache->associativity;
    int set;
    unrolled for (set = 0; set < assoc; set++) {
      if (setArray[set]dynamic->tag == tag)
        return CacheHit;
    }
    return CacheMiss;
  } /* end of dynamicRegion */
}
```

Figure 3: Example of semi-automatic dynamic optimization for C. Reproduced from [2].

Subsequent work by Grant et al. created the DyC system [19, 18]. DyC simplified the process of identifying regions and applied more elaborate optimizations at run time. This system can handle real programs, although even the streamlined process of manually designating only run-time constants is reported to be time consuming. Their methodology allowed them to evaluate the impact of different optimizations independently, including complete loop unrolling, dynamic zero and copy propagation, dynamic reduction of strength and dynamic dead assignment elimination to name a few. Their results showed that only loop unrolling had sufficient impact to speed up real programs and in fact without loop unrolling there would have been no overall speedup at all.

6 Trace-oriented Dynamic Optimization

In the late 1990s, researchers at HP labs took a different approach to dynamic re-optimization of existing programs. Rather than focusing on optimization techniques for exploiting run-time constants, the HP Dynamo project concentrated on techniques to automatically select regions of the program to improve. The technique they invented to identify interesting areas to optimize used instruction traces to record, optimize and later dispatch hot paths through a program.

Their approach was in some ways opposite to the work described in Section

5. Rather than require programmer intervention, the HP team developed online heuristics to automatically select the regions of the program to dynamically optimize and performed only relatively modest optimizations.

Dynamo achieved good speedups on the HP workstations of the day [4, 3]. Although they do not report detailed micro-architectural performance data they suggest that better branch prediction and virtual address translation lookaside buffer (TLB) effects are the reason.

Recently Bruening describes a new version of the Dynamo approach, running on the Intel x86 architecture, which has not managed to speed up benchmarks or real world applications in as clear-cut a way [7]. Bruening does not mention this might be because modern (Pentium 4) Intel x86 implementations contain a hardware trace cache. I suggest that the speedups obtained by Dynamo on HP PA-8000 computers were in part due to better use of i-cache prefetch bandwidth. The Pentium's hardware trace cache thus skims the cream of this benefit and hence Dynamo has greater difficulty speeding up applications on this architecture.

6.1 HP Dynamo

HP Dynamo's approach is counter-intuitive [3, 4, 14]. Dynamo interprets running highly optimized binaries instead of running them directly on the hardware. As interpretation continues, a Dynamo heuristic termed SPECL (for Speculative trace selection) identifies regions of the program which are termed "traces". These home in on the loops in the program and do not consider the call graph. A potential "trace entry" point is the destination of a rearward branch in the originating code or a "trace exit" point of cached code. The interpretation of each "trace entry" point is counted. When it is executed frequently enough, it is reckoned to be "hot". Dynamo then enters a mode called "trace generation" in which native instructions are issued into a trace cache as the originating code is interpreted[7]. The resulting code fragments in the trace cache include code only for the exact path through the program that was traversed during trace generation. Trace generation ends when a "trace end" point is reached or the trace becomes too long. A trace end point is either a cycle or a backward branch. This strategy is intended to promote the linkage of traces together so that execution eventually should remain in the traces. After trace generation has completed interpretation resumes. However, when the interpreter reaches an instruction corresponding to the entry point of a trace, interpretation is suspended and the trace is dispatched.

As part of the process, the trace generator reverses the sense of conditional branches so the path along the trace sees only *not taken* conditional branches. This is very significant. It means that subsequently, when the trace is dispatched, all the frequently executed conditional branches are not taken, which is the sense that many CPU branch prediction schemes[45] assume will be taken

[7] Actually a low level intermediate representation is issued, then optimized, then written into a trace fragment.

for forward branches. This may lead to significantly lower branch misprediction delays in the program. Off-trace branches are all *taken* conditional branches and lead to “trace exit” code that arranges to send execution back to the interpreter. In addition to better branch prediction, the traces should promote better use of the instruction cache prefetch bandwidth. Since we expect that fewer conditional branches are being taken by the traces, we should also expect that the portions of instruction cache lines following the conditional branches will be used more effectively.

6.1.1 Trace cache management and “reactive flushing”

Caches, in general, hold recently used items, which means that that older, unused items are at some point removed or replaced. As a general strategy, when its trace cache becomes full, Dynamo flushes the entire cache and starts afresh. Dynamo calls this approach “reactive flushing”. The hope is that some of the (older) fragments are not part of the current working set of the program and so if all fragments are discarded the actual working set will fit into the cache. According to the technical report [3], the overhead of normal cache management becomes much higher if garbage collection or some other adaptive mechanism is used.

6.1.2 Bailing out

When the trace selection rate remains high for more than a threshold number of consecutive intervals, Dynamo gives up and “bails out”, which means that it relinquishes control and simply dispatches the program binary. When this happens, the prototype does not attempt to resume trace generation, the program is allowed to execute to completion.

6.1.3 Calls and Returns

The Hewlett-Packard PA-8000, in the spirit of its RISC architecture, does not offer complex call and return instructions. A callsite to a shared routine first branches to a small chunk of glue code written by the static linker. The glue code loads the destination of the shared code from a data location that was mapped by the static linker and initialized by the dynamic loader. An indirect branch then transfers control to that location. When glue code is encountered during trace generation it is substantially optimized in a similar spirit to conditional branches but also with the flavor of inlining. Figure 4 (a) illustrates the original extern call and Figure 4 (b) shows how it is trace generated. The indirect branch is replaced by a conditional trace exit. The call guard in the figure is in fact a conditional branch comparing the target of the indirect branch to the original destination observed during trace generation[46]. That is, instead of using the destination loaded by the loader glue code as input to an indirect branch, Dynamo uses it to check that a dispatched trace contains a copy of the instructions. As before, the conditional branch is arranged so that it is not

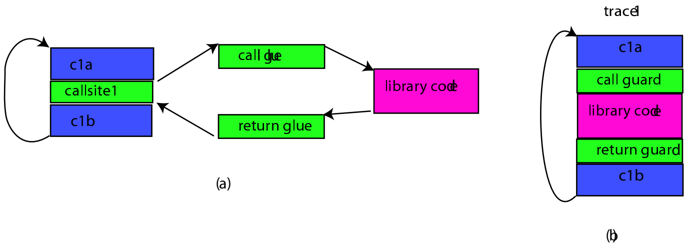


Figure 4: A simple dynamically loaded callee (a) requires an indirect branch whereas trace code (b) guards with conditional branches.

taken when control remains in the trace. The trace continues with instructions from the callee. Hence the technique straightens the glue code and replaces an expensive, taken, indirect branch with a cheaper, not-taken conditional branch, as well as inlines the callee code. Returns are handled essentially the same way.

If the destination of the shared code were to be changed by some action of the dynamic loader, the glue code replicated in the trace would load the new destination, the guard code would detect that this does not correspond to the code that was earlier inlined into the trace, and the trace would exit.

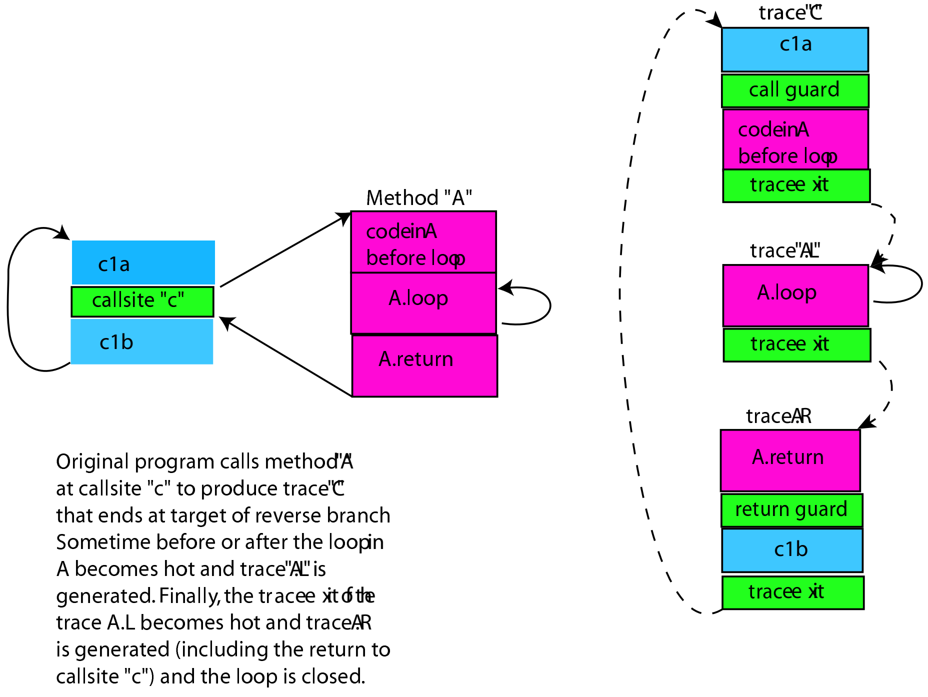
The callsite of a C++ virtual function or a regular C function pointer will also start out as an indirect call and be trace generated in a similar way. If a C++ virtual function callsite turns out to be effectively polymorphic, then the destination encountered during trace generation will be inlined into the initial trace. As overrides of the virtual method are encountered, the trace exit guarding the inlined code will fail. Eventually one of them will become hot, and a new trace will be generated from its entry point. Each time this occurs, Dynamo inserts the address of the new trace into a hash table specific to the callsite keyed by the address in the originating code. Then, when a trace exit occurs resulting from an indirect branch, Dynamo can find the destination trace by looking up the originating destination in the hash table.

In an ideal situation, the callee contains nothing that would cause a trace to end (e.g. a loop). In this case, the trace would contain both the code from the callsite and the return point and the return guard can be skipped.

6.1.4 Interaction between selection and call-return strategy

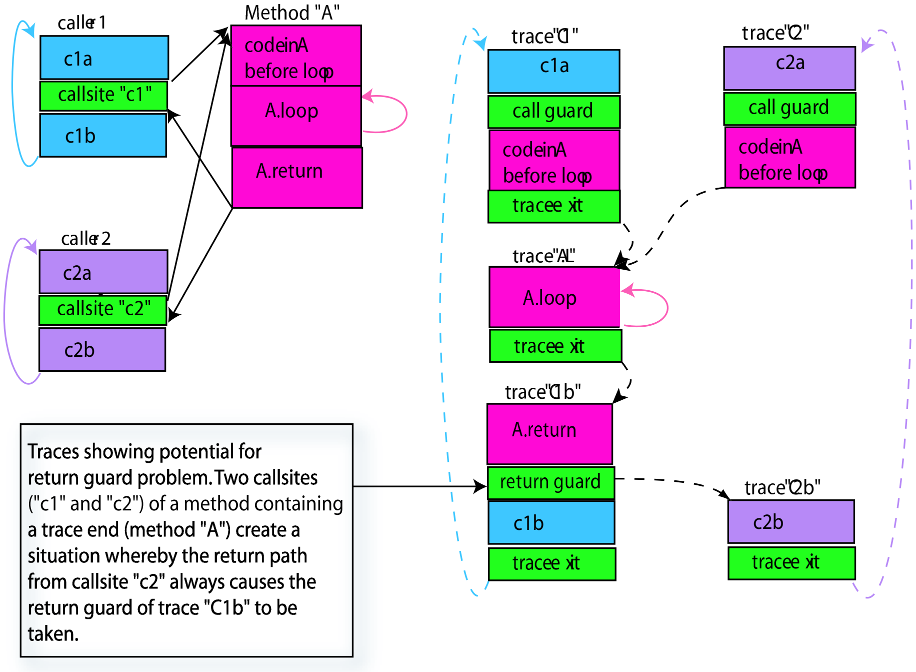
If the callee is more complicated, we may see an interesting interaction between the trace selection heuristic and the call return strategy. Especially interesting is the situation when a routine containing a "trace end" condition is called from more than one place. This scenario is quite complex, so we will discuss it in two stages. First we describe what happens when the trace generation of the callee encounters a trace end condition. This is shown in Figure 5. Then we describe the interaction between two such callsites. This is shown in Figure 6.

Suppose the callee is a commonly used routine that contains a loop. We



Original program calls method "A" at callsite "c" to produce trace "C" that ends at target of reverse branch Sometime before or after the loop in A becomes hot and trace "A.L" is generated. Finally, the trace exit of the trace A.L becomes hot and trace A.R is generated (including the return to callsite "c") and the loop is closed.

Figure 5: Callee containing trace end



Traces showing potential for return guard problem. Two callsites ("c1" and "c2") of a method containing a trace end (method "A") create a situation whereby the return path from callsite "c2" always causes the return guard of trace "C1b" to be taken.

Figure 6: Return Guard Failure

will refer to the routine as “A”. (A reasonable example might be a routine calculating the absolute value of a vector.) The target of the reverse branch closing the loop is a trace end point. Suppose the callsite of the routine is “c”, encountered during the trace generation of trace “C”. The callsite of the routine is handled as an indirect branch as described above, with the result that part of the code of routine “A” is replicated in trace “C”. In Figure 5 we labeled this portion of the routine “code in A before loop”. When trace generation reaches the target of the latch in “A”, trace “C” ends. Thus, trace generation ends having seen the call to A but not the return.

Suppose, in this case, the loop trace is already in the cache. We will refer to this trace subsequently as trace “A.L”. When Trace “C” is put into the cache it will link to trace “A.L”. A few traces may have to be generated and linked together in this manner before the return of routine “A” is trace generated. Suppose the return point occurs in trace “A.R”. For the purpose of the first example let us suppose that during this phase of execution, all calls to “A” are from the callsite “c” and hence the return takes us back to that callsite. Although, when we trace generated the return in trace A.R we observed that it returns to callsite “c” we must assume that trace “A.R” may be dispatched from other places in the program as well. The guard code that is generated handles the return like any other indirect branch, guarding that the destination address is actually to the address of callsite “c”. The return is not a trace end condition, so trace generation continues.

To recap, we seen in Figure 5 how Dynamo’s trace selection heuristic has created traces “C”, “A.L” and “A.R” which perform as intended as long as they are dispatched from places where routine “A” is called from callsite “c”. Thus, Dynamo will trace generate code that will perform best if the return is to “c” but will behave correctly when it is to some other callsite.

Next we will consider what happens when other hot callsites of “A” crop up.

6.1.5 The Return Guard problem

Here we turn our attention to what happens when other callsites to the routine“A” occur in traces. One possibility is that callsite “c” above is in a loop, so for a phase of program execution, all calls to routine A come from there. Eventually, the program continues to another loop nest where routine A is called also. This more complex scenario is shown in Figure 6. The problem is that calls to “A” other than than the callsite “c” may also lead to the return guard in trace “A.R”. The return guard there will trace exit, as the return address is not the address of callsite “c”. As a consequence, these calls to “A” will return via a lookup in the hash table backing the return guard. We will refer to this difficulty with Dynamo’s strategy as the “return guard” problem.

One interpretation of this behavior is that the inlining performed by Dynamo’s SPECL trace selection heuristic is not aggressive enough. The trace end condition, as defined, causes the return guard problem. More aggressive inlining would have attempted to replicate the body of routine A at all hot callsites and hence the call to routine “A” and the corresponding return would occur

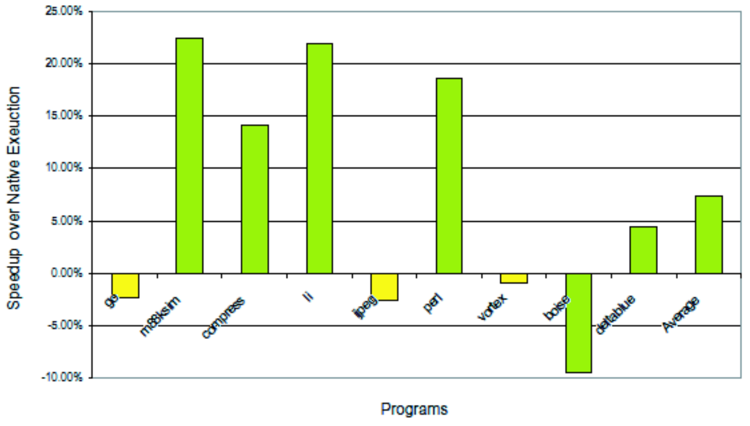


Figure 42: Dynamo performance (light/yellow bars indicate that Dynamo bailed-out)

Figure 7: Reproduction of Dynamo Figure 42 Dynamo Performance Result [3, page 81]

in the same trace. The Dynamo team would argue that small callees should be inlined by the static optimizer before the program was linked and before Dynamo was involved. Thus, the opportunities for the return guard problem to arise (hopefully) could be minimized. [8].

6.1.6 Overall Dynamo Performance

Several long running SPEC benchmarks run more quickly under Dynamo than they run natively. The average speed up is 7.37% [3, page 80]. Figure 7 breaks down the performance for each benchmark. The code straightening and inlining performed on the code (and various indirect memory hierarchy effects) by the trace generator more than compensated for the interpretation overhead and assorted trace cache housekeeping.

A large trace cache helped but was not required to achieve a speedup. Figure 9 shows how even quite small caches enabled reasonable performance. Obviously, if Dynamo's performance was close to that of the native binary (let alone better) it means the working set of traces must have fit into the trace cache (referred to as the "fragment cache" in the figure). Hence only 150K is sufficient to hold the working set of traces for these benchmarks.

Extremely interesting from our point of view is the overhead of interpretation, trace generation and trace cache management. Figure 8 shows the overhead

[8] The designers of a jit compiler do not have this luxury and so the return guard problem is important to deal with. In fact, in our preliminary work described in a separate document we will see that when SPECL is used for polymorphic java programs 1% of the bytecodes executed by Jess from its trace cache are failing return guards [48].

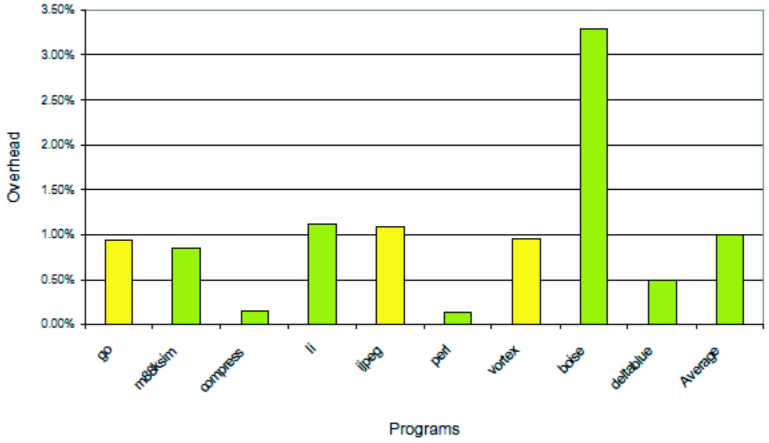


Figure 43: Dynamo overhead (light/yellow bars indicate that Dynamo bailed-out)

Figure 8: Reproduction of dynamo Figure 43. Interpretation Overhead [3, page 81]

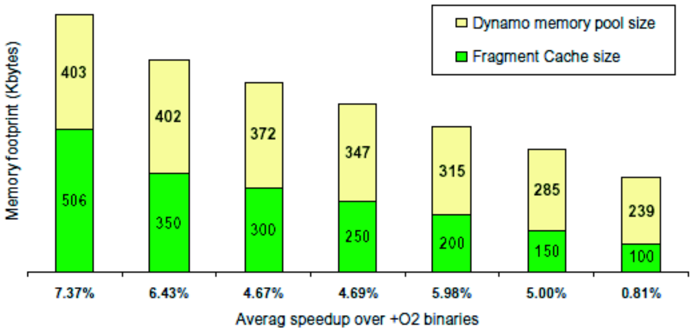


Figure 9: Reproduction of Dynamo Memory Pool Size [3, page 82]

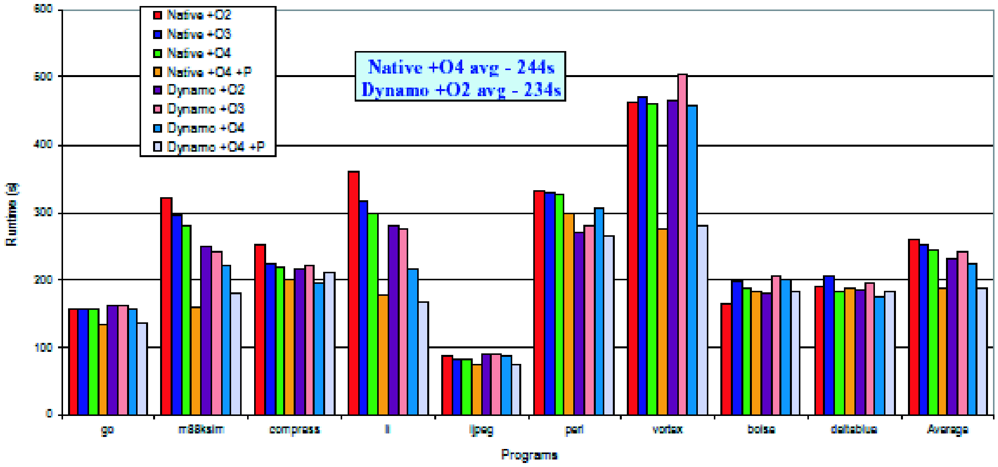


Figure 46: Performance comparison of Dynamo against native execution

Figure 10: Reproduction of Dynamo Figure Performance levels when executable has been compiled at varying levels of optimization [3, page 84]

as a fraction of runtime. It would appear that the SPECL heuristic does a very good job of identifying the hot regions of these benchmarks and linking traces together. As a consequence, Dynamo spends almost all of its time executing code from the trace cache. The benchmarks reported spend only about 1% of their time in Dynamo's interpreter; the rest of time of the time is spent running traces out of the code cache.

Dynamo found little benefit to any trace optimization other than by redundant branch removal and by the flow graph straightening as described above. Redundant load elimination, dead code elimination, code sinking beyond trace exits, and loop invariant code motion effected the speed up obtained by Dynamo very little. The efforts that were made to peephole optimize across trace boundaries seemed to have little effect.

Dynamo starts with fully optimized native code and hence does not need to worry about carrying out classical optimizations. In fact, Dynamo could even speed up executables that were compiled at the highest levels of optimization including profile driven inlining. (See Figure 10). Dynamo achieved a speed up even at optimization level 4, which in HP's parlance of the day referred to full inter-procedural optimization. It would appear that Dynamo's improvements are somewhat orthogonal to those of classical optimization.

6.1.7 Micro-architectural aspects to Dynamo Performance

The Dynamo papers do not report micro-architectural performance measurements so the reasons for the speedup are not known with certainty. The reports suggest that the speedup is caused by a combination of branch prediction and instruction Translation Look aside Buffer (TLB) effects. Part of the speedup is probably an artifact of the branch misprediction behavior of the benchmarks on the particular family of hardware it was tested on. No data is available for Dynamo other than on a HP PA-8000 running HPUX 10.20.

The Dynamo technical report [3] succinctly describes the HP PA-8000 computer.

> The PA-8000 is a four issue, dynamically scheduled processor. The machine uses a 56-entry instruction reorder queue (IRQ) from which it issues instructions. Dynamic branch prediction is used, with 256 3-bit entries in a branch history table (BHT) and a 32 entry branch target address cache (BTAC) that stores the target addresses of taken branches. The branch misprediction penalty is five cycles. Most indirect branches are not predicted at all and incur the misprediction penalty since the target address is not known until after the execute stage in the pipeline. The exception is procedure returns which are predicted using a call return register termed the Pop Latch, which effectively functions as a return address stack [Hunt 1999]. These traits characterize the PA-8000 as a highly speculative, deeply pipelined processor. Additionally, the processor has a 96 entry unified TLB and a four entry micro I-TLB located adjacent to the processor core. Our system was configured with 1 MB I- and D-caches.[3, page 85]

One theory for how Dynamo sped up these programs is that it enabled more efficient use of the branch target address cache (BTAC). Branches that are not taken don't effect the BTAC and hence executing the straightened code from the trace caches may put less pressure on BTAC than the originating code would have done. Presumably this would lead to better branch prediction for the remaining taken branches. A related micro architectural implication involving branch prediction concerns indirect branches. Dynamo's strategy is to translate indirect branches (like calls to shared libraries and virtual function dispatches) in the originating code to not taken conditional direct branches (See our Figure 4). This may avoid misprediction penalties paid by the originating code. For small originating methods Dynamo will have treated both the call and the return this way.

The detailed technical report [3] also raises the possibility that the code layout in the trace cache results in fewer TLB slots being used for instructions. Presumably the originating code branches around some basic blocks whereas code in the trace cache always falls through – hence it may take fewer instruction TLB entries to map the working set of the trace cache than were required to map the working set of the original code. No data is presented about TLB evacuations, however.

Instruction cache prefetch My studies suggest another explanation for the speedup obtained by Dynamo. While it is true that collecting frequently executing paths into traces may allow better use of BTAC resources, it is also likely to improve the use of i-cache prefetch bandwidth. I suggest that the tendency of trace exits to fall through results in more frequent use of the instructions prefetched into the i-cache. Research, such as the Software Trace Cache by Ramirez et al. [40], has shown that if the basic blocks of a program are reordered so as to maximize the sequentiality of instructions then i-cache misses are reduced. Similar reasoning by Rotenberg [41] and Patel [36] were posed to support the inclusion of trace caches in hardware. In fact, these and unrelated considerations concerning x86 instruction decoding led to Intel including a trace cache in recent models of the x86 architecture as described by Hinton [23].

6.1.8 Summary

The performance results obtained by the Dynamo project indicate several interesting facts:

- The SPECL heuristic can locate relatively small regions of an executing program in which the program spends the lion's share of its time.
- The notion of interpreting a program in order to locate its hot regions is a reasonable one. Dynamo found that at the end of the day the overhead of the interpretation was negligible.
- Trace generation is an interesting compilation technique that allows us to generate code that takes into account properties of the program that are unknown at compile, link edit or load time.
- The success of reactive flushing suggests that a sophisticated trace cache management scheme may not be required.

The Dynamo results have inspired us to investigate building a Just in Time compiler for Java bytecode based on trace generation instead of method compilation. See my research proposal [47].

6.2 Recent Dynamo related work

It probably comes as no surprise that the promising results reported by Dynamo on the PA-8000 inspired other researchers to also investigate dynamic trace optimization. Researchers have failed, so far, to obtain speedups like those obtained by Dynamo on x86 hardware, although recent efforts are closing in.

Wen-Ke Chen and others [9] built mojo, a somewhat quick and dirty system very similar to Dynamo that was aimed at large Microsoft Office applications. It performed poorly for a number of reasons. Perhaps the most important contribution made by mojo was to warn researchers that a lot of careful engineering work was required to follow Dynamo's approach.

Bruening [6] describes a new version of the Dynamo infrastructure aimed directly at commercial applications on the Intel platform. Due to the complexity of interpreting the x86 instruction set, interpretation was abandoned in favor of basic block dispatch and copying to traces. The overhead of this was shown to be acceptable. The project appeared to become somewhat mired in complexity introduced by the event-driven programming model required by Microsoft Windows. Again, their inability to speed up Microsoft Office applications is inconclusive, given the short-cuts required to manage the engineering complexity of the task.

Hazelwood and Smith [22] used Dynamo as a black box[9] to evaluate more sophisticated trace cache management schemes than Dynamo's all-or-nothing preemptive flushing approach. They found that a relatively simple least-recently-created scheme reduced cache misses by about half.

Very recently Bruening [7] describes how Dynamo has evolved into DynamoRIO, a framework for dynamic code modification. An interesting internal representation for traces is described that allows adaptive levels of detail to be supported. Several interesting dynamic optimizations are built, including redundant load removal, reduction in strength, and a technique for the dispatch of indirect branches reminiscent of Hölze's PIC. On an Intel Pentium 4 Xeon DynamoRIO outperforms the original binaries for the floating point SPEC2000 benchmarks. The integer SPEC2000 benchmarks do not fare as well: four out of twelve integer benchmarks performed more than 25% worse than binaries, produced by `gcc -O3`, running directly on the hardware. In comparing their results to the original Dynamo results on the PA-8000, they comment that the Intel Pentium Xeon (running Linux) they used for their testing had a return address predictor, whereas the PA-8000 did not. They failed to mention, however, that the Pentium 4 family includes a relatively small[10] hardware trace cache. This suggests to me that some of the benefit from trace dispatch may have already have been obtained by the underlying Intel trace cache hardware.

7 Dynamic Object-oriented optimization

A detailed survey of this very active field is beyond the scope of this paper. However, a good start can be made by describing the seminal work carried out in the mid 1990's by Dave Ungar and the Self team and then touching upon the very recent work done by IBM research and many collaborators towards the Jikes[11] Research Virtual Machine (RVM) project.

7.1 Self

Self is an ambitious, pure object-oriented, research language and run-time system. Self does not define any primitive types (`int, boolean, byte, float, double` are examples of primitive types in C++ and Java.) Omitting primitive types forces almost all work done by the Self virtual machine to require

[9] They filtered their data from Dynamo debug logs.

[10] The original revisions of the trace cache for Pentium 4 was 12K μops., which are the RISC like opcodes i86 instructions are decoded into.

[11] This project started out under the name Jalapeno and was renamed when it became an open source project.

a message send.[12] In addition, the Self system aims to support the most interactive programming environment possible. To this end, it aims to support editing and recompiling methods while a program is running with no need to restart. This requires very late binding, much like Java. The combination of the radically pure object-oriented approach and the ambitious goals regarding development environment make Self a sort of trial-by-fire for object-oriented dynamic compilation techniques.

Ungar, Chambers and Hölzle have published several papers [8, 25, 24, 26] that describe how the performance of Self was increased from more than an order of magnitude slower than compiled C to only twice as slow. A readable summary of the techniques are given by Ungar et al. in [44]. A thumbnail summary would be that effective monomorphism can be exploited by a combination of type-checking guard code (to ensure that some object's type really is known) and static inlining (to expose the guarded code to inter-procedural optimization). To give the flavor of this work we will briefly describe two specific optimizations: customization and splitting.

7.1.1 Customization

Customization is a relatively old object-oriented optimization introduced by Craig Chambers in his dissertation[8] in 1988. The general idea is that a polymorphic callsite can be turned into a static callsite (or inlined code) when the type of object on which the method is invoked is known. The approach taken by a customizing compiler is to replicate code so as to produce callsites where types are known.

Ungar et al. give a simple, convincing example in [44]. In Self, it is possible to write algorithms that can be shared by integer and floating point code. An evocative example is a method to calculate minimum. The `min` method is defined by a class called `Magnitude` in Self. All concrete number classes like `Integer` and `Float` inherit the `min` method from `Magnitude`. A customizing compiler will arrange that multiple customized definitions of `min` are compiled for `Integer` as well as `Float`. The main effect of the customization is that the polymorphic callsites of `<`[13] within the original `min` method can be replaced by a few arithmetic compare instructions[14] in each of the customized versions of integer and float `min`.

Customization can be carried out in a similar way by a Java JIT compiler. In most cases the customized code can be inlined.

7.1.2 Method Splitting

Oftentimes, customized code can be inlined only when protected by a type guard. The guard code is essentially an if-then-else construct where the "if"

[12] A virtual method invocation in Java is called a message send in Self.

[13] In Self even integer comparison requires a message send.

[14] i.e. the integer customized version of `min` can issue an arithmetic integer compare and the float customization can issue a float comparison instruction.

tests the type of an object, the "then" inlines the customized code and the "else" performs the original polymorphic method invocation of the method. Chambers [8] noted that the predicate implemented by the guard establishes the type of the invoked object for one leg of the if-then-else, but following the merge point[15], this knowledge is lost. Hence, he suggested that following code be "split" into paths for which knowledge of types is retained. This suggests that instead of allowing control flow to merge after the guard, a splitting compiler can replicate following code to preserve type knowledge.

Incautious splitting could potentially cause exponential code size expansion. This implies that the technique is one that should only be applied to relatively small regions where it is known that polymorphic dispatch is hurting performance.

7.2 Jikes Research Virtual Machine

Jalapeno, as the Jikes Research Virtual machine (RVM) was originally called, differs from most Java implementations in two principal ways. First, Jalapeno and all of its runtime support is written entirely in Java. This requires Jalapeno to be bootstrapped. Second, it includes no interpreter – all classes are quickly compiled by a fast baseline compiler when they are loaded. The dynamic optimization strategy of the RVM is to have the baseline compiler generate instrumented code with embedded counters which raise events that are used to automatically create an optimization plan and subsequent recompilation of selected methods. Until about 2000, Jalapeno publications such as [1] tended to concentrate on how these two aspects of the RVM were designed.

IBM research released the Jikes RVM sources under an open software license, presumably in the hope that it would become a popular infrastructure for researchers. This appears to have come to pass. During 2002, academic researchers published accounts of having ported many object-oriented optimization techniques to Jikes. Two examples we have singled out below are context sensitive inlining and optimistic intra-procedural optimization.

7.2.1 Context-Sensitive Inlining

Early inlining techniques depended on static heuristics to decide when a callee should be inlined. For instance, a very small callee for which the body is estimated to execute in a time similar to the time required to make the call should probably always be inlined. By the mid 1990s, the best inlining heuristics were *profile-driven,* which means that they depended on profile data from a previous training run to guide inlining decisions. Nevertheless, the heuristics were still *context-insensitive.* Profile-driven context-insensitive inlining estimates the probability of each edge in a program's call graph from the profile data and chooses to inline only those routines that appear to be commonly called. The technique is stymied by polymorphic callsites because it has no way of reacting

[15] The merge point is the point in the control flow graph where the "if" and "else" paths merge together. In a simple if-then-else construct the merge point follows the code of the else.

to the fact that the common destination of a particular callsite depends on more context than simply the routine it appears in.

Context-sensitive inlining also takes into account how a program reached a given callsite and was first described by Grove and Chambers in [20]. Their technique was an off-line profile-driven optimization. Hazlewood and Grove [21] describe how they implemented on-line context-sensitive inlining in the Jikes RVM. Obviously, the dynamic version does not have the luxury of examining profile data off-line. This complicates matters in two main ways. First, the computation required to separate profile data into contexts counts towards execution time. Second, inlining decisions must be made based only on data available so far (whereas the offline profile-driven version could avail itself of data from complete runs). Hazlewood and Groves give an Java example which is reproduced in Figure 11.

Their example illustrates how a polymorphic callsite (the call to `HashMap.get` in the `runTest` method) should be inlined differently depending on context. In both cases `Hashmap.get` polymorphically invokes `hashCode` method on its first parameter. The calls have been contrived such that the first call (commented A: in the Figure) eventually dispatches `Integer.hashValue,` whereas the second call always dispatches `MyKey.hashValue`. The point of the example is that context-insensitive inlining will conditionally inline both versions into HashMap whereas context-sensitive inlining will know what to do.

Figure 12 illustrates the difference between a context sensitive and insensitive call-graph. Call graph (a) shows the actual call graph and indicates that the call occurs in two contexts. Call graph (b) shows how a context-insensitive analysis would simplify the situation. Call graph (c) shows how context-insensitive analysis would discriminate between the contexts.

Hazlewood and Grove's implementation had a relatively small effect on the performance of most SPECjvm98 benchmarks, but this is partially because the suite has relatively few truly polymorphic callsites. One of the benchmarks which is significantly polymorphic, `jess`, sped up by 2%. On the other hand, code size shrank by 10% or more for several of the benchmarks with no slowdown.

7.2.2 Optimistic Intra-procedural Optimization

Pechtchanski and Sarkar [37] describe an ambitious optimistic scheme for optimistic dynamic intra-procedural optimization. They note that oftentimes guard code must be generated to defend against conditions that cannot happen unless a new class with very specific properties is loaded in the future. Hence, rather than pessimistically execute the guard code before it is needed, they suggest that one should optimistically omit the guard. Optimistic type assumptions are recorded in a global value graph and examined by an enhanced class loader to detect when a class that violates previously made type assumptions is loaded. This requires the JVM to support a framework which can respond to invalidation events and re-compile optimistically compiled methods whose assumptions have been found to be incorrect.

```
class HashMapTest {
    static int counter;
    public static void main(String[] args) {                 Object k1 = new MyKey(22);
        Object k2 = new Object();
        HashMap map = new HashMap();
        map.put(k1, new Integer(1));
        map.put(k2, new Integer(2));
        runTest(k1, k2, map);
    }
    public static void runTest(Object k1, Object k2, HashMap map) {
        counter += ((Integer)map.get(k1)).intValue(); //A: k1 always Integer
        counter += ((Integer)map.get(k2)).intValue(); //B: k2 always MyKey
    }
}
class MyKey {
    int key;
    MyKey(int k) { key = k; }
    public int hashCode() { return key; }
    public boolean equals(Object other) {
        return other instanceof MyKey && ((MyKey)other).key == key;
    }
}
class HashMap {
    ...
    // simplified version of HashMap.get
    public Object get(Object key) {
      int index = (key.hashCode() & 0x7FFFFFFF) % elementData.length;
      HashMapEntry entry = elementData[index];
      while (entry != null) {
        if (entry.key == key || key.equals(entry.key)) return entry.value;
        entry = entry.next;
        }
      return null;
    }
}
```

Figure 11: Context Sensitive Inlining Example reproduced from [21] Figure 1

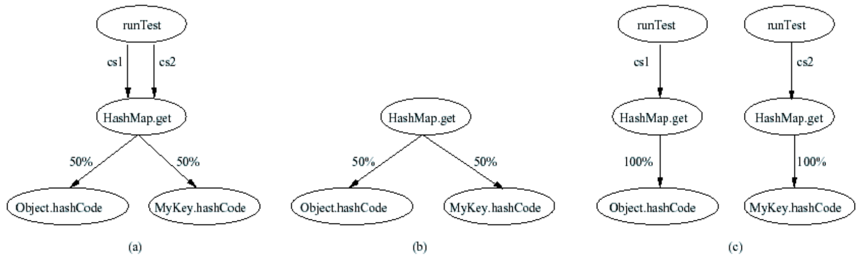


Figure 12: Call graphs derived from Java code of Figure 11. Reproduced from [21] Figure 2.

The performance of their Dynamic Optimistic Intra-procedural Type (DOIT) analysis is quite promising. A version of the Jikes RVM which includes DOIT saw an elapsed time speedup for all benchmarks except `compress`. Apart from `db` and `javac`, for which which the number of guarded interface calls decreased dramatically, the speedups were in the range of 1%. DOIT reduced the average proportion of polymorphic callsites from 39.5% to 24.4%. Similarly, the average number of polymorphic interface callsites was reduced from 96.4% to 36.2%. This suggests that the optimistic assumptions allowed by DOIT have a significant impact on code quality. It will be interesting to see whether greater speedups are realized as engineering improvements reduce the overhead of the assumption checking and de-optimization frameworks.

8 JIT as Dynamic Optimizer

The first Java JIT compilers translated methods into native instructions and improved polymorphic method dispatch by deploying techniques invented decades previously for Smalltalk. New innovations in garbage collection and thread synchronization, which are not discussed in this review, were also made. Despite all this effort, Java implementations were still slow. More aggressive optimizations, such as those described in Section 7, had to be developed to accommodate the performance challenges posed by Java's object-oriented features, particularly the polymorphic dispatch of small methods. The writers of Sun's Hotspot compiler white paper lament:

> In the Java language, most method invocations are *virtual* (potentially polymorphic), and are more frequently used than in C++. This means not only that method invocation performance is more dominant, but also that static compiler optimizations (especially global optimizations such as inlining) are much harder to perform for method invocations. Many traditional optimizations are most effective between calls, and the decreased distance between calls in the Java language can significantly reduce the effectiveness of such opti-

> mizations, since they have smaller sections of code to work with.[33, pp 17]

The result is that JIT compilers began to generate code that assumed the types of many objects. The gambles often were right, but new techniques had to be invented to recover when the bets were wrong. Again, from the Hotspot white paper:

> So the Java HotSpot VM must be able to dynamically de-optimize (and then re-optimize, if necessary) previously optimized hot spots, even while executing code for the hot spot. Without this capability, general inlining cannot be safely performed..[33, pp 19]

A further complicating factor is that current JIT compilers compile entire methods at a time. Thus, in addition to callsites whose destination are known, there are callsites whose destination is not known. The result is that heuristics must be invented that decide to re-optimize methods as previously-unknown or infrequently executed regions of already compiled methods become hot or change their behavior.

9 Summary

From the earliest Smalltalk work by Deutsch and Shiffman in 1984[12] to very recent products like Hotspot [35] we have seen dynamic compilation techniques come to the fore. Early techniques exploited relatively predictable properties of object oriented programs, like the high probability of a callsite having low effective polymorphism. In-line caching techniques generated code that was fast if the callsite behaved as expected and slow otherwise.

Today, the most extreme approaches involve optimistic dynamic techniques such as those described by Pechtchanski and Sarkar in [37]. These techniques generate code that is correct only if subtle type assumptions continue to hold. A reasonable fear is that correct, well-engineered applications will load classes in an unfortunate sequence that will trick such an optimizer into making optimistic assumptions prematurely. Since the optimistic code may be embedded in aggressively inlined methods, significant re-compilation may be required to undo the earlier optimism. Aggressive inlining, needed to reduce the cost of small methods, amplifies the cost of optimistic assumptions that turn out to be wrong.

My feeling is that the current combination of method-based compilation, aggressive inlining and optimistic dynamic optimization gambles too aggressively. Optimistic dynamic optimization would be more palatable if it were possible to repair incorrect assumptions without re-compiling entire methods and their inlined callees.

I observe that current JIT compilers use the method as the unit of granularity for both translation (from bytecode to native code) and optimization. Is this necessary? The results of the Dynamo project suggest that traces could be used to identify regions of code to translate. A trace-based infrastructure must anyway handle trace-exits and the resumption of trace-generation after the failure of an assumption. Presumably we can extend this mechanism to guard optimistic assumptions also.

The original departure point for my own research was the idea that traces, apart from their understood and potential properties, contain nothing but hot code. Thus, traces present an ideal environment for optimizing polymorphic callsites. Furthermore, a trace-based infrastructure must include a mechanism to continue from taken trace exits. My intuition is that this trace exit mechanism can be reasonably extended to include assertions of more speculative assumptions. This realization came only after reading papers such as [37] and realizing that a trace-based approach would allow the optimistic dynamic optimizer to add code following trace exits when assumptions prove false rather than re-compiling the containing methods.

Although the notion of a speculative, object-oriented trace-based JIT is intriguing, there is still a missing piece. Aggressive inlining within a method-based JIT provides a way of removing method overhead and optimization barriers from multiply nested loops. How can a trace-based approach achieve this? How can we optimize across the multiple traces that make up a loop nest? I suggest that as traces are linked into the trace cache the emerging control flow graph can be monitored. When multiply nested regions are detected the traces can be combined into a combined compilation unit (CCU) and optimized.

This topic will be discussed further in my research proposal[47].

References

[1] Bowen Alpern, Dick Attanasio, John Barton, Michael Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Ton Ngo, Mark Mergen, Vivek Sarkar, Mauricio Serrano, Janice Shepherd, Stephen Smith, VC Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeno virtual machine. In *IBM Systems Journals, Java Performance Issue*, 2000. http://www.zaleski.ca/papers/Thesis/jalapeno/jalapenoJVMibmJournal2000.pdf .

[2] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 149–159, 1996. http://citeseer.nj.nec.com/auslander96fast.html.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical report, Hewlet Packard, 1999. http://www.hpl.hp.com/techreports/1999/HPL-1999-78.html.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000. http://citeseer.nj.nec.com/bala00dynamo.html.

[5] Derek Bruening and Evelyn Duesterwald. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3*, 2000. http://www.eecs.harvard.edu/fddo/papers/108.ps.

[6] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2000. http://www.zaleski.ca/papers/Thesis/DynamicOpt/bruening01fddo4.pdf .

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *In Proceedings of the First Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, March 2003. http://www.cag.lcs.mit.edu/dynamorio/CGO03.pdf.

[8] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, 1988.

[9] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David Gillies. Mojo: A dynamic optimization system. In *Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, 2000. http://www.cs.washington.edu/homes/lerns/mojo.pdf.

[10] Standard Performance Evaluation Corporation. SPEC jvm98 benchmarks. http://www.spec.org/osg/jvm98.

[11] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling Java just in time. *IEEE Micro*, 17(3):36–43, 1997. http://ieeexplore.ieee.org/iel1/40/12908/00591653.pdf.

[12] Peter L. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 297–302, Salt Lake City, Utah, January 1984. ACM Press. http://www.zaleski.ca/papers/Thesis/Smalltalk/deutsch-schiffman-popl84.pdf .

[13] Karel Driesen. *Efficient Polymorphic Calls.* Klumer Academic Publishers, 2001.

[14] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *ACM SIGPLAN Notices*, 35(11):202–211, 2000.

[15] Paolo Faraboschi, Joseph A. Fisher, and Cliff Young. Instruction scheduling for instruction level parallel processors. In *Proceedings of the IEEE*, 2001. http://www.zaleski.ca/papers/Thesis/TraceScheduling/Farab-Fisher-YoungSchedule2001.pdf.

[16] Stephen Gilmore. Programming in standard ML '97: A tutorial introduction. http://www.dcs.ed.ac.uk/home/stg, 1997.

[17] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[18] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000. http://www.zaleski.ca/papers/Thesis/DynamicOpt/grant98dyc.pdf .

[19] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan.J. Eggers. An evaluation of staged runtime optimizations in Dyc. In *Conference on Programming Language Design and Implementation*, May 1999. http://www.cs.washington.edu/research/projects/unisw/DynComp/www/Papers/pldi99.pdf.

[20] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, November 2001. http://www.zaleski.ca/papers/Thesis/GrovesHints/toplas2118.pdf .

[21] Kim Hazelwood and David Grove. Adaptive online context-sensitive inlining. In *International Symposium on Code Generation and Optimization*, March 2002. http://www.research.ibm.com/people/d/dgrove/papers/cgo03.pdf.

[22] Kim Hazelwood and Michael D. Smith. Code cache management schemes for dynamic optimizers. In *Sixth Annual Workshop on Interaction between Compilers and Computer Architectures held in conjunction with the Eighth International Symposium on High-Performance Computer Architecture*, Feb 2002. http://www.eecs.harvard.edu/hube/publications/interact6.pdf.

[23] Glenn Hinton, Dave Sagar, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, Q1, 2001. http://www.intel.com/technology/itj/q12001.htm.

[24] Urs Hölzle. *Adaptive Optimization For Self:Reconciling High Performance With Exploratory Programming*. PhD thesis, Stanford University, 1994.

[25] Urs Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Conference on Programming Language Design and Implementation*, 1992. http://www.cs.ucsb.edu/labs/oocsb/papers/pldi92.pdf.

[26] Urs Hölzle and David Ungar. A third-generation Self implementation: Reconciling responsiveness with performance. In *Proceedings of the OOPSLA '94 conference on Object Oriented Programming Systems Languages and Applications*, 1994. http://research.sun.com/self/papers/third-generation.html.

[27] Ronald L. Johnston. The dynamic incremental compiler of apl 3000. In *Proceedings of the international conference on APL: part 1*, pages 82–87, 1979. http://doi.acm.org/10.1145/800136.804442.

[28] William N. Joy, Susan L. Graham, Charles B. Haleyà, Marshall Kirk McKusick, and Peter B. Kessler. Berkeley pascal user's manual version 3.1. http://citeseer.nj.nec.com/188577.html.

[29] Thompson K. Regular expression search algorithm. *CACM*, June 1968. http://www.zaleski.ca/papers/Thesis/DynamicOpt/regexp-thompson.pdf.

[30] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 137–148, 1996. http://www.zaleski.ca/papers/Thesis/DynamicOpt/MLlee_leone_pldi96.pdf .

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

[32] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993. http://citeseer.nj.nec.com/lowney92multiflow.html.

[33] Sun Microsystems. The Java hotspot virtual machine, v1.4.1, technical white paper. http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html, 2002.

[34] Steven S Muchnick. *Advanced Compiler Design and Construction*. Morgan Kaufman, 1997.

[35] M. Paleczny, C. Click, and C. Vick. The Java hotspot server compiler. In *2001 USENIX Java Virtual Machine Symposium*, 2001. http://www.usenix.org/events/jvm01/full_papers/paleczny/paleczny.pdf .

[36] S. Patel and S. S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, 50(6):590–608, June 2001. http://www.crhc.uiuc.edu/~steve/papers/rePLay.ps.

[37] Igor Pechtchanski and Vivek Sarkar. Dynamic optimistic interprocedural analysis: A framework and an application. In *2001 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2001. http://www.cs.nyu.edu/phd_students/pechtcha/pubs/oopsla01.pdf.

[38] Steven Pemberton and Martin C. Daniels. *Pascal Implementation, The P4 Compiler.* Ellis Horwood, 1982. http://www.cwi.nl/ steven/pascal/book/10pcode.html.

[39] Rob Pike, Bart Locanthi, and John Reiser. Hardware/software tradeoffs for bitmap graphics on the blit. *Software - Practice and Experience*, 15(2):131–151, 1985. http://citeseer.nj.nec.com/324101.html.

[40] Alex Ramirez, Josep-Lluis Larriba-Pey, Carlos Navarro, Josep Torrellas, and Mateo Valero. Software trace cache. In *International Conference on Supercomputing*, pages 119–126, 1999. http://citeseer.nj.nec.com/15361.html.

[41] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *International Symposium on Microarchitecture*, pages 24–35, 1996. http://www.zaleski.ca/papers/Thesis/hwTrace/rotenberg96trace.pdf .

[42] Bjarne Stroustrup. *The C++ Programming Language, Second Edition.* Addison Wesley, 1991.

[43] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Systems Journals, Java Performance Issue*, 39(1), February 2000. http://www.research.ibm.com/journal/sj/391/suganuma.html.

[44] David Ungar, Randall B. Smith, Craig Chambers, and Urs Hölzle. Object, message, and performance: how they coexist in Self. *IEEE-COMPUTER*, 25(10):53–64, October 1992. http://www.zaleski.ca/papers/Thesis/Self/self_ieee_comp_00161280.pdf.

[45] Hank S Warren, Jr. Instruction scheduling for the IBM RISC system/6000 processor. *IBM Systems Journals*, 34(1), Jan 1990.

[46] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996. citeseer.nj.nec.com/witchel96embra.html.

[47] Mathew Zaleski. A better way of running object-oriented bytecode? a research proposal. http://www.cs.toronto.edu/˜matz/prop.pdf, April 2003.

[48] Mathew Zaleski. Jootch, a simulation of bytecode trace selection and creation in a JVM. http://www.cs.toronto.edu/˜matz/traceCreation.pdf, April 2003.