Dynamic Analyses for Just-in-Time Compilers

M. Tech. Seminar Report

Submitted in partial fulfillment of the requirements
for the degree of

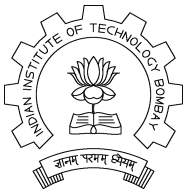
Master of Technology

by

Rohan Padhye
Roll No: 113050017

under the guidance of

Prof. Uday Khedker



Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Contents

1 Introduction **1**

1.1 Why JIT? . . . 2

1.2 Evolution of JIT compilers . . . 2

1.2.1 Origins . . . 2

1.2.2 Dynamic translators . . . 3

1.2.3 Modern JITs . . . 3

2 Dynamic dispatch **4**

2.1 Inline caching . . . 5

2.2 Customization . . . 6

2.3 Type analysis . . . 7

2.4 Message splitting . . . 8

2.5 Run-time feedback-directed specialization . . . 9

3 Engineering JIT systems **10**

3.1 Dynamic optimizations . . . 10

3.1.1 Common subexpression elimination . . . 10

3.1.2 Register allocation . . . 10

3.1.3 Exception handling . . . 11

3.2 Adaptive optimization . . . 11

4 Trace-based JIT compilation **13**

4.1 A motivating example . . . 13

4.2 How tracing works . . . 15

4.3 Optimizations for free! . . . 17

4.4 Methods vs. Loops vs. Traces . . . 19

5 Conclusion **20**

Abstract

Just-in-time compilers have been gaining popularity recently due to the increasing use of scripting languages in the Web and embedded devices, as well as the need for executing programs that are generated on-the-fly. This seminar report attempts to contrast the analyses and optimizations performed by JIT compilers as compared to traditional static compilers, as well as discover the addtional optimization opportunities present in JIT compilers due to the possiblity of gaining additional run-time information about an exeucting program before performing dynamic compilation.

Chapter 1

Introduction

There are two broad approaches for supporting execution of programs written in a high level language - *compilation* and *interpretation*.

Compilation involves translating the high level program to a low level representation which is more amneable to execution by a computing system. This can be done in two ways:

- **Static compilation** consists of a separate translation phase whose resulting binary executable can be stored and executed later, an arbitrary number of times. This has several benefits such as:

    1. The compiler can spend a significant amount of time performing static analysis and optimizations on the code, as the translation phase is distinct from execution time.
    2. The resulting binary exeuctable consists of machine code which can execute at native speeds without any run-time overheads.

- **Dynamic compilation** involves performing the translation to native machine code when the program is invoked by the user, thus interleaving compilation with execution. The benefits of dynamic compilation are discused later in this chapter.

Interpretation is performed by parsing the high level program when it is invoked and executing the required instructions one by one depending on the semantics of the high level language. The benefits of interpration are:

1. Portability, as the program can be distributed in a platform-independant manner as long as an interpreter exists for a given platform.

2. Smaller code size, as a high-level language can capture the semantics of a program in a much more compact manner (using complex language-specific features) than native machine code.

3. The ability to quickly execute programs after modification, as there is no waiting time for a separate compilation phase.

However, the overheads of dispatching the operation to perform for a given high level instruction can reduce performance as compared to native code.

Hybrid approaches that try to combine benefits of both systems include **bytecode systems** that use an intermediate platform-independent low-level representation and **just-in-time** compilers that perform the task of dynamically compiling portions of the program lazily, i.e. only when the particular region of code will benefit from compilation.

1.1 Why JIT?

Just-In-Time (JIT) compilers have been receiving attention recently due to:

1. The increasing use of scripting languages for interactive applications,
2. The need to execute programs which are dynamically generated on-the-fly (e.g. GPU applications or languages that support the `eval()` function).
3. The difficult task of meeting portability with resource constraints to run programs on embedded devices.

Dynamic/JIT compilers cannot perform all the analyses and optimiztions that are done by traditional static compilers as compilation time is a significant constraint because it adds to the overall execution time of the program (and responsiveness in the case of interactive applications). However, as this compilation occurs during execution, it is possible to use certain run-time information to perform optimizations selectively and specialized to the context in which the program will benefit the most, given it's current dynamic profile. This seminar report focuses on investigating the various dynamic analyses used in existing JIT compilers to answer three main questions: [1]

1. **What to compile?** JIT compilers can work in compile-only mode by compiling every function they come across, or selectively compile only certain portions of code that are executed frequently. An important parameter here is the unit of compilation (e.g. whole methods, traces, extended basic blocks, etc).
2. **When to compile?** Mixed-mode interpreters which selectively JIT compile "hot" code are faced with the question of how to decide when it is the right time to perform compilation. A trade-off has to be made between aggressive compilation (resulting in large overheads) and lazy approaches (possibly too much time in slow interpretation).
3. **What optimizations to perform?** Given the time constraints and shortage of information about the whole program, how can the code be optimized? Code specialization is an important dynamic optimization that uses run-time information of an exeucting program.

1.2 Evolution of JIT compilers

Although the term "just-in-time" compiler became popular in the 1990s, the technique of dynamic compilation goes back several decades. A detailed account of this history is given in [1], from which the following milestones can be noted:

1.2.1 Origins

Dynamic translation of data objects to code was first mentioned by John McCarthy while implementing the `eval()` function for LISP (1960). Thompson's pattern matching work (1968) compiled regular expressions to native code on-the-fly from arbirary strings.

Donald Knuth's observation (1971) that majority of the time is spent in minority of the code gave rise to the idea of mixed-mode systems that would generally interpret code, except for "hot spots" that were compiled, thus achieving a trade-off between code size and performance.

Hansen's Adaptive FORTRAN system (1974) used counters for each basic block that were incremented each time the block was executed. Also, on each counter increment, the system

would check whether it was profitable to optimize this block of code by comparing it with a preset threshold.

Hansen's design supported multiple levels of optimization: each time the frequency counter associated with a block of code crossed the set threshold, the code was recompiled with an additional optimization. For example, code could be initially interpreted, then compiled without optimization, then later re-compiled with constant folding, then again with common subexpression elimination, etc. This scheme limited the amount of time spent in the compiler at any given stage, while attempting to perform optimizations on code directly proportional to it's usage.

1.2.2 Dynamic translators

The Xerox PARC Smalltalk-80 implementation [2] by Deutsch and Schiffman was very much like a modern JIT compiler, in that it performed dynamic translation of v-code (bytecode for the virtual machine) to n-code (native machine instructions), the first time a procedure was called. The main motivation for this was to reduce the memory footprint and paging overhead as n-code would take about 5 times as much space as the compact v-code.

A novel aspect was to consider the compiled n-code as a *cache*, which could be flushed if the combined code size was bloating. The authors found that re-compiling the procedure the next time it was needed was often faster than the overheads of swapping pages from virtual memory. Thus, every procedure call could either result in a direct transfer of control (if it's n-code was present in cache) or would cause a call-fault, thus invoking the dynamic translator.

SELF is a dynamically typed pure object-oriented language [3], largely influenced by Smalltalk. A lot of research on dynamic optimizations was carried out by David Ungar (one of the creators of the language) and his students in the early '90s, which resulted in three distinct generations of the optimzing SELF compiler, the focus being on improving performance under dynamic dispatch. These techniques are discussed in Chapter 2.

1.2.3 Modern JITs

The Java platform was the first to introduce JIT compilation in the mainstream market. Sun Microsystems, the developer of the Java language, used much of the features from the SELF compilers to implement their Java Virtual Machine (JVM). Several other companies such as Intel and IBM had their own Java platforms. This market competition surged a lot of research into dynamic optimizations within a virtual machine, which is discussed in Chapter 3.

More recently, JIT compilation has become a buzzword owing to the rapid development efforts to increase JavaScript performance in Web browsers, which have led to wide-spread usage of tracing JITs. This approach is discussed in Chapter 4.

Chapter 2

Dynamic dispatch

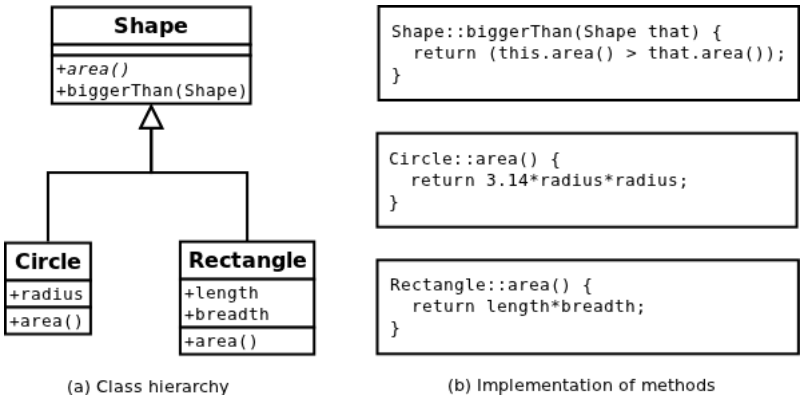


Figure 2.1: An example class hierarchy shows how both `Circle` and `Rectangle` inherit the `biggerThan()` method from `Shape`, but implement their own `area()` methods.

One feature of the object oriented paradigm is virtual methods, which lets the programmer invoke methods of an object without knowing it's exact type (although perhaps constrained by an inheritence hierarchy). In some languages such as C++ that support overloading of methods, the actual procedure to invoke may also depend on the run-time types of the arguments.

Traditionally, call sites to virtual methods invoked a routine which would look-up the address of the actual procedure to call based on the current run-time type of the receiver. This is expensive because it involves not just looking up the dynamic type of the receiver, but also traversing the inheritance tree of this type to find the implementation of the method.

As it is not possible to determine what method will be invoked at compile-time, dynamic dispatch prohibits several inter-procedural optimizations such as method inlining. Moreover, object-oriented design principles encourage abstractions and this results in an explosion of call sites, mostly calling very small methods (such as *getters* or *setters*) that greatly increase overheads and whose call sites frequently kill precious information for intra-procedural optimizations. See Figure 2.1 for example, where the `biggerThan()` method does not know which exact implementation of `area()` will be called.

Deutsch and Schiffman designed a novel technique called inline-caching that optimized for the common case: the type of the receiver at a given call site is the same more than 90% of the time [2].

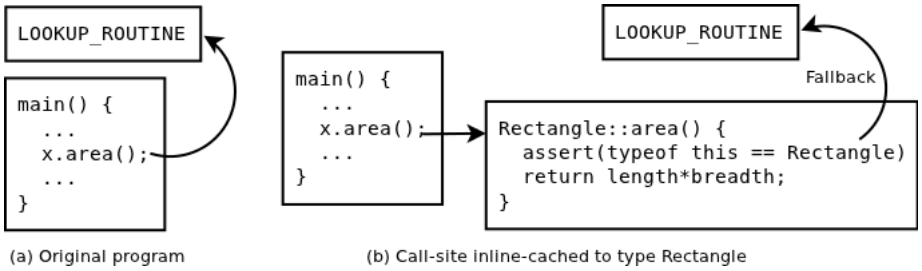


Figure 2.2: Caching the most recently looked-up type inline in the call site allows direct jumping to the predicted method, with a fallback to the look-up routine in case the type guard fails.

2.1 Inline caching

A simple optimization that can be performed using the above mentioned observation is to maintain a cache-table of recently encountered receiver types at the call-site. Although faster than the class-hierarchy lookup, this still involves accessing a table that may be located far away in memory (thus requiring fetching of data).

Inline caching is a technique in which the target method for the recently-encoutered type is embedded in the code itself, thus avoiding a seperate memory look-up but yet jumping to the required procedure quickly. It is implemented as follows:

- The first time a call-site is encountered, the dynamic look-up routine must be called to determine the type of the receiver. The result is used to call the actual procedure. The call-site is then back-patched to make a direct procedure to this method the next time.
- On a subsequent visits to this call-site, the procedure of the cached type is called directly, thus saving the expensive look-up. However, the actual type of the receiver may have changed. Hence, a few instructions are added to the start of each procedure to verify the intended type of the receiver. In case this test fails, the dynamic look-up routine is then called. The result of this look-up is back-patched on to the call site.

Performance improvements of about 10% on average were achieved using this technique. Figure 2.2 depicts inline-caching a call to the method `x.area()`, as implemented in the example of Figure 2.1.

Polymorphic inline caches

The above implementation of inline caching is essentially monomorphic, i.e. only the most recent receiver type is cached. The SELF compilers extended this idea to cache a small number of possible receiver types (e.g. 4 or 8) that are frequent at a call site. This technique is called polymorphic inline caching, and it provided up to 27% speed-up [4]. Call sites which exhibit more than the maximum possible polymorphic types are termed as megamorphic sites, whose performance approaches that of the expensive method look-up.

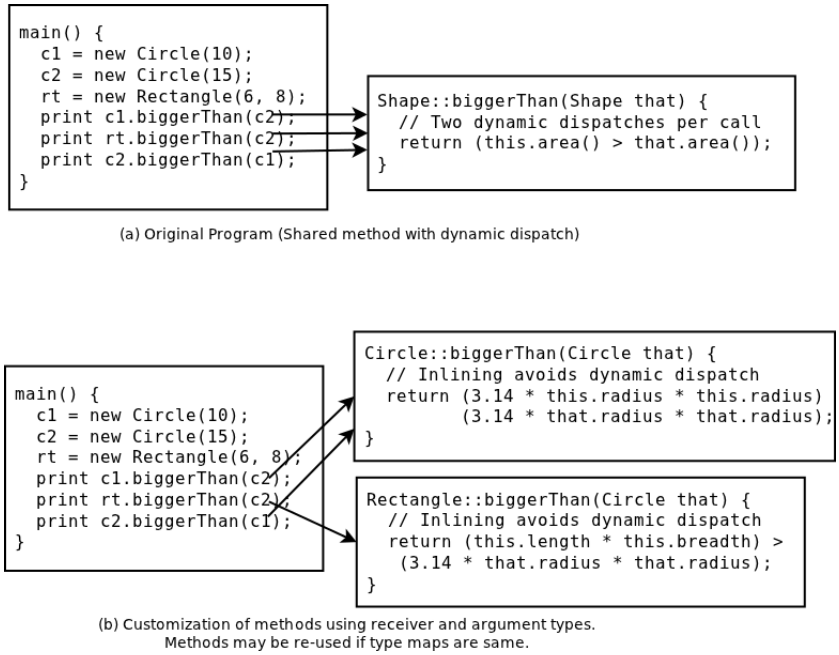


Figure 2.3: Customized compilation of shared method `biggerThan()` for different type-maps at distinct call sites.

2.2 Customization

The first generation SELF compiler introduced the idea of customized methods [5] which traded-off increased code-size for faster execution speed.

When a method was dynamically compiled, the run-time system was aware of various characteristics of the call site such as types of the receiver, arguments, etc. and thus the compiled code was customized to these types. In fact, specializing the types in the method also led to recursive customization opportunities for call-sites within that method. Aggresive inlining could be performed on the customized code. For example, Figure 2.3 shows how an inherited method `biggerThan` which makes a virtual call could be saved from the look-up and have the call inlined by customizing it into two separate versions: one for receiver type `Circle` and argument `Circle`, while the other for receiver type `Recntagle` and argument `Circle`.

Of course, this meant that more than one version of a method would be needed in the code cache as the call site need not always have the same types, however, the customized code was much faster due to inlining and branch pruning. Distinct call sites that shared the same characteristics (identified using receiver type maps) could share the same version of the customized method.

2.3 Type analysis

The second generation SELF compiler performed type analysis [6] to predict the set of possible types a variable could be bound to at a given program point.

Because the SELF compiler would perform dynamic translation of customized methods (see Section 2.2), it would know the type of the receiver (and optionally it's arguments too). Also, each local variable is initialized to a particular type (either explicitly to a constant, or implicitly to `nil`, which is the unknown type).

The data flow values in this analysis were one of the following:

1. **Value**: A constant value of a definite type (e.g. 3 or 'e').
2. **Class**: A particular class type (e.g. `Circle`), whose methods could be looked up at compile-time for inlining.
3. **Integer subrange**: A sequential range of integer values with a particular lower and upper bound. The bounds could get tighter on either side of an integer comparison branch (e.g. `if(x < y)`) or wider after the result of a primitive operation such as addition.
4. **Union**: A union of two or more types as defined above. This was generally the result of a merge node in the control flow.
5. **Difference**: A set difference of possible data flow values. This was generally the result of the `false` branch of a type test. Naturally, the `true` branch contained the class type which was being tested.
6. **Unknown**: The variable may contain any possible value. As this analysis was being performed in a dynamic (JIT) environment, the compiler did not have the whole program in view and thus could not infer types accross method calls using techniques such as bi-directional type inferencing [7]. Thus, the unknown type was assigned to results of methods that could not be inlined or folded (see below).

The type analysis allowed the compiler to perform several optimizations such as:

1. **Constant Folding**: If operands (or arguments) of an operation were of a constant value, the operation could be executed at compile-time and replaced with it's result.
2. **Method inlining**: Method calls on variables which were of a unique class type could be inlined, as the compiler could perform the dynamic virtual method lookup at compile-time.
3. **Type-test elimination**: One of the branches of a run-time type test (e.g. `if(typeof x == int)`) could be eliminated if the analysis revealed that the variable was definitely of (or not of) that particular type.
4. **Comparison branch elimination**: One of the branches of a comparison (e.g. `if(x < y)`) could be eliminated if the analysis revealed that the integer sub-ranges of the operands did not overlap.

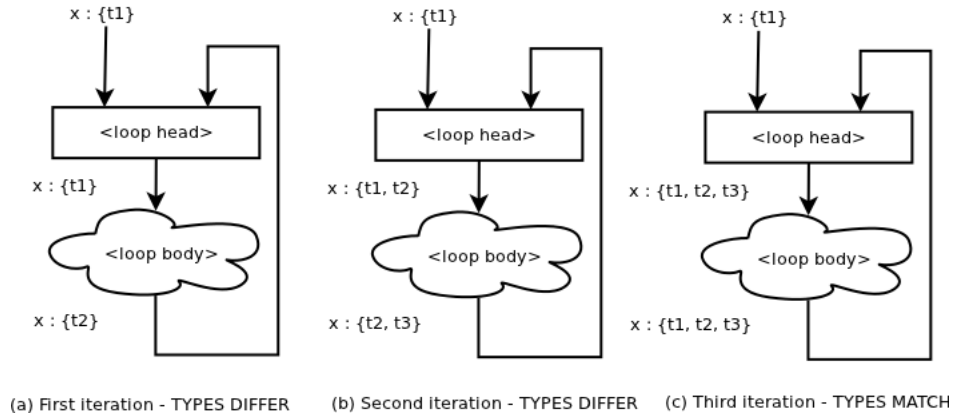


Figure 2.4: Performing three iterations of type analysis on a loop to reach a fixed-point.

Type analysis for loops

In the case of loops, the compiler is faced with a merge node, which has one predecessor whose data flow values have not yet been computed.

A problem with the use of traditional iterative data flow analyses [8] is that many variables will converge to the unknown type, because the return type of un-inlined method calls is unknown (as noted above), and methods cannot be inlined unless the receiver type is fixed.

Thus, the SELF compiler took a novel approach which the authors termed iterative type analysis [6].

The loop body was first compiled assuming the types at the head of the loop, while performing as much inlining, folding, etc. as possible. If the types of every variable remained the same at the end of the loop body, the compilation was successful. If not, the assumptions made about the types were modified to the union of the types at the head and tail of the loop, and the body was re-compiled. This iterative re-compiling approach continued till a fixed-point was reached.

The main difference between this approach of iterative type analysis and traditional data flow frameworks is the use of a different control-flow graph during each iteration of analysis, as the type assumptions at each iteration were used to perform inlining and other optimizations on the loop body in order to discover more type information.

Essentially, this technique attempted to observe the changes in the dynamic types of variables in each iteration of the loop and then assign all these types to the variables' data flow set. Figure 2.4 illustrates an example where it takes 3 iterations to reach a set of types for the variable `x` which encompass all possible iterations of the loop.

2.4 Message splitting

The second generation SELF compiler also used the results of type analysis to perform message splitting [6]. In this technique, the compiler would duplicate the call-site and all nodes backwards upto the merge point that caused dilution of type information. Figure 2.5 illustrates a split for the message `x.area()`, where the variable `x` could have been either type `Circle` or `Rectangle`. After splitting, each call-site could either be inlined or optimized otherwise, now that the compiler was sure of the type of the receiver.

Message-splitting is only performed if the number of replicated nodes is below a fixed threshold, to avoid an explosion of code if every call site were to be split.

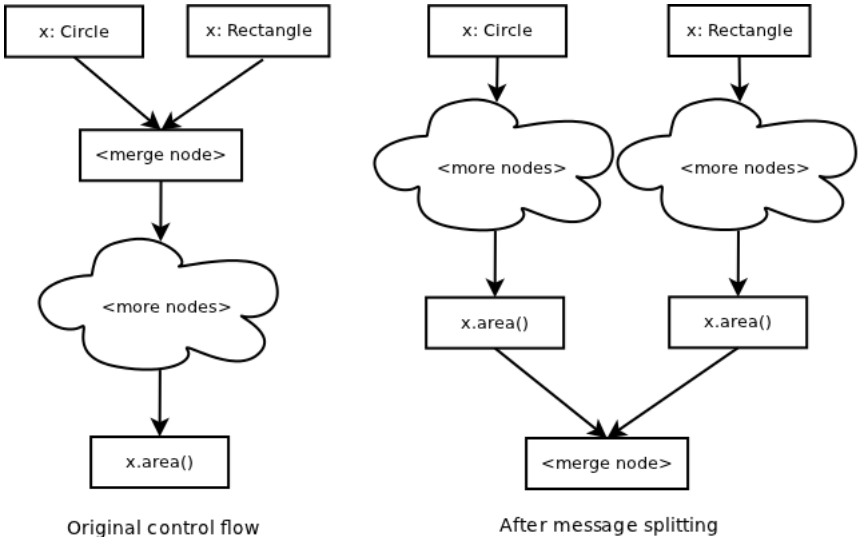


Figure 2.5: Transformation of a program's control flow under message splitting.

2.5 Run-time feedback-directed specialization

The first and second generation SELF compilers performed specializations for call sites based on their types, with the intention of optimizing for the common case. However, the assumption was that all type signatures that occur at run-time (thus triggering dynamic compilation) are equally common.

The third generation compiler instrumented the compiled code to profile the characteristics of variables and call sites at run-time to provide a feedback to the compiler, which could then optimize only for those types that actually occured frequently, rather than greedily at the first instance [9]. In fact, the instrumentation was nothing but using the information stored in the polymorphic inline caches (Section 2.1) of each call site to establish a type profile of the program.

Figure 2.6 shows an example specialization performed on a call site in which the variable `x` commonly turned out to be of type `Rectangle`. The code could be recompiled giving preferential treatment to this type by, for example, inlining the call. Other aggressive optimizations such as message-splitting (Section 2.4) could be selectively performed using the type feedback.

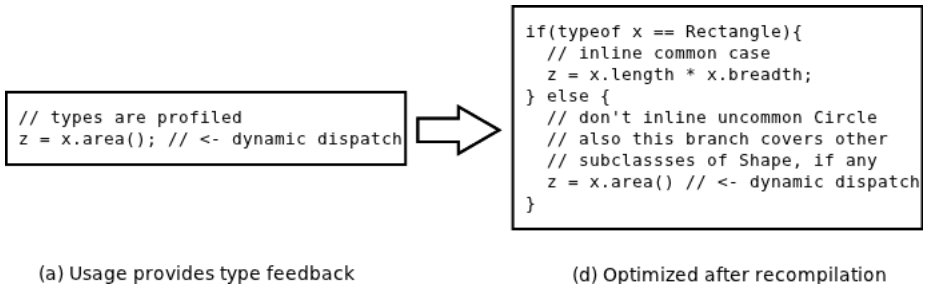


Figure 2.6: An example of a specialization opportunity using run-time type feedback. In this case, the `Rectangle` type was the most common at this particular call site.

Chapter 3

Engineering JIT systems

The popularity of Java-based software in web applets, embedded systems and the enterprise market since the late '90s could be attributed to it's *write once, run anywhere* ideology as well as the strong safety and reliability guarantees offered by it's run-time exception model.

However, these features have a drastic effect on the performance of Java programs: (1) The bytecode model forces interpretation or dynamic compilation that mixes with execution time and (2) the exception model adds further overhead due to run-time checks for pointer dereferencing, array accesses, type casts, etc.

Thus, a lot of research has been carried out to increase execution speed of Java programs while managing the constraints of run-time compilation overheads and resource limits. This section discusses various approaches taken by competing Java Virtual Machine products.

3.1 Dynamic optimizations

Although theoretically a JIT compiler could perform all the advanced optimizations performed by static compilers such as GCC, this is not practically possible due to two main reasons:

1. JIT compilation is usually performed on-demand, on a small code unit such as a method or loop body, and thus the compiler cannot optimize accross method calls to perform whole-program inter-procedural optimizations.

2. JIT compilation occurs during program execution, adding to the overall execution time as well as creating undesirable pauses in execution if performed in the same thread. Hence, the compilation time must be extremely small, which often eliminates many expensive optimizations.

3.1.1 Common subexpression elimination

For example, the IBM just-in-time compiler [10] performs common subexpression elimination using scalar replacement (for array accesses) and partial redundancy elimination (for loop-invariant code hoisting), but forgoes sophisticated global value numbering techniques because construction of SSA form is considered too expensive.

3.1.2 Register allocation

Similarly, JIT compilers seldom have time to perform global register allocation using graph-colouring approaches, and thus use fast algorithms such as linear scan allocation [11] or simple least-recently-used (LRU) type schemes with some additional heuristics [12].

The linear scan algorithm was developed specifically for dynamic compilation environments. It's complexity of $O(V \log R)$, where V is the number of live variables and R is the number of registers, causes the allocation time to be significantly less compared to graph coloring (which is NP-complete), while the penalty in performance of the compiled program due to relatively suboptimal allocation was only 10% of the original running time with graph coloring.

3.1.3 Exception handling

To make the handling of run-time exceptions more efficient, the IBM JIT [10] deferred some common exception cases such as null-pointer dereferencing, divide-by-zero error and stack overflow to the underlying operating system level exception mechanisms, where the user-space exception handler would perform the necessary transfer of control to the Java program's appropriate reaction code.

However, some exception cases such as accessing an array with an out-of-bounds index, required a bounds-check condition before each access, as this could not be caught by the operating system.

Some JIT compilers perform bounds-check elimination, an analysis in which bounds that have guaranteed to have been checked are saved to avoid re-checking for accesses with lower or equal bounds [12]. For example, a statement containing `a[i+2]` followed by `a[i]` will cause only one bounds-check, as the second statement will never cause an exception if the first one does not.

3.2 Adaptive optimization

Implementations of Smalltalk and SELF that were discussed before only had a dynamic compilation mode - a call to a method that did not have a compiled version in memory triggered a new JIT compilation for that method.

Sun Microsystems' JVM implementation popularized the mixed-mode approach, in which bytecode was initially interpreted, and counters were associated with methods and loop back-edges. If the execution count at these points exceeded a pre-set threshold, this portion of code was deemed a *hot spot*, which is likely to be executed frequently in the future as well, and hence a JIT compilation of the method was triggered [13].

Sun's HotSpot™ compiler came in two modes:

1. The **client mode** was tuned to reduce application start-up time and memory footprint, and thus used fast register allocation, lightweight threading, efficient garbage collection and a JIT compiler which performed only very fast optimizations such as constant propogation and local common subexpression elimination.

2. The **server mode** was aimed at long-running server applications and was thus optimized for peak performance rather than start-up time. Hence, more aggressive optimizations were performed by the JIT compiler such as class hierarchy analysis for inlining, global value numbering, optimal instruction selection, graph-colouring register allocation, etc.

IBM took the adaptive optimization strategy further by introducing multiple levels of optimization [14]:

1. **Mixed-mode interpreter**: This was similar to the HotSpot interpreter, which maintained counts for method invocations and loops. Methods having loops were given a boost in their counters, as they were more likely to be performance bottlenecks.

2. **Quick compiler**: The aim of this stage was to get the small- step improvement from interpretation to native code. Simple data flow analysis based optimizations such as constant propogation, common subexpression elimination, array bounds-check elimination, null pointer check elimination and dead code elimination were performed, sometimes with a limited number of iterations to reduce compilation overheads.

   This compiled code was also continuously profiled by a low-overhead periodic sampling that provided staistical insights into execution frequencies of various portions of the code, which were used by a recompilation controller to guide the decisions for further optimizations.

3. **Optimizing compiler**: If deemed necessary by the controller, methods may be recompiled a second time with many more optimizations such as aggressive full method inlining, complete DFA-based optimizations with all iterations, escape analysis of pointers for memory allocation, scalar replacement of array accesses, elimination of synchronization barriers, etc.

   The optimized code was not sampled periodically but contained some instrumentation code at the start of methods and loops that profiled not-only for execution frequency but also values of variables such as arguments of a method, which are likely to remain statisitcally constant. The instrumented code uninstalled itself after some time if there was no need for further recompilation to reduce overheads.

4. **Specialized compiler**: In case the above value-profiler did produce interesting results (e.g. the argument of `f(x)` is equal to the value "1" about 80% of the time), the method could be re-optimized with specialized branches, similar to the run-time feedback-directed optimizations used in SELF (see Section 2.5).

Chapter 4

Trace-based JIT compilation

Traditional JIT compilers have always considered the unit of compilation to be a method. However, if only a small portion of a method is contributing to the performance bottleneck of a program, it may be wasteful to compile the entire method. Tracing JITs generalize the idea of compiling hot methods by detecting frequently executing *paths*, which is known as a trace.

4.1 A motivating example

A good example that depicts the problem with method JITs can be found in [15]. Consider a sample Java program that repeatedly adds items to the end of a list implemented as a vector.

```
/* Original program */
ArrayList<Integer> list = ...;
for (int i=0; i < 1000; i++) {
    list.add(new Integer(i));
}
```

The repeated invocation of the `ArrayList.add()` method will trigger it's JIT compilation:

```
/* In class ArrayList */
void add(Object value)
{
    // First check if storage array 'elementData' has enough space
    int oldCapacity = this.elementData.length;
    if (elementCount + 1 > oldCapacity) {
        // Resize required!
        Object[] oldData = this.elementData ;
        int newCapacity = (this.capacityIncrement > 0) ?
            (oldCapacity + this.capacityIncrement) :
            (oldCapacity * 2);
        if (newCapacity < elementCount + 1) {
            newCapacity = elementCount + 1;
        }
        this.elementData = Arrays.copyOf(this.elementData, newCapacity);
    }
    this.elementData[elementCount++] = value ;
}
```

Notice that a large portion of the code in this method is dedicated to handling the uncommon case that the number of items has exceeded the capacity of the storage array, which therfore needs to be doubled.

Not only is this wasteful in compilation overhead, it also increases the code size if this method were to be inlined:

```
/* Loop with inlined method */
for (int i = 0; i < 1000; i++) {
    Integer value =  new Integer(i);
    // Inlined Method
    int oldCapacity = list.elementData.length;
    if (elementCount + 1 > oldCapacity) {
        // Resize required!
        Object[] oldData = list.elementData ;
        int newCapacity = (list.capacityIncrement > 0) ?
            (oldCapacity + list.capacityIncrement) :
            (oldCapacity * 2);
        if (newCapacity < elementCount + 1) {
            newCapacity = elementCount + 1;
        }
        list.elementData = Arrays.copyOf(list.elementData, newCapacity);
    }
    list.elementData[elementCount++] = value;

}
```

Compiling the branch for capacity doubling is completely unnecessary, as it will be required in just about $\log_2(1000) = 10$ iterations, while the most commonly executed line is the last one, which actually inserts the value into the storage array.

Not only has this caused overheads in compilation time and code size, it also prevents optimizations on the most frequently executed last line of code because of the "merge" in the control flow from the capacity changing branch.

Tracing JIT approach

A tracing JIT would have detected that hot path within the loop which is actually executed, containing only the `false` branch of the capacity test. Thus, it would JIT-compile only those statements on the hot trace, with a guard to make sure that the conditions are valid.

```
/* After tracing */
for (int i = 0; i < 1000; i++) {
    Integer value =  new Integer(i);
    int oldCapacity = list.elementData.length;
    // Guard to stay on-trace
    if (elementCount + 1 > oldCapacity) {
        /* EXIT TRACE */
    }
    list.elementData[elementCount++] = value;
}
```

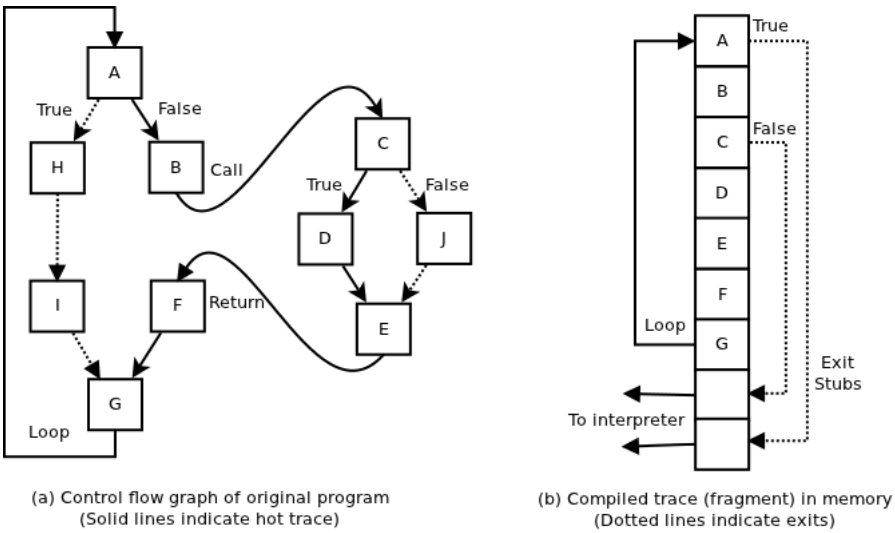


Figure 4.1: Tracing a hot path accross procedure boundaries and compiling into a linear fragment, with exit stubs in case guard tests for branches fail.

In case the guard fails (the uncommon case), the trace is exited and control never returns to this compiled code. Not only has this reduced code size and compilation time, the guarantee that that the code can never reach the last line without satisfying the guard also tells the compiler that the value of `elementCount` will always be less than the size of the storage array, `oldCapacity`, and thus the internal `ArrayIndexOutOfBounds` check can be eliminated.

4.2 How tracing works

The mechanics of tracing and management of compiled code fragments originated in Dynamo [16], a native binary translator meant for adaptive optimization of compiled executables. Here, information such as procedure boundaries, symbol tables, etc. were not available to the optimizer and hence the system had to detect hot paths by tracing the frequent execution paths of the program.

Trace detection, recording and fragment compilation

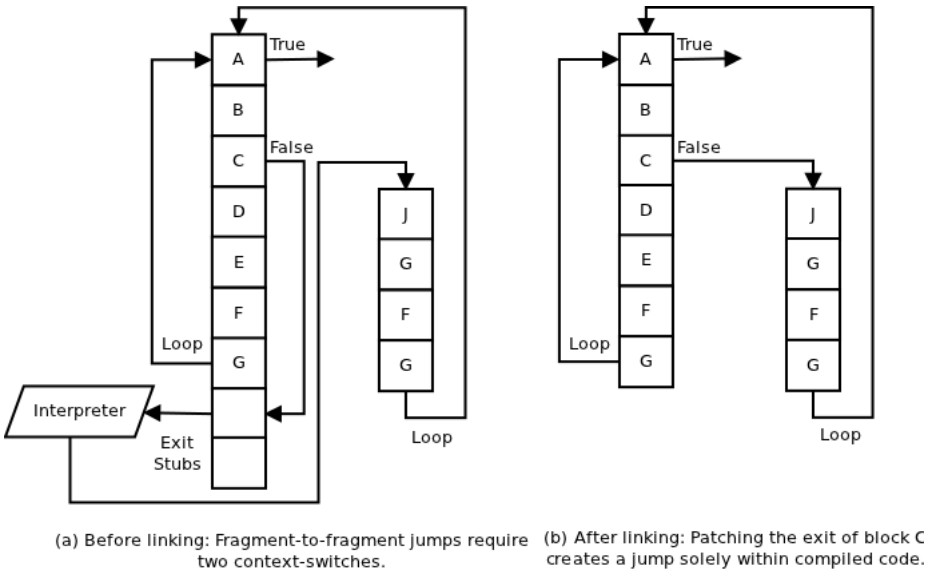
The basic steps to perform tracing are as follows:

1. Interpret instructions until a *start-of-trace* condition is met. The typical start condition is a backward jump (most likely a loop). Keep track of the program counter (PC) while interpreting.

2. If a fragment[1] exists for this PC then transfer control to the compiled code. An exit from the compiled code is also considered a likely start-of-trace, so continue from Step 1 when done.

3. If no fragment exists yet for this PC, then increment a counter associated with this PC (initialize one if it does not exist).

[1]The term "trace" refers to the hot path within the original program and "fragment" refers to the compiled version of the trace in memory.



(a) Before linking: Fragment-to-fragment jumps require two context-switches.

(b) After linking: Patching the exit of block C creates a jump solely within compiled code.

Figure 4.2: Fragment linking reduces overheads when jumping between two fragments.

4. If the counter value exceeds a threshold, then this path is "hot". Continue interpreting but start recording every executed instruction. When recording a conditional jump instruction, remember if the branch was taken or not.

5. When an *end-of-trace* condition is reached (such as another backward jump, or a jump to a PC for which a fragment exists in the code cache), stop recording. The recorded trace is then compiled. Guards are inserted at branch points to ensure that compiled code remains on-trace. The code is then optimized and an entry made for it in the fragment cache. Linking is performed if necessary (explained in Section 4.2).

For example, in Figure 4.1 (a) the back-edge G-A was taken so frequently that the counter for block A exceeded it's threshold, resulting in a start-of-trace. The path recorded was A-B-C-D-E-F-G-A. The last backward jump caused an end-of-recording, and the recorded blocks were compiled into a fragment shown in Figure 4.1 (b).

Exits and fragment linking

In case guard tests for conditional branches fail, a direct jump is made to an exit stub, which is a piece of code that compensates for any assumptions made in the optimized fragment, restores broken semantics such as the call stack (which isn't maintained in fragments due to implicit inlining), and then transfers control back to the interpreter.

Note that the next instruction after an exit from a fragment is a very good candidate to become a new trace. For example, in Figure 4.1 (a), if the branch of block C has a 50% likelihood of going either way, then exits from the compiled block C in the fragment of Figure 4.1 (b) would be so frequent as to create a new hot trace J-E-F-G-A. The recording would stop at A, because a fragment already exists for this instruction, and thus a direct jump to A's fragment in Figure 4.1 (b) would suffice. Although this causes tail-duplication (blocks F-G appear in both fragments), each tail can be optimized within it's own specialized context.

Fragments exiting to create new traces is most commonly seen in nested loops. The inner loop gets hot first, so it is traced and compiled. However, if the outer loop is run for many iterations, the exit of the inner loop is so frequent that it causes a new trace to be recorded.

Because this situation is so common, it is wasteful for the fragment to exit to the interpreter, only to learn that the next PC is also in code cache. These two context switches could be avoided if there was a fragment-to-fragment jump. This is possible by fragment linking as shown in Figure 4.2. The exit of block C is patched so that a direct jump is made to the fragment starting at J. Thus, more time is spent inside compiled code and unnecessary context switches are eliminated.

However, linking does have disadvantages [17]. It makes the removal or relocation of fragments in memory very expensive, as the changed starting addresses need to be updated in every linked patch, not just the code cache index that the interpreter maintains. Fragment removal is necessary in case memory is falling short and new fragments need to be created (i.e. a replaccement scheme). Fragment relocation may be necessary for compacting and defragmentation of the memory layout.

Linking may be done every time a new fragment is created, or on-demand, when the first time the interpreter notices a fragment-to-fragment jump. The latter requires one extra instance of context-switching but avoids wasteful linking of rarely taken branches.

4.3 Optimizations for free!

As seen in Figure 4.1, tracing can take place across procedure boundaries as well, and thus inlining is implicit.

Also notice that the compiled code fragment is a linear sequence of instructions. Thus, some more automatic benefits are good instruction scheduling (most likely branches are now in the fall-through instead of a jump, thus better pipelining) and code locality (thus better cache performance).

The most promising feature of traces is the absence of merge nodes in the interior of the fragment. This is good news for several data flow analysis based optimizations that are non-distributive [8], such as constant propogation.

Also, construction of the SSA form is trivial because there will be no ϕ-nodes except in the first block at the head of the loop [18]. Thus, SSA-based optimizations which were sometimes avoided in JIT compilers (see Section 3.1) are now easily possible.

Further, because of the guarantee that at any point within the fragment, all run-time guards for staying on-trace have been satisfied, several specialized optimizations can be performed that would otherwise be possible only inside a conditional branch. The example in Section 4.1 showed how an `ArrayIndexOutOfBounds` check could be eliminated due to tracing. These specialized optimizations trade-off the additional space requirements due to tail-duplication as mentioned in the previous section.

Most traditional optimizations such as copy propogation, common subexpression elimination, etc. can be performed on recorded traces during compilation.

Dead code elimination can be performed more aggressively on traces by exploiting partial redundancies that arise due to some statements being unncessary as long as control stays on-trace. Consider the example in Figure 4.3, where an assignment to variable `X` is made, but `X` is not live in the remaining trace. Thus the assignment is redundant, but only partially (as `X` may be live in case the guard fails). The optimization that can be performed here is the sinking of the redundant assignment to the compensation code that is executed in case of a trace-exit. This is only possible because there is a guarantee that control will not return to the middle of the trace.

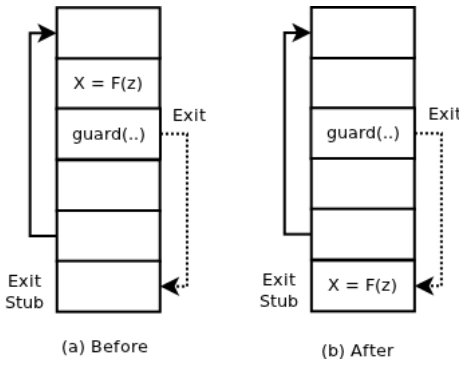


Figure 4.3: Code sinking performed to move partially redundant assignment of variable `X`, which is not live on-trace, to exit block as a compensation code in case the trace exits.

It is clear now that longer a trace, more is the potential benefit from specializations because a larger portion of code is running under a larger number of assumptions (one for each branch), which increases the optimization opportunities.

Thus, in the case of fragment linking as described in Section 4.2, the "secondary" fragment will have less aggressive optimizations as compared to the "primary" fragment (A-B-C-D-E-F in Figure 4.2).

In the case of nested loops, the inner loop will become hot first, and thus form the primary fragment, while the outer loop becomes the secondary fragment as it becomes hot later. Figure 4.4 shows how this pulls the control flow graph inside-out. This is, in fact, desirable because the inner loop would be the more critical section of code, and thus is rightfully the primary trace as compared to the outer loop [18].

In the case where this discrepancy between primary and secondary fragments is not desirable, a compiler may choose to recompile the primary trace along with the secondary trace when linking is detected, thus performing optimizations on the aggregate fragment. This can extend for arbitrary number of linking opportunities, thus forming a trace tree [19].

Unlike traditional linear fragments, trace trees may branch out. However, the absence of arbirary in-branches still provide the optimization opportunities that arise from the absence of merge nodes.

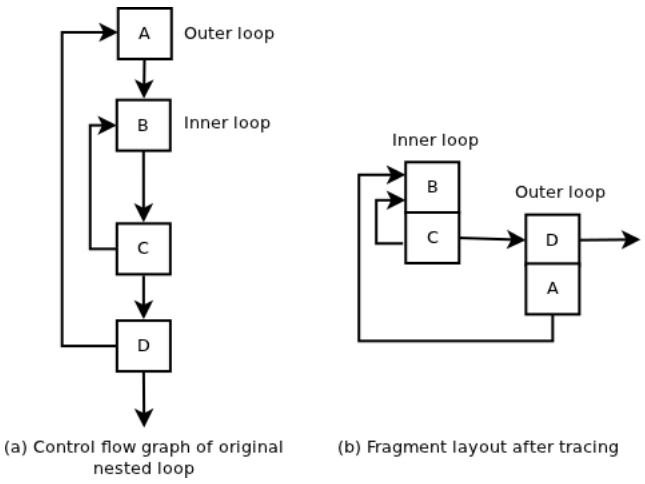


Figure 4.4: Inside-out transformation of nested loops due to tracing.

4.4 Methods vs. Loops vs. Traces

Bruening and Deusterwald performed expirements [20] on combinations of different compilation units such as methods, loops and traces each with varying parameters of their own. Their motivation was to use the well-known 90/10 rule (90% of execution time is spent in 10% of the code), in order to find an optimal JIT compilation policy which results in 90% of the execution time in compiled code, with compiled code taking only 10% of overall code size.

If only method compilation was used, compiled code size was too large, while only compiling loops resulted in small code size but missed many critical program points (only 35% of execution time in compiled regions). Tracing alone led to a 70/10 combination which was fairly decent, and performed even better for a combined strategy of "compile whole methods if small, otherwise use tracing", which reached the 90/10 goal.

Chapter 5

Conclusion

It is evident that although JIT compilers can borrow a lot of know-how from static compilers directly, there are several issues unique to dynamic compilation which must be accounted for. The chief amongst these issues is the trade-off between compilation time, which interleaves with program execution, and the quality of compiled code.

The most common approach to meeting this trade-off is to perform selective compilation. The choice of code to optimize is driven by predicting the future based on the past - frequently executing code is favoured in the hope that it will continue to remain a critical section of the program. Techniques pioneered by Sun's HotSpot VM [13] and the IBM just-in-time compiler [10] suggest that more than one tunable level of optimzation may be needed instead of looking for a one-size-fits-all solution.

Although faced with definite constraints, JIT compilers have the advantage of using run-time profile information to guide their optimization decisions. This is different from offline profile-guided optimization which may be inaccurate or even dangerous if old profile data is used which does not resemble the next execution instance [17]. Run-time information can be used to optimize dynamic dispatch by inlining object types [2, 4] or by specializing code for individual call sites [5, 9].

A combination of selective compilation and code specialization comes in the form of traces, which provide several optimization opportunies for free. Tracing JITs have gained mileage recently, popularized by their prominence in Web-based scripting environments [21, 22].

With the widespread use of multi-core systems today, the next generation JIT compilers seem to be focussing on issues in concurrent dynamic optimizations. However, as these concerns are more related to design issues rather than dynamic analyses, they have been left out of scope of this seminar report.

Bibliography

[1] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys*, vol. 35, pp. 97–113, June 2003.

[2] L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the smalltalk-80 system," in *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pp. 297–302, ACM, 1984.

[3] D. Ungar and R. B. Smith, "Self: The power of simplicity," in *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pp. 227–242, ACM, 1987.

[4] U. Hölzle, C. Chambers, and D. Ungar, "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches," in *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, (London, UK, UK), pp. 21–38, Springer-Verlag, 1991.

[5] C. Chambers and D. Ungar, "Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, pp. 146–160, ACM, 1989.

[6] C. Chambers and D. Ungar, "Iterative type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pp. 150–164, ACM, 1990.

[7] U. Khedker, D. Dhamdhere, and A. Mycroft, "Bidirectional data flow analysis for type inferencing," *Computer Languages, Systems & Structures*, vol. 29, no. 1-2, pp. 15–44, 2003.

[8] U. Khedker, A. Sanyal, and B. Karkare, *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., 1st ed., 2009.

[9] U. Hölzle and D. Ungar, "Optimizing dynamically-dispatched calls with run-time type feedback," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pp. 326–336, ACM, 1994.

[10] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Java just-in-time compiler," *IBM Systems Journal*, vol. 39, pp. 175–193, January 2000.

[11] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 21, pp. 895–913, Sept. 1999.

[12] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani, "Design, implementation, and evaluation of optimizations in a just-in-time compiler," in *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99, pp. 119–128, ACM, 1999.

[13] M. Paleczny, C. Vick, and C. Click, "The Java HotSpot^{TM} server compiler," in *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pp. 1–1, USENIX Association, 2001.

[14] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani, "A dynamic optimization framework for a Java just-in-time compiler," in *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pp. 180–195, ACM, 2001.

[15] M. Bebenita, *Trace-Based Compilation and Optimization in Meta-Circular Virtual Execution Environments*. PhD thesis, University of California, Irvine, 2011.

[16] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pp. 1–12, ACM, 2000.

[17] E. Duesterwald, "Dynamic compilation," in *The Compiler Design Handbook: Optimizations & Machine Code Generation* (Y. N. Srikant and P. Shankar, eds.), ch. 19, CRC Press, 2003.

[18] A. Gal, C. W. Probst, and M. Franz, "HotpathVM: an effective JIT compiler for resource-constrained devices," in *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pp. 144–153, ACM, 2006.

[19] A. Gal and M. Franz, "Incremental dynamic code generation with trace trees," tech. rep., University of California, Irvine, 2006.

[20] D. Bruening and E. Duesterwald, "Exploring optimal compilation unit shapes for an embedded just-in-time compiler," in *In Proceedings of the 2000 ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3*, pp. 13–20, 2000.

[21] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz, "Tracing for web 3.0: trace compilation for the next generation web applications," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pp. 71–80, ACM, 2009.

[22] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, "Trace-based just-in-time type specialization for dynamic languages," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pp. 465–478, ACM, 2009.