Dynamic Profile Driven Optimizations in a Java Virtual Machine

S T E F A N  S Ä R N E

Master’s Thesis in Computer Science (20 credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2006
Supervisors at CSC were Lars Engebretsen and Henrik Eriksson
Examiner was Stefan Arnborg

TRITA-CSC-E 2006:113
ISRN-KTH/CSC/E--06/113--SE
ISSN-1653-5715
Dynamic Profile Driven Optimizations in a Java Virtual Machine

Abstract

A Virtual Machine has several advantages over static compilers. One is the ability to get to know the application in the applications environment, since it is both compiling and executing. This thesis project shows that it is possible to use this advantage and profile the application and dynamically regenerate and optimize the code on a per method basis according to profiling data.

The results of this thesis have been implemented and tested in BEA JRockit®; a high performing JVM specialized for the server side, where long running performance gain is of interest.

Dynamiska optimeringar baserade på profildata med en virtuell javamaskin.

Sammanfattning

En virtuell maskin har flera fördelar gentemot en statisk kompilator. En av dessa är förmågan att lära känna en applikation i dess rätta miljö, eftersom den både kompilerar och exekverar. Detta exjobb visar att det är möjligt att utnyttja denna fördel genom att profilera en applikation och dynamiskt generera om och optimera koden på metodbasis baserad på de profildata som samlats.

Detta exjobb är implementerat och testat i BEA JRockit®, en JVM med höga prestanda som specialiserat sig på serversidejava, där man tjänar på långsiktiga prestandaökningar.
Thanks to

Joakim Dahlstedt, supervisor at BEA.
Lars Engebretsen, supervisor at KTH
Henrik Eriksson, supervisor at KTH
Olof Lindholm, Marcus Lagergren, questions and answers.
Josefin Hallberg, Malena Söderlund general support.
Content

Introduction ................................................................................................................................... 1
Assignment ................................................................................................................................ 1
Compiler tasks ............................................................................................................................... 2
   Input ...................................................................................................................................... 2
   Intermediate representation ................................................................................................. 2
   Register allocation ................................................................................................................. 2
   JIT / Interpreter ..................................................................................................................... 3
Code analysis ............................................................................................................................... 3
   Loop analysis ........................................................................................................................ 3
   Domination ............................................................................................................................ 4
   SSA ....................................................................................................................................... 5
Classic optimizations ................................................................................................................ 5
   Dead code elimination ........................................................................................................... 6
   Algorithm for eliminating dead code .................................................................................... 6
   Inlining .................................................................................................................................. 7
   Copy and constant propagation ............................................................................................. 8
   Arrays .................................................................................................................................... 8
   Peep hole optimizations ....................................................................................................... 9
   Register allocation ................................................................................................................. 9
   Block scheduling .................................................................................................................. 10
   Processor properties ............................................................................................................. 12
Profiling ....................................................................................................................................... 13
   What should be profiled? ...................................................................................................... 13
      Values ................................................................................................................................. 13
      Where to use value profiling? .......................................................................................... 13
      Integers ............................................................................................................................ 14
      Flow ................................................................................................................................... 15
      Objects ............................................................................................................................... 16
   Correctness in profiling ....................................................................................................... 17
Profiling method ......................................................................................................................... 18
   Profiling is not free .............................................................................................................. 18
   Invocation counters .............................................................................................................. 18
   Sampling ............................................................................................................................... 18
   The world of this thesis ....................................................................................................... 18
   When is profiling done? ...................................................................................................... 19
Introduction

A Virtual Machine (VM) is a container program for other programs. It translates and executes the program, handles the interaction with the underlying OS and hardware layer as well as provides and collects resources the program uses and needs. When a VM first translates (compiles or interprets) the code and then executes it both steps are done during the same execution of the VM. This gives the VM and in particular the compiler in the VM the opportunity to know something about the applications behavior according to the environment the application runs in.

This thesis tries to take advantage of this opportunity and evaluates whether it is possible to gather information during runtime and optimize according to this information. There are several areas in the code generation that could benefit from such information. This thesis has focused on flow analysis but also explored the possibilities of value specializations.

The thesis is divided into several parts. The first part discusses some optimizations done during code generation in general and gives an insight in what can be and is done in a static context. The second part discusses what can be profiled, how it can be done and of what use it could be. The third part summaries what this theses project did try and the result from it.

The project was carried out at BEA Systems Inc in the Java Runtime Product Group, where the subject of this thesis has been implemented and tested in JRockit, which is a high performing JVM specialized for the server side where long run performance gains are of interest.

Assignment

The objective for this project is to be able to optimize and regenerate the code in an application during runtime considering the special needs of this application. To be able to do this there are several steps to fulfill:

- To design a method level framework to collect information and use it to analyze the application code during runtime.
- To create a way to map the collected information to the intermediate representation of the code to enable optimizations.
- To optimize the code and evaluate whether the effort is worth the potential speedup. Areas of optimizations to look into are value optimizations, code flow optimizations and specializations. Focus will be set on implementing one optimization within these areas.
Compiler tasks

As stated in the introduction a Java Virtual Machine (JVM) is the layer between the written code and the platform it is executed upon. To be able to be this, it has to either interpret or as JRockit compile the code for it to be usable on the platform where it executes. This section gives a high level description of the steps from byte code to executable machine instructions and a description of some of the optimizations done in these steps. The purpose is to give an insight to the compiler and understand what tasks the compiler already does, which this project then will provide with extra info.

This first part of the thesis serves as the study part for the thesis project.

Input

A JVM is given byte code as input, which are the normal class files that java source code is compiled to. The byte code format is the standard format that java is distributed in and one of its greatest strengths is that it is platform independent. But the byte code format is not the best representation for the VM to work with. The first step is therefore to translate the byte code to an Intermediate Representation. It is this intermediate representation that will be instrumented in this thesis project.

Intermediate representation

Intermediate Representation (IR) is a way to represent code inside a compiler. There are several different structures that can be used for the higher structure of the code. Appel [2] uses a tree structure where as JRockit uses a Control Flow Graph (CFG). In the IR in JRockit, each method is represented by one CFG. A method consists of one or more blocks that are connected to each other and they can have none, one or several successors as well as none, one or several predecessors. Every block also has a list of instructions.

The IR is separated into High IR (HIR), Medium IR (MIR) and Low IR (LIR). The HIR is a high level representation, basically only byte code mapped to the internal structure. During compilation the IR is transformed, first from HIR to MIR and during this transition several transforms are made on the way, where it is appropriate. Some of them are what can be read about in the rest of the study part. When converting MIR to LIR, the representation must consider which platform it executes on, since there are different kinds of machine instructions, which the code shall be converted into. In this step, registers are allocated and for variables that need it, space is allocated on the stack. In the final LIR-representation, every operand represents a machine instruction. During the transform of LIR the blocks are placed in order. This is called block scheduling and is one of the areas where this thesis project has tried different approaches in order to optimize the performance. Finally the code is written to memory, as machine code, to be called by other methods.

Register allocation

To allocate registers is to take the step from platform independent to platform dependent code. What is done is to assign where operands fit, in registers or on the stack or both and also define the instructions for moving values from the memory to registers. Since different architectures have different amounts of (and different) registers, the general IR can no longer be maintained.

A CPU can (at least on some architectures) work with operands directly on the stack, but accessing memory is more expensive than accessing registers. Therefore it is beneficial to have the currently most frequently used operands in registers, which is something to keep in
mind for later in this thesis. To allocate registers is not only far from trivial, to do it in an optimal way is an NP hard problem (scheduling).

**JIT / Interpreter**

An Interpreter, in the java virtual machine context, would read the byte code and simulate a run, which means it calls an existing routine that executes each specific byte code command. A JIT-compiler, Just In Time-compiler, would translate the byte code, to actual machine code and let the processor execute it.

To compile the code and execute it is more work than interpreting it, but if the part of the application is run several times, a lot is gained by compiling it, since an interpreter has a rather big overhead when executing, compared to compiled code, which only has to be compiled once.

**Code analysis**

The byte code has not kept all the information that you are used to look at in normal code syntax, for example where there is a loop, or where a synchronized region is. The representation is made for a stack machine that interprets each command at a time and actual loop information is not needed then, nor where the region is, as long as you know where to take and release a lock.

Since loop information is useful to know when compiling the code and doing different kinds of optimizations, there are some analysis steps done of the CFG, to retain access to this information.

**Loop analysis**

To store the loop information, a loop nesting forest-structure is created. There are several ways of doing this, resulting in different forests. Ramalingman [8] speaks of three of them (Steensgaard, Sreedhal et al and Havlak) and how to compute them, for example using a depth first search or a breadth first search algorithm on the CFG. What is common to the different loop forests are that a loop is represented by two sets, one including all the members (the basic blocks) and one with the loop headers, which is a subset of the first set. A loop header is a block that is directly reachable by a block outside the loop.

Questions that are easy to answer when this is done, is whether a block is member and submember of a loop, which block is the header of the loop (potentially more than one), which all the members are, etc, etc.
**Domination**

**Dominate**

One block dominates another, if the program flow has to pass through the dominating block to get to the dominated block. In the example block A dominates all blocks. Block B does not dominate any block, not even block E. Block C dominates Block D.

**Domination front**

The domfront, as it is called in short, contains the blocks, which are the first reachable blocks from the block that is not dominated by the block itself. In the example, block E and F is in the domfront of C.

**Reverse dominate**

To reverse dominate is to dominate in the reversed CFG. The reversed CFG is copy, but with all the directed edges in the graph reversed. In the example block D reverse dominates C.

**Reverse domination front**

Just as in the case of reverse dominate the reversed CFG is used. Considering that, the reverse domination front is the same as the domfront. In the example, the reverse domination front of D is A.

**When is domination analysis used?**

Domination analysis is useful in several places, for example to get at good order at the block layout when dumping the IR at physical addresses and it can be used as a step to achieve SSA form. In this thesis, it is used in the divide and conquer scheduler, where it helps to divide the method in smaller independent chunks of block.

---

*Loop nesting forest, for the CFG to the left, where A is the outer loop header and B and D are loop headers for nested loops.*

*Loop nesting forest for the CFG to the right, created using Steengard’s algorithm. B and D are the outer loop headers and C and E are nested loop headers.*

---

**Example:**

```
A
  B
    C
D
  E
    F
```
SSA

SSA, which is short for Single Static Assignment, is a form of representing the code. Variables are what is assigned in the title and the point is that every variable may only be assigned once. If it is reassigned, a new variable is created in the transformation from the earlier representation to SSA-form.

Normal version:  
$ a = 1 + 2; 
 a = a + 3; 
 a = 4 + 5; 
 a = a + 6; $

SSA version:  
$ a_1 = 1 + 2 
 a_2 = a_1 + 3 
 a_3 = 4 + 5 
 a_4 = a_3 + 6 $

When the code flow is separated and a variable is assigned in one or both of the paths, the value will depend on which path was selected. But since a variable can only be assigned once, at one place, a new variable is created at each assignment and where the paths meet, a join instruction assigns a new variable the value depending on the others using what is called the phi operator.

Normal version:  
$ \text{if (cond1)} 
 a = 1 + 2; 
 \text{else} 
 a = 4 + 5; $

SSA version:  
$ \text{if (cond1)} 
 a_1 = 1 + 2 
 \text{else} 
 a_2 = 4 + 5 
 a_3 = \phi(a_1, a_2) $

The SSA-form can be used in many of the optimizations described later in the thesis. It is useful because:

- It is easier and quicker to do data flow analysis, since each variable has one and only one definition.
- The resource usage for a representation in SSA-form is linear, compared to an alternative representation, the def-use chain, which can have a quadratic resource usage.
- Easy to use for other analysis, like the CFG and a domination analysis.
- Unrelated uses of the same variable are discovered.

How to do the transform is described by Cytron [6], by Morgan [1] and by Appel [2]. For a normal program, this transform can be done in (almost) linear time.

Classic optimizations

There are several kinds of static optimizations that can be done only by looking at the code. Of course they are done during runtime, since JRockit is a JIT-compiler, but the techniques are the same as those used in a static compiler, for example gcc*

The compiler techniques described below are applied several times in the compiling of a method, until nothing more can be improved. The reason is that optimizations enable opportunities for other optimizations, which in return enable more opportunities, also for those already used.

In an example which uses the techniques that will be described, method a inlines method b. Since method a pass a constant argument to b, the constant propagation enables the dead code eliminator to discover that the if-statement in b only can have one out come and then removes dead code. This means that the inline engine can afford to inline another method c called from b.

* GCC stands for Gnu C Compiler, a compiler used on Unix and Linux for compiling c-code.
Dead code elimination

There are two kinds of dead code, but the term is used for both of them.

The first one is for unreachable code; code that never is executed. For example the false branch where it is known that the flow always takes the true branch. In the example below, the method bar() can never be reached.

```java
boolean foo = true;
if (!foo)
    bar();
```

The call to the method bar would lie in a block by itself. By finding this dead code, the whole block can be removed.

The other case is when the code is executed, but the result of the execution is never used, hence nobody cares about it.

```java
int method(int a) {
    int b = 2 * a;
    return a;
}
```

In the example above the multiplication on line 2 is reached, but b is never used.\(^5\) Since it is unnecessary to do the multiplication it is classified as dead code and can be removed.

What makes this a bit tricky in a JVM is that it is responsible for reporting when things go bad, for example checking for null and bounce checks.

```java
int method(int [] arr, int b) {
    int i;
    for (i = 0; i < b; i++)
        arr[i] = i * i;
    return i;
}
```

In the example above, arr is never used by anything else. Can then all the operations on arr be removed? No, because if arr would be null, then the method should throw a NullPointerException instead of returning i. If b would be larger than or equal to the length of arr it should throw an ArrayIndexOutOfBoundsException.

This is not impossible to solve but it is an example that shows a complexity that you might not think of at first glance.

Algorithm for eliminating dead code

An algorithm for finding\(^5\) dead code is presented by Cytron [6]. It assumes that the code has been converted to SSA-form and that a so called Control Dependence Graph (CDG) has been calculated.

The basic steps are

1. Mark all prelive instructions as live. A prelive instruction is an instruction that has or at least can have side effects. Calls are example of prelive instructions, since they do

\(^5\) Of course even the comparison and the conditional jump can be removed, but that is an optimization called dynamic dispatching or constant propagation.

\(^6\) There are a couple of glitches in the algorithm, but there are workarounds for them as well. For example the reversed domination tree is not computable with an infinite loop without exits.
things out of control before they return. Other examples are stores into instance or class variables.

2. Work from the prelive instruction and mark all dependent instructions as live; that is for all source variables in a prelive instruction, all instruction that define it, should be marked as live. And of course recursively, for all instructions marked as live, mark their dependencies. Also, for all live instructions, the last instruction in all blocks, in their reversed dominance frontier, should be marked as live. This is because for all live instructions, we have to be able to get there. The last branch was made at the last instruction in the reverse dominance frontier (usually a compare).

3. Eliminate all instructions that are not marked as alive.

Inlining

To inline a method is to insert the code of the inlined method in the calling method, instead of creating a call to it. By doing this, you gain several things. You need not perform the call, and save all callee-save registers. You might also want to be able to do some extra optimizations with variables, another better optimized register allocation.

This is a classical example of a method that is good to inline:

```java
int max(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}
```

Other examples are get- or set-methods, which are very common in Java as an object oriented language.

Several kinds of inlining in a JVM will be discussed in the following sections.

Fixed inlining

The simplest kind of inlining is Fixed Inlining. It is performed on non-virtual calls, i.e. when we know exactly to which method the call is made. Methods declared static or final in Java, or these declared in a class declared final, are typical targets for this kind of inlining. There are also some other occasions when you can do a fixed inlining, even if the call is actually a virtual call. When you really know which type an object is, you can make a direct call. (This is an example of a situation where you can use SSA to easily prove such a thing, if the object is created inside the method.) Another situation is if you only have one implementation of the method, which is spoken of in the next passage.

Inlining with SSIA and pre-existence

Single Subclass Implementation Assumption (SSIA) means that we assume that a virtual method have only one implementation. That is, in the currently loaded class hierarchy, only one class has a definition of this method. The method can not be overridden in neither any sub- nor any super class.

By checking in the class hierarchy we can know that for the moment, there is only one definition. Then we can inline it. But at any time in the future a subclass to this can be loaded which overrides this one. Then we need to reevaluate the method and remove the inlining.

Will there be a problem when someone is executing the inlined method and we need to reevaluate it, because something is loaded, so the single implementation is no longer true? No, since those executing will keep running the old version and if the object existed before reevaluation, it must be of the type that had or should use the SSIA. Or must it? One easy way of knowing that it was of pre-evaluation type, is if it is given as a parameter. In other cases, for example when it is given as a return or is an instance variable, it may not be provable, or even true, that the given object is of pre-existence.
Thus, beside the SSIA requirement, the object needs (provable) preexistence, before the call to the method. To use this, there has to be some control structure that has a reference to all methods that need to be re-evaluated when the method is overridden.

**Inlining using class comparison**

In the case where this is a single static assignment, but the pre-existence condition is not met, we have one more option. We can keep the original version of the method and inline it as well, and before entering the inlining part, check whether this is the right class. If it is not, use the original virtual call, to get the correct version.

**Copy and constant propagation**

Copy Propagation is to replace a variable with another variable. Constant propagation is to replace a variable with a constant instead. When it is provable that a variable has a specific value, it is (most often) better to replace it. It is a way to open up for more powerful optimizations, like dead code elimination but as itself also to shorten the live range and save space in the registers by removing some variables. A simple example:

```java
int b = a;
if (b == a)
    foo();
else
    bar();
```

Here you can replace b by the variable a, and since a == a, the evaluation need not take place and the complete else block can be removed. Of course it is not common that code is written in this way, but when inlining methods, it quite often occurs.

Using the SSA-form is a good way to find the spots where this is doable.

Closely related to Constant Propagation is Constant Folding, where you replace an expression at compile time, with the constant it evaluates to (or a simpler expression). In the example below, the only thing that would happen on execution is that a is assigned the value 10.

```java
int a = 1 + 2 + 3 + 4;
```

**Arrays**

A JVM is responsible for checking the bounds of an array, when all loads and stores are done to it. This is not anything a programmer can choose, since it is implicit in the Java language, but for a programmer many mistakes that in an unchecked world would have resulted in nonsense values, destruction of other values and complete crashes results in an exception which points to where in the code this happens.

The direct effect for the code generator in the JVM is that all loads and stores in arrays need a comparison, before they can be performed. As an indirect effect, this also limits the possibilities for rescheduling operations, since the Java virtual machine specification requires exact exception handling.

However, when it is provable that the array index will be within the limits of the arrays, the check can be removed and this will save the number of instructions executed. Here are some examples:
Compiler tasks

Bodik et al. [7] speak of one way of doing analysis for removing bound checks, using an extended version of SSA, called phi-SSA, where phi is a function that calculates the ranges of a variable. Quin et al. [13] uses a similar approach but with one major difference that makes it better. The phi-function is removed and instead of only assigning every variable once, a VCG, Value Control Graph, is calculated, where every variable has a special range at any given point of the method.

**Bopee hole optimizations**

A global analysis is sometimes too expensive. When it comes to a method, it might be too expensive to try to look at all the operations when trying to do an optimized op-scheduling. The solution is then to look at only a part of the method a time, use a “peep hole”. There are several other areas this technique can be applied on.

The point is to limit the world and do a complete optimization there. Limiting the world is done by only be allowed to look at a small area, though the peep hole.

In the case of the method, this is often not a bad idea, since most things that are used at the same time tend to be close to each other. But it is not enough to optimize one area and move on to the next. Overlapping peep holes is a technique often used; where when one area is optimized the latter part of it is part of the next peep hole. In this way, things are not missed because of bad layout of the peep holes.

**Register allocation**

Register allocation was earlier briefly described and as being a complicated problem. There is much to gain by doing it well and even more can be lost when it is badly done. This thesis only touches at the subject and tries one thing, but it is worth a few more words.

Accessing a register is much faster than accessing a location in the memory. Therefore you want to diminish the number of loads and stores (moves of values from and to memory). By doing this, as many values as possible and the most frequently used ones are kept in registers when executing the method.
When a method is called, as many as possible of the in-parameters are stored in the dedicated registers, just as the return value is stored in a register. The live range of each variable is calculated. Based on the live range info a variables value is kept while it is alive, either in register or on stack. In an ideal world all variables would be in registers, but the number of registers is limited. Having an infinite number of registers is not a solution either (the hardware cost disregarded), since it must be stored when switching thread of execution, or for some other reason doing a context switch.

Distributing the registers in a good way is the hardest part and there are several ways to do this in linear time and reach a fairly ok result. But to reach optimal result, the most common way is to use an algorithm based on graph coloring. This is described by Briggs et al. [9] in more detail.

**Block scheduling**

The block scheduling area is what most of this thesis is about. However, our focus is profile driven scheduling, whereas this section will be about static scheduling.

When scheduling the blocks of a method, the goal is to place blocks that lead to each other in a line. But since the CFG can, and most of the time is, more complex than one straight line, there has to be jump-instructions inserted. The block scheduling tries to insert as few jumps as possible. There are diamond structures, loops and more complex nets and of course, combinations of them, for example a diamond inside a loop.

Examples:

```
Diamond
A
  B
  |
  C
D

Diamond look alike
A
  B
  |
  C
  |
  D
E

Loop
A
  C
  |
  D
B
```

**Block compaction**

Where a block only has one successor and the successor only has one predecessor there is actually no reason to have two separate blocks. (See the right branch of the diamond-look-alike) The obvious and correct solution is to compact, to make one block of them.

**Turning a loop**

When scheduling blocks, the most intuitive thing would probably be to just lay them in the order they appear. However, sometimes it can be profitable to turn a loop.
In the second example, there will be an extra jump to enter the loop and an extra jump to exit the loop. But what is gained, is that there will be no unconditional jump when the end of the loop is reached. This jump is instead taken in the operation “jic”, but in the jic-case, it can both test and perform the jump at the same time, compared to the unturned version where the comparison is made at one time and the jump at another. Since most of the time will be spent in the loop, block B and C, you gain by minimizing the work in the loop.

However, if the loop is a multi-exit-loop, there is nothing to gain by turning it. Then it is better to have a straight flow in the entrance and exit of the loop instead and keep the straight forward layout.

**Invert conditions**

The basic block scheduler tries to put on of the successors to a conditional block immediately after the conditional block. If this is achieved, it is good to check if the condition should be inverted, to lower the number of instructions and jumps made. The block scheduled as immediate successor shall be the false one in the condition executed. In the example below, to the right this is the case, whereas in the example to the left, it is not.

To invert a condition is no problem. The conditional jump jne (jump not equal) is inverted to jeq (jump equal). The jgt (jump greater than) is inverted to jle (jump less or equal) and so on.

If the condition is the wrong one, the code will have two jump instructions in the end of the block. A first conditional one, just passing the second unconditional one to the block scheduled
elsewhere. To invert a condition will not only make the number of instructions less, but it will also create fewer bubbles in the CPU pipeline, as described in the next section.

**Processor properties**

**Branch prediction**

A modern processor has something called an instruction pipeline. This enables the processor to handle several instructions at the same time; before the processor has completed the work of one instruction it has already begun working on the next. Those instructions are all in the processor pipeline. As an example, the Pentium 4 pipeline length is 20 steps (simplified).

To be able to fill up the pipeline, the processor predicts if a conditional branch will be taken or not. It will of course never complete an instruction that is dependent on the outcome of an earlier one, until the earlier one is completed and the outcome is known. But by starting to process in advance; if the instructions shall be used much is gained. If not, nothing is lost by trying. When the prediction is wrong and the processor has to start over, it gets what is called a bubble in the pipeline, several cycles where nothing is done in most of the pipeline, because execution has to start over. To have as few bubbles as possible, the prediction has to be fairly good.

If the processor reaches a conditional branch (branch is a short jump) to a lower address, a back branch, it predicts the branch to be taken, with the assumption that it is the end of a loop. If it is a forward branch, it predicts it not to be taken and starts reading from the instruction behind. By helping the processor when scheduling there are much to gain and this is why block scheduling and turning conditions are important.

**Branch profiling in the processor**

The processor manufacturers are aware of how important branch prediction is and in processor of today (for example the Pentium 4) profiling is done during execution. Branches taken are stored for a while and the outcome of a conditional branch is predicted to be the same as last time. The profiling is done in hardware and has no negative impact on execution speed. It is also the case that this profiling is fairly good, although it is not perfected (which is fortunate for this thesis).

Read more about this in the reference literature [10].
Profiling

By profiling an application during runtime, there is knowledge to be gained about how the application behaves. With the profiling data it is possible to draw conclusions that there is no information about when compiling in a static context. Integer values and other variables can be dependent of different sorts of inputs or being set to far off too be able to do the analysis and know anything about them, thus we have no knowledge of them. Since input is unknown information at static compile time, the flow through a method will be unknown, when (and since) it is dependent of the values.

What should be profiled?

Everything is a good answer. But when the amount of profiling data grows larger, the cost of doing the profile increases. It is ok to accept a hit when profiling in the beginning, but there is no use in profiling, if there are not over all performance gains.

In this thesis I divided the potential targets for profiling into three different areas. To some extend they interfere, and by having knowledge of one of them, you have possible knowledge in the others. I divided them into Values, Objects and Flow.

Values

The hypothesis when profiling values is that a variable often has the same value, it is quasi invariant. If this is the case it is detectable by profiling. It can be worth to profile and store the values, or at least the most common ones for a variable, when it has some good properties. By knowing a variable is quasi-invariant, a specialized version of the code can be made where the variable is used. If a specialized version is added, a check that guards it is added as well. If the special condition is not fulfilled, a generic version will be used. If there shall be any point in doing a specialization, the specialized version has to be reasonably faster than the original one. Potentially the specialized version is inlined and information propagated while the generic version is still called, to not bloat more than necessary.

Muth et al. [3], bring up an example of where this is a good approach. One of the benchmarks in SPECcpu95 calls a general (and fairly big) method for doing one operation in 80% of the cases. The complete method is bad to inline, while the one instruction used in 80% of the cases is a very good inline candidate.

Where to use value profiling?

The hard part in value profiling is not to do the profiling, but to identify the spots where it might be useful to do it. One example, the one mentioned above, is when you can replace a call by a fairly small set of operations. In this case it is a function that copies a blob of the memory from one place to another and in 80% of the cases the move was only one byte to copy (specified as an in-argument). So method parameters are a good target for profiling. If these are quasi invariant, it is easy and a direct benefit to create a specialized version. Easiest are boolean and integers but also objects, for example if null is frequent, are possible and potential to profile. The profiling should be done at the call sight, where an inlined version can be generated.

Another candidate for value profiling are switches, where it would be good to compare for the most common value first, to reduce the number of comparisons, or in a compact case, not having to do a lookup in a call table.

A third case, which is more unlikely to occur, is when doing multiplications by the value 0 and making all subsequent calculations to the variable meaningless.
Constant propagation - Specialization

A more general way of using value profiling is propagation of constants that are not constant. If a value is frequent, it is possible to perform a check to verify the frequent value and then run a specialized version, where it is treated as a constant. For this to be useful, there has to be something more to it than just exchanging the variable. Not much is gained by just loading a constant from memory, it could both optimize as well as reduce efficiency but in either way an extra check is introduced, that has to be made up for. There is also the issue of code bloat when having a duplication of the code made.

Value prediction

Loading a variable from memory can take several cpu-cycles. By knowing in advance the value of the variable, the load can be predicted and the value prefetched or even read into memory, and work can continue immediately.

But when the prediction is wrong, and there has to be a check for this, some backtracking has to be done or rather some error eliminating code introduced. The complexity of a program increases quickly when inserting recovery code.

Integers

With value profiling, the goal is to find most frequent values. Keeping track of all values is too much waste of memory and disregarding that, it would still not be interesting. If there is a wide spectra of equally used values, not one or two more frequent, there is nothing to gain.

Therefore, when profiling values, using Top-N-Values-tables (TNV-tables) described by Calder [5] is the way to go. TNV-tables are used for keeping track of how many times the top-n-most frequent values occur.

Every variable to profile has its own TNV-table. When a value is inserted in a TNV-table, a pair is inserted, the value, and a counter that says how frequent it is. If the value already exists in the table, the counter will only be incremented. If the table is full, the value will be disregarded.

This way there is a potential issue that values may be missed. If it is not one of the n first values, it can not make it in the table. The solution to this is to empty the less frequent part of the table and allow new values to enter a couple of times during the profile.

Method used

In this thesis the TNV-tables were used for profiling the values. The n-value was chosen to 8 and 4 positions was kept steady. According to Ball and Larus [4] this is more than enough.

A verification of this claim showed that it was true. What was discovered when testing these values is that using a TNV-table of size 3 with only one stable value gave quite good result. There was a real increase in the correctness of the result when increasing the table size to 5, but hardly any further increase when increasing the precision to 10. These measurements were done using SPECjvm98 and SPECjbb2000.

A stable part of size four in the TNV-tables are according to these measurements rather luxurious, but in this thesis, the goal is to examine whether it is possible to gain anything, not to find the exact limit where the gain is the highest, compared to the job done.

The unstable part, also this of size four, was emptied about ten times during profile.

Groups, rather than values

Another approach of profiling might be to not record the actual value but rather profile it into a group, whether it is 0, small, large, less than 0 or odd/even, depending on the situation. One typical example to this would be a variable determining how many turns to take in a loop and then know if it is worth heavily optimizing this or not, regarding unrolling and register pressure.
This is not evaluated in the implementation part of this thesis.

**Floating point values**

Profiling floating point values would be done in the exact same way as integers, but it is harder to see a real situation where it is useful, a situation where the profiling will lead to any usable optimization and how to get around the uncertainty in precision from dealing with floating numbers. This speaks for grouping the values for profiling but it is still hard to see the usefulness.

**Flow**

To profile the flow in a method, the goal is to see what way the application takes, the so-called hot path. This can be done in two ways, either a counter is inserted in the beginning of all blocks, or a counter is inserted on all edges, by basically inserting a new block in between only containing the counter.

A lot of optimizations can be done if the flow is known.

- Block rescheduling is one of them. Put blocks that are likely to follow after each other and create a “free way” for the application through the method for most cases.
- Hot spots can be recognized in the method. Places in the method where the application spends most of the time. Given this knowledge, a choice can be made to optimize for these spots, give those spots most importance in the block scheduling and make sure registers are optimally allocated at the hot spot.
- Cold spots, parts of the method that hardly ever is used, can be disregarded of the optimizer, even moved away far from the method. The gain of moving them away would be to compact the used code on a less memory space. This is beneficial since more code can fit in the cache, which results in less cache misses and ITLB (Instruction Table Look aside Buffer) penalties, which both slow performance.
- Instructions can be moved around aggressively, even cross conditional block boundaries. Even if this means that recovery code has to be inserted in the cold path to clean up, it can still with high certainty help gain performance.
- Super blocks can be created. A super block is a sequence of blocks concatenated together. It has to be entered at the first position; however, it is possible that the super block is exited at different spots. This means some code duplication – called tail duplication, since it is adding copies of blocks to the “tail” of the instruction sequence – will be done but it enables not only a straighter flow for the hot path, but also makes it easier to move instructions without interfering.

**Method used**

In this thesis edges in the CFG are profiled, simply because that is the value wanted from the profile information. The other alternative is to profile in each basic block. But it does not matter which alternative that is used, since it is still possible to calculate the sums for the edges and vice versa.

As mentioned, the way of doing it is inserting a new block on every edge and redirect the CFG to pass through the inserted block. And in every block an increment is inserted. Since the data should be reachable from the compiler in the JVM, it is preferable to have access to it from native code. Because of this, the increment operation was no ordinary java operation, but an increment of a value stored in the native memory. This is possible since the space is allocated at code generation and known by the compiler. The data will not take up space on the java heap and will not be touched by the garbage collector.

Since this is a JVM, exceptions have to be accounted for. These are handled separately, not with normal edges in the CFG, but with handler blocks instead. But since they are rare, there is no use in optimizing the code for it and they will be disregarded as profile candidates. In either
case, throwing an exception most of the time reroutes through the JVM and does not stay in the method and it is not clear what can be gained by profiling it from a scheduling point view.

**Hot paths**

To find the hot path as above, every single branch possibility is considered, but on case by case basis. This will miss if there is any interesting correlation between two different path-choices. Let us say, purely hypothetically, that the method consists of small diamonds and in every path separation there is an equal probability of whether to take one choice or the other. But looking at the bigger perspective, you could see that the flow was more like two slalom skiers, crossing each others tracks. And by actually knowing the first choice, the others will follow. Not proved, but likely to be taken. By knowing this, specialized versions for each case can be created.

In order to know this, there has to be a profiling that considers the relations between the different spots in the code.

**Method used**

This was implemented and tested, using an integer, with one bit for every interesting spot. In the end of the method a value is retrieved that was profiled as an ordinary value. This is an interesting idea; however does seem to turn out to be useful. In the applications examined, no method was found that imitates this behavior.

**Objects**

The third thing to profile is objects. What might be interesting to know about them? Where are candidates for optimizations? As for values, all given objects can be profiled, including the this-object, which is a candidate.

This is set as a third thing to profile but it could be argued that it is a sub class of value profiling. The reason breaking it out is that there are several questions that can be asked for objects that not apply on regular values.

**Thread local**

If the compiler is certain that an object is thread local, there would be no need to check a lock or consider volatile. A thread local object is an object that is only reachable from one thread.

But to know this demands a pretty heavy analysis and it is important that you are certain here. Since profiling can not prove anything for certain, it could be used to provide a hint about whether this object reference often is thread local. You can say that it probably is and worth to try to prove it.

**Locking**

It would be interesting to know how often an object is locked, when some thread is trying to lock on it, even recursively and how many different threads that tries to lock it.

By knowing this, it would be easier to decide, which kind of lock should be used, a simpler spin lock or a fairer but more expensive monitor implementation.

By also knowing if there is more than one thread that tries to take the lock, another locking mechanism could be used, a locking mechanism that by default doesn’t release the lock to be able to acquire it again for free.

**Values/attributes in objects**

It is of course possible to profile values in objects, just as it is possible to profile parameters to methods or other (normal) variables. Those are treated in the value profile section.
Is Null

Whether an object is null most of the time, can be more interesting to know. If this often occurs, it could be valuable to insert a check that shortcuts the normal route via interrupts from the OS when the address 0 is referenced.

Virtual calls

Virtual calls were discussed earlier in the block “Fixed Inlining”. It was concluded that they are more expensive than non virtual calls but in some special cases it was possible to treat it as a static call.

A virtual call is done in several steps. The first is to lookup what class this object is an instance of and then lookup what implementation of the method is the correct to execute.

Given that in most cases the class is the same, a class comparison can be made and then an inlined version or a static call can be made.

Therefore it would be interesting to profile what class the object is an instance of and then lessen the lookup to be a quick pointer comparison in the most frequent case and combine it with adding the method as an inlining candidate as a specialization.

Correctness in profiling

During this thesis project, no synchronization was made when gathering data. This means that in some cases the profiling data may not be completely correct, if for example two threads are updating the same counters at the same time. On the other hand, the loss of the correctness is not very expensive given the alternative and to support this, it has to be said that it is only profiling data, which shall give a hint of how the application behaves at this given time and assuming that the application will continue to behave in a similar way. But the assumption is only that the application will continue in a similar way, not in the exact same as way. So even if given slightly incorrect profile data, nothing is really lost. The data is used in a defensive way and only strong indications are acted upon, because only then is it useful.

Potential errors

When optimizing from profile data, no conclusion that leads to an error in the application can be drawn, because no certain conclusion can be drawn. Nothing can be proven for all cases using the data, which means that it always has to handle all the cases. Worst case is that the data is misleading and the potential optimization will instead decrease the performance, by additional checks, code or poor scheduling.

The profile data can be misleading for two reasons. Either it is not a good profile of how the application behaves, because the data was gathered at a misleading time or it is not a good profile because there are errors when gathering the data which are assumed not to matter. The first case is something that should be fixed by profiling long enough. How long that is, is not a subject for this thesis; here the data is representative. The second case can be due to many things. Below is discussed what dangerous assumptions this project did and why they are considered acceptable.

*Potential profiling data errors in this thesis*

It is assumed that all gathering cannot be synchronized, because this would have a too large impact on the application. How bad is this?

When profiling flow, nothing serious can happen, since the profiling is only incrementing counters. In the situation where two threads increment at the same time, the worst case will only be a loss of one of the increments.
With value profiling, the situation is a little more complicated. The same thing as with flow profiling is true for value profiling regarding the counters. It is a bit more severe if two threads at the same time try to reorganize the TNV-table and switch two values. In the worst case, the same value will be at both of these positions. This will loose one of the values in the table and you have to know if this matters or not, if the loss of the data is ok or if it should be disregarded. It is easy to detect if such a thing has happened since one value will be at two positions. It is also good to keep in mind that most of the time when value profiling, the goal is to find a very frequent value, and if two values are competing to be the most frequent value, then it probably is not interesting anyway.

### Profiling method

#### Profiling is not free
To insert instructions, collect and store the data and then analyze and use it are all things that cost, to different extents. That is why it only is interesting when the benefit potentially is high. JRockit is a server JVM and can work with the assumption that it will run during longer periods of time, meaning doing some extra work once is often worth it in the end, which is an environment where profiling is well suited.

It is a fact that most of the execution in a normal program takes place in a smaller set of the code. There is a benefit in finding these spots, the hot spots, and put the extra effort in making the code better on them.

#### Invocation counters
One way of finding the methods that will benefit most from being more optimized is to insert invocation counters. This means to insert a counter on every method and record every invocation of the method. A hot spot detector, a background thread which reads these counters, will be used and it will look for more frequently used methods and optimize those. This is exactly the same way as flow profile data is gathered and the drawback is that it is expensive. Since this is expensive, you only want to gather profile data where it is needed the most. This is expensive because when the counters are updated from different CPU’s it will frequently destroy the caches.

#### Sampling
Another way to find the hot spots in the application is to sample it. Then the hot spot detector thread instead works by looking at the executing threads and book keep where the stopped thread is executing right now. After finding the same method a number of times, it is classified as a hot spot. The benefit of this approach is that you find the methods that are executed the most, not invoked the largest number of times. Using sampling, it is true that every recording costs more (sampling a thread, compared to incrementing an invocation counter), but the factor between the two values and how many recordings are needed is in the area of 100 000, which means that sampling is the way to go.

#### The world of this thesis
JRockit uses the hot spot sampling described in previous section. When it finds a hot method it optimizes it. When optimizing the method, JRockit generates a new version of it and while doing so, the same and more static analysis are done to the code as when the method was first compiled and a more aggressive inlining takes place.

In this thesis, profiling counters are inserted in the optimized code. This means that an extra pass is needed for each optimized method, a pass when the profiling is done and it generates the
code a third time according to the profile data. To handle it this way introduces some extra work, but the primary interest for this thesis is not to collect the data in the best way, but to show that it can be gathered and used to generate better code.

**When is profiling done?**

In this thesis, a method is fully profiled when it is invoked 100 000 times. This works in most cases but misses some, typically large top or close to top level methods where a great amount of time is spent even though they are not invoked very often, so the invocation counter does not justify the importance of them and they will keep running with profile data gathering all through the run. This is handled here in that after a certain amount of time, all methods that are being profiled are also optimized whether they are profiled enough or not.

This is of course not the way you want to decide if data gathering is complete. It should rather be a combination of sampling and invocations and invocation at the hottest branch in the method.

**Lower the cost of profiling**

Even if the primary target for this thesis is not to optimize the profiling, this section speaks about what can be done to lower the cost for it.

**Minimize counters**

It is described here that all edges are profiled using a counter on every edge. This is not necessary, just as Ball and Larus describes [4], edges can be identified that can be calculated from others and the counters can then be removed. A very simple example is if you have a block with one edge in and two edges out. Then you will not have to have counters on all three, because the third can easily be calculated from the others. In the same example, it is of course preferable to remove the in-counter, since every pass through the block never will be registered more than once.

**Report less frequently**

Instead of reporting to the global data structures on every increment, the counters can be treated as local variables inside the method and not until method exit be written to the global structure. This will lessen the writes to the shared memory and the caches will not be as frequently emptied.

Writing a summary in the end will also give the opportunity to add a more intelligent check if the method is enough profiled, regarding how much is done in the method in addition to the number of times it is executed.

**Sample in hardware**

DPGO (Dynamic Profile Guided Optimizations) is something that Intel provides with their modern CPUs. Beside that the name indicates it will solve the entire problem, it is more about support for profiling done in hardware. Other CPU-vendors also supply similar counters to different extents. They give the means for doing sampling based profiling with very little overhead. You request to listen to an event and a frequency and it will respond to you with an interrupt after the requested number of times this event has occurred. Examples of events are: executed instruction; miss predicted branch or cache misses. There are several benefits to this model, you will not have to instrument your code in any way and the cost for profiling (sampling based) is very low (an interrupt and the few instructions executed at the interrupt) and the accuracy is very good since the sampling frequency is high.
The drawbacks are that this is only limited supported by OSes today and the information is only presented to the OS, not to an application in user space, which has to register and only receive events. It is also not possible to do a value profiling with DPGO.

There is more on this topic in Staffan Friberg’s master thesis Dynamic Profile Guided Optimization in a VEE on IA-64 [16].
Optimizations implemented

This section describes the optimizations implemented from the profile data gathered.

Block scheduling according to flow

Goal for scheduler
Minimize the numbers of performed jumps. This is done by removing the jumps from the most common path. This is of course not possible in the case of a loop, but in the case of a diamond, this can be done all the way. The less frequently used block will be moved and placed completely out of order. This means that in the less common case, an extra jump is introduced. A normal scheduler would avoid having the double jump.

Example:

![Diagram of ordinary diamond, scheduled normally, and scheduled using flow prof]

Ordinary diamond  Scheduled normally  Scheduled using flow prof

Why not trivial?
1. Loops. Loops cannot be scheduled without at least one jump somewhere. Since loops often are executed more than once, to schedule what is inside the loop well is more important compared to what is outside it.
2. Break and break-label is a construction that does not follow the standard pattern. They are no different from other jumps in the MIR, but with break label constructions there are several exits and exit blocks out of loops.

Several different approaches are implemented to do the scheduling.

Total search
Scheduling is an NP-hard problem. To find the perfect schedule will then mean a total search of all the combinations $O(n!)$, where $n$ is the number of blocks. This is not possible under normal conditions, since the number of blocks often is quite large due to method inlining. But a total search is the naive way to solve the problem and that is the first attempt.
To know if one solution was better than another, a score function was implemented giving points to a schedule. As expected it quite soon was obvious that this was not usable for real, not even as reference. Around 20 blocks was the limit for what could be scheduled, which is the classic break for NP hard problems, while a method with several hundred blocks was not uncommon.

**Divide and conquer**

If the total search is not possible, the next approach is to break it down into pieces and part by part schedule it perfectly. The hot path of all the parts would be chained together and the others scheduled according to what was possible. Using the information from the domination analysis, it was divided according to the principle that all blocks dominated by one chosen block form a group and a group may only have one exit. The last demand, since they were to be scheduled together. But even with these grouping many methods couldn’t be broken down into pieces possible to schedule.

**Simple greedy**

A heuristic was then needed since the optimal approach was not possible to use and this was the simplest of them.

1. Start at the first block.
2. When there is more than one successor to the block, schedule the most likely as the next (of the ones that is not scheduled). Continue with that block.
3. If there are no successors available to schedule, pick an unscheduled block and apply the simple logic in 2.
4. Turn loops.

1 to 3 gives pretty good result and is a clean algorithm, but as mentioned earlier, it is beneficial to turn a loop under some circumstances. By adding loop turning as a post step to the heuristic the complexity increased some but also the result.

**Greedy**

There are cases that cannot be handled in the best way by the simple-greedy heuristic; cases where the main flow slowly decreases and by just following it, another greater flow is built up elsewhere. To handle such a case, both the flow from a block, and the flow to the next block was considered. It resulted in a greedy algorithm that in step one connects all the blocks that are obvious should be connected, where a blocks most common successor has itself as the most common predecessor.

1. Connect all blocks that choose each other, where the predecessor’s first choice of successor is the successor who as a first choice for predecessor choose the predecessor.
2. Form meta blocks of the chains and all lonely blocks.
3. Schedule the meta blocks with the simple greedy algorithm.

Again loops have to be treated as a special case or else they would be linked in circles.

**Edge greedy**

This heuristic looks at the flow on all edges and start by linking the edge with the largest flow first and continues as long as it is possible to link the blocks together, again with loops as a special case. This heuristic is described by Pettis and Hansen [14] and was implemented as a pure profile data driven scheduling mechanism. It was written for two reasons, to see the result with this approach but also to be able to use the result as a sanity check of the greedy algorithms and the profile data gathered. Since it worked from a different angle, it should be correct if the different approaches yield the same result.
Block duplication

Sometimes it is not easily decidable which the best branch choice is, both are likely to happen. One way is to not trust the profile data in those cases and schedule as without any info. Another is to create duplicates of the block behind both of them, so none of them will have to have the extra jump that is introduced for the extra block. With this approach, there is a risk of code bloat since copies are made, so this way is only interesting when the code to copy is in the hot part of the method, which can be looked up in the profile data. As long as no jump is needed on a normal case, the duplication continues, for example a complete loop.

Invert loops

To invert a loop, as is mentioned in several of the heuristics above, is to move the loophead to the end of the loop, just as in the static optimizing case. The difference is that when the profile data is available, it is possible to ask it, if it is beneficial to turn or not. Just as in the static case, it is not useful to turn a loop in the multi exit loop case.

The score function

In the naive approach to schedule the blocks, a score function was used to compare two schedules which is the best of them. They were given scores according to:

Cost:
- A predicted jump or a predicted fall through costs 1.
- A non predicted jump costs 10.
- A scheduled block without condition or jump costs 0.

Prediction:
- A backwards branch is predicted to be taken.
- A forward branch is predicted not to be taken.

An edge cost is the cost for the edge times the number of times it is taken.
The total score is the sum of all the edge costs.
The lower score a method schedule get, the better.

Value profiling

The hard thing with value profiling is to identify where it can be useful. In this thesis project, the call to arraycopy is identified as a potential spot to profile.

Specialized version of System.arraycopy

The arraycopy method is both common and demanding in Java.

```java
static void System.arraycopy(Object src, int srcPos,
Object dest, int destPos, int length)
```

Several different versions of this method is today identified and used internally, for example which type of array it is and if it is proven that this copy is safe or not, regarding bounds check.

The best candidate for value profiling and specialization is the length of the array to copy. Other candidates could be the offset to copy from but did not seem as interesting and was not implemented.

What type of array to copy is quite limited, since it is either a primitive type, or an object, which is just a reference and not the object itself. This means it will always be about small elements that shall be copied, but it is often clear what type it is at the call sight.
In this thesis, a specialized version created from the profile data was inlined and unrolled to be able to handle two copy instructions at the same time, just as Morgan [2] suggests.

**Register pressure**

When registers are allocated in a static context some simple assumptions are made to help the register allocator to choose the right variables for the registers. For example is a loop considered to have precedence over what is outside with a factor X. Given that profile data is available, it is possible to feed the register allocator with profiled data for how important a loop is, compared to the surrounding.

**Potential optimizations**

A lot more optimizations can be done using the given profile data. Some of them are discussed below.

**Decision of inlining**

The decisions for whether to inline a method or not is today based on the size of the candidate. The potential dangers and the reasons for not inlining are twofold, a too large method means more work to do a good job when optimizing and if too much is inlined the code bloat problem overhead with destroyed caches might be larger than the gains from skipping the call is. Every inlining means that a copy of the method is done. That means that inlining most of all is a tradeoff between faster code without calls and potential optimizations compared to memory usage and potential increase of page faults.

By knowing how often a method call is made, the decision of whether to inline or not has better potential of being the right one, which Sugama [12] has proven.

**Loop unrolling**

The same decision as in the inlining case, but with some different parameters has to be made when you want to unroll a loop. When unrolling a loop, there are several things to gain. There will be lesser branches. With some proving, there might be fewer comparisons made and there are also potentials for gaining speed by doing a different operation scheduling inside the loop, see arraycopy in this thesis for example.

And for the same reasons as in the inlining case there are dangers of enlarging the code.

An example where loop unrolling is used and very useful is the highly optimized GIMP-C-library [17]. At first glance, maybe 2, 4 or even 8 times of unrolling a loop can seem reasonable. It turned out that for this specific package, unrolling 80 times at some occasions was the optimal way to go. This decision was however made after a lot of profiling and can not serve as an example on how all loops should be handled.

**Load prediction**

Loads take time. Given that you know the value with great certainty it can be useful to start executing and just verify in the end that the load actually was the value expected. But as with all value profiling, given the slight potential gain, this has to be a real good candidate for this.

**Speculative execution**

The ia64 processor is a RISC processor and has the ability to execute several instructions in different units in the processor at the same time. On such a processor the operation scheduling is more important than for example on the ia32 processor and the possibility to slip
instructions outside of the block for the hot path is attractive. This is especially true since it runs in-order-execution. In-order-execution means that the instructions will be executed in exactly the order they are presented. This architecture demands more of the compiler to get it really efficient.

Profiling data could help to say where the hot path is, and where it is hot, so the compiler can let operations slip and accept the cost for recovery code in the non hot branches and know it will benefit from it.
Results

The different optimizations are tested on two different test suites, a set of micro benchmarks and SPECjvm98.

The different algorithms describe earlier are named:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>No profiling used.</td>
</tr>
<tr>
<td>Prof</td>
<td>Both flow profiling and specialized array copy used. The basic block scheduler named Greedy is used.</td>
</tr>
<tr>
<td>Prof BBP</td>
<td>As Prof, with the addition that the register allocator is fed with the true basic block pressure.</td>
</tr>
<tr>
<td>Prof CD</td>
<td>As Prof, with the addition that code duplication is allowed.</td>
</tr>
<tr>
<td>ACopy</td>
<td>Only specialized ArrayCopy is used.</td>
</tr>
<tr>
<td>Best</td>
<td>Best result of the different scheduling algorithms and different choices from above.</td>
</tr>
</tbody>
</table>

They are tested on two different CPU’s. The first one is an AMD Athlon and the second one is an Intel P4.

Micro benchmarks

This is a set of small tests written as a proof of concept targeting the different optimizations. For each part, time measured is how long it takes to execute the function a certain number of times.

Diamond

A big diamond structure is run through a large number of times, each time, the same route is chosen. This is the ideal world for the scheduler to place the basic blocks in order and put the cold block out of the way.

Loops

The benchmark is a function with nested loops that are run several times. In each loop, integers are incremented, to add some work in the loop, so they are not optimized away. This benchmark verifies that the scheduler turns the loops as it should to not pay a penalty in each iteration for a smooth entrance to the loop.

LoopD

The benchmark consists of a function with loops, similar to Loops. But as work to be done inside the loop, a diamond similar to the Diamond benchmark is inserted. This is more realistic than previous benchmarks, since it combines constructions and will also serve as an example where it is useful to schedule blocks outside the hot path.

CopyArrays

In this benchmark, several array copies are made. They will have different lengths, but for each place in the code, it will always be of the same length and also possible to prove that bounds checks are not needed. This serve as an example on how beneficial it can be to use a specialization and create the optimized version given a lot more knowledge.
Duplicate
In this benchmark, a for-loop contains a diamond, but unlike LoopD both paths are chosen equally frequently. This benchmarks shows the penalty of using the profile info in a place where it should not be used, but also that code duplication can be profitable in a case where the data shows two common paths.

HotPathD
This benchmark includes a chain of diamonds. The path through them will be the same as that of a skier, but at half of the times one way and half of the times the other. This will show the same as Duplicate does, that there are occasions where simple profiling are bad to use and in some of them, like this one, duplication can help. But it shall also show that in some cases, looking at the complete method shows more than looking at each branch by itself. However, since this was just a theory and and not visible in any of the real life applications used, this advantage was never used in this thesis.

Results from the micro benchmark runs.
In the diagrams below are the time spent to perform a certain amount of work reported with the different ways of running JRockit. This means that the lower the bar is, the better the result. The time for each run is normalized for the diagrams.

It shows that in the right conditions, a lot is to be gained with profiling. Laying out the blocks in an order where the code blocks don’t interfere can gain you as much as 40%. It is also possible to see that at under the optimal conditions, you can gain up to 10% on code duplication. But as you see on the P4 runs, it is also possible to make the wrong decisions and when that happens loose from it. However, that would be solved by not using the profile data, if block duplication is not an option.

The results also show that the most powerful optimization is the specialization, which can give more than 5 times speedup. This is the case since the work that is removed when a candidate is found can be and is in this case substantial.

![Bar chart showing results from micro benchmarks on an AMD Athlon](image)
Results

The difference in results between the Athlon processor and the P4 processor is substantial. The P4 processor has a branch profiler build into the processor, which the Athlon doesn’t and that profiler exist to partly solve the problem this thesis address. On the Athlon processor, two things improve the result, the work that has to be done is less since branches are not taken when not needed, but also are good hints provided to the CPU on how to predict branches, based on direction. The P4 processor doesn’t need and doesn’t use these hints and the benefits are solely what the extra instructions give.

**SPECjvm98**

SPECjvm98 is a benchmark suite for JVMs. This benchmark is produced by SPEC [18] and is here used as a tool to measure the JVM in different real life scenarios.

The eight small applications are run several iterations in a row. When the JVM has stabilized, the time for an iteration is the same as previous, the actual benchmark iteration is done and the time for the iteration is reported. In the measurements here, the number of iterations are forced to a high number, 30, and the average time of the iterations for the second half of the runs are calculated, to reach the same intention, but with more stable data.

In the diagrams below, the speedup for the different benchmarks in the suite are reported in percent. The speedup is defined as:

\[ \text{speedup} = \frac{T_o - T_p}{T_o} \]

- \(T_o\) Time for original run
- \(T_p\) Time using profiling

The diagrams show that also in real world applications, there are candidates where it is beneficial to use profile data and improvements to up to 6% are possible to get. The results
are not only good, which is due to the naivety of this thesis, to always act on the data. This is discussed more in the next blocks Improvements and Conclusions. Also interesting to note is that the results follow the results from the micro benchmarks regarding the different CPU architectures.

SPECjvm98 results from an AMD Athlon Best is either any of or a combination between the optimizations used.

SPECjvm98 results from an Intel p4. Best is either any of or a combination between the optimizations used.
**Improvements**

What the implementations in this thesis project miss is the possibility to go fall back to a classic generating of the code. Especially the scheduling algorithms have to recognize when a branch is not likely enough to be taken and then fall back to a decent solution for both of them. Right now, one of the path choices will be selected as the hot one and optimized for. The cost for taking the other way is larger, than the benefit of taking the first one, both compared to the static case where both are ok to use.

What also could be seen in the results was that sometimes one of the schedulers produced the best result, sometimes another. The conclusion is that none of them is perfect, so all could use improvements and with these improvements the results would look even better.
Conclusion

In this thesis it is shown that it is possible to gather profile data on method level and map it to the corresponding method in the code pipeline, using a method level framework.

What is more interesting is that it proves beneficial to use the gathered profile data and optimize according to the information in it. This can make a significant positive difference, which is shown by the micro benchmarks.

In the real world, where applications are not as straight-forward and predictable, the task is harder and the benefits not as visible due to application overhead, also working side by side with the CPU which tries to solve the same problem, there are still performance gains in all scenarios of 1 to 7%.
References

Literature

Articles
3. MUTH, ROBERT. WATTERSON, SCOTT. DEBRAY, SAUMYA. 2000, Code Specialization based on Value Profiles. University of Arizona, Department of Computer Science
6. CYTRON, RON. FERRANTE, JEANNE. ROSEN, BARRY K. NEWMAN, MARK N. ZADECK, KENNETH F. 1991, Efficiently Computing Single Static Assignment Form and the Control Dependency Graph. IBM Research and Division and Brown University
7. BODIK, RASTISLAV. GUPTA, RAHUL SARKAR, VIVEK. 2000 ABCD: Eliminating Array Bounds Checks on Demand. University of Wisconsin and University of Arizona and IBM T.J. Watson Research Center
8. RAMALINGMAN, G. 2000, On loops, Dominators, and Dominance Frontier. IBM T.J. Watson Research Center
9. BRIGGS, COOPER, KENNEDY, TORCZON, 1989, Coloring Heuristics for Register Allocation. Rice University: Department of Computer Science
10. INTEL CORPORATION, 1999 IntelPentium 4 and Intel Xeon Processor Architecture Intel Corporation
13. QUIN, FENG. HENDREN, LAURI J. VERBRUGGE, CLARK. 2002, A Comprehensive Approach to Array Bounds Check Elimination for Java. McGill University: School of Computer Science
15. ANTONSSON, BJÖRN. DAHLSTEDT, JOAKIM. LAGERGREN, MARCUS. LINDHOLM. OLOF, WILUND, CALLE. 2001 Documentation caramel’s, The Java Runtime Product Group - BEA Systems Inc
16. STAFFAN FRIEBERG, 2004, Dynamic Profile Guided Optimization in a VEE on IA-64, KTH IT and BEA Systems Inc

Web