Abstract

Superblock compilation techniques such as control flow graph (CFG) or trace compilation have become a widely adopted approach to increase the performance of dynamically compiling virtual machines even further. While this was shown to be successful for many conventional virtual machines, it did not result in a higher performance for Java-based DBT machine emulators so far. These emulators dynamically translate application binaries of a target machine into Java bytecode, which is then eventually compiled into the native code of the emulating host by the Java Virtual Machine (JVM). Successful superblock compilation techniques for this class of emulators must consider the special requirements that result from the two-stage translation as well as the characteristics of the JVM, such as the inability of most Java JIT compilers to handle large bytecode methods efficiently.

In this paper, we present a superblock compilation approach for a Java-based DBT machine emulator that generates a performance increase of up to 90 percent and of 32 percent on average. The key idea of our design is to provide a large scope over the control flow of target applications across basic block boundaries for the JVM, while still keeping small bytecode methods for the execution units. In addition, we also present two further optimizations – interpreter context elimination and program counter elimination – which increase the emulation performance by 16 percent again. In total, the optimization techniques discussed in this paper provide an average performance gain of 48 percent for the surveyed emulator.

1. Introduction

Superblock compilation techniques have become a widely adopted approach for virtual machines that feature just-in-time compilation. While traditional DBT JIT compilers translate the target application at the unit of one basic block, superblock compilers use larger translation units and may increase the performance even further. Firstly, larger execution units mean less time is spent in the dispatch loop of a VM to select the next execution unit. Secondly, they also provide a larger scope for possible optimizations by the compiler during the JIT compilation.

While superblock compilation has been employed successfully in many conventional virtual machines, it did not result in a performance increase for Java-based DBT machine emulators so far. These emulators dynamically translate target machine application binaries into Java bytecode, which is then eventually compiled by the JVM into the native code of the emulating host. The advantage of this technique is that it is entirely platform-independent although it provides a high emulation performance due to dynamic compilation. In addition, it exploits the JVM as an external back end for native code generation that performs even optimizations specific to the host platform. However, a successful superblock compilation approach that actually results in a performance increase for this class of emulators must respect the special requirements that result from this two-stage translation as well as the characteristics of the JVM. Superblock compilation has already been researched for Java-based DBT machine emulators, but did not work very well due to the inability of many Java JIT compilers to efficiently translate large bytecode methods.

Our implementation pursues the strategy of providing a large scope over the target application’s control flow across basic block boundaries for the JVM while still keeping small bytecode methods for the execution units. This is achieved by a separation between the control flow logic of a CFG that implements the dispatch between the single CFG nodes and the actual behavior of these CFG nodes. While compiled superblocks actually only contain the control flow logic of the corresponding CFG, the behavior of the basic blocks is contained in separate Java classes that are statically invoked by the superblocks. Our experiments prove this approach to be successful with an average performance increase of 32 percent. Also, the separation between control flow and node behavior results in a very low code generation overhead for superblocks, which is smaller than 10 percent compared to the code generated in the plain basic block compilation mode.

In addition, we present two more optimizations that increase the speed of the emulation framework by further 16 percent. Sec. 2 presents the state of the art on superblock compilation and related work concerning Java-based DBT machine emulators. In Sec. 3, we
give a short description of the underlying emulation framework before the optimizations that we applied to it are discussed in Sec. 4. Our experimental results are shown in Sec. 5. Sec. 6 concludes this paper and provides a summary of our results.

2. Related Work
As argued in Sec. 1, the usage of translation units which are larger than one basic block enables a higher emulation performance for virtual machines. In the remainder of this paper, we use the term superblock compilation for all these compilation techniques. Large translation and execution units that combine more than one basic block of the target application are called superblocks.

Different superblock compilation techniques vary in their profiling strategy and in their strategy to select a subgraph of the target application control flow graph to be compiled into a superblock. Basically, two main approaches can be distinguished: CFG compilation and trace compilation. For CFG compilation, the control flow graph (CFG) of a superblock may contain many execution paths. For trace compilation, a superblock contains only one execution path whose nodes are compiled in sequence.

Prominent examples for CFG compilation are implementations of the Java and the .NET runtime environments. Here, the translation unit typically is one method. However, in the case of Java, recent implementations also apply trace compilation [7, 8]. Another example for CFG compilation is EHS by Jones and Topham [10]. The authors implemented three different CFG compilation modes and compared them to each other in terms of simulation speed and average size of a superblock. In SCC mode, superblocks contain only strongly connected components (SCC), which means in fact that they contain exactly the CFG of a program loop. In CFG mode, superblocks are not restricted to loops, while in page mode, the CFG of a whole physical memory page is compiled into a superblock. While all these modes superior to raw basic block compilation mode by the factor of 1.6 in terms of performance, none of them proved superior to the others in their experiments.

This might hint that superblocks that cross (outer) loop boundaries do hardly bear any benefit because most runtime is spend within loops anyway.

Trace compilation performs a sophisticated runtime profiling in order to determine the most frequently executed paths through an application CFG. One such a path, called a trace, is then compiled node by node into a sequential superblock. This sequential control flow inside the execution unit provides massive potential for optimizations by the JIT compiler. Also, branching is generally an expensive operation. However, there is still guarding code required for each CFG node that could possibly exit the trace. Therefore, superblocks are not free of branch instructions at all.

Because each superblock contains only one execution path, code duplication that occurs when two paths include the same CFG node is an even bigger problem for trace compilation than it is for CFG compilation. Anchor points for trace compilation must therefore be chosen more carefully. Bala et al. [5, 6] propose the targets of backward branches as anchors for trace compilation for their dynamic binary optimization system Dynamo, because they mark the entry point of a loop very likely. While this is a commonly accepted strategy, Hayashizaki et al. [9] argue that it also encounters false loops, which could degrade the performance of a virtual machine. False loops do actually not represent program loops but could instead occur when multiple callers invoke the same method. They propose a stack-based false loop filter to avoid them. This, however, requires knowledge and makes assumptions about the target machine that are not fulfilled in the case of a generic machine emulator.

Meanwhile, Java-based machine emulators have emerged. These emulators translate target machine code dynamically into Java bytecode, which is then eventually compiled into the native code of the emulating host. Their benefits have been discussed in Sec. 1. Examples for this approach are Pearcolator/MRP [4, 12], JPC [3], JDosbox [2], as well as Jahris [11], the emulation framework that we built our superblock compilation upon. However, they feature the compilation of plain basic blocks only. While jxEmu/VEELS [15] also supports superblock compilation, the authors report poor performance when the size of a bytecode method for a superblock exceeds a certain threshold. They attribute this to the inability of many Java JIT compilers to translate large bytecode methods efficiently.

3. Underlying Emulation Framework
The Java-based DBT machine emulator that we built our optimizations upon is a generic emulation framework. Simulation models are constructed with an instruction set description language called HPADL. In this language, ISA models are split into three parts: the structural section describing the register set and the memory system of the target, the decoder description and the behavioral description. The decoder description models the instruction decoder of the target machine. For an input instruction word, it produces a sequence of micro operations that resembles the behavior of the decoded instruction. The behavioral description implements the behavior for the individual micro operations. The instruction control
flow is modeled within the behavioral part by assigning values to the program counter register.

Next to a model of the target machine, emulation requires an application binary compiled for the target machine. The binary is loaded into the memory system of the virtual target and decoded by the instruction decoder in units of basic blocks. The resulting micro operation sequences, called simple functions, reproduce the behavior of the basic blocks of the target application. They are stored in the simulation code cache for later reuse. Each basic block, as far as it is not removed from the cache in the meantime, is therefore decoded only once. Simple functions are interpreted by the simulation engine. As soon as the execution count for a simple function reaches a certain threshold, it is dynamically compiled into Java bytecode. The associated Java class, called a translated function (TF), is then dynamically loaded into the JVM and instantiated as a singleton via the Java Reflection API. It replaces the corresponding simple function in the simulation code cache. Further compilation and optimizations steps on the level of the TF, such as its translation into the native code of the emulating host platform, are then eventually performed by the JVM. The workflow of the framework is depicted in Fig. 1.

4. Optimization Techniques

4.1 Elimination of the Interpreter Context

When executing a basic block of the target application in interpreter mode (Simple Function), the machine context of the simulation target is mapped to a set of Variable objects, with each of these objects representing one machine register. This set is called the interpreter context. Variable objects provide a suitable form for the internal representation (IR) of HPADL statements, on which the simulation interpreter operates. They include a get and a set method in order to access the variable contents.

On the other hand, basic blocks compiled to Java bytecode (Translated Function) use another representation of registers. Here, the machine context of the simulation target is mapped to a Java class, in which each machine register is represented by a static field. This class is generated by the simulation compiler and is called the compiled context. The compiled context enables the more efficient execution of translated functions, in which each register read and write operation is implemented by a simple static field access (getstatic, putstatic).

This, however, implies that the machine context has to be copied from the interpreter to the compiled context whenever execution switches from a simple to a translated function, and vice versa when switching from a translated to a simple function. While this rarely happens for the mature stage of application execution when most of the simple functions have been compiled into translated functions, it causes a measurable impact during the warm-up phase.

In order to avoid this runtime overhead, we eliminate the interpreter context and use the compiled context also for the interpreter mode. Since field access via Java reflection might be orders of magnitude slower than direct field access [13], reflection is not feasible for this purpose and would slow down the interpreter up to impracticability. Instead, we let the simulation compiler generate a subclass of Variable for each static field in the compiled context, whose get and set methods contain a simple getstatic and putstatic bytecode to access the respective field. These classes are then instantiated as singletons that replace the original Variable objects of the interpreter context in the IR.

4.2 Program Counter Elimination

The control flow of target instructions is modeled by modifying the program counter register in HPADL architecture descriptions. It is incremented by sequential control flow instructions while branch instructions write the address of the branch target to it. From the view of the architecture description, it is just a regular machine register. It is handled as a regular register also by the emulation engine. Like for any other register, HPADL statements that address the program counter are translated one to one by the simulation compiler. For the emulation engine, its only speciality is that, between two execution units, the next instruction fetch address (IFA) is loaded from this register.

While providing an elegant way to model the control flow, this induces a huge overhead at emulation runtime because each instruction loads, modifies and writes the program counter. This is depicted in Listing 1 for the example of an ARMv4 Translated Function with R15 as the program counter. This overhead is, however, unnecessary as the program counter contains at the execution time of an instruction is already known to the instruction decoder when fetching the instruction word from a given address. Therefore, we removed the program counter from the simulation while still keeping it in the architecture description. HPADL expressions that read from the program counter are replaced by constant load operations by the instruction decoder. Assign statements that write to it are simply ignored and do not produce any code in the case of sequential control flow and direct branch instructions (Listing 2). Only for indirect branch instructions, they are translated into a bytecode sequence that passes the value of the assign expression as the next IFA to the instruction decoder.

Direct and indirect branches can be distinguished by a data flow analysis on the micro operation sequence generated by the instruction decoder. In the case the branch target address is a constant expression in the context of the decoded instruction word, the branch is a direct branch and an indirect branch otherwise. If the assign statement occurs within a conditional statement, the branch is conditional. Finally, unconditional direct branches whose target is the address of the next instruction in the sequential control flow are no branches but sequential control flow instructions. Note, that this scheme is fully transparent for existing HPADL architecture models and does not require to modify them.

4.3 Superblock Compilation

A successful superblock compilation approach for Java-based DBT machine emulators must respect the special requirements that result from the two-stage compilation approach on the one hand (Sec. 3), and from the characteristics of Java Runtime Environments (JREs) on the other hand.

The optimization of translated functions on the bytecode level by the simulation compiler is largely irrelevant because code optimization is performed by the JVM anyway. Typical Java JIT com-

Listing 1. TF bytecode for an instruction sequence A, B, ... without program counter elimination
1 getstatic ctx0.r15 // increment PC
2 iconst 4
3 iadd
4 putstatic CTX0.r15
5 // remainder behavior of A
6 getstatic CTX0.r15 // increment PC
7 iconst 4
8 iadd
9 putstatic CTX0.r15
10 // remainder behavior of B
11 ...
12 ldc 0x40007F38
13 putstatic CTX0.r15 // PC = 0x40007F38 (branch)
14 getstatic CTX0.r15
15 ireturn // return PC
Superblocks, CFGs and TFs

Our basic idea is to provide a large view to the JVM across basic block boundaries of the target application while still keeping small bytecode methods. Like in the original emulation framework without superblock compilation (Sec. 3), the target application is decoded basic block by basic block into simple functions (SFs) that are interpreted, compiled basic blocks (TFs) that gradually replace SFs, and superblocks (CFGs) that gradually replace TFs.

When a CFG is compiled into a superblock, its nodes are not inline, but instead, they are invoked statically from the superblock. Thus, the bytecode generated for a CFG contains only the dispatch logic between TFs but no behavior while the bytecode for a TF resembles the behavior of one basic block of the target application. This keeps bytecode methods small and yet provides large execution units. The control flow inside a superblock is visible to the JVM, which can then apply optimizations to it that go across basic block boundaries. Additionally, compared to conventional superblock compilation techniques, our approach reduces code duplication heavily because each TF exists only once, no matter how many superblocks contain it as a CFG node.

In the original emulation framework, TFs were invoked only by the dispatch loop. This invocation had to be non-static because otherwise the loop had to invoke them via reflection API. TFs contained a non-static execute method that implemented the behavior of the basic block and returned the IFA for the next look-up in the code cache. However, with superblock compilation TFs are also invoked by CFGs. This invocation is static and is not required to return an IFA, because the control flow logic is compiled into the superblock directly. For this purpose, a second, static method exec has been introduced that now implements the basic block behavior (Fig. 2). It is invoked by CFGs as well as the TFs own execute method. The execute method additionally returns an IFA, contains profiling code and is still only invoked from the dispatch loop. While this means some overhead when a TF is invoked directly from the dispatch loop, most runtime is spent inside CFGs after the application warm-up phase anyway.

CFG Anchors: Execution Count vs. Loop Headers

In order to maximize the performance gain by superblock compilation, most runtime must be spent within superblocks (i.e., in CFGs or in TFs that are invoked by CFGs) rather than in basic blocks (i.e., TFs that are invoked directly by the dispatch loop). We implemented two

Listing 2. TF bytecode for instruction sequence A, B, ... with program counter elimination
1  <B>  // behavior of A
2  <B>  // behavior of B
3  ...
4  ldc 0x40007F38
5  ireturn  // return 0x40007F38 (next IFA)

In order for these highly sophisticated runtime optimization techniques to score the best results, control and data flow analyses over a large portion of the target application are required. This is not a problem for conventional Java applications. For Java-based DBT machine emulators, however, the control flow between two execution units of the target application is masked by the emulator’s code cache and its dispatch loop. Thus, it is not visible to the JVM so optimization cannot cross execution unit boundaries.

An obvious solution to this problem would be to combine more than one basic block into a translation unit. This would generate larger execution units and therefore provide a greater view of the target application control flow to the JVM. As discussed in Sec. 2, this, however, barely yields a performance increase due to the inability of many Java JIT compilers to handle big bytecode methods efficiently. We therefore conclude three design criteria for a successful superblock compilation in a Java-based DBT machine emulator:

1. Many small execution units – i.e., bytecode methods – are better than a few big ones.
2. The control flow between these units must be visible to the JVM.
3. The optimization of translated functions on the bytecode level is rather irrelevant. We leave optimizations up to the JVM.
strategies of profiling and anchor selection in our superblock compiler: counting the executions of a TF (execution count) and counting how often it has been the target of a backward branch (backward branch count). The latter is a strategy widely used to identify loop headers (Sec. 2). As soon as the respective count exceeds a certain threshold, a TF is used as an anchor for superblock compilation.

In general, loop headers are suitable anchor points for superblock compilation because most target application runtime is spent within loops. Also, the resulting superblocks for our compilation approach should comprise at least the complete CFG of a loop in this case. In our experiments, however, it turned out that even after a long warm-up phase when no more superblocks were added to the emulation code cache, most runtime was still spent in TFs that were invoked by the dispatch loop directly rather than from superblocks.

The explanation for this behavior is indirect branches. Because of the generic nature of the emulation framework, no assumptions but the given architecture description are made about the emulated target machine. Therefore, superblock compilation is unable to perform a control flow analysis across the subroutine boundaries of the target application. While the direct call instruction for a subroutine is handled as a direct branch, the corresponding return instruction is an indirect branch where the branch target address is not encoded within the instruction word but fetched from a link register or fetched from the stack. If a subroutine is called within a loop, this results in an incomplete CFG for the loop. If only loop headers are selected as superblock compilation anchors, the remainder of the loop, although executed as frequently as the loop header, will never get compiled into a superblock in such a case.

Thus, loop headers as the only CFG anchors is not a good design choice for our emulation framework. Instead, we use the execution count to trigger superblock compilation. Each TF that is executed frequently enough will be used as an anchor for superblock compilation whether or not it is a loop header. Note that this would result in massive code duplication for traditional superblock compilation approaches. Due to the separation between the control flow logic and node behavior of CFGs, this is not the case for our design however.

Path Profiling and Hotpath Reordering Our superblock compiler features a hybrid approach between traditional CFG compilation and trace compilation. CFG compilation in general compiles a part of the CFG of the target application into one execution unit at a time. The resulting execution unit may contain many execution paths. Trace compilation on the other hand is a special case of this. It uses a runtime profile to determine the most frequently executed paths through the CFG of the application. One such path, called a trace, is then selected for compilation and is compiled into a sequential instruction stream. Unlike for traditional CFG compilation, this provides a sequential control flow within execution units. Note, however, that there are still guarding code elements required after each CFG node that could possibly exit the trace. Thus, it is not free of branch instructions at all.

We do not compile single traces. Our superblocks may contain many execution paths. However, we also perform a runtime profiling on how often branches are taken. For each TF that exits with a conditional direct branch, there is a counter that indicates its branch count. For TFs that exit with an unconditional branch, there is no such counter required because their branch count always equals their execution count. Finally, for TFs that exit with a conditional indirect branch, the branch count is irrelevant because their branch target is not considered in the CFG analysis anyway. This results in a CFG whose edges are weighted by the branch count.

When a CFG is compiled into a superblock, its nodes are reordered so that the most frequently executed paths are compiled into a sequence. Starting with the compilation anchor, the node linked into the execute method at the offset behind the current node is always the one targeted by the heaviest-weighted edge. This is similar to trace compilation except that the guarding code elements may not only exit a superblock but may also transfer control to another execution path within the same superblock. This is illustrated in Fig. 3 with an example control flow graph including four traces and A as the compilation anchor.

5. Experimental Results
5.1 Emulation performance

We used the EEMBC AutoBench 1.1 Automotive/Industrial Benchmark Suite [1] in order to measure the simulation performance for the optimization techniques described in Sec. 4. The entire suite consists of 16 individual benchmarks that resemble a representative set of applications for embedded platforms in terms of their runtime profile and instruction mix. The suite was compiled with a GCC toolchain version 4.4.3 for ARM, floating point emulation enabled, optimization level O2 and with the default compiler settings otherwise. It was run with a HPADL ARMv4 machine model on our emulator. The host machine features an Intel® Core i7 with 6 GB memory and 64-Bit Ubuntu 11.04, kernel version 3.0.0-15-server. The emulator was executed with the Oracle JRE 7 HotSpot 64-Bit Server VM. The iteration number for the benchmarks was chosen so that a run took between 5 and 10 seconds. A benchmark was run 5 times for each mode with always the best score recorded.

In order to evaluate the performance increase achieved by superblock compilation and the other optimization techniques, we compared three operation modes of the emulator. In TF-Only mode, superblock compilation is disabled but program counter elimination and interpreter context elimination are still enabled. In Exit-CFG mode, superblock compilation is also enabled, but superblocks do not invoke each other. Instead, if a CFG node of a superblock during CFG analysis turns out to be another superblock rather than a TF, the compiled superblock will not invoke this node but return to the dispatch loop at this point. In contrast, superblocks do also invoke each other in Invoke-CFG mode. Note that this will never result in the recursive invocation of superblocks. This is because when a superblock B is being compiled and found to invoke another superblock A, A cannot invoke B in turn, because B did not exist at the time when A was compiled. A would instead statically invoke the TF that was selected as the compilation anchor for B at a later point in time.
The results of our experiments are shown in Tab. 1 and Fig. 4. The baseline is the simulation performance of the same emulation framework without any of the optimizations applied as discussed in Sec. 4. Compared to the baseline, TF-Only increases simulation performance by up to 27 percent, and by 16 percent on average. Exit-on-CFG increases the performance by up to 87 percent and by 40 percent on average, while Invoke-CFG delivers the best performance with an increase of 100 percent for the a2time benchmark and of 48 percent on average. This means that program counter elimination and interpreter context elimination cause a significant performance gain of 16 percent for the emulator with the referred benchmark set. Invoke-CFG provides an average performance that is even 32 percentage points higher than this, which is attributed only to superblock compilation. Our superblock compilation approach described in Sec. 4.3 as well as our other optimization techniques discussed in Sec. 4 can therefore be considered successful.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Invoke-CFG</th>
<th>Exit-on-CFG</th>
<th>TF-Only</th>
<th>Baseline</th>
<th>Invoke-CFG</th>
<th>Exit-on-CFG</th>
<th>TF-Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2time</td>
<td>205158</td>
<td>184631</td>
<td>112794</td>
<td>102329</td>
<td>200</td>
<td>180</td>
<td>110</td>
</tr>
<tr>
<td>basefp</td>
<td>46679</td>
<td>41722</td>
<td>29439</td>
<td>27001</td>
<td>173</td>
<td>155</td>
<td>109</td>
</tr>
<tr>
<td>bitmnp</td>
<td>9878</td>
<td>9778</td>
<td>7620</td>
<td>6654</td>
<td>148</td>
<td>147</td>
<td>115</td>
</tr>
<tr>
<td>cacheb</td>
<td>1324908</td>
<td>1334154</td>
<td>1388592</td>
<td>1161336</td>
<td>114</td>
<td>115</td>
<td>120</td>
</tr>
<tr>
<td>canrdr</td>
<td>1533481</td>
<td>1301330</td>
<td>1342482</td>
<td>1054975</td>
<td>145</td>
<td>123</td>
<td>127</td>
</tr>
<tr>
<td>aifftr</td>
<td>593</td>
<td>590</td>
<td>572</td>
<td>511</td>
<td>116</td>
<td>116</td>
<td>112</td>
</tr>
<tr>
<td>aifirf</td>
<td>115735</td>
<td>113884</td>
<td>103638</td>
<td>97490</td>
<td>119</td>
<td>117</td>
<td>106</td>
</tr>
<tr>
<td>idctrn</td>
<td>9864</td>
<td>9896</td>
<td>7865</td>
<td>5285</td>
<td>187</td>
<td>187</td>
<td>149</td>
</tr>
<tr>
<td>iirflt</td>
<td>79601</td>
<td>76849</td>
<td>72819</td>
<td>55788</td>
<td>143</td>
<td>138</td>
<td>131</td>
</tr>
<tr>
<td>aifitt</td>
<td>652</td>
<td>645</td>
<td>629</td>
<td>557</td>
<td>117</td>
<td>116</td>
<td>113</td>
</tr>
<tr>
<td>matrix</td>
<td>201</td>
<td>190</td>
<td>131</td>
<td>124</td>
<td>162</td>
<td>153</td>
<td>105</td>
</tr>
<tr>
<td>pntrch</td>
<td>22798</td>
<td>22759</td>
<td>5046</td>
<td>14164</td>
<td>161</td>
<td>161</td>
<td>106</td>
</tr>
<tr>
<td>puwmod</td>
<td>1991238</td>
<td>1934235</td>
<td>1643655</td>
<td>1488538</td>
<td>134</td>
<td>130</td>
<td>110</td>
</tr>
<tr>
<td>rspeed</td>
<td>2007528</td>
<td>1759014</td>
<td>1779755</td>
<td>1451115</td>
<td>138</td>
<td>121</td>
<td>123</td>
</tr>
<tr>
<td>tblook</td>
<td>65876</td>
<td>56361</td>
<td>46162</td>
<td>41814</td>
<td>158</td>
<td>135</td>
<td>110</td>
</tr>
<tr>
<td>tssprk</td>
<td>48043</td>
<td>47195</td>
<td>36348</td>
<td>31873</td>
<td>151</td>
<td>148</td>
<td>114</td>
</tr>
</tbody>
</table>

**Table 1.** Emulation Performance by Optimizations

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Average</th>
<th>Geo. mean</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>148</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>140</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>116</td>
<td>116</td>
</tr>
</tbody>
</table>

Figure 4. Emulation Performance by Optimizations Compared to Baseline

5.2 Code Generation Overhead

Table 2 and Fig. 5 depict the total size of the classes generated for superblocks (CFG) and basic blocks (TF) for one run of each benchmark in Sec. 5.1. It turns out that the code generation overhead for superblock compilation is rather low. While the geometric mean of the total class size is 221800 bytes for generated TFs, the geometric mean of the total class size for CFGs is 20393 bytes. This means that on average only 8.4 percent of all class bytes dynamically generated are due to superblock compilation while the remaining 91.6 percent are attributed to basic block compilation.
Table 2. Superblock Code Generation Overhead

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total size in bytes</th>
<th>ARMv4 machine code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TF classes</td>
<td>CFG classes</td>
</tr>
<tr>
<td>a2time</td>
<td>265328</td>
<td>30578</td>
</tr>
<tr>
<td>basefp</td>
<td>211512</td>
<td>28257</td>
</tr>
<tr>
<td>bitmnp</td>
<td>288504</td>
<td>18507</td>
</tr>
<tr>
<td>cacheb</td>
<td>146949</td>
<td>8188</td>
</tr>
<tr>
<td>canrdr</td>
<td>190240</td>
<td>18926</td>
</tr>
<tr>
<td>aifftr</td>
<td>300449</td>
<td>30084</td>
</tr>
<tr>
<td>aifirf</td>
<td>127278</td>
<td>7411</td>
</tr>
<tr>
<td>idctrn</td>
<td>275416</td>
<td>12556</td>
</tr>
<tr>
<td>irfft</td>
<td>202403</td>
<td>20622</td>
</tr>
<tr>
<td>idptrch</td>
<td>277286</td>
<td>30343</td>
</tr>
<tr>
<td>matrix</td>
<td>503186</td>
<td>128207</td>
</tr>
<tr>
<td>puwmod</td>
<td>137438</td>
<td>9587</td>
</tr>
<tr>
<td>rspeed</td>
<td>173472</td>
<td>11272</td>
</tr>
<tr>
<td>tblook</td>
<td>122519</td>
<td>10931</td>
</tr>
<tr>
<td>tssprk</td>
<td>304212</td>
<td>54105</td>
</tr>
<tr>
<td>Geo. mean</td>
<td>227800</td>
<td>20393</td>
</tr>
</tbody>
</table>

Table 3. Average Generated Class Count and Size

<table>
<thead>
<tr>
<th>Class type</th>
<th>Class count</th>
<th>Class size in B</th>
<th>Bytecode size in B</th>
</tr>
</thead>
<tbody>
<tr>
<td>TF</td>
<td>248.9</td>
<td>965</td>
<td>215 to 290</td>
</tr>
<tr>
<td>CFG</td>
<td>45.4</td>
<td>612</td>
<td>100 to 120</td>
</tr>
</tbody>
</table>

6. Summary

This paper presented a superblock compilation approach for a Java-based DBT machine emulator, that, in contrast to earlier approaches, significantly increases the emulation performance by 32 percent on average compared to a plain basic block compilation. We achieved this due to a new strategy that generates small bytecode methods but still provides a large view on the control flow of the target application to the JVM. For this purpose, we separated the control flow logic for CFGs from the actual behavior of their nodes. While CFG classes (superblocks) contain only the control flow logic, TF classes, which are statically invoked by CFGs, implement only the behavior of a single basic block.

Our superblock compiler also features a hybrid technique between the traditional compilation of CFGs and trace compilation. While superblocks may contain many execution paths, branches between basic blocks are profiled at runtime. When a superblock is generated, the nodes of its CFG are reordered so that its most
frequently executed paths are compiled to a sequence. Contrary to existing approaches that prefer loop headers as anchor points for superblock compilation, we use every basic block that is executed frequently enough as an anchor, whether or not it is the head of a loop.

While this would cause massive code duplication for traditional superblock compilers, our separation between the control flow logic and the node behavior ensures a low code generation overhead. In fact, our experiments show that not even 10 percent of the overall generated classes size is due to superblocks (CFG) while more than 90 percent are due to basic blocks (TF).

Additionally, we implemented two more optimizations. Interpreter Context Elimination removes the interpreter context and maps the registers of the simulation target onto a compiled machine context even in interpreter mode. Program counter elimination removes the program counter register from the simulation model and replaces it by constants, which are then compiled directly into Java bytecode by the simulation compiler. These two optimizations increase the simulation performance by 16 percent on average again, which results in an average performance gain of 48 percent when also superblock compilation is applied.

Acknowledgments

This work has been supported by the European Social Fund (ESF) and by pls Programmierbare Logik & Systeme GmbH.

References