Adaptive Inlining and On-Stack Replacement in the CACAO Virtual Machine

Edwin Steiner Andreas Krall Christian Thalinger
Technische Universitaet Wien
Institut fuer Computersprachen
Argentinierstr. 8
A-1040 Wien, Austria
{edwin,andi,twisti}@complang.tuwien.ac.at

ABSTRACT

Method inlining is a well-known and effective optimization technique for object-oriented programs. In the context of dynamic compilation, method inlining can be used as an adaptive optimization in order to eliminate the overhead of frequently executed calls. This work presents an implementation of method inlining in the CACAO virtual machine. On-stack replacement is used for installing optimized code and for deoptimizing code when optimistic assumptions of the optimizer are broken by dynamic class loading. Three inlining heuristics are compared using empirical results from a set of benchmark programs. The best heuristic eliminates 51.5% up to 99.96% of all executed calls and improves execution time up to 18%.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation, run-time environments*

General Terms

Performance

Keywords

virtual machines, method inlining, on-stack replacement, just-in-time compiler

1. INTRODUCTION

Modern object oriented programming languages offer sophisticated abstraction mechanisms to the programmer like classes or virtual methods. The challenge for a language implementation is to support these abstractions with as little overhead as possible. Additionally a virtual machine has to support features as exceptions, automatic garbage collection, multiple threads of execution, synchronization mechanisms, and dynamic class loading. In such an environment, classical program optimizations often have to be modified in order to still be applicable and effective. Object-oriented programming styles favor lots of very small subroutines (*methods*) and frequent, deeply nested calls.

An important technique for minimizing the call overhead is *inlining*. Inlining refers to replacing subroutine calls with a modified version of the body of the called subroutine. This effectively reverts the abstraction introduced by the programmer, eliminating call overhead, and providing larger scopes for subsequent optimization. There are, however, several issues which complicate inlining: First, inlining is not without costs. Excessive inlining can greatly increase compiled code size. Another complication is caused by *polymorphic calls* that must be dispatched at runtime. The receivers of such calls cannot be determined at compile time, making straight-forward inlining impossible.

In the presence of dynamic class loading, parts of the program code may only become available during execution of the program. This precludes the use of classic whole-program optimizations. However, it is possible to modify optimizations such that they act upon *preliminary* results obtained at runtime. In order to guarantee correctness, such speculative optimization must be supplemented by mechanisms that track the assumptions made during optimization and take proper measures as soon as any assumption becomes invalid.

Replacing program code during runtime poses great challenges for a virtual machine. Particular difficulties arise with *on-stack replacement*, i.e. replacing the code of methods that are currently activated. Nevertheless, on-stack replacement allows timely replacement of methods that execute for a very long time and enables speculative optimizations like inlining currently-monomorphic calls without paying the runtime cost of guard code to protect the optimized call sites.

The techniques described in this work have been implemented in CACAO [15], an open source Java virtual machine providing just-in-time compilation for many different architectures (`www.cacaojvm.org`).

2. RELATED WORK

Inlining can be counted among the most effective optimizing program transformations for a variety of programming languages, with examples of execution time improvements of 5 to 28% [18] for CLU 15% [8] for C code, 24% [4] for intermediate code, and 10 to 44% [13] or up to 40% [2] for Java programs.

PPPJ 2007, September 5–7, 2007, Lisboa, Portugal.

Arnold, Fink, Sarkar, and Sweeney conducted a comparative study [2] of inlining heuristics. They demonstrated that the effectiveness of inlining rises with the precision of available profile data. In particular, heuristics based on a dynamic call graph with edge weights proved to outperform all algorithms using coarser data, even if given much narrower limits for code expansion.

Ishizaki et al. [13] reported that inlining only very small methods gave performance improvements to within 15% of the peak performance, while increasing compilation time by only 6% on average. On the other hand, in order to obtain the peak performance, an increase of compile time by up to 50% had to be accepted.

Object-oriented languages typically make use of dynamically dispatched *virtual methods*. The purpose of *devirtualization* is to identify virtual call sites that can be proven to be monomorphic and turn them into statically bound calls. Dean, Grove, and Chambers proposed *class hierarchy analysis* [9] (CHA) as a means to limit the set of potential receivers of virtual calls.

Lee et al. [16] reported that in the Java programs of the SPECjvm98 benchmark suite, about 85% of virtual calls are monomorphic and about 90% have a single target method.

Guarded inlining techniques can be used to inline methods that cannot be proven to be monomorphic. Detlefs and Agesen proposed the *method test* [10] as a refinement of the class test for guarded inlining. Arnold and Ryder introduced *thin guards* [3] in order to further reduce the cost of guarded inlining.

The Jalapeño Virtual Machine [1] developed by Burke et al. [5] was the first *compile-only* system for dynamic optimizing compilation of Java code.

The Java HotSpot^{TM} Server Compiler [17] uses a mixed-mode approach for executing Java code. HotSpot employs method-entry and backward-branch counters to select hot methods for compilation. On-stack replacement is used for deoptimization and for replacing long-running methods.

Suganuma et al. [19] developed another mixed-mode framework that provides a three-level optimizing compiler, a lightweight continuous sampling profiler, and an instrumenting profiler. Their instrumenting profiler allows dynamic installation and de-installation of profiling code at method entries by code patching.

On-stack replacement was first described by Chambers and Ungar in the context of deferred compilation in the Self-91 compiler [7]. Hölzle, Chambers and Ungar subsequently used on-stack replacement for debugging optimized code via deoptimization [12]. The *interrupt points* defined by the authors are analogous to the replacement points used in this paper.

In his PhD Thesis [6] Chambers described *scope descriptions* and *byte code mappings* created by the SELF Compiler in order to facilitate deoptimization. He also presented *dependency links* as a means to record assumptions and perform selective code invalidation.

Fink and Qian implemented and evaluated [11] on-stack replacement in the Jikes RVM. Their design compiles a specialized version of the method for each activation that is replaced. Each version has a specialized prologue prepended to the original bytecode that sets up local variables and the stack and then jumps to the current program counter. An advantage of this scheme is that it requires minimal changes to the underlying compilers and that it may provide additional opportunities for optimization. On the other hand more compiled code has to be generated.

3. ADAPTIVE OPTIMIZATION FRAMEWORK

Adaptive optimization refers to the application of optimization techniques at runtime by monitoring the behavior of the running program and using the collected data to guide optimization decisions. Several parts of a virtual machine have to cooperate to make adaptive optimization possible. This section briefly describes the modules that are responsible for profiling, recompilation, and optimization in the CACAO virtual machine.

CACAO is an open source research Java Virtual Machine. The native code generation is based on a JIT compiler with a compile-only approach.

3.1 Modules

In order to provide adaptive optimizations, several modules of the virtual machine must work together. Figure 1 shows the most important modules and how they interface with each other.

The *JIT compiler* is responsible for compiling bytecode to machine code. The *code repository* manages the generated machine code, including the invalidation of obsolete code. In the *method database* properties of methods and assumptions about methods are kept. The *replacement mechanism* performs replacement of old versions of compiled code with new versions, including on-stack replacement. The replacement module also provides services to the garbage collector, like setting traps, reading the source state, and writing back a modified source state. The *linker* determines object layout, performs method overwriting, and builds the virtual method and interface tables. The *inliner* is a separate pass in the compiler that performs the inlining transformation. The inliner is depicted as a module in order to emphasize its separation from other parts of the compiler and to clearly show its interaction with the method database. The exact *garbage collector* interfaces with the replacement mechanism for setting GC traps, finding live objects in the source state, and redirecting references after compaction. The *architecture layer* bundles architecture-dependent functions needed by the replacement mechanism.

3.2 Adaptive Recompilation

As the optimizing compiler has higher compile time and can produce larger code (by inlining) than the baseline compiler, it should only be used for code that is executed frequently enough to amortize these costs. CACAO uses instrumentation of the code generated by the baseline compiler in order to select methods for which recompilation is likely to be profitable.

In CACAO a method can go through a sequence of states and transitions.

The baseline compiler generates code for the method and inserts countdown traps in order to trigger recompilation after a certain number of method entries or loop iterations. The majority of methods does never reach this threshold (see Section 5).

When a method triggers a countdown trap, it is recompiled with instrumentation code for creating a dynamic profile of the method, including execution counts for each basic

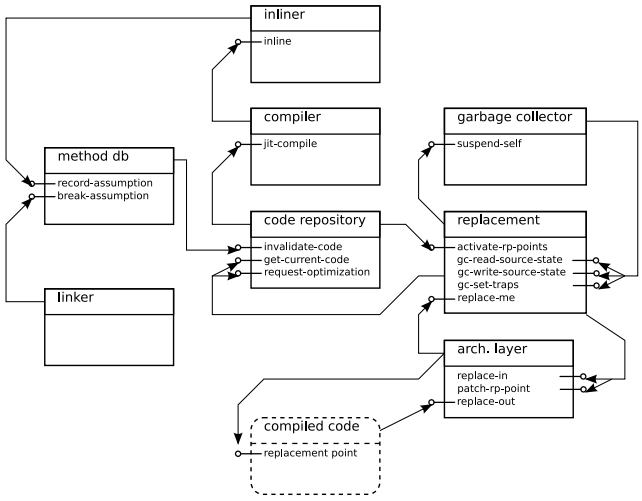

Figure 1: Modules of the adaptive optimization framework

block. Countdown traps are used to limit the time the code runs with full instrumentation.

When the instrumented code triggers recompilation, it is recompiled by the optimizing compiler. The optimized code contains no intrusive instrumentation.

3.3 Inlining Mechanism

The input to the inlining mechanism is the intermediate representation (IR) of a method—subsequently called the *root method*—possibly augmented with profiling information. The output is an intermediate representation of the root method in which selected `INVOKE` instructions have been replaced by inlined code (which in turn may include inlined code of nested callees).

3.4 Inlining Decisions

When the compiler has determined whether a call site can be inlined, it must decide whether the call site *should* be inlined. In order to make these inlining decisions, three heuristic algorithms have been implemented in CACAO.

3.4.1 Aggressive Depth-first Inlining

When the depth-first algorithm finds a call site that can be inlined, it inserts the corresponding node into the inlining tree, parses the callee, and enters a recursive analysis of the callee's code. Only two conditions limit the building of the inline tree. Call sites below a certain depth from the root node are not inlined and, when code expansion reaches a certain limit, inlining is stopped. For the experimental results given in Section 5 a maximum inlining depth of 3 was used and inlining was stopped when the resulting code would have had more than ten times as many basic blocks as the original root method. The final inlining tree used was the tree before adding the callee that crossed this threshold.

A variation of this scheme was also implemented, in which inlining is completely cancelled for a root method if the code expansion threshold is reached. So the root method is either left unchanged, or *all* monomorphic call sites down to the maximum inlining depth are inline expanded. As can be seen in Section 5, this variation caused much less overall code expansion than the one described above, with comparable execution times in some cases. However, there were also benchmarks for which this all-or-nothing variation performed significantly worse.

3.4.2 Aggressive Breadth-first Inlining

The breadth-first algorithm first inlines all possible call sites in the root method. After that it inlines the call sites in the previously inlined callees. The algorithm iterates, inlining call sites at a certain depth only after all candidate call sites at lower depths have been inlined. Inlining stops when code expansion—either measured as the increase in intermediate instructions or the increase in basic blocks—reaches a threshold.

3.4.3 Knapsack Heuristics

In contrast to the aggressive heuristics, another algorithm was implemented that tries to select only inlining candidates with an attractive ratio of expected benefit to expected cost of inlining. The algorithm is a variant of the greedy heuristics used to approximately solve the KNAPSACK problem: The algorithm starts with a certain inlining *budget* and at each step selects the call site for inlining that has the highest ratio of benefit to cost of all candidates fitting within the budget. The costs of the selected site are subtracted from the budget. Then the selected callee is parsed and all of the contained call sites that could be inlined are added to the set of candidate call sites. The algorithm iterates until there is no candidate left that fits within the remaining budget.

The hardest problem when implementing this algorithm is calculating good estimates of the benefits and costs of inlining a call site.

3.5 Requirements of Adaptive Optimization

There are several requirements on the replacement mechanism when used for adaptive optimizations.

Newly compiled versions of methods must be installed to replace older versions. As the new versions are usually better optimized, the replacement should happen as soon as possible. An important case are methods that are rarely entered or left but contain frequently executed loops. Such methods should be replaced *on-stack*, i.e. while they are activated.

When code optimized on preliminary assumptions becomes invalid, this code *must* be replaced before it can create an inconsistent program state. This requires *replacement traps* that can be set at the start of invalidated code regions.

Old and new versions of code may differ in the allocations of variables. For example, one version may keep a value in a register, while another version puts the same value on the stack.

By changing inlining decisions, the grouping of source-level frames into machine-level stack frames can change. The replacement mechanism must be able to translate in both directions between code that performs a machine-level call, and code that is inlined within the same compilation unit.

4. ON-STACK REPLACEMENT

In the context of on-stack replacement it is fundamental to distinguish two different representations of program state.

The **machine-level** state, hereafter called the *execution state*, which depends on compiler optimizations and the **source-level** state, hereafter called the *source state*—a representation that is independent of any optimizations used.

The source-level state is the state that would have been created by interpreting the unmodified bytecode of the program up to the current point. Thus all optimizations that preserve the semantics of the program are transparent with respect to the source-level state. This allows the source-level state to serve as a common ground for translating state between differently optimized versions of the same program.

The following sections give a definition of these concepts.

4.1 Execution State

We define the *execution* state to be the machine-level snapshot of a thread at a certain time. The execution state of a thread comprises the current values of the CPU registers and the contents of the machine stack of this thread.

4.2 Source State

The source state of a thread comprises the currently active stack frames of the Java virtual machine stack. (Our notion of *source frame* is equivalent to the *JVM scope descriptor* in [11].)

For each source-level stack frame the currently executing method, the bytecode position within this method, the types and values of local variables that are currently live in the frame, the types and values of the operand stack slots that are currently live in the frame and the object that the method synchronizes on, if any, are given.

The most important property of the source state is that at any given point in the execution of the program the source state is independent of any past optimization decisions. In other words, all optimizations are transparent with regard to the source state. Thus when we revert optimization decisions at some point, program correctness is guaranteed if we replace the current execution state by a new execution state corresponding to exactly the same source state.

In order to be able to recreate an execution state from the source state, we also need the values of all saved registers before the activation of the bottom-most stack frame of the source state and the value of the stack pointer at this time.

4.3 Replacement Points

A *replacement point* is a position in compiled code where we have sufficient information to reconstruct the source state from the execution state taken at this point. We will hereafter use the term *replacement point* to refer to both the actual compiled code position and the data structure associated with this position that is needed for the transformations between execution state and source state.

We call a replacement point P *mappable* if it is guaranteed that all compiled versions of the containing method will have a replacement point corresponding to the same program point as P. Replacement points used for switching between differently compiled versions of a method must be mappable.

Notice that a replacement point at a call site has two aspects, depending on how it is reached during replacement: If execution is trapped at the point of invocation, all variables reaching the call site are live, including the arguments to the call. If, on the other hand, the replacement point is reached in the course of unwinding activation records, only the variables living through the call are guaranteed to be live. This has important consequences for the instance argument of non-static methods: When such a method is active while a replacement point is reached, there is no guarantee that there is a live variable referring to the object instance of this invocation.

5. RESULTS

This section reports experimental results obtained with CACAO and the SPECjvm98 benchmark suite using adaptive inlining on an i386 architecture (Intel Pentium M). The SPECjvm98 is the standard benchmark suite for evaluating Java virtual machines.

The programs of the SPECjvm98 benchmark suite were executed with recompilation triggered by countdown traps. Hot methods were recompiled with inlining and without instrumentation code. The benchmarks were also run with recompilation done by the baseline compiler (no inlining). The purpose of the latter runs, in which recompilation was only used to remove instrumentation code, was to separately measure the overhead introduced by the countdown traps.

As Figure 2 shows, depth-first heuristics performed well for `compress` and `mtrt`, improving overall execution time by 8.6% and 15% respectively. However, for `mpegaudio` performance got slightly worse, although over 80% of executed method calls were eliminated (see Figure 3). In this case, side effects of inlining degrade code quality. The large number of eliminated calls, however, indicates that even in this case the potential benefits of inlining could be great. Better performance may be expected when a linear scan register allocator and copy elimination are available in CACAO replacing the simple register allocator [14].

The downside of aggressive depth-first inlining is severe expansion of compiled code size. Thus a version of depth-first inlining was implemented that completely cancels inlining for a root method if the code expansion threshold is reached. As to be expected, this change reduces code expansion significantly The modified algorithm, however, yields bad results for two benchmarks. In the case of `javac` execution time increases by significant five percents compared to the unoptimized case. Clearly, a more sophisticated solution is needed to limit code expansion.

Aggressive breadth-first inlining was implemented in order to obtain more balanced inlining trees than those built by depth-first inlining. For the results presented in Figure 2, expansion of intermediate code was limited to a factor of five. For some benchmarks, breadth-first inlining yielded performance better than or comparable to depth-first inlining with less code expansion. Total code expansion is still high, especially for `javac`, which also becomes slower than the baseline version by 4%.

The knapsack algorithm was the only algorithm that improved execution time for all benchmarks in the SPECjvm98 suite. As can be seen in Figure 2, the variance of the achieved speedups is very large, with improvements ranging from 0.8% to 18.2%. The knapsack heuristics eliminated on average over 72% of executed method calls (somewhat less than aggressive depth-first inlining which eliminated over 76%).

Figure 3 shows how effective the various heuristics were in reducing the dynamic number of executed calls for each benchmark. The numbers show that adaptive inlining could in all cases reduce the number of calls significantly. In the case of `compress`, only 0.05% of the original number of calls was performed when using aggressive depth-first inlining.

For each recompiled method the change of code size relative to the code generated by the baseline compiler was

measured. Here we present the results for the Knapsack heuristic. Table 1 show the frequency distribution of the expansion factor, and its geometric mean and maximum. Note that these factors refer to individual recompiled methods and not to total code size change. An interesting observation is that in many cases, inlining does not increase the size of the compiled code at all, and quite often even reduces it. The reason for this can probably be found in the object-oriented programming style that favors very small methods, for example getter/setter methods containing a single statement, and empty initializers. Code size expansion—while for some benchmarks and heuristics rarer than reduction—still dominates on the whole.

6. CONCLUSION

Current virtual machines use dynamic compilation and adaptive optimization in order to deliver high performance while keeping compilation times low. A framework for adaptive optimization has been implemented in the CACAO virtual machine using sophisticated on-stack replacement of active methods. The replacement mechanism can switch between unoptimized and optimized code in both directions and between different optimized versions. Stack frames can be combined or split, as replacement translates between inlined and non-inlined method calls.

Adaptive inlining supports several heuristics for making inlining decisions. Of the implemented heuristics, a variant of the greedy knapsack algorithm proved to yield the best overall performance. Improvements of execution time up to 18% were achieved for the SPECjvm98 benchmark suite. As adaptive inlining could eliminate up to 99.96% and on average over 70% of the executed calls, further improvements can be expected when a linear scan register allocator is available in CACAO.

7. REFERENCES

[1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, October 2000.

[2] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *DYNAMO '00: Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, pages 52–64, New York, NY, USA, 2000. ACM Press.

[3] Matthew Arnold and Barbara G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. *Lecture Notes in Computer Science*, 2374:498–524, 2002.

[4] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 134–145, New York, NY, USA, 1997. ACM Press.

[5] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, New York, NY, USA, 1999. ACM Press.

[6] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, 1992.

[7] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 26, pages 1–15, New York, NY, 1991. ACM Press.

[8] Jack W. Davidson and Anne M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18(2):89–102, February 1992.

[9] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952:77–101, 1995.

[10] David Detlefs and Ole Agesen. Inlining of virtual methods. *Lecture Notes in Computer Science*, 1628:258–277, 1999.

[11] Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 241–252, Washington, DC, USA, 2003. IEEE Computer Society.

[12] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 32–43, New York, NY, USA, 1992. ACM Press.

[13] Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Effectiveness of cross-platform optimizations for a Java just-in-time compiler. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 187–204, New York, NY, USA, 2003. ACM Press.

[14] Andreas Krall. Efficient JavaVM just-in-time compilation. In Jean-Luc Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, Paris, 1998. North-Holland.

[15] Andreas Krall and Reinhard Grafl. CACAO — A 64-bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[16] Junpyo Lee, Byung-Sun Yang, Suhyun Kim, Kemal Ebcioğlu, Erik Altman, Seungil Lee, Yoo C. Chung, Heungbok Lee, Je Hyung Lee, and Soo-Mook Moon. Reducing virtual call overheads in a Java VM just-in-time compiler. *SIGARCH Comput. Archit. News*, 28(1):21–33, 2000.

[17] Michael Paleczny, Christopher A. Vick, and Cliff Click. The Java HotSpot$^{\text{TM}}$ server compiler. In *Java^{TM} Virtual Machine Research and Technology Symposium*. USENIX, 2001.

[18] Robert W. Scheifler. An analysis of inline substitution for a structured programming language. *Commun. ACM*, 20(9):647–654, 1977.

[19] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 180–195, New York, NY, USA, 2001. ACM Press.

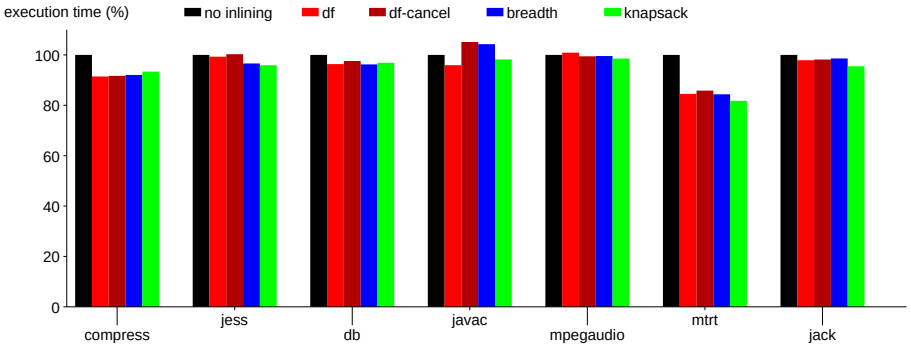

Figure 2: Relative execution times with inlining

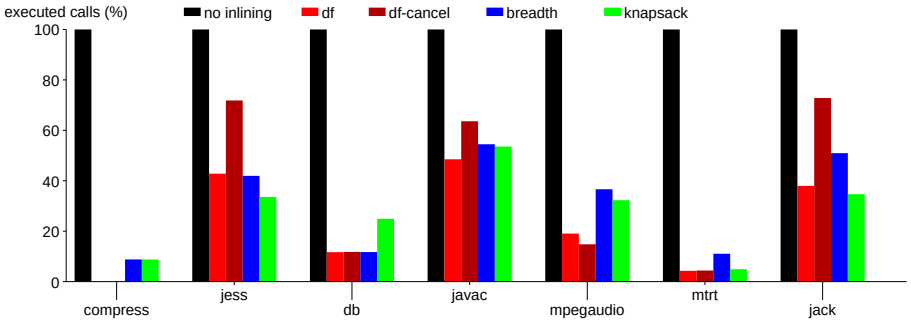

Figure 3: Relative number of executed calls

factor	compress	jess	db	javac	mpegaudio	mtrt	jack
≤ 0.2	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
≤ 0.5	0.0%	1.1%	0.0%	5.9%	0.0%	0.9%	2.0%
≤ 1.0	29.3%	23.2%	26.2%	29.4%	44.0%	35.3%	25.5%
≤ 2.0	41.5%	24.7%	31.0%	31.0%	32.0%	37.9%	25.5%
≤ 5.0	29.3%	40.5%	34.5%	27.9%	21.3%	22.4%	38.7%
≤ 10.0	0.0%	8.9%	6.0%	4.9%	1.3%	2.6%	5.9%
> 10.0	0.0%	1.6%	2.4%	0.9%	1.3%	0.9%	2.5%
geometric mean	1.5	2.0	1.8	1.5	1.4	1.5	1.9
maximal	3.7	24.1	12.5	16.8	13.0	12.2	20.0
total size (unopt)	327376	429301	338273	620963	653582	385663	498019
total size (opt)	375996	640347	422720	1163337	794648	534157	802259
change	+15%	+49%	+25%	+87%	+22%	+39%	+61%

Table 1: Code size change through inlining and recompilation, knapsack heuristics