SAPIENZA
UNIVERSITÀ DI ROMA

New Techniques for Adaptive Program Optimization

School of Information Engineering, Computer Science and Statistics

Dottorato di Ricerca in Ingegneria Informatica – XXVIII Ciclo

Candidate

Daniele Cono D'Elia
ID number 1208297

Thesis Advisor

Prof. Camil Demetrescu

A thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Engineering

March 2016

Abstract

Adaptive optimization techniques are a key ingredient for the performance of modern runtime systems. In this thesis we present novel ideas for adaptive optimization, ranging from profiling techniques (at both intra- and inter- procedural level) based on elegant data structures, to providing support for continuous program optimization through a flexible infrastructure for On-Stack Replacement (OSR).

We implement our ideas in production systems such as the Jikes RVM and the LLVM compiler infrastructure, and evaluate their performance against prominent benchmarks. We also present a first attempt to prove the correctness of on-stack replacement transitions, identifying sufficient conditions for their correctness and devising an algorithm for automatically generating compensation code to support OSR at arbitrary program locations in the presence of common compiler optimizations.

Contents

Chapter 1

Introduction

Translating programming languages into a form that can *efficiently* execute on a target platform is a very challenging problem for computer scientists. Historically, there are two approaches to translation: interpretation and compilation. An interpreter reads the source code of a program, stepping through its expressions to determine which operation to perform next. A compiler instead translates a program into a form that is more amenable to execution, analyzing its source code only once and generating code that would give the same effects as interpreting it.

The two approaches have different benefits in terms of execution speed, portability, footprint, and optimization opportunities. Compiled programs typically execute faster, as a compiler can devote an arbitrary amount of time to *static* (i.e., prior to run-time) code analysis and optimization. On the other hand, an interpreter can access run-time information such as taken control-flow, input parameter values, and variable types, thus enabling optimizations that static compilation would miss. Indeed, this information may be subject to changes across different runs, or may not be obtainable in general through sole source code inspection.

Additionally, the evolution of programming languages over the years has provided software developers with a plethora of useful features such as dynamic typing and class loading, reflection, and closures that may hinder efficient code generation in a static compiler. In response, industry and academia have significantly invested in *adaptive optimization* technology, which consists in observing the run-time behavior of a program in order to drive optimization decisions.

1.1 Context and Motivations

The past two decades have witnessed the widespread adoption of programming languages designed to run on *application virtual machines* (VMs). Compared to statically compiled software, these execution environments provide several advantages from a software engineering perspective, including portability, automatic memory and concurrency management, safety, and ease of implementation for *dynamic* features of a programming language such as adding new code, extending object definitions, and modifying the type system.

Application virtualization technology has been brought to the mainstream market by the Java programming language and later by the Common Language Runtime for the execution of .NET programs. Virtual machines are nowadays available for every popular language, including JavaScript, MATLAB, Python, R, and Ruby.

Modern virtual machines typically implement a mixed-mode execution environment, in which an interpreter is used for executing portions of a program until it becomes profitable to compile them through *just-in-time* (JIT) compilation and continue the execution in the native code. For efficiency reasons, source code is usually translated into an *intermediate representation* (IR) - also known as *bytecode* - that is easier to analyze and process. Multiple levels of JIT compilation are possible, each with a different trade-off between compilation time and expected code quality.

Adaptive optimization technology is a central element for the performance of runtime systems. JIT compilation indeed does not come for free: a virtual machine should be able to exploit run-time information to tailor optimized code generation to the current workload, so that the expected performance gains can counterbalance the overheads coming from collecting the profile and performing the optimizations.

Analyzing the run-time behavior of a program is useful also in the context of statically compiled code. *Profile-guided optimization* (PGO) techniques adopt a dual-compilation model in which a program is compiled and executed on representative input sets during an initial training stage, and is eventually recompiled using feedback information to generate the final optimized version.

1.2 Addressed Problems

Collecting accurate profiling information with a minimal impact on a running program is a key factor for an effective deployment of adaptive optimization techniques. In principle, developers and VM builders may leverage hardware performance counters provided by modern processors to collect low-level profiling information with no impact on the performance of a running program. However, the difficulty in mapping low-level counter data to high-level constructs such as classes and objects discourages their use for implementing complex analyses in runtime systems.

In the past three decades many sophisticated software-based techniques have been proposed for collecting fine-grained information regarding individual statements, objects, or control-flow paths. These techniques are typically based on program instrumentation, sampling, or a combination of both. For some problems, however, the size of the domain can be particularly large and extant techniques do not scale well or may even run out of space when analyzing real-world programs. In this thesis we investigate how data structure-based techniques from the algorithmic community can be used to devise efficient performance profiling tools.

Another key ingredient for adaptive optimization is the ability of a runtime to divert the execution to the newly generated optimized code *while* the original function is executing. In fact, in the presence of long-running methods it is not sustainable for a VM to wait for the original function to complete and let only subsequent invocations run the optimized version. The problem of handling transitions between different compiled versions is formally known as *On-Stack Replacement* (OSR). Modern VMs implement OSR techniques to dynamically replace code with (more) optimized code, and also to invalidate aggressive, speculative optimizations and continue in a safe code version when an assumption made during the compilation no longer holds.

Supporting OSR in a runtime system raises a number of fundamental questions. What is the set of program points at which OSR can safely occur, and how is it affected by compiler optimizations? Can we guarantee the soundness of an OSR transition? What is the impact of OSR machinery on running code?

1.3 Contributions of the Thesis

The contributions of this thesis aim at covering both methodological and practical aspects of the adaptive optimization cycle, ranging from performance profiling to continuous program optimization through online code generation. Methodological contributions of this thesis include:

- an *interprocedural* analysis to identify the most frequently encountered calling context across function invocations, based on data streaming algorithms that enable a reduction of space usage by orders of magnitude without sacrificing accuracy or performance;
- an *intraprocedural* analysis to identify cyclic paths taken across the control-flow graph of a function, overcoming the limitations of previous approaches and enabling the profiling of very long cyclic paths using efficient data structures;
- a new abstraction for on-stack replacement based on compensation code, to enable OSR at places that previous approaches do not support as they expect a new function to resume from the very same state of the original function;
- a first step towards a provably sound methodological framework for OSR, identifying sufficient conditions to determine the set of points where OSR can safely occur and devising an algorithm to automatically generate compensation code in the presence of certain classes of compiler optimizations.

We evaluate the practicability of all our techniques through extensive experimental studies on both industry-strength benchmark and real-world applications. All our techniques have been implemented as libraries for mainstream systems, including the `gcc` compiler, the Jikes RVM, and the LLVM compiler infrastructure, and their source code is publicly available. To back our results and to empower other researchers to build upon the contributions of our papers, we submitted two software artifacts that have been reviewed and endorsed by the scientific community. We believe that some techniques presented in this paper might have some technology transfer potential in VM construction; private conversations we had with developers from major ICT players about our OSR library for LLVM have been quite encouraging in this sense.

1.4 Structure of the Thesis

The rest of this thesis is structured as follows. Chapter 2 reviews the state of the art in techniques for adaptive program optimization. Chapter 3 presents our intra- and inter-procedural techniques for performance profiling, while Chapter 4 tackles the on-stack replacement problem for enabling continuous program optimization. Chapter 5 illustrates the results of our experimental studies. Chapter 6 presents examples of application of our techniques in program optimization. Conclusions and possible directions for future work are discussed in Chapter 7.

Chapter 2

State of the Art

2.1 Performance Profiling

2.1.1 Means for Collecting Profiling Information

2.1.2 Intra-Procedural Profiling Techniques

2.1.3 Inter-Procedural Profiling Techniques

2.2 Optimized Code Generation

2.2.1 Profile-Guided Optimization

2.2.2 Just-In-Time Compilation

2.2.2.1 On-Stack Replacement

2.2.2.2 Tracing JIT Compilation

2.2.3 Other Related Work

Chapter 3

Performance Profiling Techniques

In this chapter, we present two run-time analyses for collecting fine-grained profiling information based on efficient and elegant algorithmic techniques. The first analysis is *interprocedural* and focuses on identifying the calling contexts of function invocations that are most frequently encountered during the execution of a program. The second analysis works at *intraprocedural* level and identifies cyclic paths that are taken in the control-flow graph of a procedure, thus spanning multiple loop iterations. Both techniques can provide valuable information for program understanding and performance analysis, as they can be used to direct optimizations to portions of the code where most resources are consumed.

3.1 Mining Hot Calling Contexts in Small Space

The first contribution we present in this thesis is an interprocedural technique for mining the most frequently encountered calling contexts for function invocations at run time. We show that the traditional approach of constructing a *Calling Context Tree* (CCT) might not be sustainable for real-world applications, as their CCTs often consist of tens of millions of nodes, making them difficult to analyze and also hurting execution time because of poor access locality. We thus introduce a novel data structure, the *Hot Calling Context Tree* (HCCT), in the spectrum of representations for interprocedural control flow. The HCCT is defined as the subtree of the CCT containing only its most frequently visited nodes, which we call *hot*, and their ancestors. We show how to construct it independently of the CCT using fast, space-efficient algorithms for mining frequent items in data stream.

3.1.1 Motivation and Contributions

The dynamic *calling context* of a routine invocation is defined as the sequence of functions that are concurrently active on the run-time stack at the time of the call. A calling context leads to an exact program location: in fact, it corresponds to the sequence of un-returned calls from a program's root function to the routine invocation of interest.

Context-sensitive profiling information provides valuable information for program understanding, performance analysis, and run-time optimizations. Many works have demonstrated its effectiveness for tasks such as residual testing [83, 103], function inlining [27], statistical bug isolation [43, 70], performance bug detection [80], object allocation analysis [79], event logging [112], and anomaly-based intrusion detection [22]. Calling-context information has also been employed in reverse engineering of protocol formats [71], unit test generation [104], and testing of sensor network applications [65].

Application	|Call graph|	Call sites	|CCT|	|Call tree|
amarok	13 754	113 362	13 794 470	991 112 563
ark	9 933	76 547	8 171 612	216 881 324
audacity	6 895	79 656	13 131 115	924 534 168
bluefish	5 211	64 239	7 274 132	248 162 281
dolphin	10 744	84 152	11 667 974	390 134 028
firefox	6 756	145 883	30 294 063	625 133 218
gedit	5 063	57 774	4 183 946	407 906 721
ghex2	3 816	39 714	1 868 555	80 988 952
gimp	5 146	93 372	26 107 261	805 947 134
gwenview	11 436	86 609	9 987 922	494 753 038
inkscape	6 454	89 590	13 896 175	675 915 815
oocalc	30 807	394 913	48 310 585	551 472 065
ooimpress	16 980	256 848	43 068 214	730 115 446
oowriter	17 012	253 713	41 395 182	563 763 684
pidgin	7 195	80 028	10 743 073	404 787 763
quanta	13 263	113 850	27 426 654	602 409 403
sudoku	5 340	49 885	2 794 177	325 944 813
vlc	5 692	47 481	3 295 907	125 436 877
botan	3 388	27 114	308 550	26 272 804 980
cairo-perf-trace	1 408	3 696	137 920	15 976 619 734
crafty	107	516	36 434 095	10 403 074 070
fhourstones	18	32	OOM	39 272 563 944
gobmk	1 133	4 049	OOM	21 909 088 291
ice-labyrinth	2 335	8 050	2 160 052	1 637 076 406
mount-herring	2 318	8 269	3 733 120	3 311 257 932
overworld	14 173	50 394	3 774 937	4 112 679 880
scotland	13 932	51 206	1 813 368	5 982 612 379
sjeng	57	221	OOM	28 370 207 811

Table 3.1. Number of nodes of call graph, call tree, calling context tree, and number of distinct call sites for different applications. OOM stands for *Out Of Memory* (i.e., the CCT is too large to be constructed in main memory on a 32-bit architecture). We provide a detailed description of the applications and their workloads in Section **XXX**.

Calling context trees (CCTs) offer a compact representation for context-sensitive information. In fact, a CCT yields a more accurate profile than a *call graph* (which can sometimes drive to misleading conclusions [87, 94]) in a space that is typically several orders of magnitude smaller than that required to maintain a *call tree*. Also, many techniques have been proposed over the years to reduce the overhead for its construction.

However, even CCTs may be very large and difficult to analyze in several applications [22, 113]; their sheer size might also hurt execution time, because of poor access locality during construction and query. As an example, we report in Table 3.1 numbers collected for short usage sessions of off-the-shelf Linux applications and for benchmarks from popular suites. Under the optimistic assumption that each CCT node requires 20 bytes for its representation on a 32-bit architecture[1], nearly 1 GB of memory is needed just to maintain OpenOffice Calc's 48-million-node CCT.

In a performance profiling scenario, we remark that only the most frequently encountered contexts are of interest, as they represent the hot spots to which optimizations must be directed. As observed in [113]: "Accurately collecting information about hot edges may be more useful than accurately constructing an entire CCT that includes rarely called paths".

In Figure 3.1 we report the cumulative distribution of calling contexts for different applications, using frequency counts as metric. We observe that for all the applications only a small fraction of contexts are hot: in conformance with the Pareto principle, nearly 90% of routine calls take place in only 10% of contexts. The skewness of the distribution suggests that space could be greatly reduced by keeping information about hot contexts only and discarding on the fly likely cold (i.e., having low frequency) contexts.

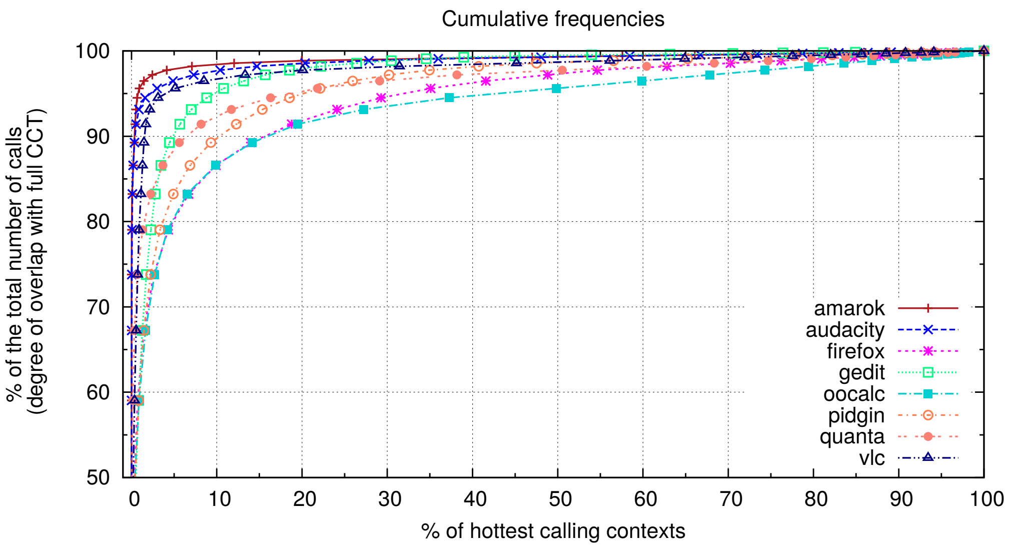


Figure 3.1. Skewness of calling contexts distribution on a representative subset of applications. For instance, in `oocalc`, 10% of the hottest calling contexts account for more than 86% of all routine calls.

Contributions. In this thesis, we introduce a novel run-time data structure, called *Hot Calling Context Tree (HCCT)*, that compactly represents all the hot calling contexts encountered during a program's execution, offering an additional intermediate point in the spectrum of data structures for representing interprocedural

[1]From maintaining routine ID, call site, and a performance metric as `int` fields, along with two pointers for a first-child, next-sibling tree representation. Previous works [4, 94] use larger nodes.

control flow. The HCCT is a subtree of the CCT that includes only hot nodes and their ancestors, also maintaining estimates of performance metrics (e.g., frequency counts) for hot calling contexts. We cast the problem of identifying the most frequent contexts into a data streaming setting: we show that the HCCT can be computed without storing the exact frequency of all calling contexts, by using fast and space-efficient algorithms for mining frequent items in data streams. These algorithms allow us to distinguish between hot and cold contexts on the fly, and typically provide tight guarantees on the accuracy of returned frequency estimates.

3.1.2 Approach

Background. A *calling context tree* (CCT) can be used to compactly represent all the calling contexts encountered during the execution of a program. In fact, calling contexts can be straightforwardly mapped to paths in a tree: nodes represent un-returned function calls, and each path from the root to a node v encodes the calling context of the call associated with v. As in a tree the path from the root to any other node is always unique, we can also say that each calling context is uniquely represented by a node, which aggregates metrics for identical contexts recurring in the execution. Note that a routine with multiple calling contexts will instead appear more than once in the tree. Slightly extended CCT definitions can be given to bound its depth in the presence of direct recursion, and to distinguish calls that take place at different call sites of the same calling procedure [4].

Introducing the HCCT. In order to introduce the hot calling context tree, we have first to define when a context can be called hot. Let N be the number of calling contexts encountered during a program's execution: N equals the number of nodes of the call tree, the sum of the frequency counts of CCT nodes, as well as the number of routine invocations in the execution trace.

Definition 1 *A calling context is* hot *with respect to a frequency threshold* $\phi \in [0, 1]$ *if and only if the frequency count of its corresponding CCT node is* $\geq \lfloor \phi N \rfloor$.

Any calling context that is not hot is said to be *cold*.

Definition 2 *The* Hot Calling Context Tree (HCCT) *is the (unique) subtree of the CCT obtained by pruning all cold nodes that are not ancestors of a hot node.*

In graph theory, the HCCT corresponds to the Steiner tree of the CCT with hot nodes and the root used as terminals, i.e., to the minimal connected subtree of the CCT spanning hot nodes and the root. The HCCT includes all the hot nodes, and all its leaves are necessarily hot. An example of HCCT is given in Figure 3.2(b).

A Data Streaming Problem

The execution trace of routine invocations and terminations can be naturally regarded as a stream of items. Each item is a triple containing routine name, call site, and event type (i.e., routine invocation or termination). Figures reported in Table 3.1 indicate that, even for complex applications, the number of distinct routines (i.e.,

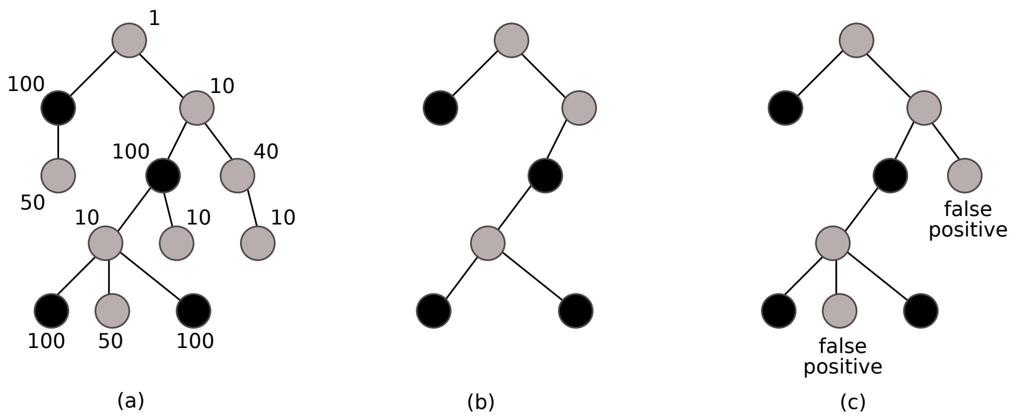


Figure 3.2. (a) CCT annotated with calling-context frequency counts; (b) HCCT; and (c) (ϕ, ε)-HCCT. Hot nodes are black. In this example $N = 581$, $\phi = 1/10$, and $\varepsilon = 1/30$: the approximate HCCT includes all contexts with frequency $\geq \lfloor \phi N \rfloor = 58$ and no context with frequency $\leq \lfloor (\phi - \varepsilon) N \rfloor = 38$.

the number of nodes of the call graph) is small compared to the stream length (i.e., to the number of nodes of the call tree). Hence, non-contextual profilers – such as vertex profilers – can easily collect performance metrics for all the routines using a hash table. This may not be sustainable for contextual profiling when the number of distinct calling contexts (i.e., the number of CCT nodes) is too large, and hashing would be inefficient. Motivated by the fact that execution traces are typically very long and their items (calling contexts) are taken from a large universe, we cast the problem of identifying the most frequent contexts into a data streaming setting.

In the data streaming computational model, algorithms should be able to perform near-real time analyses on massive data streams, where input data come at a very high rate and cannot be stored entirely due to their huge, possibly unbounded size [38, 77]. This line of research has been mainly motivated by networking and database applications: for instance, a relevant IP traffic analysis task consists in monitoring the packet log over a given link in order to estimate how many distinct IP addresses used that link in a given period of time. Space-efficient data streaming algorithms can maintain a compact data structure that is dynamically updated upon arrival of new input data, supporting a variety of application-dependent queries. Approximate answers are allowed when it is impossible to obtain an exact solution using only limited space. Streaming algorithms are therefore designed to optimize space usage and update/query time while guaranteeing high solution quality [77].

We remark that the practical requirements for the design of effective dynamic analysis tools – which have to collect and process large amounts of data in nearly real time and with a minimal impact on the running program – make it natural to look for the connections between these two research areas.

Finding Frequent Items in a Stream. The problem of computing the HCCT online can be reconducted to the *frequent items* (a.k.a. heavy hitters) problem, which has been extensively studied in the data streaming model. Given a frequency threshold $\phi \in [0, 1]$ and a stream of length N, the problem (in its simplest formulation) is to find all items that appear in the stream at least $\lfloor \phi N \rfloor$ times, i.e., having frequency $\geq \lfloor \phi N \rfloor$. For instance, for $\phi = 0.1$ the problem seeks all items that appear in the stream at least 10% of the times. Notice that at most $1/\phi$ items can have frequency larger than $\lfloor \phi N \rfloor$. It can be proved that any algorithm that outputs an exact solution requires $\Omega(N)$ bits, even using randomization [77]. Note that this lower bound result extends to the problem of computing the HCCT, which cannot be calculated exactly in a space asymptotically smaller than the entire CCT. Hence, researchers have focused on solving an approximate version of the heavy hitters problem [77]:

Definition 3 *(ϕ, ε)-heavy hitters problem. Given two parameters $\phi, \varepsilon \in [0, 1]$, with $\varepsilon < \phi$, an algorithm has to return all items with frequency $\geq \lfloor \phi N \rfloor$ and no item with frequency $\leq \lfloor (\phi - \varepsilon) N \rfloor$.*

In the approximate solution, *false negatives* are not allowed, i.e., all frequent items must be returned. Instead, some *false positives* can exist, but their actual frequency is guaranteed to be at most εN-far from the threshold $\lfloor \phi N \rfloor$. For the HCCT construction, we focus on a variant of the problem where, besides returning the heavy hitters, it is necessary to estimate their true frequencies accurately, the stream length N is not known in advance, and all the items in the stream have equal weight.

Counter-based streaming algorithms solve this problem by tracking a subset of items from the input and monitoring counts associated with them. For each new arrival, the algorithms decide whether to store the item or not, and, if so, what count to associate with it. Update times are typically dominated by a small (constant) number of dictionary or heap operations. These algorithms, according to extensive experimental studies [33, 74], have superior performance with respect to space, running time, and accuracy compared to other classes of algorithms for (ϕ, ε)-heavy hitters that have been proposed in the last 15 years.

Approximating the HCCT

Streaming algorithms for mining frequent items can be used to solve a relaxed version of the HCCT construction problem. We thus rely on them to compute an *Approximate Hot Calling Context Tree* that we denote by (ϕ, ε)-*HCCT*, where $\varepsilon < \phi$ controls the degree of approximation:

Definition 4 *Given a set A of (ϕ, ε)-heavy hitters, the* (ϕ, ε)-HCCT *is the minimal connected subtree of the CCT spanning all the nodes in A and their ancestors.*

A (ϕ, ε)-HCCT contains all hot nodes (*true positives*), but may possibly contain some cold nodes without hot descendants (*false positives*). The true frequency of these false positives, however, is guaranteed to be at least $\lfloor (\phi - \varepsilon) N \rfloor$. Unlike the HCCT, a (ϕ, ε)-HCCT is not uniquely defined, since the set of (ϕ, ε)-heavy hitters is not unique: nodes with frequencies smaller than $\lfloor \phi N \rfloor$ and larger than $\lfloor (\phi - \varepsilon) N \rfloor$ may be either in such a set or not depending on the streaming algorithm's decisions. On the other hand, the HCCT is always a subtree of any (ϕ, ε)-HCCT, as the latter always contains all the hot nodes and their cold ancestors up to the CCT root.

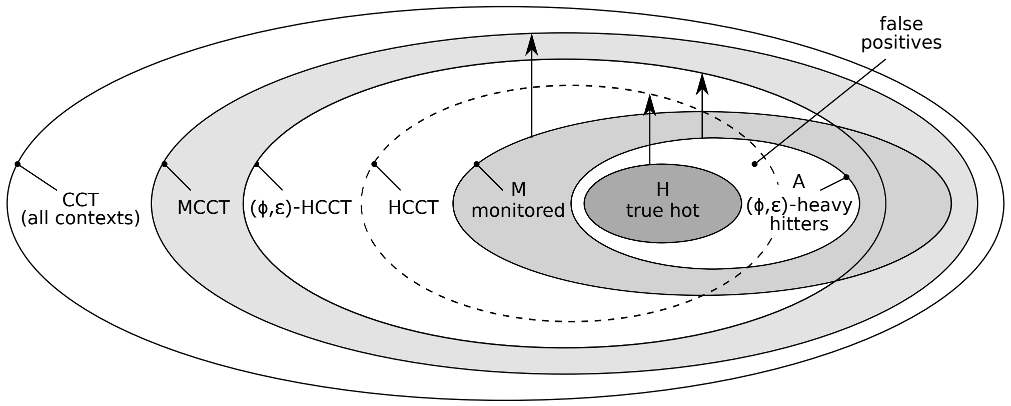


Figure 3.3. Tree data structures and calling contexts classification. We use the graphical notation S $\uparrow$ T to indicate that T is the minimal subtree of the CCT spanning all nodes in S and their ancestors.

3.1.3 Algorithms

Computing a (ϕ, ε)-HCCT online requires extending the canonical CCT construction algorithm with an online pruning strategy, driven by an underlying streaming routine. Constructing a CCT on-the-fly during the execution of a program is rather simple. Let v be a cursor pointer that points to the current context, i.e., to the node corresponding to the calling context of the currently active routine (v is initialized to the CCT root node). At each routine invocation, the algorithm checks whether v has a child associated with the called routine. If this is the case, the existing child is used and its metrics are updated, if needed. Otherwise, a new child of v is added to the CCT. In both cases, the cursor is moved to the callee. Upon routine termination, the cursor is moved back to the parent node in the CCT. This approach can be implemented by either instrumenting every routine call and return, or performing stack-walking when sampling is used to inhibit redundant profiling [7, 105, 113].

In order to computer the set A of (ϕ, ε)-heavy hitters, counter-based streaming algorithms need to monitor a slightly larger set M $\supseteq$ A of elements. Nodes in M $\setminus$ A can be either ancestors of nodes in A and thus already in the (ϕ, ε)-HCCT, or nodes not in the (ϕ, ε)-HCCT; for the latter category, we have to retain information about their ancestors as well, which might not be in the (ϕ, ε)-HCCT. We denote as MCCT the minimal subtree of the CCT spanning all the nodes in M and the CCT root. Figure 3.3 graphically illustrates the relationships among all our data structures.

Our (ϕ, ε)-HCCT construction algorithm dynamically maintains the MCCT while the underlying streaming routine processes the execution trace and updates M. At query time, the streaming algorithm analyzes M to discard all the elements in M $\setminus$ A: the MCCT is thus pruned appropriately and the (ϕ, ε)-HCCT $\subseteq$ MCCT is returned.

Example 1 *To understand why the heavy hitters and the approximate HCCT are not maintained directly, but derived by pruning M and MCCT, respectively, we discuss a scenario where M is larger than the number of heavy hitters. Consider the following example: the execution trace contains the initial invocation of the* `main` *function, which in turn invokes once a routine p, and $N-2$ times a different routine q. Hence, we have three distinct calling contexts:* `main`*,* `main`$\rightarrow$`p`*, and,* `main`$\rightarrow$`q`*.*

Assume that $N \geq 8$, $\varepsilon = 1/4$, $\phi = 1/2$, and that the counter-based streaming subroutine can maintain three counters, one for each calling context. Then, only context `main`$\rightarrow$`q` *has frequency larger than $\lfloor(\phi - \varepsilon)N\rfloor$ and is a (ϕ, ε)-heavy hitter, but – as we assumed there is room in M for all contexts – a streaming algorithm can maintain the exact frequencies of both* `main`$\rightarrow$`p` *and* `main`$\rightarrow$`q`*. Since* `main`$\rightarrow$`p` *has frequency 1, it would be an error returning it as a heavy hitter. For this reason, M needs to be post-processed in order to eliminate low-frequency items that may be included when there are more available counters than heavy hitters.*

Data Structure Operations

At each function call, the set M of monitored contexts is updated by a counter-based streaming algorithm. When M is changed, the subtree MCCT spanning nodes in M needs to be brought up to date as well. To describe how this happens, we assume that the interface of the streaming algorithm provides two main functions:

`update(x,M)`$\rightarrow$`V` Given a calling context x, update M to reflect the new occurrence of x in the stream (e.g., if x was already monitored in M, its frequency count may be increased by one). This function might return a set V of *victim* contexts that were previously monitored in M and are evicted during the update (as a special case, x itself may be considered as a victim if the algorithm chooses not to monitor it).

`query(M)`$\rightarrow$`A` Remove low-frequency items from M and return the subset A of (ϕ, ε)-heavy hitters (see Figure 3.3).

As with the CCT, during the construction of the MCCT we maintain a cursor pointer that points to the current calling context, creating a new node if the current context x is not already represented in the tree. Additionally, we prune the MCCT according to the victim contexts returned by the streaming `update` operation (these contexts are no longer monitored in M). The pseudocode of the pruning algorithm is given in Algorithm 1. Since the tree must remain connected, victims can be removed from the MCCT only if they are leaves. Moreover, removing a victim might expose a path of unmonitored ancestors that no longer have descendants in M: these nodes

Input: M; MCCT; node x to be pruned.
Output: Pruned MCCT.

```
1  $V \leftarrow$ update(x, M);
2  foreach context $v \in V \setminus \{x\}$ do
3      while ($v$ is a leaf in MCCT) $\wedge$ ($v \notin M$) do
4          remove $v$ from MCCT
5          $v \leftarrow$ parent($v$)
6      end
7  end
```

Algorithm 1: Online pruning algorithm for MCCT construction.

are pruned as well. The node for the current context x is never removed from the MCCT, even if the context is not necessarily monitored in M. This guarantees that no node in the path from the tree root to x will be removed: these nodes have at least x as a descendant and the leaf test (line 3 in Algorithm 1) will always fail.
A similar pruning strategy can be used to compute the (ϕ, ε)-HCCT from the MCCT. The streaming `query` operation is first invoked on M, returning the support A of the (ϕ, ε)-HCCT. All MCCT nodes that have no descendant in A are then removed, following bottom-up path traversals as in the prune operation.

Choosing a Streaming Algorithm

Space-Saving [76] is a deterministic, counter-based algorithm for finding the heavy hitters and the top-k elements [28] in data streams. The algorithm is memory-efficient, as its space requirements are within a constant factor of the lower bound for counter-based algorithms solving the (ϕ, ε)-heavy hitters problem. Additionally, Space-Saving provides tight error guarantees on maintained frequency estimates. In an earlier work [37], we presented a thorough experimental evaluation of the *Lossy Counting* [75] algorithm, resulting in similar accuracy but higher running time and memory usage compared to Space-Saving. Experimental studies [33, 74] also show that Space-Saving outperforms other counter-based algorithms across a wide range of data sets and parameters. For all these reasons, in the remainder of this thesis we will focus on the Space-Saving algorithm only.

Space-Saving monitors a set of $1/\varepsilon = |M|$ pairs of the form $(item, count)$, initialized by the first $1/\varepsilon$ distinct items and their exact counts. After the init phase, when a context c is observed in the stream the `update` operation works as follows:

1. if c is monitored, the corresponding counter is incremented;
2. if c is not monitored, the $(item, count)$ pair with the smallest count is chosen as a victim and has its item replaced with c and its count incremented.

It can be shown that the minimum counter value *min* among monitored items is never greater than εN, and that the *count* maintained for an item is an overestimation of its true frequency by at most *min*. This overestimation derives from the initial assignment to *count* from the evicted pair. Observe that items that are stored early in the stream and never removed will have very accurate frequency estimates.

The update time is bounded by the dictionary operation of checking whether an item is monitored, and by the operations of finding and maintaining the item with minimum count. In our setting, we can avoid the dictionary operation by maintaining a flag for each tree node, which can directly be accessed for the current context using the cursor pointer to the MCCT.

Space-Saving answers `query` operations by simply returning entries in M such that $count \geq \lfloor \phi N \rfloor$; associated frequency estimates are guaranteed to be at most εN far from actual frequencies.

Engineering Space-Saving. In [76], the authors present an implementation of Space-Saving based on the *Stream-Summary* data structure, which is essentially an ordered bucket list where each bucket points to a list of items with the same count, and buckets are ordered by increasing count values.

We devise a more efficient variant based on a lazy priority queue that uses an unordered array M of size $1/\varepsilon$, where each entry points to an MCCT node. The queue supports two operations, `find-min` and `increment`, which return the item with minimum count and increment a counter, respectively.

We (lazily) maintain the value `min` of the minimum counter and the smallest index `min-idx` of an array entry that points to a monitored node with counter equal to `min`. The `increment` operation does not change M, since counters can be stored directly inside MCCT nodes. However, `min` and `min-idx` may become temporarily out of date after an `increment`: this is why we call the approach lazy. The `find-min` operation described in Algorithm 2 restores the invariant property on `min` and `min-idx`: it finds the next index in M with counter equal to `min`. If such an index does not exist, it completely rescans M in order to find a new `min` value and its corresponding `min-idx`.

Input: M, `min`, and `min-idx`.
Output: Min value in the lazy priority queue.

```
while ($M[min\text{-}idx] \neq min$ ) $\wedge$ ( $min\text{-}idx \leq M$) do
    min-idx ← min-idx +1
end
if $min\text{-}idx > M$ then
    min ← minimum in $M$
    min-idx ← smallest index $j$ s.t. $M[j]$ = min
end
return $min$
```

Algorithm 2: `find-min` operation used in *lazy* Space Saving.

By proving that `find-min` requires constant amortized time, we show that an `update` operation can be performed in constant time during the MCCT construction:

Theorem 1 *After a* `find-min` *query, the lazy priority queue correctly returns the minimum counter value in O(1) amortized time.*

Proof. Counters are never decremented. Hence, at any time, if a monitored item with counter equal to `min` exists, it must be found in a position larger than or equal to `min-idx`. This yields correctness.

To analyze the running time, let Δ be the value of `min` after k `find-min` and `increment` operations. Since there are $|M|$ counters $\geq \Delta$, counters are initialized to 0, and each `increment` operation adds 1 to the value of a single counter, it must be $k \geq |M| \cdot \Delta$. For each distinct value assumed by `min`, the array is scanned twice. We therefore have at most $2 \cdot \Delta$ array scans each of length $|M|$, and the total cost of `find-min` operations throughout the whole sequence of operations is upper bounded by $2 \cdot |M| \cdot \Delta$. It follows that the amortized cost is $(2 \cdot |M| \cdot \Delta)/k \leq 2$. □

Using a simple amortized analysis argument, it can be shown that the running time of Algorithm 1 for tree pruning is constant as well.

3.1.4 Discussion

Compared to the standard approach of maintaining the entire CCT, our solution requires storing the heavy hitters data structure M and the subtree MCCT spanning nodes in M. The space required by M depends on the specific streaming algorithm that is used as a subroutine and on the value chosen for the error threshold ε. For Space-Saving this space is proportional to $1/\varepsilon$ and can be customized by appropriately choosing ε, e.g., according to the desired accuracy or to the amount of memory available for profiling. Such a choice appears to be crucial for the effectiveness of our approach: smaller values of ε guarantee more accurate results (i.e., fewer false positives and more precise counters), but imply a larger memory footprint. In Section **XXX** we will see that the high skewness of context frequency distribution guarantees the existence of very convenient tradeoffs between accuracy and space in the analysis of real-world programs.

The MCCT consists of nodes corresponding to contexts monitored in M and all their ancestors, which may be cold contexts without a corresponding entry in M. Hence, the space required by the MCCT dominates the space required by M. The number of cold ancestors is difficult to analyze theoretically: it depends on properties of the execution trace and on the structure of the CCT. In Section **XXX** we will see that in practice this amount is negligible compared to the size of M.

Updates of the MCCT can be performed very quickly. We propose an engineered implementation of Space-Saving that hinges upon very simple and cache-efficient data structures, and might be of independent interest.

Unlike previous approaches such as, e.g., adaptive bursting [113], the MCCT adapts automatically to the case where the hot calling contexts vary over time, and new calling patterns are not likely to be lost. Contexts that are growing more popular are added to the tree as they become more frequent, while contexts that lose their popularity are gradually replaced by hotter contexts and are finally discarded. This guarantees that heavy hitters queries can be issued at any point in time, and will always be able to return the set of hot contexts up to that time.

3.1.5 Comparison with Related Work

CCTs have been introduced in [4] as a practical data structure to associate performance metrics with paths through a program's call graph: Ammons, Ball, and Larus suggest to build a CCT by instrumenting procedure code and to compute metrics by exploiting hardware counters available in modern processors. It has been later observed, however, that exhaustive instrumentation can incur large slowdowns.

Reducing Overhead. To reduce the overhead from instrumentation, in [13] the authors generate path profiles including only methods of interest, while statistical profilers [7, 47, 53, 105] attribute metrics to calling contexts through periodic sampling of the call stack. For call-intensive programs, sample-driven stack-walking can be orders of magnitude faster than exhaustive instrumentation, but may incur significant loss of accuracy with respect to the complete CCT: sampling guarantees neither high coverage [22] nor accuracy of performance metrics [113], and its results may be highly inconsistent in different executions.

A variety of works explores the combination of sampling with bursting [6, 56, 113]. Most recently, Zhuang *et al.* suggest to perform stack-walking followed by a burst during which the profiler traces every routine call and return [113]: experiments show that adaptive bursting can yield very accurate results. In [92], the profiler infrequently collects small call traces that are merged afterwards to build large calling context trees: ambiguities might emerge during this process, and the lack of information about where the partial CCTs should be merged to does not allow a univocal reconstruction of the entire CCT.

The main goal of all these works is to reduce profiling overhead without incurring significant loss of accuracy. Our approach is orthogonal to this line of research and regards space efficiency as an additional resource optimization criterion besides profile accuracy and time efficiency. When the purpose of profiling is to identify hot contexts, exhaustive instrumentation, sampling, and bursting might all be combined with our approach and benefit of our space reduction technique. In Section **XXX** we present an integration of our technique with static bursting [113], which results in faster running times without substantially affecting accuracy.

Reducing Space. A few previous works have addressed techniques to reduce profile data (or at least the amount of data presented to the user) in context-sensitive profiling. Incremental call-path profiling lets the user choose a subset of routines to be analyzed [13]. Call path refinement helps users focus the attention on performance bottlenecks by limiting and aggregating the information revealed to the user [52]. These works are quite different in spirit from our approach, where only hot contexts are profiled and identified automatically during program's execution.

Probabilistic calling contexts have been introduced as an extremely compact representation (just a 32-bit value per context), especially useful for tasks such as residual testing, statistical bug isolation, and anomaly-based intrusion detection [22]. Bond and McKinley target applications where coverage of both hot and cold contexts is necessary, but their inspection is unnecessary. This is not the case in performance analysis, where identifying and understanding a few hot contexts is typically sufficient to guide code optimization. Hence, although sharing with [22] the common goal of space reduction, our approach targets a rather different application context.

Somner et al. proposed a technique called *Precise Calling Context Encoding* (PCCE) that encodes acyclic paths in the call graph of a program into one number, while recursive call paths are divided into acyclic subsequences and encoded independently [97, 98]. Different calling contexts are guaranteed to have different IDs that can be faithfully decoded, and experiments on a prototype implementation for C programs show negligible overhead. However, PCCE would not work in the presence of virtual methods and dynamic class loading in object-oriented languages, and the encoding scheme shows scalability problems when handling large-scale software [23, 111]. These limitations, which are absent from our solution, have been recently addressed in [111], and it would be interesting to investigate whether their approach can be extended to collect performance metrics and in turn filter the collected data on-the-fly using a data streaming approach as we do in this thesis.

3.2 Multi-iteration Path Profiling

Path profiling is a powerful *intraprocedural* methodology for identifying performance bottlenecks in a program, and has received considerable attention in the last 15 years for its practical relevance. The well-known Ball-Larus numbering algorithm [10] can efficiently encode *acyclic* paths that are taken across the control flow graph of a function. Previous attempts to extend it to *cyclic* paths – thus spanning multiple loop iterations – to capture more optimization opportunities, are based on rather complex algorithms that incur severe performance overheads even for short cyclic paths. In this thesis we present a new, data-structure based approach to *multi-iteration* path profiling built on top of the original Ball-Larus numbering technique. Starting from the observation that a cyclic path can be described as a concatenation of Ball-Larus acyclic paths, we show how to accurately profile all executed paths obtained as a concatenation of up to k Ball-Larus paths, where k is a user-defined parameter.

3.2.1 Motivation and Contributions

Path profiling associates performance metrics, usually frequency counters, to paths taken in the control flow graph of a routine. Identifying the hottest paths can direct optimizations to portions of the code where most resources are consumed, often yielding significant speedups. For instance, trace scheduling can be used to increase instruction-level parallelism along frequently executed paths [45, 108]. Basic-block and edge profiles, albeit inexpensive and widely available, may not correctly predict frequencies of overlapping paths, and are thus inadequate for such optimizations.

The seminal paper by Ball and Larus [10] introduced a simple and elegant path profiling technique. The main idea was to implicitly number all possible acyclic paths in the control flow graph so that each path is associated with a unique compact path identifier (ID). The authors showed that path IDs can be efficiently generated at run time and can be used to update a table[2] of frequency counters. Although in general the number of acyclic paths may grow exponentially with the graph size, in typical control flow graphs this number is usually small enough to fit in current machine word-sizes, making this approach very effective in practice.

While the original Ball-Larus approach was restricted to acyclic paths obtained by cutting paths at loop back edges, profiling paths that span consecutive loop iterations is a desirable, yet difficult, task that can yield better optimization opportunities. Consider, for instance, the problem of eliminating redundant executions of instructions, such as loads and stores [17], conditional jumps [15], expressions [16, 19], and array bounds checks [18]. A typical situation is that the same instruction is redundantly executed at each loop iteration, which is particularly common for arithmetic expressions and load operations [19, 17]. To identify such redundancies, paths that extend across loop back edges need to be profiled. Another application is trace scheduling [108]: if a frequently executed cyclic path is found, compilers may unroll the loop and perform trace scheduling on the unrolled portion of code. The benefits of multi-iteration path profiling are discussed in depth in [101].

[2]Large routines can have too many potential paths to use an array of counters. In this case, a slower (but more space-efficient) hash table is used to record only paths that actually execute [10].

Different authors have proposed techniques to profile cyclic paths by modifying the original Ball-Larus path numbering scheme in order to identify paths that extend across multiple loop iterations [101, 90, 69]. Unfortunately, all known solutions require rather complex algorithms that incur severe performance overheads even for short cyclic paths, leaving the interesting open question of finding simpler and more efficient alternative methods.

Contributions. In this thesis, we present a novel, data structure-based approach to multi-iteration path profiling. Our method stems from the observation that any cyclic path in the control flow graph of a routine can be described as a concatenation of Ball-Larus acyclic paths (BL paths). In particular, we show how to accurately profile *all executed paths* obtained as a concatenation of up to k BL paths, where k is a user-defined parameter. We reduce multi-iteration path profiling to the problem of counting n-grams, i.e., contiguous sequences of n items from a given sequence. To compactly represent collected profiles, we organize them in a forest of prefix trees (or tries) [46] of depth up to k, where each node is labeled with a BL path, and paths in a tree represent concatenations of BL paths that were actually executed by the program, along with their frequencies. We also present an efficient construction algorithm based on a variant of the k-SF data structure presented in [8].

3.2.2 Approach

Differently from previous techniques [101, 90, 69], which rely on modifying the Ball-Larus path numbering to cope with cycles, our method does not require any modification of the original numbering technique described in [10].

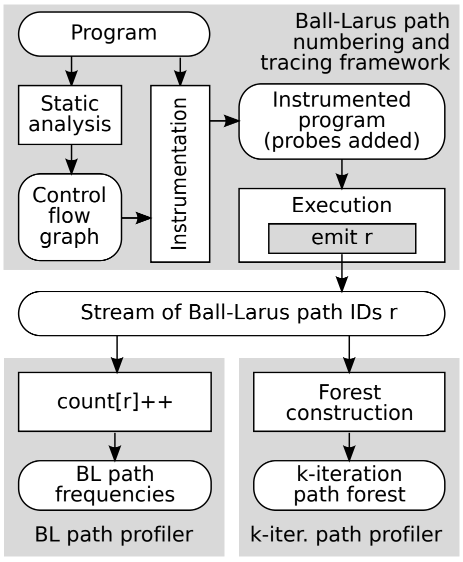


Figure 3.4. Overview of our approach: Ball-Larus profiling versus k-iteration path profiling, cast in a common framework.

The main idea behind our approach is to fully decouple the task of tracing Ball-Larus acyclic paths at run time from the task of concatenating and storing them in a data structure to keep track of multiple iterations.

Figure 3.4 illustrates from a high-level point of view our two-stage process:

1. instrumentation and execution of the program to be profiled (top);
2. profiling of paths (bottom).

We let the Ball-Larus profiling algorithm issue a stream of BL path IDs, where each ID is generated when a back edge in the control flow graph is traversed or the current procedure is abandoned. As a consequence of this modular approach, our method can be implemented on top of existing Ball-Larus path profilers, making it simpler to code and maintain.

The first phase is almost identical to the original approach described in [10]. The target program is statically analyzed and a control flow graph (CFG) is constructed for each routine of interest. The CFG is used to instrument the original program by inserting probes, which allow paths to be traced at run time. When the program is executed, taken acyclic paths are identified using the inserted probes. The main difference with the Ball-Larus approach is that, instead of directly updating a table of frequency counters here, we emit a stream of path IDs, which is passed along to the next stage of the process. This allows us to decouple the task of tracing taken paths from the task of profiling them.

The profiling phase can be either the original hash table-based method of [10] used to maintain BL path frequencies (bottom-left of Figure 3.4), or other approaches such as the one we propose, i.e., profiling concatenations of BL paths in a forest-based data structure (bottom-right of Figure 3.4). Different profiling methods can be therefore cast into a common framework, increasing flexibility and helping us make more accurate comparisons.

We start the description of our approach with a brief overview of the Ball-Larus path tracing technique, which we use as the first stage of our profiling technique.

Ball-Larus Path Tracing Algorithm

The Ball-Larus path profiling (BLPP) technique [10] identifies each acyclic path that is executed in a routine. Paths start on the method entry and terminate on the method exit. Since loops make the CFG cyclic, loop back edges are substituted by a pair of dummy edges: the first one goes from the method entry to the target of the loop back edge, and the second one from the source of the loop back edge to the method exit. After this transformation, which preserves the number of acyclic paths and is reversible, the CFG of a method becomes a directed acyclic graph (DAG), and acyclic paths can be easily enumerated.

```
procedure bl_path_numbering():
foreach basic block $v$ in reverse topological order do
    if $v$ is the exit block then
        numPaths($v$) $\leftarrow 1$
    else
        numPaths($v$) $\leftarrow 0$
        foreach outgoing edge $e = (v, w)$ do
            val($e$) = numPaths($v$)
            numPaths($v$) += numPaths($w$)
        end
    end
end
```

Algorithm 3: The Ball-Larus path numbering algorithm.

The Ball-Larus path numbering algorithm (Algorithm 3) assigns a value $val(e)$ to each edge e of the CFG such that, given N acyclic paths, the sum of the edge values along any entry-to-exit path is a unique numeric ID in [0, N-1]. A CFG example and the corresponding path IDs are shown in Figure 3.5: notice that there are eight

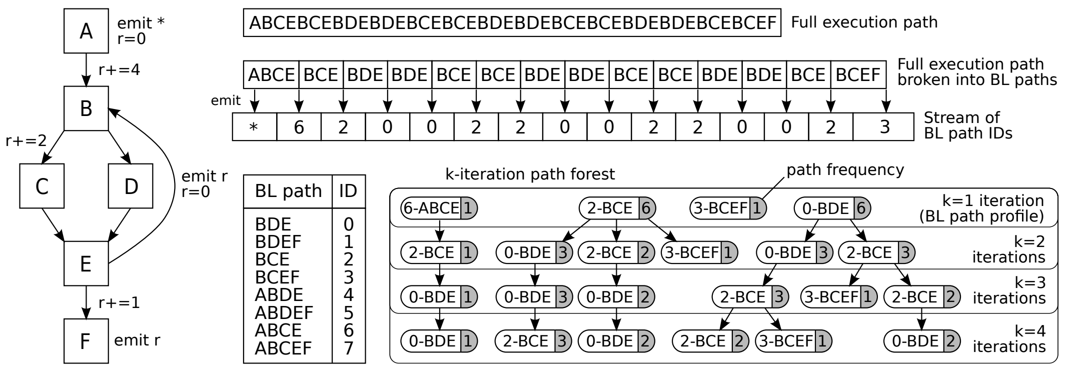


Figure 3.5. Control flow graph (CFG) with Ball-Larus instrumentation modified to emit acyclic path IDs to an output stream and running example of our approach that shows a 4-iteration path forest (4-IPF) for a possible small execution trace. Loop back edges in the CFG have been restored after the path numbering phase.

distinct acyclic paths, numbered from 0 to 7, starting either at the method's entry A, or at loop header B (target of back edge (E, B)).

BLPP places instrumentation on edges to compute a unique path number for each possible path that is taken at run time. In particular, it maintains a variable `r`, called *probe* or *path register*, to compute the path number. Variable `r` is first initialized to zero upon method entry and is then updated as edges are traversed. When an edge that reaches the method exit is executed, or a back edge is traversed, variable `r` represents the unique ID of the taken path. As observed, instead of using the path ID `r` to increase the associated path frequency counter (`count[r]++`), we defer the profiling stage by emitting the path ID to an output stream (`emit r`). To support profiling over multiple invocations of the same routine, we annotate the stream with the special marker $*$ to denote a routine entry event. Instrumentation code for our CFG example is shown on the left of Figure 3.5.

k-iteration Path Profiling

The second stage of our profiling technique takes as input the stream of BL path IDs generated by the first stage and uses it to build a data structure that keeps track of the frequencies of *each and every distinct taken path* consisting of the concatenation of up to k BL paths, where k is a user-defined parameter. This problem is equivalent to counting all n-grams, i.e., contiguous sequences of n items from a given sequence of items, for each $n \leq k$. Our solution is based on the notion of *prefix forest*, which compactly encodes a list of sequences by representing repetitions and common prefixes only once. A prefix forest can be defined as follows:

Definition 5 (Prefix forest) *Let $L = \langle x_1, x_2, \ldots, x_q \rangle$ be any list of finite-length sequences over an alphabet H. The* prefix forest *$\mathcal{F}(L)$ of L is the smallest labeled forest such that, $\forall$ sequence $x = \langle a_1, a_2, \ldots, a_n \rangle$ in L there is a path $\pi = \langle \nu_1, \nu_2, \ldots, \nu_n \rangle$ in $\mathcal{F}(L)$ where ν_1 is a root and $\forall j \in [1, n]$:*

1. ν_j is labeled with a_j, i.e., $\ell(\nu_j) = a_j \in H$;

2. ν_j has an associated counter $c(\nu_j)$ that counts the number of times sequence $\langle a_1, a_2, \ldots, a_j\rangle$ occurs in L.

By Definition 5, each sequence in L is represented as a path in the forest, and node labels in the path are exactly the symbols of the sequence, in the same order. The notion of minimality implies that, by removing even one single node, there would be at least one sequence of L not counted in the forest. Note that there is a distinct root in the forest for each distinct symbol that occurs as first symbol of a sequence.

The output of the second stage of our profiling technique is a prefix forest, which we call *k-Iteration Path Forest* (k-IPF), that compactly represents all observed contiguous sequences of up to k BL path IDs:

Definition 6 (k-Iteration Path Forest) *Given an input stream Σ representing a sequence of BL path IDs and $*$ markers, the* k-Iteration Path Forest *(k-IPF) of Σ is defined as k-IPF $= \mathcal{F}(L)$, where $L = \{$ list of all n-grams of Σ that do not contain $*$, with $n \le k\,\}$.*

By Definition 6, the k-IPF is the prefix forest of all consecutive subsequences of up to k BL path IDs in Σ. Each path $\langle \nu_1, \nu_2, ..., \nu_q\rangle$ in the forest, with $q \le k$, corresponds to a consecutive sequence of items $\langle \ell(\nu_1), \ell(\nu_2), ..., \ell(\nu_q)\rangle$ that occurs in Σ.

Example 2 *Figure 3.5 provides an example showing the 4-IPF constructed for a small sample execution trace consisting of a sequence of 44 basic blocks encountered during one invocation of the routine described by the control flow graph on the left. Notice that the full (cyclic) execution path starts at the entry basic block A and terminates on the exit basic block F. The first stage of our profiler issues a stream Σ of BL path IDs that are obtained by emitting the value of the probe register r each time a back edge is traversed, or the exit basic block is executed. Observe that the sequence of emitted path IDs induces a partition of the execution path into Ball-Larus acyclic paths. Hence, the sequence of executed basic blocks can be fully reconstructed from the sequence Σ of path IDs.*

The 4-IPF built in the second stage contains exactly one tree for each of the 4 distinct BL path IDs (0, 2, 3, 6) that occur in the stream. We observe that path frequencies in the first level of the 4-IPF are exactly those that traditional Ball-Larus profiling would collect. The second level contains the frequencies of taken paths obtained by concatenating 2 BL paths, etc. Notice that the path labeled with $\langle 2, 0, 0, 2\rangle$ in the 4-IPF, which corresponds to the path $\langle B, C, E, B, D, E, B, D, E, B, C, E\rangle$ in the control flow graph, is a 4-gram that occurs 3 times in Σ and is one of the most frequent paths among those that span from 2 up to 4 loop iterations.

Properties. A k-IPF has some relevant properties:

1. $\forall$ nodes $\alpha \in k$-IPF, $k > 0$:

$$c(\alpha) \ge \sum_{\beta_i\,:\, edge\ (\alpha,\beta_i) \in\, k\text{-IPF}} c(\beta_i);$$

2. $\forall k > 0$, k-IPF $\subseteq (k+1)$-IPF.

By Property 1, since path counters are non-negative, they are monotonically non-increasing as we walk down a tree in the k-IPF. The inequality $\geq$ in Property 1 may be strict ($>$) if the execution trace of a routine invocation does not end at the exit basic block; this may be the case when a subroutine call is performed at an internal node of the CFG.

Property 2 implies that, for each tree T_1 in the k-IPF there is a tree T_2 in the $(k+1)$-IPF such that T_2 is equal to T_1 after removing the leaves at level $k+1$. Notice also that a 1-IPF includes acyclic paths only, and yields exactly the same counters as a canonical Ball-Larus path profiler.

3.2.3 Algorithms

We observe that building explicitly a k-IPF concurrently with a program's execution would require updating up to k nodes for each traced BL path: indeed, this may considerably slow down the program even for small values of k. In this section we show that, given a stream of BL path IDs, a k-IPF profile can be constructed by maintaining an intermediate data structure that can be updated quickly, and then converting it into a k-IPF when the stream is over. As intermediate data structure, we use a variant of the *k-slab forest* (k-SF) introduced in [8].

Main idea. The variant of the k-SF we present in this paper is tailored to keep track of all the n-grams from a sequence of symbols, for all $n \leq k$. The organization of our data structure stems from the following simple observation: if we partition a sequence into chunks of length $k-1$, then any subsequence of length up to k will be entirely contained within two consecutive chunks of the partition. The main idea is therefore to consider all the subsequences that start at the beginning of a chunk and terminate at the end of the next chunk, and join them in a prefix forest. Such a forest will contain information for all the subsequences of length up to k starting in an arbitrary position of the stream, and also for subsequences of length up to $2k-2$ starting at the beginning of a chunk. Moreover, this forest will contain a distinct tree for each distinct symbol that appears at the beginning of any chunk.

The partition of the sequence into chunks induces a division of the forest into upper and lower regions (*slabs*) of height up to $k-1$. As we will see later on in this section, this organization implies that the k-SF can be constructed on-line as stream items are revealed to the profiler by adding or updating up to two nodes of the forest at a time, instead of k nodes as we would do if we incremented explicitly the frequencies of n-grams as soon as they are encountered in the stream.

Example 3 *Let us consider again the example given in Figure 3.5. For $k=4$, we can partition the stream into maximal chunks of up to $k-1=3$ consecutive BL path IDs as follows:*

$$\Sigma = \langle *, \underbrace{\boxed{6,\ 2,\ 0}}_{c_1}, \underbrace{\boxed{0,\ 2,\ 2}}_{c_2}, \underbrace{\boxed{0,\ 0,\ 2}}_{c_3}, \underbrace{\boxed{2,\ 0,\ 0}}_{c_4}, \underbrace{\boxed{2,\ 3}}_{c_5} \rangle.$$

The 4-SF of Σ, defined in terms of chunks $c_1, \ldots, c_5$, is shown in Figure 3.6. Notice for instance that 2-gram $\langle 0, 0\rangle$ occurs three times in Σ and five times in the 4-SF.

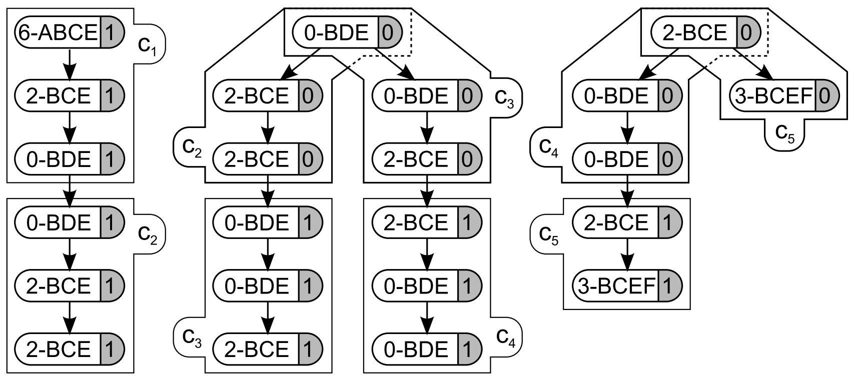


Figure 3.6. 4-SF resulting from the execution trace of Figure 3.5.

However, only three of them end in the bottom slab and hence are counted in the frequency counters.

To obtain a k-IPF starting from the k-SF, for each BL path ID that appears in the stream we eventually construct the set of nodes in the k-SF associated with it and join the subsequences of length up to k starting from those nodes into a prefix forest.

Definition 7 (k-slab forest) *Let $k \geq 2$ and let $c_1, c_2, c_3, \ldots, c_m$ be the chunks of Σ obtained by: (1) splitting Σ at $*$ markers, (2) removing the markers, and (3) cutting the remaining subsequences every $k-1$ consecutive items. The k-*slab forest *(k-SF) of Σ is defined as k-SF $= \mathcal{F}(L)$, where $L = \{$list of all prefixes of $c_1 \cdot c_2$ and all prefixes of length $\geq k$ of $c_i \cdot c_{i+1}$, $\forall i \in [2, m-1]\}$ and $c_i \cdot c_{i+1}$ denotes the concatenation of c_i and c_{i+1}.*

By Definition 7, since each chunk c_i has length up to $k-1$, then a k-SF has at most $2k-2$ levels and depth $2k-3$. As observed above, the correctness of the k-SF representation stems from the fact that, since each occurrence of an n-gram with $n \leq k$ appears in $c_i \cdot c_{i+1}$ for some i, then there is a tree in the k-SF representing it.

Example 4 *In accordance with Definition 7, the forest of Figure 3.6 for the stream of Example 3 is $\mathcal{F}(L)$, where $L = \langle\ \langle 6\rangle, \langle 6,2\rangle, \langle 6,2,0\rangle, \langle 6,2,0,0\rangle, \langle 6,2,0,0,2\rangle, \langle 6,2,0,0,2,2\rangle, \langle 0,2,2,0\rangle, \langle 0,2,2,0,0\rangle, \langle 0,2,2,0,0,2\rangle, \langle 0,0,2,2\rangle, \langle 0,0,2,2,0\rangle, \langle 0,0,2,2,0,0\rangle, \langle 2,0,0,2\rangle, \langle 2,0,0,2,3\rangle\rangle$.*

k-SF construction algorithm. Given a stream Σ formed by BL path IDs and $*$ markers, which we remind the reader denote routine entry events, the k-SF of Σ can be constructed by calling the procedure `process_bl_path_id`(r) shown in Algorithm 4 on each item r of Σ. The stream processing algorithm, which is a variant of the k-SF construction algoritm given in [8] for the different setting of bounded-length calling contexts, keeps the following information:

- a hash table R, initially empty, containing pointers to the roots of the trees in the k-SF, hashed by node labels; since no two roots have the same label, the lookup operation `find`(R, r) returns the pointer to the root containing label r, or `null` if no such root exists;

- a variable n that counts the number of BL path IDs processed since the last $*$ marker;
- a variable τ (top) that points either to `null` or to the current k-SF node in the upper part of the forest (levels 0 through $k-2$);
- a variable β (bottom) that points either to `null` or to the current k-SF node in the lower part of the forest (levels $k-1$ through $2k-3$).

```
procedure process_bl_path_id($r$):
if $r = *$ then
    $n \leftarrow 0$
    $\tau \leftarrow$ null
    return
end
if $n$ mod $(k-1) = 0$ then
    $\beta \leftarrow \tau$
    $\tau \leftarrow$ find$(R, r)$
    if $\tau =$ null then
        add root $\tau$ with $\ell(\tau) = r$ and $c(\tau) = 0$ to $k$-SF and $R$
    end
else
    find child $\omega$ of node $\tau$ with label $\ell(\omega) = r$
    if $\omega =$ null then
        add node $\omega$ with $\ell(\omega) = r$ and $c(\omega) = 0$ to $k$-SF
        add arc $(\tau, \omega)$ to $k$-SF
    end
    $\tau \leftarrow \omega$
end
if $\beta \neq$ null then
    find child $\upsilon$ of node $\beta$ with label $\ell(\upsilon) = r$
    if $\upsilon =$ null then
        add node $\upsilon$ with $\ell(\upsilon) = r$ and $c(\upsilon) = 0$ to $k$-SF
        add arc $(\beta, \upsilon)$ to $k$-SF
    end
    $\beta \leftarrow \upsilon$
    $c(\beta) \leftarrow c(\beta) + 1$
else
    $c(\tau) \leftarrow c(\tau) + 1$
end
$n \leftarrow n + 1$
```

Algorithm 4: Stream processing algorithm for k-SF construction.

The main idea of the algorithm is to progressively add new paths to an initially empty k-SF. The path formed by the first $k-1$ items since the last $*$ marker is added to one tree of the upper part of the forest. Each later item r is added at up to two different locations of the k-SF: one in the upper part of the forest (lines 13–17) as a child of node τ (if no child of τ labeled with r already exists), and the other one in the lower part of the forest (lines 21–25) as a child of node β (if no child of β labeled with r already exists). Counters of processed nodes already containing r are incremented by one (either line 27 or line 29).

Both τ and β are updated to point to the child labeled with r (lines 18 and 26, respectively). The running time of the algorithm is dominated by lines 8 and 10 (hash table accesses), and by lines 13 and 21 (node children scan). Assuming that operations on the hash table R require constant time, the per-item processing time is $O(\delta)$, where δ is the maximum degree of a node in the k-SF. An experimental investigation revealed that δ is typically a small constant value on average.

As an informal proof that each subsequence of length up to k is counted exactly once in the k-SF, we first observe that, if the subsequence extends across two consecutive chunks, then it appears exactly once in the forest (connecting a node in the upper slab to a node in the lower slab). In contrast, if the subsequence is entirely contained in a chunk, then it appears twice: once in the upper slab of the tree rooted at the beginning of the chunk, and once in the lower slab rooted in at the beginning of the preceding chunk. However, in this case only the counter in the lower part of the forest is updated (line 27): for this reason, the sum of all counters in the k-SF is equal to the length of the stream.

k-SF to k-IPF conversion. Once the profiling phase has terminated, we convert the k-SF into a k-IPF using the procedure shown in Algorithm 5. The key intuition behind the correctness of the conversion algorithm is that for each sequence in the stream of length up to k, there is a tree in the k-SF containing it.

```
procedure make_k_ipf():
1   $I \leftarrow \emptyset$
2   foreach node $\rho \in k$-SF do
3       if $\ell(\rho) \notin I$ then
4           add $\ell(\rho)$ to $I$ and let $s(\ell(\rho)) \leftarrow \emptyset$
5       end
6       add $\rho$ to $s(\ell(\rho))$
7   end
8   let the $k$-IPF be formed by a dummy root $\phi$
9   foreach $r \in I$ do
10      foreach $\rho \in s(r)$ do
11          join_subtree($\rho, \phi, k$)
12      end
13  end
14  remove dummy root $\phi$ from the $k$-IPF
```

Algorithm 5: Algorithm for converting a k-SF into a k-IPF.

The algorithm creates a set I of all distinct path IDs that occur in the k-SF and for each r in I builds a set $s(r)$ containing all nodes ρ of the k-SF labeled with r

(lines 2–7). To build the k-IPF, the algorithm lists each distinct path ID r and joins to the k-IPF all subtrees of depth up to $k-1$ rooted at a node in $s(r)$ in the k-SF, as children of a dummy root, which is added for the sake of convenience and then removed. The join operation is specified by procedure `join_subtree` (Algorithm 6), which performs a traversal of a subtree of the k-SF of depth less than k and adds nodes to k-IPF so that all labeled paths in the subtree appear in the k-IPF as well, but only once. Path counters in the k-SF are accumulated in the corresponding nodes of the k-IPF to keep track of the number of times each distinct path consisting of the concatenation of up to k BL paths was taken by the profiled program.

```
procedure join_subtree($\rho, \gamma, d$):
1  $\delta \leftarrow$ child of $\gamma$ in the $k$-IPF s.t. $\ell(\delta) = \ell(\rho)$
2  if $\delta =$ null then
3      add new node $\delta$ as a child of $\gamma$ in the $k$-IPF
4      $\ell(\delta) \leftarrow \ell(\rho)$ and $c(\delta) \leftarrow c(\rho)$
5  else
6      $c(\delta) \leftarrow c(\delta) + c(\rho)$
7  end
8  if $d > 1$ then
9      foreach child $\sigma$ of $\rho$ in the $k$-SF do
10         join_subtree($\sigma, \delta, d-1$)
11     end
12 end
```

Algorithm 6: Subroutine for joining trees during k-SF conversion.

[**Note:** *Cost of conversion? Also, give some intuition on why the various steps are needed!*]

3.2.4 Discussion

Our multi-iteration path profiling technique introduces a technical shift based on a smooth blend of the path numbering methods used in intraprocedural path profiling with data structure-based techniques typically adopted in interprocedural profiling, such as calling-context profiling. Our solution combines the original Ball-Larus path numbering technique with a prefix tree data structure to keep track of concatenations of acyclic paths across multiple loop iterations.

Maintaining a prefix forest during a program's execution would be too costly: we thus devise an intermediate data structure that supports updates in nearly constant time (i.e., proportional to the degree of the node, which is typically small in practice). As a k-SF captures information on all distinct paths of bounded length taken at run time, its space requirements mainly depend on intrinsic structural properties of analyzed programs. To capture even longer paths, our technique might be extended with heuristics aiming at reducing space usage, for instance by periodically pruning branches of the k-SF with small frequency counters. Also, it might be integrated with sophisticated sampling techniques for reducing time overhead used in intraprocedural (e.g., [20]) and interprocedural (e.g, [113] and Section **XXX**) profilers.

As we will see in Section **XXX**, the key to efficiency of our approach is to replace costly hash table accesses with substantially faster operations on trees. This allows us to profile paths that extend across many loop iterations, while previous techniques do not scale well even for short cyclic paths. Profiling longer paths can reveal interesting optimization opportunities that "short" profiles would miss [36]. In **XXX** we present an optimization case study on masked convolution filters for image processing: using a k-IPF profile collected for $k = 10$, we devise a selective loop unrolling optimization resulting in non-negligible speedups on all of our tests.

3.2.5 Comparison with Related Work

The seminal work of Ball and Larus [10] has spawned much research interest in the development of new path profiling techniques in the last 15 years. In particular, several works focus on profiling acyclic paths with a lower overhead by using sampling techniques [20, 21] or choosing a subset of *interesting* paths [5, 60, 103]. On the other hand, only a few works have dealt with cyclic-path profiling.

Tallam *et al.* [101] extend the Ball-Larus path numbering algorithm to record slightly longer paths across loop back edges and procedure boundaries. The extended Ball-Larus paths overlap and, in particular, are shorter than two iterations for paths that cross loop boundaries. These overlapping paths enable very precise estimation of frequencies of potentially much longer paths, with an average imprecision in estimated total flow of those paths ranging from -4% to $+8\%$. However, experimental results reveal that the average cost of collecting frequencies of overlapping paths is about 4.2 times that of canonical BLPP.

Roy and Srikant [90] generalize the Ball-Larus algorithm for profiling k-iteration paths, showing that it is possible to number these paths efficiently using an inference phase to record executed backedges in order to differentiate cyclic paths. One problem with this approach is that, since the number of possible k-iteration paths grows exponentially with k, path IDs may overflow in practice even for small values of k. Furthermore, very large hash tables may be required. In particular, their profiling procedure aborts if the number of static paths exceeds $60,000$, while this threshold is reached on several small benchmarks already for $k = 3$ [69]. This technique incurs a larger overhead than BLPP: in particular, the slowdown may grow to several times the BLPP-associated overhead as k increases.

Li *et al.* [69] propose a new path encoding that does not rely on an inference phase to explicitly assign identifiers to all possible paths before the execution, yet ensuring that any finite-length acyclic or cyclic path has a unique ID. Their path numbering algorithm needs multiple variables to record probe values, which are computed by using addition and multiplication operations. Overflowing is handled by using *breakpoints* to store probe values: as a consequence, instead of a unique ID for each path, a unique series of breakpoints is assiged to each path. At the end of program's execution, a *backwalk* algorithm reconstructs the executed paths starting from breakpoints. This technique has been integrated with BLPP to reduce the execution overhead, resulting in a slowdown of about 2 times on average with respect to BLPP, but also showing significant performance loss (up to a 5.6 times growth) on tight loops[3]. However, the experiments reported in [69] were performed on single methods of small Java programs, leaving further experiments on larger industry-strength benchmarks to future work.

The common trait of all these works it to extend the Ball-Larus encoding algorithm to capture longer paths. These techniques typically maintain more than one probe value, and incur higher average costs compared to BLPP. Our approach builds instead on top of the original Ball-Larus algorithm: a single probe value is used to track acyclic paths taken at run time, and each BL path ID in the output stream can be processed in nearly constant time. Experimental results presented in Section **XXX** will reveal that our approach incurs an overhead competitive with BLPP.

Of a different flavor is the technique introduced by Young [109] for profiling *general paths*, i.e., fixed-length sequences of taken branches that might span multiple loop iterations. Unfortunately, this technique scales poorly for increasing path lengths l both in terms of space usage and running time. In particular, the running time is proportional not only to the length of the stream of taken branches, but also to the number of possible sequences of length l, that is likely to be exponential in l. In order to reduce the per-taken-branch update time, the algorithm uses also additional space with respect to that required for storing the path counters and identifiers; such space is proportional to the number of possible sequences of length l as well.

[3]A loop is *tight* when it contains a small number of instructions and iterates many times.

Chapter 4

Continuous Program Optimization Techniques

In this chapter, we focus on a fundamental aspect for deploying adaptive optimization techniques in runtime systems: the *On-Stack Replacement* (OSR) problem. OSR consists in dynamically transferring execution between different versions of a function at run time. Modern virtual machines implement OSR to *continuously* optimize a program as it executes, for instance by interrupting a long-running function and recompiling it at a higher optimization level, or by replacing a function version with another when a speculative assumption made during its compilation no longer holds.

In the first part of the chapter, we present a platform-independent framework for OSR that introduces novel ideas and combines features of existing techniques that no previous solution provided simultaneously. In particular, we introduce a *compensation code* abstraction that increases the flexibility of OSR mechanisms, and we present our OSR implementation for the LLVM MCJIT compiler that allows OSR to happen at arbitrary locations of a function.

In the second part, we make a first step towards a provably sound methodological framework for OSR. We formalize the concept of *multi-version program*, and we identify sufficient conditions for the correctness of an OSR transition. We also devise an algorithm for automatically generating compensation code possibly required at a program location in the presence of several common compiler optimizations.

4.1 A New Framework for On-Stack Replacement

Modern language runtimes dynamically adapt the execution to the actual workload, maintaining different versions of the code generated with different, often speculative, optimizations. For this reason, they typically implement on-stack replacement mechanisms to dynamically transfer execution between them *while* a version of the method to optimize is still running.

Pioneered in the SELF language runtime in the early 90's, OSR mechanisms have drawn considerable attention from the community of VM builders as the Java language grew popular. OSR is nowadays used in a relevant number of virtual machines to implement optimization techniques such as profile-driven and deferred compilation, and can also be employed to support debugging of optimized code.

OSR can be a very powerful tool for implementing dynamic languages, for which most effective optimization decisions can typically be made only at run time, when critical information such as type and shape of objects becomes available. In this scenario, OSR becomes useful also to perform deoptimization, i.e. when the running code has been speculatively optimized and the assumption used for the optimization does not hold anymore, the optimized function is interrupted and the execution continues in a safe version of the code.

Contributions. In this thesis, we propose a general-purpose, target-independent framework for OSR. Specific goals of our solution include:

- The ability for a function reached via OSR to fire an OSR itself: this would allow switching from a base function f to an optimized function f', and later on to a further optimized version f'', and so on.
- Supporting deoptimization, i.e., transitions from an optimized function to a less optimized function from which it was derived.
- Supporting transitions at arbitrary program points, including those that would require adjusting the transferred program state to resume the execution in the OSR target function.
- Supporting OSR targets either generated at run-time (e.g., using profiling information) or already known at compilation time.
- Hiding from the front-end that generates the different function versions all the implementation details for handling OSR transitions between them.

We show the feasibility of our approach by implementing OSRKit, a prototype library for the MCJIT just-in-time compiler from the LLVM compiler infrastructure.

4.1.1 Approach

The key to generality and platform-independence in our approach is to express the OSR machinery entirely at intermediate representation (IR) level, without resorting to native-code manipulation or special intrinsics of a compiler.

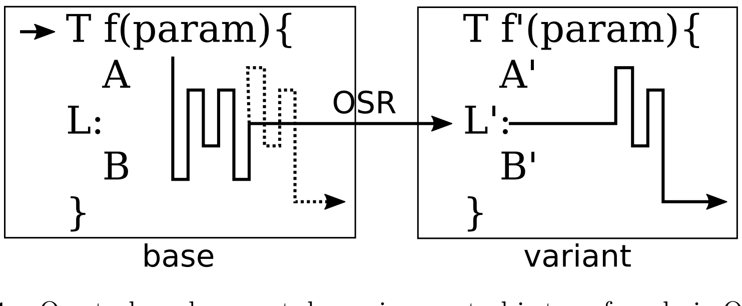


Figure 4.1. On-stack replacement dynamics: control is transferred via OSR from a point L of a base function f to a point L' in a variant f' of f.

Consider the generic OSR scenario shown in Figure 4.1. A base function f is executed and it can either terminate normally (dashed lines), or an OSR event may transfer control to a variant f', which resumes the execution. The decision of whether an OSR should be fired at a given point L of f is based on an *OSR condition.* A typical example in JIT-based virtual machines is a profile counter reaching a certain hotness threshold, which indicates that f is taking longer than expected and is worth optimizing. Another example is a guard testing whether f has become unsafe and execution needs to fall back to a safe version f'. This scenario includes deoptimization of functions generated with aggressive speculative optimizations.

Several OSR implementations adjust the stack so that execution can continue in f' with the current frame [26, 25, 57, 96]. This requires manipulating the program state at machine-code level and is highly ABI- and compiler-dependent. A simpler approach, which we follow in this thesis, consists in creating a new frame every time an OSR is fired, essentially regarding an OSR transition as a function call [66, 86]. Our solution targets two general scenarios:

1. *resolved OSR*: f' is known before executing f as in the deoptimization example discussed above;
2. *open OSR*: f' is generated when the OSR is fired, supporting deferred and profile-guided compilation strategies.

In both cases, f is instrumented before its execution to incorporate the OSR machinery. We call such OSR-instrumented version $\texttt{f}_{\texttt{from}}$.

In the resolved OSR scenario (see Figure 4.2), instrumentation consists of adding a check of the OSR condition and, if it is satisfied, a tail call that fires the OSR. The called function is an instrumented version of f', which we call $\texttt{f'}_{\texttt{to}}$. We refer to $\texttt{f'}_{\texttt{to}}$ as the *continuation function* for an OSR transition. The assumption is that $\texttt{f'}_{\texttt{to}}$ produces the same side-effects and return value that one would obtain by f if no OSR was performed. Differently from f', $\texttt{f'}_{\texttt{to}}$ takes as input all live variables of f at L, executes an optional *compensation code* to fix the computation state (`comp_code`), and then jumps to a point L' from which execution can continue.

Compensation code adds flexibility to our framework, as it extends the range of points where OSR transitions can be fired. In fact, the OSR practice often makes the conservative assumption that execution can always continue with the very same program state as the base function. This assumption can however be restrictive, as it may reduce the number of program locations eligible for OSR (i.e., one has to wait to a point where the states would realign). Our solution provides a front-end with means to encode a glue code, tailored to the specific optimizations involved between

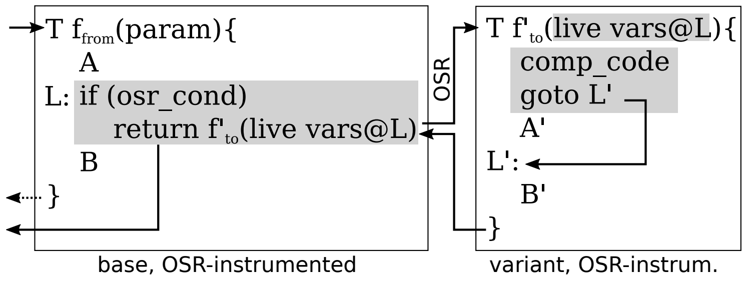


Figure 4.2. Resolved OSR scenario.

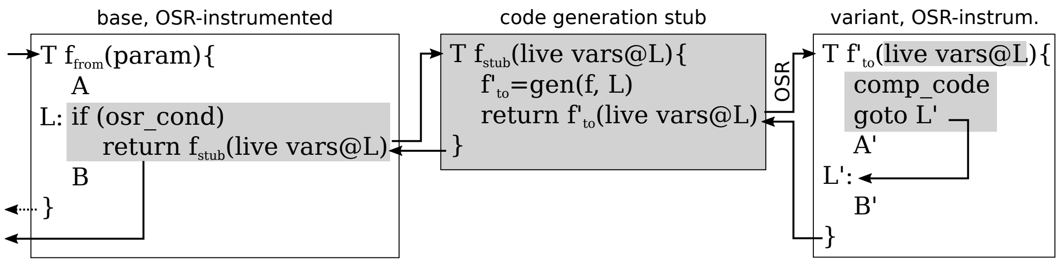


Figure 4.3. Open OSR scenario.

two function versions, to adjust the program state and perform an OSR transition. This code can be used, for instance, to modify the heap, or to reconstruct values for variables that are live at `L'` but not at `L`.

The open OSR scenario is similar, with one main difference (see Figure 4.3): instead of calling $\texttt{f'}_{\texttt{to}}$ directly, $\texttt{f}_{\texttt{from}}$ calls a stub function $\texttt{f}_{\texttt{stub}}$, which first creates $\texttt{f'}_{\texttt{to}}$ and then calls it. Function $\texttt{f'}_{\texttt{to}}$ is generated by a function `gen` that takes the base function `f` and the OSR point `L` as input. The reason for having a stub in the open OSR scenario, rather than directly instrumenting `f` with the code generation machinery, is to minimize the extra code injected into `f`. Indeed, instrumentation may interfere with optimizations, e.g., by increasing register pressure and altering code layout and instruction cache behavior. [**Note:** *JIT compilation?*]

4.1.2 LLVM Implementation

The LLVM compiler infrastructure [68] provides a Just-In-Time compiler called MCJIT that is currently being used for generating optimized code at run-time in virtual machines for dynamic languages. MCJIT is employed in both industrial and research projects, including Webkit's Javascript engine, the open-source Python implementation Pyston, the Rubinius project for Ruby, Julia for high-performance technical computing, McVM for MATLAB, CXXR for the R language, Terra for Lua, and the Pure functional programming language. The MCJIT compiler shares the same optimization pipeline with LLVM front-ends for static languages such as `clang`, and it provides dynamic features such as native code loading and linking, as well as a customizable memory manager for code and data sections.

Currently VM builders using MCJIT are required to have a deep knowledge of the internals of LLVM in order to mimic an OSR mechanism. In particular, they can rely on two experimental intrinsics, *Stackmap* and *Patchpoint*, to inspect the details of the compiled code generated by the back-end and to patch it manually with a sequence of assembly instructions. A *Stackmap* is used to record the run-time location (i.e., register, stack offset or constant) for a set of variables (e.g., the set of live variables) at a given IR instruction. The intrinsic generates no code in place[1], as the back-end emits its data into a designated section in the object code. A Stackmap allows the runtime to disruptively patch the original code in response to an event

[1]Although a front-end can ask LLVM to insert a number of NOP instructions into the function to prevent overwriting program text or data outside the function during the run-time patching.

```
int isord(long* v, long n, int (*c)(void*,void*)) {
    for (long i=1; i<n; i++)
        if (c(v+i-1,v+i)>0) return 0;
    return 1;
}
```

Figure 4.4. Example for OSR instrumentation in LLVM.

triggered from outside, thus executing a new code sequence when the location for the original IR instruction is reached. A *Patchpoint* instead creates a function call to a target typically not known at compile time, and implies a StackMap generation to track the run-time location for the set of variables given as argument. A Patchpoint reserves space for injecting new code (e.g., to update the target of the call), so that the other instructions in the function are preserved. An example application of the Patchpoint intrinsic is the implementation [86] of an inline caching mechanism [39] for polymorphic method dispatch in the WebKit JavaScript engine. Both intrinsics are currently marked as experimental, and are treated along the optimization pipeline as instructions that can potentially read and write all memory[2].

We prototyped our idea of flexible OSR infrastructure working entirely at IR level in a library for MCJIT called OSRKit. OSRKit provides a number of useful abstractions that include open and resolved OSR instrumentation of IR base functions without breaking the SSA (Static Single Assignment) form [35], liveness analysis, generation of OSR continuation functions, and mapping of LLVM values between different function versions along with an interface for compensation code generation.

We also implemented a proof-of-concept VM called TinyVM that provides an interactive environment for LLVM IR manipulation, JIT compilation, and benchmarking. All of our code is publicly available and has been endorsed by the joint Artifact Evaluation process of CGO-PPoPP 2016.

A Running Example

We present our OSR embodiment for LLVM through a simple running example that illustrates a profile-driven optimization scenario. We start from a simple base function (`isord`) that checks whether an array of numbers is ordered according to some criterion specified by a comparator (see Figure 4.4). Our goal is to instrument `isord` so that, whenever the number of loop iterations exceeds a certain threshold, control is dynamically diverted to a faster version generated on the fly by inlining the comparator. The IR code shown in this section has been generated with `clang` and later instrumented with OSRKit inside TinyVM. Virtual register names and basic block labels have been refactored for the sake of readability.

IR Instrumentation. To defer the compilation of the continuation function until the comparator is known at run time, we used OSRKit to instrument `isord` with an open OSR point at the beginning of the loop body, as shown in Figure 4.5. Portions added to the original code by OSR instrumentation are highlighted in grey.

[2]A `store` instruction clearly cannot be moved across a Stackmap, but also a `load` must be handled conservatively (i.e., cannot be hoisted above it) as it might trigger an exception.

```
define i32 @isordfrom(
  i64* %v, i64 %n, i32 (i8*, i8*)* nocapture %c) {
entry:
  %t0 = icmp sgt i64 %n, 1
  br i1 %t0, label %loop.body, label %exit

loop.header:
  %t1 = icmp slt i64 %i1, %n
  br i1 %t1, label %loop.body, label %exit

loop.body:
  %i = phi i64 [%i1, %loop.header], [1,%entry]
  %p.osr = phi i64 [%p.osr1, %loop.header],
                   [1000, %entry]
  %p.osr1 = add nsw i64 %p.osr, -1
  %osr.cond = icmp eq i64 %p.osr, 0
  br i1 %osr.cond, label %osr,
                   label %loop.body.cont
loop.body.cont:
  %t2 = getelementptr inbounds i64* %v, i64 %i
  %t3 = add nsw i64 %i, -1
  %t4 = getelementptr inbounds i64* %v, i64 %t3
  %t5 = bitcast i64* %t4 to i8*
  %t6 = bitcast i64* %t2 to i8*
  %t7 = tail call i32 %c(i8* %t5, i8* %t6)
  %t8 = icmp sgt i32 %t7, 0
  %i1 = add nuw nsw i64 %i, 1
  br i1 %t8, label %exit, label %loop.header

exit:
  %res = phi i32 [1, %entry], [1, %loop.header]
                 [0, %loop.body.cont],
  ret i32 %res

osr:
  %val = bitcast i32 (i8*, i8*)* %c to i8*
  %osr.res = call i32 @isordstub(i8* %val,
      i64* %v, i64 %n, i32 (i8*, i8*)* %c, i64 %i)
  ret i32 %osr.res
}
```

Figure 4.5. LLVM IR version of base function `isord` (Figure 4.4) instrumented for open OSR. Additions resulting from the instrumentation are in grey. The OSR is fired at the beginning of the loop body after 1000 iterations, i.e., when the counter reaches 0.

New instructions are placed at the beginning of the loop body to increment a hotness counter `p.osr` and jump to an OSR-firing block if the counter reaches the threshold (1000 iterations in this example). The OSR block contains a tail call to the target generation stub, which receives as parameters the four live variables at the OSR point (`v`, `n`, `c`, `i`). OSRKit allows the stub to receive the run-time value `val` of an IR object that can be used to produce the continuation function – in our example, the pointer to the comparator function to be inlined. The stub (shown in Figure 4.6) calls a code generator that:

1. builds an optimized version of `isord` by inlining the comparator, and

2. uses it to create the continuation function `isordto` shown in Figure 4.7.

The stub passes to the code generator four parameters:

1. a pointer to the `isord` IR code;
2. a pointer to the basic block in `isord` from which the OSR is fired;
3. a pointer to a user-defined object to support code generation in MCJIT;
4. the stub's `val` parameter.

The first three parameters are provided by the front-end and hard-wired by OSRKit. In particular, the third parameter is a handle to the environment for code generation (e.g., in our dynamic inliner the object contains pointers to the MCJIT engine and to a map between addresses of compiled functions and their IR counterparts). The stub terminates with a tail call to `isordto`.

```
define i32 @isordstub(
   i8* %val, i64* %v_osr, i64 %n_osr,
   i32 (i8*, i8*)* nocapture %c_osr, i64 %i_osr) {
entry:
  %cont.func = call
     ; generator returns ptr to isordto
     i32 (i64*, i64, i32 (i8*, i8*)*, i64)*
        (i8*, i8*, i8*, i8*)* inttoptr

     ; generator function address is 4357824
     (i64 4357824 to
           i32 (i64*, i64, i32 (i8*, i8*)*, i64)*
                 (i8*, i8*, i8*, i8*)*)

     ; hard-coded parameters passed to generator:
     ;  46993664 = addr of isord IR function
     ;  46995056 = addr of basic block at loop.body
     ;  47005408 = addr of code generation env
     (i8* inttoptr (i64 46993664 to i8*),
      i8* inttoptr (i64 46995056 to i8*),
      i8* inttoptr (i64 47005408 to i8*), i8* %val)

  %osr.res = call i32 %cont.func(i64* %v_osr,
    i64 %n_osr, i32 (i8*, i8*)* %c_osr, i64 %i_osr)
  ret i32 %osr.res
}
```

Figure 4.6. IR stub that generates the continuation function when an open OSR is fired by `isordfrom` (Figure 4.5).

To generate the continuation function (shown in Figure 4.7) from the optimized version created by the inliner, OSRKit replaces the function entry point, removes dead code, replaces live variables with the function parameters, and fixes ϕ-nodes accordingly. As the OSR transition does not require modifications to the program state, the new entry point does not contain any form of compensation code. Preserving the SSA form while constructing the continuation function is a challenging task, as it might require inserting new ϕ-nodes in the control-flow graph in addition to simply updating some of the existing ones as in this example. When generating a continuation function, we rely on the SSAUpdater component of LLVM to account for the available values – transferred as parameters or reconstructed in the OSR entrypoint – of all the variables that are live at the OSR landing pad.

```
define i32 @isordto(
  i64* nocapture readonly %v_osr,
  i64 %n_osr, i32 (i8*, i8*)* %c_osr, i64 %i_osr) {

osr.entry: ; no compensation code needed...
  br label %loop.body

~~entry:~~
  ~~%t1 = icmp sgt i64 %n_osr, 1~~
  ~~br i1 %t1, label %loop.body, label %exit~~

loop.header:
  %t2 = icmp slt i64 %i1, %n_osr
  br i1 %t2, label %loop.body, label %exit

loop.body:
  %i = phi i64 [ %i1, %loop.header ],
               ~~[ 1, %entry ],~~
               [ %i_osr, %osr.entry ]
  %t3 = add nsw i64 %i, -1
  %t4 = getelementptr inbounds i64* %v_osr, i64 %t3
  %t5 = load i64* %t4, align 8, !tbaa !1
  %t6 = getelementptr inbounds i64* %v_osr, i64 %i
  %t7 = load i64* %t6, align 8, !tbaa !1
  %t8 = icmp sgt i64 %t5, %t7
  %i1 = add nuw nsw i64 %i, 1
  br i1 %t8, label %exit, label %loop.header

exit:
  %res = phi i32 ~~[ 1, %entry ],~~
                 [ 0, %loop.body ],
                 [ 1, %loop.header ]
  ret i32 %res
}
```

Figure 4.7. Faster variant of isord (Figure 4.4) in LLVM IR with comparator inlining, instrumented as OSR continuation function. Instrumentation additions are in grey. The original function entry block is unreachable after instrumentation and is eliminated (struck-through code fragments).

x86-64 Lowering. Figure 4.8 shows the x86-64 code generated by the LLVM back-end for isordfrom and isordto. For the sake of comparison with the native code that would be generated for the original non-OSR versions, additions resulting from the IR instrumentation are in grey, while removals are struck-through.

Notice that the OSR intrusiveness in isordfrom is minimal, consisting of just two assembly instructions with register and immediate operands. As a result of induction variable canonicalization in the LLVM back-end, loop index i and hotness counter p.osr are fused in register %r12. We also note that tail call optimization is applied in the OSR-firing block, resulting in no stack growth during an OSR.

```
isordfrom:
   pushq %r15
   pushq %r14
   pushq %r12
   pushq %rbx
   pushq %rax
   movq %rdx, %r14 #c
   movq %rsi, %r15 #n
   movq %rdi, %rbx #v
   movl $1, %r12d  #i
   cmpq $1, %r15
   jle .LBB0_1
.LBB0_4: # %loop.body
   cmpq $1001, %r12
   je .LBB0_7
   movq %rbx, %rdi
   leaq 8(%rbx), %rbx
   movq %rbx, %rsi
   callq *%r14
   movl %eax, %ecx
   xorl %eax, %eax
   testl %ecx, %ecx
   jg .LBB0_6
   incq %r12
   cmpq %r15, %r12
   jl .LBB0_4
   movl $1, %eax
   jmp .LBB0_6
.LBB0_1:
   movl $1, %eax
.LBB0_6: # %exit
   addq $8, %rsp
   popq %rbx
   popq %r12
   popq %r14
   popq %r15
   retq
.LBB0_7: # %osr
   movq %r14, %rdi # c
   movq %rbx, %rsi # v
   movq %r15, %rdx # n
   movq %r14, %rcx # c
   movq %r12, %r8  # i
   addq $8, %rsp
   popq %rbx
   popq %r12
   popq %r14
   popq %r15
   jmp isordstub
```

```
isordto:
   ~~movl $1, %edx~~
   ~~cmpq $1, %rsi~~
   ~~jle .LBB0_1~~
.LBB0_4: # %loop.body
   movq -8(%rdi,%rdx,8),%rcx
   xorl %eax, %eax
   cmpq (%rdi,%rdx,8),%rcx
   jg .LBB0_5
   incq %rdx
   cmpq %rsi, %rdx
   jl .LBB0_4
~~.LBB0_1:~~
   movl $1, %eax
.LBB0_5: # %exit
   retq
```

Figure 4.8. OSR-instrumented functions `isordfrom` (base) and `isordto` (faster continuation) after IR-to-x86-64 lowering in LLVM. Additions resulting from the IR instrumentation are in grey, while removals are struck-through.

The continuation function `isordto` is identical to the specialized version of `isord` with inlined comparator, except that the loop index is passed as a parameter in `%rdx` and no preamble is needed since OSR jumps directly in the loop body.

4.1.3 Discussion

Instrumenting functions for OSR at a higher level than machine code yields several benefits:

1. *Platform independence*: the OSR instrumentation code is lowered to native code by the compiler back-end, which handles the details of the target ABI.
2. *Global optimizations*: lowering OSR instrumentation code along with application code can generate faster code than local binary instrumentation. For instance, dead code elimination can suppress from $\texttt{f'}_{\texttt{to}}$ portions of code that would no longer be needed when jumping to the landing pad `L'`, producing smaller code and enabling better register allocation and instruction scheduling.

3. *Debugging and Profiling*: preserving ABI conventions in the native code versions of $\mathsf{f}_{\mathsf{from}}$, $\mathsf{f}_{\mathsf{stub}}$, and $\mathsf{f'}_{\mathsf{to}}$ helps debuggers and profilers to more precisely locate the current execution context and collect more informative data.
4. *Abstraction*: being entirely encoded using high-level language constructs (assignments, conditionals, function calls), the approach is amenable to a clean instrumentation API that abstracts the OSR implementation details, allowing a front-end to focus on where to insert OSR points independently of the final target architecture.

A natural question is whether encoding OSR at a higher level of abstraction can result in poorer performance than binary code approaches. Our solution relies on the backend's compilation pipeline to generate the most efficient native code for $\mathsf{f}_{\mathsf{from}}$ and $\mathsf{f'}_{\mathsf{to}}$. We provide performance numbers in Section**XXX**by measuring the overhead of OSRKit on classic benchmarks.

To the best of our knowledge, our framework is the first to support OSR point insertion at arbitrary locations in the code. There was considerable implementation and experimentation effort to show its feasibility. In order to enable OSR at any program point, two engineering aspects are involved: manipulating the code of an optimized function to generate a continuation function – a task that can be daunting as the SSA form must be preserved – and providing a front-end with a clean interface to specify glue code that might be required to perform a transition. Encoding compensation code with our API is currently delegated to the front-end. In Section**XXX** we will provide an algorithm to automatically build it for a number of common compiler optimizations.

A possible scenario in which supporting OSR at arbitrary locations can be particularly useful is the implementation a flexible deoptimization mechanism in the presence of aggressive speculative optimizations. In general, it might be necessary to have a deoptimization point in the middle of a heavily optimized code fragment. Our framework provides VM builders with means to perform deoptimization without requiring a fallback to an interpreter. Also, by supporting both open and resolved OSR points we allow them to explore the trade-off between the latency from creating continuation functions on-the-fly and the code bloat from doing it ahead-of-time.

XXX

[**Note:** *Deferred compilation, case study*]

4.1.4 Comparison with Related Work

Early Approaches. OSR has been pioneered in the SELF programming language implementations [57] to enable source-level debugging of optimized code, which requires deoptimizing the code back to the original version. To reconstruct the source-level state, the compiler generates *scope descriptors* recording locations or values of arguments and locals. Execution can be interrupted only at certain interrupt points where its state is guaranteed to be consistent (i.e., method prologues and backward branches in loops), allowing optimizations between interrupt points. SELF also implements a deferred compilation mechanism [26] for branches that are unlikely to occur at run-time: the system generates a stub that invokes the compiler to generate a code object that can reuse the stack frame of the original code. Open

OSR points proposed in this thesis can be used to implement deferred compilation in a similar manner.

Java Virtual Machines. The success of the Java language has drawn more attention to the design and implementation of OSR techniques, as bytecode interpreters began to work along with JIT compilers. In the high-performance HotSpot Server JVM [81] performance-critical methods are identified using method-entry and backward-branches counters; when the OSR threshold is reached, the runtime transfers the execution from the interpreter frame to an OSR frame and thus to compiled code. Deoptimization is performed when class loading invalidates inlining or other optimization decisions: execution is rolled forward to a safe point, at which the native frame is converted into an interpreter frame.

The Jikes RVM uses an OSR mechanism [44] that extracts a scope descriptor from a thread suspended at a method's entrypoint or backward branch, creates specialized code to setup the stack frame for the optimized compiled code and resumes the execution at the desired program counter. OSR is used as part of an automatic, online, profile-driven deferred compilation mechanism. The specialized code used by Jikes is essentially equivalent to the continuation function generated by our framework. A more general approach has then been proposed in [93], with the OSR implementation decoupled from program code to ease more aggressive specializations triggered by events external to the executing code (e.g., class loading, exception conditions). Execution state information is maintained in a variable map - a per-method list of thread-switch points and associated live bytecode variables - that is incrementally updated across a wide range of basic compiler optimizations.

In the Graal VM - a modified version of HotSpot centered on the principle of speculative optimizations - the execution falls back to the interpreter during deoptimization, while a runtime function restores the stack frames in the interpreter using the metadata associated with the deoptimization point [40, 107, 41].

Prospect. Prospect [100] is an LLVM-based framework for parallelizing a sequential application. The IR is instrumented through two LLVM passes to enable switching at run-time between a slow and a fast variant of the code, which are both compiled statically. Helper methods are used to save and eventually restore registers, while stack-local variables are put on a separate `alloca` stack rather than on the stack frame so that the two variants result into similar and thus interchangeable stack layouts. Speculative variables [99] are introduced when the slow variant needs to track state (e.g., information for out-of-bound checks) that is missing in the fast variant. Switching operations are performed by Prospect at user-specified checkpoints in the original code. Although both Prospect and `OSRKit` support switching execution between two variants of a function, they target different applications. Notice also that speculative variables act as a form of compensation code for transitions to the slow variant.

Other Related Work. Dynamic Software Updating (DSU) is a methodology for permitting programs to be updated while they run and is thus useful for systems that cannot afford to halt service. DSU techniques (e.g., [78, 73]) are required to update all functions active on the call stack at the same time, so their code should be instrumented and data types wrapped to support future extensions. Albeit DSU and OSR both manipulate the stack to replace running functions, they target different applications, and the performance constraints they are subject to are dissimilar.

In tracing JIT compilers deoptimization techniques are used to safely leave an optimized trace when a guard fails. SPUR [12] is a trace-based JIT compiler for Microsoft's Common Intermediate Language (CIL) with three levels of JIT-ting plus a transfer-tail JIT used to bridge the execution from an instruction in a block generated at the second or third level to a safe point for deoptimization to the first JIT level. Deoptimization can thus happen without falling back to an interpreter; similarly, our approach enables VM builders to perfom OSR transitions working at native-code level only.

In RPython, guards are implemented as a conditional jump to a trampoline that analyzes resume information for the guard and executes compensation code to leave the trace; resume data is compactly encoded by sharing parts of the data structure between subsequent guards [91]. A similar approach is used in LuaJIT, where sparse snapshots are taken to enable state restoration when leaving a trace [82]. For deoptimization purposes, it would be interesting to investigate whether, given a sequence of source instructions, a single continuation function could be created, by adding a dispatcher in its entry block that compensates the state according to the current source for the OSR.

4.2 Proving On-Stack Replacement Sound

On-stack Replacement is employed in modern adaptive compilation systems to dynamically switch between different versions of a function depending on program's run-time state. Traditionally, code optimizers are responsible for marking the points where such transitions can take place, and generating required meta-data or ad-hoc code to get the program state to a correct resumption point. OSR is usually at the core of large and complex JIT compilers employed by popular production virtual machines. Indeed, the engineering effort to integrate this technique in a language runtime can be daunting, making it rarely accessible to the research community.

Contributions. In this thesis we investigate how to provide VM builders with a rich "menu" of possible program points where OSR can safely occur, relieving code optimizers from the burden of generating all the required machinery to realign the program state during an OSR transition.

To capture OSR in its full generality, we define a notion of *multi-program*, which is a collection of different versions of a program along with support to dynamically transfer execution between them. Execution in a multi-program starts from a designated base version. At any time, an oracle decides whether execution should continue in the current version, or an OSR should divert it to a different version, modeling any conceivable OSR-firing strategy. One of the goals of our work is to characterize sufficient conditions for a multi-program to be *deterministic*, yielding the same result regardless of the oracle's decisions. This captures the intuitive idea that any sequence of OSR transitions is *correct* if it does not alter the intended

$$
\begin{array}{rcl}
\mathit{Instr} & ::= & \mathit{Var}\ \texttt{:=}\ \mathit{Expr} \\
 & & \texttt{|}\ \texttt{if (}\ \mathit{Expr}\ \texttt{) goto}\ \mathit{Num} \\
 & & \texttt{|}\ \texttt{goto}\ \mathit{Num} \\
 & & \texttt{|}\ \texttt{skip} \\
 & & \texttt{|}\ \texttt{abort} \\
 & & \texttt{|}\ \texttt{in}\ \mathit{Var}\cdots\mathit{Var} \\
 & & \texttt{|}\ \texttt{out}\ \mathit{Var}\cdots\mathit{Var} \\
\mathit{Expr} & ::= & \mathit{Num}\ \texttt{|}\ \mathit{Var}\ \texttt{|}\ \mathit{Expr}\ \texttt{+}\ \mathit{Expr}\ \texttt{|}\ \ldots \\
\mathit{Var} & ::= & \texttt{X | Y | Z | ...} \\
\mathit{Num} & ::= & \texttt{... | -2 | -1 | 0 | 1 | 2 | ...}
\end{array}
$$

Figure 4.9. Program Syntax

semantics of a program.

We distill the essence of OSR to a simple imperative calculus with an operational semantics. Using program bisimulation, we prove that an OSR can correctly divert execution from one program version to the other if they are *live-variable bisimilar*, i.e., the live variables they have in common at any corresponding execution states are equal. As prominent examples of how bisimulation can be used to prove this property, we consider classic optimizations that eliminate or move code around, such as dead code elimination, constant propagation, and code hoisting. We show how to construct OSR machinery by devising an algorithm that automatically generates compensation code to reconstruct the values of variables that are live in the OSR target, but not in the source.

4.2.1 Language Syntax and Semantics

Our discussion is based on a minimal imperative programming whose syntax is reported in Figure 4.9. In this section we introduce some basic definitions used in our representation of programs, and provide a big-step semantics for the language.

Definition 8 (Program) *A program is a sequence of instructions the form:*

$$\pi = \langle I_1, I_2, \ldots, I_n \rangle \in \mathit{Prog} = \bigcup_{i=2}^{\infty} \mathit{Instr}^i$$

where:

- $I_i \in \mathit{Instr}$ *is the i-th instruction of the program, indexed by program point* $i \in [1, n]$
- $I_1 = \texttt{in} \cdots$ *is the initial instruction*
- $\forall i \in [2, n-1] : I_i \neq \texttt{in} \cdots \wedge I_i \neq \texttt{out} \cdots$
- $I_n = \texttt{out} \cdots$ *is the final instruction*

Instruction `in`, which must appear at the beginning of a program, specifies the variables that must be defined prior to entering the program. Similarly, `out` occurs at the end and specifies the variables that are returned as output.

By `e[x]` we indicate that `x` is a variable of expression `e` $\in$ *Expr*. We also denote by $\mathit{vars}(\texttt{e})$ the set of variables that occur in expression `e`. By $|\pi| = n$ we indicate the number of instructions in $\pi = \langle I_1, I_2, \ldots, I_n \rangle$.

Definition 9 (Memory Store) *A* memory store *is a total function $\sigma : Var \to \mathbb{Z} \cup \{\bot\}$ that associates integer values to defined variables, and $\bot$ to undefined variables. We denote by Σ the set of all possible memory stores.*

By $\sigma[\texttt{x} \leftarrow v]$ we denote the same function as σ, except that `x` takes value v. Furthermore, for any $A \subseteq Var$, $\sigma|_A$ denotes σ restricted to the variables in A, i.e., $\sigma|_A(\texttt{x}) = \sigma(\texttt{x})$ if $\texttt{x} \in A$ and $\sigma|_A(\texttt{x}) = \bot$ if $\texttt{x} \notin A$.

Definition 10 (Program State) *The* state *of a program $\pi = \langle I_1, I_2, \ldots, I_n \rangle$ is described by a pair (σ, l), where σ is a memory store and $l \in [1, n]$ is the program point of the next instruction to be executed. We denote by $State = \Sigma \times \mathbb{N}$ the set of all possible program states.*

We provide a big-step semantics using transition relation $\Rightarrow_\pi \subseteq State \times State$, which specifies how a single instruction of a program π affects its state. Our description relies on relation $\Downarrow \subseteq (\Sigma \times Expr) \times \mathbb{Z}$ to describe how expressions are evaluated in a given memory store.

Definition 11 (Big-Step Transitions) *For any program π, we define relation $\Rightarrow_\pi \subseteq State \times State$ as follows, with meta-variables $\texttt{x}, \texttt{y} \in Var$, $\texttt{e} \in Expr$, and $\texttt{m} \in Num$:*

$$\frac{I_l = \texttt{x:=e} \quad \wedge \quad (\sigma, \texttt{e}) \Downarrow v}{(\sigma, l) \Rightarrow_\pi (\sigma[\texttt{x} \leftarrow v], l+1)} \tag{4.1}$$

$$\frac{I_l = \texttt{if (e) goto m} \quad \wedge \quad (\sigma, \texttt{e}) \Downarrow 0}{(\sigma, l) \Rightarrow_\pi (\sigma, l+1)} \tag{4.2}$$

$$\frac{I_l = \texttt{if (e) goto m} \quad \wedge \quad (\sigma, \texttt{e}) \Downarrow v \quad \wedge \quad v \neq 0}{(\sigma, l) \Rightarrow_\pi (\sigma, \texttt{m})} \tag{4.3}$$

$$\frac{I_l = \texttt{goto m}}{(\sigma, l) \Rightarrow_\pi (\sigma, \texttt{m})} \tag{4.4}$$

$$\frac{I_l = \texttt{skip}}{(\sigma, l) \Rightarrow_\pi (\sigma, l+1)} \tag{4.5}$$

$$\frac{I_1 = \texttt{in x y} \cdots \quad \wedge \quad \sigma(\texttt{x}) \neq \bot \quad \wedge \quad \sigma(\texttt{y}) \neq \bot \quad \wedge \quad \cdots}{(\sigma, 1) \Rightarrow_\pi (\sigma, 2)} \tag{4.6}$$

$$\frac{I_n = \texttt{out x y} \cdots \quad \wedge \quad \sigma(\texttt{x}) \neq \bot \quad \wedge \quad \sigma(\texttt{y}) \neq \bot \quad \wedge \quad \cdots}{(\sigma, n) \Rightarrow_\pi (\sigma|_{\{\texttt{x},\texttt{y},\cdots\}}, n+1)} \tag{4.7}$$

For a transition to apply, we implicitly assume that I_l is defined, i.e., $l \in [1, n]$.

Definition 12 (Program Semantic Function) *We define the semantic function $[\![\pi]\!] : \Sigma \to \Sigma$ of a program π as:*

$$\forall \sigma \in \Sigma : \quad [\![\pi]\!](\sigma) = \sigma' \quad \Longleftrightarrow \quad (\sigma, 1) \Rightarrow^*_\pi (\sigma', |\pi| + 1)$$

where $\Rightarrow^*_\pi$ is the transitive closure of $\Rightarrow_\pi$.

Note that a program has undefined semantics if its execution on a given store does not reach the final `out` instruction. This accounts for infinite loops, abort instructions, exceptions, and ill-defined programs or input stores.

We define the notion of program semantic equivalence as follows:

Definition 13 (Program Equivalence) *Two programs π_1 and π_2 are* semantically equivalent *iff* $[\![\pi_1]\!] = [\![\pi_2]\!]$.

A notion that will be useful in proving correctness in our framework is that of *trace* of a transition system:

Definition 14 (Traces) *A* trace *in a transition system $(S,\ R \subseteq S^2)$ starting from $s \in S$ is a sequence $\tau = \langle s_0, s_1, \ldots,\ s_i, \ldots\rangle$ such that $s_0 = s$ and $\forall i \geq 0 :\ \ s_i \in \tau \wedge s_i\ R\ s_{i+1} \iff s_{i+1} \in \tau$. By $\mathcal{T}_{R,s}$ we denote the system of all traces of $(S, R \subseteq S^2)$ starting from s. By $\tau[i]$ we denote the i-th state of τ, i.e., $\tau[i] = s_i$. Furthermore, if trace τ is finite then $|\tau|$ denotes the index of its final state, i.e., $\tau = \langle s_0, s_1, \ldots, s_{|\tau|}\rangle$, otherwise $|\tau| = \infty$. Finally, $dom(\tau) = \{i : s_i \in \tau\}$ denotes the set of indices of states in τ.*

Notice that, since $\Rightarrow_\pi$ is deterministic in our language, then for any initial store σ, the system of traces $\mathcal{T}_{\Rightarrow_\pi,(\sigma,1)}$ of the execution transition system $(Store, \Rightarrow_\pi)$ contains a single trace, which we denote by $\tau_{\pi\sigma}$.

Finally, we provide a formal definition of control flow graph, which will be useful in defining computation tree logic operators for reasoning on program properties:

Definition 15 (Control Flow Graph) *The* control flow graph *G for a program $\pi = \langle I_1, I_2, \ldots, I_n\rangle$ is described by a pair $(V, E \subseteq V \times V)$ where:*

$$
\begin{aligned}
V &= \{I_1, I_2, \ldots, I_n\}\\
E &= \{(I_i, I_{i+1}) \mid I_i \neq \textsf{abort} \wedge I_i \neq \textsf{goto m},\ \textsf{m} \in Num\}\\
&\quad \cup\ \{(I_i, I_m) \mid I_i = \textsf{goto m} \vee I_i = \textsf{if (e) goto m},\ \textsf{m} \in Num,\ \textsf{e} \in Expr\}.
\end{aligned}
$$

4.2.2 Program Properties and Transformations

In this section we present a formalism based on *computation tree logic* (CTL) to reason about program properties operators and describe program transformations through rewrite rules with side conditions [30, 64, 61].

Reasoning about Program Properties

To analyze properties of a program, we use Boolean formulas with free meta-variables that combine facts that must hold globally or at certain points of a program. Formulas can be checked against concrete programs by a *model checker*. For any program π and formula ϕ, the checker verifies if there exists a substitution θ that binds free meta-variables with program objects so that $\theta(\phi)$ is satisfied in π. In this paper, by $\mathcal{A} \models \phi$ we mean that ϕ is true in $\mathcal{A}$, i.e., formula ϕ is satisfied by structure $\mathcal{A}$ (or equivalently, $\mathcal{A}$ models ϕ) [30].

Two global predicates that we will use later on are `conlit(c)`, which states that an expression `c` is a constant literal, and `freevar(x, e)`, which holds if and only if `x` is a free variable of expression `e`.

To support analyses based on facts that involve finite maximal paths in the control flow graph (CFG), such as liveness and dominance, we use formulas based on CTL operators. In order to introduce these operators, we need to formalize the concept of finite maximal paths first.

Definition 16 (Set of Complete Paths) *Given a control flow graph $G = (V, E)$ and an initial node $n_0 \in V$, the* set of complete paths *$CPaths(n_0, G)$ starting at n_0 consists of all finite sequences $\langle n_0, n_1, \ldots, n_k \rangle$ such that $(n_i, n_{i+1}) \in E$ for all n_i with $i < k$, and such that there does not exist a n_{k+1} such that $(n_k, n_{k+1}) \in E$.*

Complete paths from a specified node (i.e., instruction) are thus maximal finite sequences of connected nodes through a control flow graph from an initial point to a sink node, which in our setting is unique (unless `abort` instructions are present) and corresponds to the final instruction I_n.

First-order CTL can be used to specify properties of nodes and paths in a CFG. In particular, temporal CTL operators can be used to express properties of some or all possible future computational paths, any one of which might be an actual path that is realized. Before formalizing the temporal operators that we are going to use in the remainder of this chapter, we provide an intuitive definition for them. We say that, given a point l in a program π and two formulas ϕ and ψ, the following predicates are satisfied at l if:

- $\overrightarrow{AX}(\phi)$: ϕ holds for all immediate successors of l;
- $\overrightarrow{EX}(\phi)$: ϕ holds for at least one immediate successor of l;
- $\overrightarrow{A}(\phi\ U\ \psi)$: ϕ holds on all paths from l, until ψ holds;
- $\overrightarrow{E}(\phi\ U\ \psi)$: ϕ holds on at least one path from l, until ψ holds.

Corresponding operators $\overleftarrow{AX}$ and $\overleftarrow{EX}$ are defined for immediate predecessors of l, while $\overleftarrow{A}$ and $\overleftarrow{E}$ refer to backward paths from l.

Definition 17 (Temporal Operators) *Given a node n in the control flow graph $G = (V, E)$ of a program π, we define the following CTL* temporal operators *as:*

$$\begin{aligned}
n &\models \overrightarrow{AX}(\phi) \iff \forall m : (n, m) \in E : \pi, m \models \phi \\
n &\models \overrightarrow{EX}(\phi) \iff \exists m : (n, m) \in E : \pi, m \models \phi \\
n &\models \overrightarrow{A}(\phi\ U\ \psi) \iff \forall p : p \in CPaths(n, G) : Until(\pi, p, \phi, \psi) \\
n &\models \overrightarrow{E}(\phi\ U\ \psi) \iff \exists p : p \in CPaths(n, G) : Until(\pi, p, \phi, \psi)
\end{aligned}$$

where predicate $Until(\pi, p, \phi, \psi)$ holds for $p = \langle n_0, n_1, \ldots, n_k \rangle \in CPaths(n_0, G)$ if:

$$\exists j : 0 \le j \le k : \pi, n_j \models \psi \ \wedge\ \forall 0 \le i < j : \pi, n_i \models \phi$$

Operators $\overleftarrow{AX}$, $\overleftarrow{EX}$, $\overleftarrow{A}$, and $\overleftarrow{E}$ can be defined similarly on the reverse control flow graph $\overleftarrow{G}$, which is identical to G but with every edge in $\overleftarrow{E}$ flipped.

Operators A and E are quantifiers over paths, while X and U path-specific quantifiers. Notice that $\phi\ U\ \psi$ requires that ϕ as to hold at least until at some node ψ is satisfied: this implies that ψ will be verified in the future.

$$\begin{array}{rcl}
\texttt{def(x)} & \triangleq & I_l = \texttt{x:=e} \;\;\vee\;\; I_l = \texttt{in}\ \cdots\ \texttt{x}\cdots \\
 & & [\texttt{x}\ \textit{is defined by instruction}\ I_l\ \textit{in}\ \pi] \\
\texttt{use(x)} & \triangleq & I_l = \texttt{y:=e[x]}\ \vee \\
 & & I_l = \texttt{if (e[x]) goto m}\ \vee \\
 & & I_l = \texttt{out}\ \cdots\ \texttt{x}\cdots \\
 & & [\texttt{x}\ \textit{is used by instruction}\ I_l\ \textit{in}\ \pi] \\
\texttt{trans(e)} & \triangleq & I_l = \texttt{x:=e'}\ \wedge\ \neg\texttt{freevar}(\texttt{x},\texttt{e})\ \vee \\
 & & I_l \neq \texttt{x:=e'} \\
 & & [\textit{no constituent of}\ \texttt{e}\ \textit{is modified by instruction}\ I_l\ \textit{in}\ \pi] \\
\texttt{is\_live(x)} & \triangleq & \overleftarrow{AX}\overleftarrow{A}(\text{true}\ U\ \texttt{def(x)})\ \wedge \\
 & & \overrightarrow{E}(\neg\texttt{def(x)}\ U\ \texttt{use(x)}) \\
 & & [\texttt{x}\ \textit{is live at program point}\ l\ \textit{in}\ \pi] \\
\texttt{urdef}(\texttt{x},l') & \triangleq & \overleftarrow{AX}\overleftarrow{A}(\neg\texttt{def(x)}\ U\ \texttt{point}(l') \wedge \texttt{def(x)}) \\
 & & [\textit{unique definition of}\ \texttt{x}\ \textit{at}\ l'\ \textit{reaching}\ l\ \textit{in}\ \pi] \\
\texttt{stmt}(I) & \triangleq & I = I_l \quad [I\ \textit{is the instruction at}\ l\ \textit{in}\ \pi] \\
\texttt{point(m)} & \triangleq & \texttt{m} = l \quad [\textit{program point}\ \texttt{m}\ \textit{is}\ l\ \textit{in}\ \pi]
\end{array}$$

Figure 4.10. Predicates expressing local properties of a point $l \in [1, n]$ in a program $\pi = \langle I_1, \ldots, I_n\rangle$, with meta-variables $\texttt{e}, \texttt{e'} \in \mathit{Expr}$, $\texttt{x}, \texttt{y} \in \mathit{Var}$, and $l, \texttt{m} \in \mathit{Num}$.

Figure 4.10 shows a number of local predicates that will be useful throughout the paper. For instance, $\pi, l \models \texttt{urdef}(\texttt{x}, l')$ (*unique reaching definition*) holds if and only if variable x is defined at l and on all paths in the control flow graph starting from an immediate successor of l, x is not redefined until point l' is reached, i.e., there is a unique definition of x that reaches l', and this definition is at l. urdef's formulation relies on nested CTL operators: $\overrightarrow{AX}$ is used to encode a property for all successors of l, while the nested $\overrightarrow{A}$ captures all forward paths starting at such nodes.

The following definition will be useful, too:

Definition 18 (Live Variables) *The set of live variables of a program π at point l is defined as:*

$$\texttt{live}(\pi, l) \triangleq \{\ \texttt{x} \in \mathit{Var} \mid \pi, l \models \texttt{is\_live(x)}\ \}$$

Example 5 *Dominance analysis is widely employed in a number of program analysis and optimization techniques. In a CFG, we say that a node n dominates another node m if every path from the CFG's entry node to m must go through n. Using CTL operators, we can easily encode this property. Given a program π as in Definition 8, we can write:*

$$\texttt{dom}(n, m) \Longleftrightarrow \pi, I_1 \models \neg E(\neg\texttt{point}(n)\ U\ \texttt{point}(m))$$

which captures the idea that there does not exist a path starting at the entry node (i.e, the first instruction in π) that reaches m without reaching n first.

Program Transformations

To describe program transformations, we use rewrite rules with side conditions in a similar manner as [64, 62]. We consider generalized rules that transform multiple instructions simultaneously, with side conditions drawn from CTL formulas:

Definition 19 (Rewrite Rule) *A rule T has the form:*

$$T = \quad m_1 : \hat{I}_1 \Longrightarrow \hat{I'}_1 \quad \cdots \quad m_r : \hat{I}_r \Longrightarrow \hat{I'}_r \quad \texttt{if } \phi$$

where $\forall k \in [1, r]$, m_k is a meta-variable that denotes a program point, $\hat{I}_k$ and $\hat{I'}_k$ are program instructions that can contain meta-variables, and ϕ is a side condition that states whether the rewriting rule can be applied to the input program. We denote by $\mathcal{T}$ the set of all possible rewrite rules.

An elementary example of rewrite rule with meta-variables `m`, `x`, and `y` is:

$$m : \; \texttt{y} := 2 * \texttt{x} \quad \Longrightarrow \quad \texttt{y} := \texttt{x} + \texttt{x} \quad \texttt{if } \mathit{true}$$

which implements a peephole optimization based on a weak form of operator strength reduction [31].

Rules can be applied to concrete programs by a transformation engine based on model checking: when the checker finds a substitution θ that binds free meta-variables with program objects so that $\theta(\phi)$ is satisfied in π and $\theta(\hat{I}_k) = I_{\theta(m_k)} \in \pi$ for some $k \in [1, t]$, then $I_{\theta(m_k)}$ is replaced with $\theta(\hat{I'}_k) = I'_{\theta(m_k)} \in \pi'$, as formalized next:

Definition 20 (Rule Semantics) *Let T be a rewrite rule as in Definition 19. Transformation function $[\![T]\!] : Prog \to Prog$ is defined as follows:*

$$\forall \pi, \pi' \in Prog : \pi' = [\![T]\!](\pi) \Longleftrightarrow \exists\, \theta : \; \pi \models \theta(\phi) \; \wedge \\ \forall k \in [1, r] : \theta(\hat{I}_k) = I_{\theta(m_k)} \in \pi \; \wedge \; \theta(\hat{I'}_k) = I'_{\theta(m_k)} \in \pi'$$

In this thesis, we focus on transformations that do not alter the semantics of a program:

Definition 21 (Semantics-Preserving Rules) *A rewrite rule T is* semantics-preserving *if for any program π it holds $[\![\pi]\!] = [\![\pi']\!]$, where $\pi' = [\![T]\!](\pi)$.*

Examples of semantics-preserving rules for classic compiler optimizations (as proven in [63, 64]) are given in Figure 4.11.

The constant propagation (CP) rule replaces uses of a variable v at a node m with a constant c. Its side condition is satisfied when in all backward paths starting at m, the first definition of v we encounter is always $v := c$.

The dead store elimination (DSE) rule deletes an intruction at a node m if the result of its computation will never be used later in the program's execution. As we are not interested in uses of the variable itself at m, in the side condition we skip past it with AX and specify that there should not exist a forward path that eventually uses (i.e., reads from) the variable.

Constant propagation (CP):
$m: \texttt{x := e[v]} \implies \texttt{x := e[c]}$
$\texttt{if}\ \ \texttt{conlit(c)} \ \wedge\ m \models \overleftarrow{A}(\neg\texttt{def(v)}\ U\ \texttt{stmt(v := c)})$

Dead store elimination (DSE):
$m: \texttt{x := e} \implies \texttt{skip}$
$\texttt{if}\ \ m \models \overrightarrow{AX}\ \neg\overrightarrow{E}(\mathit{true}\ U\ \texttt{use(x)})$

Code hoisting (Hoist):
$p: \texttt{skip} \implies \texttt{x := e}$
$q: \texttt{x := e} \implies \texttt{skip}$
$\texttt{if}\ \ p \models \overrightarrow{A}(\neg\texttt{use(x)}\ U\ \texttt{point}(q))\ \wedge$
$\quad q \models \overleftarrow{A}((\neg\texttt{def(x)} \vee \texttt{point}(q)) \wedge \texttt{trans}(e)\ U\ \texttt{point}(p))$

Figure 4.11. Rewriting rules for defining CP, DCE, and Hoist transformations.

Finally, the code hoisting (Hoist) rule moves an assignment of the form $x := v[e]$ from a node q to a node p provided that two conditions are met. The first requires that in all forward paths starting at the insertion point p, x is not used until the original location q is reached. The second requires that in all backward paths starting at q, x is not reassigned at any node other than q and the constituents of e are not redefined, until the insertion point p is reached.

4.2.3 On-Stack Replacement Framework

OSR consists in dynamically transferring execution from a point l in a program π to a point l' in a program π' so that execution can transparently continue from π' without altering the original intended semantics of π. To model this behavior, we assume there exists a function that maps each point l in π where OSR can safely be fired to the corresponding point l' in π' from which execution can continue. As we observed in Section 4.1.1, the OSR practice often makes the conservative assumption that π' can always continue from the very same memory store as π. However, this assumption may reduce the number of points where sound OSR transitions can be fired. To overcome this limitation and support more aggressive OSR transitions, our model includes a *store compensation code* χ to be executed during an OSR transition from point l in π to point l' in π'. The goal of the compensation code is to fix the memory store of π at l so that execution can safely continue in π' from l' with the fixed store. Note that, if no compensation is needed for an OSR transition, $[\![\chi]\!]$ is simply the identity function. We formalize these concepts in the next sections.

4.2.3.1 OSR Mappings

The machinery required for performing OSR transitions between two programs can be modeled as an *OSR mapping*:

Definition 22 (OSR Mapping) *For any $\pi, \pi' \in Prog$, an* OSR mapping *from π*

to π' is a (possibly partial) function $\mu_{\pi\pi'} : [1, |\pi|] \to [1, |\pi'|] \times Prog$ such that:

$$\forall \sigma \in \Sigma, \forall s_i = (\sigma_i, l_i) \in \tau_{\pi\sigma} \ s.t. \ l_i \in dom(\mu_{\pi\pi'}),$$
$$\exists \sigma' \in \Sigma, \exists s_j = (\sigma_j, l_j) \in \tau_{\pi'\sigma'} \ s.t.$$
$$\mu_{\pi,\pi'}(l_i) = (l_j, \chi) \ \wedge \ [\![\chi]\!](\sigma_i|_{\texttt{live}(\pi,l_i)}) = \sigma_j|_{\texttt{live}(\pi',l_j)}$$

We say that the mapping is strict *if $\sigma' = \sigma$. We denote by OSRMap the set of all possible OSR mappings between any pair of programs.*

Intuitively, an OSR mapping provides the information required to transfer execution from any realizable state of π, i.e., an execution state that is reachable from some initial store by π, to a realizable state of π'. Notice that this definition is rather general, as a non-strict mapping allows execution to be transferred to a program π' that is *not* semantically equivalent to π. For instance, π' may contain speculatively optimized code, or just some optimized fragments of π [51, 9, 49]. In those scenarios, one typically assumes that execution in π' can be invalidated by performing an OSR transition back to π or to some other recovery program. We also observe that Definition 22 uses a weak notion of store equality restricted to live variables. To simplify the discussion, we assume that the memory store is only defined on scalar variables (we address extensions to memory `load` and `store` instructions in Section**XXX**). Hence, the behavior of a program only depends on the content of its live variables, as stated in the following lemma:

Lemma 1 *For any program $\pi \in Prog$, any $\sigma, \sigma' \in \Sigma$, and any $l, l' \in \mathbb{N}$, it holds:*

$$(\sigma, l) \Rightarrow_\pi (\sigma', l') \iff (\sigma|_{\texttt{live}(\pi,l)}, l) \Rightarrow_\pi (\sigma'|_{\texttt{live}(\pi,l')}, l')$$

PROOF. We reason on the structure of the transition relation $\Rightarrow_\pi$ for our big-step semantics shown in Definition 11. We rewrite our claim as:

$$(\sigma, l) \Rightarrow_\pi (\sigma', l') \iff (\sigma|_{\texttt{live}(\pi,l)}, l) \Rightarrow_\pi (\hat{\sigma}, l') \ \wedge \ \hat{\sigma}|_{\texttt{live}(\pi,l')} = \sigma'|_{\texttt{live}(\pi,l')}$$

When Equation (4.1) applies, both states advance to location $l+1$, and the evaluation $(\sigma, \texttt{e}) \Downarrow v$ for the assignment yields the same result in both stores, as each operand in e is either a constant literal or a live variable for π at l. Indeed, having a variable operand for e not in $\texttt{live}(\pi, l)$ would contradict the definition of liveness. When the instruction at l is a conditional expression, $\Rightarrow_\pi$ applies either Equation (4.2) or Equation (4.3) to both states: as discussed for assignments, the evaluation of expression e yields the same result in σ and $\sigma|_{\texttt{live}(\pi,l)}$, and both states advance to the same location without affecting the store. When one of Equations (4.4) to (4.7) applies, trivially both states advance to the same location, while values in their stores are not affected. Finally, from Definition 18 it follows that $\texttt{live}(\pi, l') \supseteq \texttt{live}(\pi, l) \cup \{ \texttt{x} \mid I_l = \texttt{x:=e} \}$ and thus $\hat{\sigma}|_{\texttt{live}(\pi,l')} = \sigma'|_{\texttt{live}(\pi,l')}$. □

Notice that $dom(\mu_{\pi\pi'}) \subseteq [1, |\pi|]$ is the set of all possible points in π where OSR transitions to π' can be fired. If $\mu_{\pi\pi'}$ is partial, then there are points in π where OSR cannot be fired. In Section**XXX**, we discuss an algorithm whose goal is to minimize the number of these points.

4.2.3.2 OSR Mapping Generation Algorithm

We now discuss an algorithm that, given a program π and a rewrite rule T, generates:

1. a program $\pi' = [\![T]\!](\pi)$;
2. an OSR mapping $\mu_{\pi\pi'}$ from π to π';
3. an OSR mapping $\mu_{\pi'\pi}$ from π' to π.

Mappings $\mu_{\pi\pi'}$ and $\mu_{\pi'\pi}$ produced by the algorithm are based on compensation code that runs in $O(1)$ time and support bidirectional OSR between π and π', enabling invalidation and deoptimization. The algorithm, which we call `OSR_trans`, is shown in Algorithm 7. In Section**XXX**, we prove that the algorithm is correct under the sufficient condition that variables that are live at corresponding points in the original and rewritten program contain the same values.

Input: Program π, transformation T.
Output: Program π', OSR mappings $\mu_{\pi\pi'}$ and $\mu_{\pi'\pi}$.

```
algorithm OSR_trans($\pi, T$)→($\pi'$,$\mu_{\pi\pi'}$,$\mu_{\pi'\pi}$):
1  ($\pi', \Delta, \Delta'$) ← apply($\pi, T$)
2  foreach $l \in dom(\Delta)$ do
3      $\chi$ ← build_comp($\pi, l, \pi', \Delta(l)$)
4      if $\chi \neq$ undef then $\mu_{\pi\pi'}(l) \leftarrow (\Delta(l), \chi)$
5  end
6  foreach $l' \in dom(\Delta')$ do
7      $\chi$ ← build_comp($\pi', l', \pi, \Delta'(l')$)
8      if $\chi \neq$ undef then $\mu_{\pi'\pi}(l') \leftarrow (\Delta'(l'), \chi)$
9  end
10 return ($\pi', \mu_{\pi\pi'}, \mu_{\pi'\pi}$)
```

Algorithm 7: `OSR_trans` algorithm for OSR mapping construction. Functions Δ and Δ' are used to map OSR program points between π and π' (and viceversa).

`OSR_trans`. The algorithm relies on two subroutines: `apply` and `build_comp`. Procedure `apply` takes as input a program π and a program rewriting function T, and returns a transformed program π' and two functions $\Delta : [1, |\pi|] \to [1, |\pi'|]$, $\Delta' : [1, |\pi'|] \to [1, |\pi|]$ that map OSR program points between π and π'. Algorithm `build_comp` (shown in Algorithm 8) takes as input π, l, π', l' and aims to build a *store compensation code* χ that allows firing an OSR from π at l to π' at l'. `OSR_trans` first calls `apply` and then uses `build_comp` on π, π', Δ, Δ' to build OSR mappings $\mu_{\pi\pi'}, \mu_{\pi'\pi}$. Lines 2–5 build the forward mapping $\mu_{\pi\pi'}$ from l in π to $\Delta(l)$ in π', while lines 6–9 build the backward mapping $\mu_{\pi'\pi}$ from l' in π' to $\Delta'(l')$ in π. If any of the live variables at the OSR destination cannot be guaranteed to be correctly assigned, no entry is created in $\mu_{\pi\pi'}$ or $\mu_{\pi'\pi}$ for the OSR origin point (lines 4 and 8). Hence, those points will not be eligible for OSR transitions. In Section **XXX** we will analyze experimentally the fraction of points for which a compensation code can be created by algorithm `build_comp` in a variety of prominent benchmarks.

Input: Program π, point l, program π', point l'.
Output: Store compensation code χ.

```
algorithm build_comp($\pi$, $l$, $\pi'$, $l'$)$\to$ $\chi$:
1   $\chi \leftarrow$ in $x_1\ x_2\ \cdots\ x_k\ :\ \forall i \in [1,k] : \pi, l \models$ live($x_i$)
2   mark all program points of $\pi'$ as unvisited
3   try
4       foreach x : $\pi', l' \models$ live(x) $\wedge\ \pi, l \models \neg$live(x) do
5           $\chi \leftarrow \chi \cdot$ reconstruct(x, $\pi, l, \pi', l', l'$)
6       end
7   catch
8       return undef
9   $\chi \leftarrow \chi \cdot$ out $x_1\ x_2\ \cdots\ x_{k'} : \forall i \in [1,k'] : \pi', l' \models$ live($x_i$)
10  return $\chi$
```

Algorithm 8: `build_comp` algorithm for compensation code construction.

`build_comp`. Code shown in Algorithm 8 generates a program χ that starts with an `in` statement with the live variables at origin l in π (line 1), and ends with an `out` statement with the live variables at destination l' in π' (line 9). The goal of χ is to make sure that all `out` variables are correctly assigned, either because they already hold the correct value upon entry, or because they can be computed in terms of the input variables. The algorithm iterates on all variables x_i that are live at destination, but not at origin (line 4). For each of them, it calls a subroutine `reconstruct` that builds a code fragment that assigns x_i with its correct value using live variables at origin (line 5). If this value cannot be determined, `reconstruct` throws an exception and `build_comp` returns an undefined compensation code (line 8), which implies that OSR cannot be performed at l. To avoid code duplication in χ and unnecessary work, the algorithm assumes all points in π' are initially unvisited (line 2) and lets `reconstruct` mark them visited along the way. Algorithm `build_comp` can be implemented with a running time linearly bounded by the size of π'.

`reconstruct`. The procedure (shown in Algorithm 9) takes a variable `x`, the OSR origin and destination points l and l' in π and π', respectively, and an additional point l'' in π'. It builds a straight-line code fragment that assigns `x` with the value it would have had at l'' just before reaching l' if execution had been carried on in π' instead of π. The algorithm first checks if there is a unique definition of `x` of the form `x` := `e` at some point $\hat{l}$ that reaches point l'' in π'' (`urdef` at line 1). If such reaching definition is not unique, then the live information available in π at l is deemed insufficient to determine what value `x` would have assumed in π', and the algorithm gives up (line 10). The algorithm assumes liveness and reaching definition analyses are available to compute predicates `live` and `urdef` (see Section **XXX**). If `x` is live both at the origin l and at the destination l', and the definition of `x` at $\hat{l}$ that reaches l'' is also a unique definition reaching l' (line 4), then `x` would have assumed at l'' the same value available at l'. In Section **XXX** we will see that, if we can guarantee that live variables at origin have the same values they would

```
procedure reconstruct(x, $\pi$, $l$, $\pi'$, $l'$, $l''$):
1  if $\exists\hat{l} : \pi', l'' \models$ urdef(x, $\hat{l}$) $\wedge\ \pi', \hat{l} \models$ stmt(x:=e) then
2      if $\hat{l}$ is visited then return $\langle\rangle$
3      mark $\hat{l}$ as visited
4      if $\pi', l' \models$ urdef(x, $\hat{l}$) $\wedge$ $\pi', l' \models$ live($x$) $\wedge$ $\pi, l \models$ live($x$) then
       return $\langle\rangle$
5      $\chi \leftarrow \langle\rangle$
6      foreach y : y $\in$ freevar(e) do
7          $\chi \leftarrow \chi \cdot$ reconstruct(y, $\pi, l, \pi', l', \hat{l}$)
8      end
9      $\chi \leftarrow \chi \cdot$ x:=e
10 else throw undef
11 return $\chi$
```

Algorithm 9: Value reconstruction procedure used by `build_comp`.

have had at destination if execution had been performed in π', then the algorithm correctly assumes that `x` is available at origin and no compensation code is needed to reconstruct it (`return` at line 4). If `x` is not available at l, then the algorithm iterates over all free variables y of the expression e computed at $\hat{l}$ and recursively builds code that computes the values that they would have assumed at $\hat{l}$ just before reaching l' if execution had been carried on in π'. After the recursively generated code for assigning the constituents of `e` has been added to χ (lines 6–8), the assignment `x` := `e` is appended to χ (line 9).

4.2.3.3 Algorithm Correctness

We now prove the correctness of `OSR_trans`, showing that it yields strict OSR mappings if the applied rewrite rules satisfy the property that variables that are live at corresponding points in the original and rewritten program contain the same values. To characterize this property, we need to introduce some formal machinery based on bisimilarity of programs.

Definition 23 (Program Bisimulation) *A relation $R \subseteq State \times State$ is a bisimulation relation between programs π and π' if for any input store $\sigma \in \Sigma$ it holds:*

$$\begin{array}{l} s \in \tau_{\pi\sigma} \ \wedge\ s' \in \tau_{\pi'\sigma} \ \wedge\ s\ R\ s' \Longrightarrow \\ \qquad 1)\ \ s \Rightarrow_{\pi} s_1 \quad \Longrightarrow \quad s' \Rightarrow_{\pi'} s'_1 \ \wedge\ s_1\ R\ s'_1 \\ \qquad 2)\ \ s' \Rightarrow_{\pi'} s'_1 \quad \Longrightarrow \quad s \Rightarrow_{\pi} s_1 \ \wedge\ s_1\ R\ s'_1 \end{array}$$

Notice that our notion of bisimulation between programs π and π' requires that R be a bisimulation between transition systems $(\tau_{\pi\sigma}, \Rightarrow_{\pi})$ and $(\tau_{\pi'\sigma}, \Rightarrow_{\pi'})$ for any store $\sigma \in \Sigma$.

Lemma 2 *Let R be a reflexive bisimulation relation between programs π and π'. Then for any $\sigma \in \Sigma$ it holds:*

$$|\tau_{\pi\sigma}| = |\tau_{\pi'\sigma}| \qquad (4.8)$$

$$\forall i \in dom(\tau_{\pi\sigma}), \quad \tau_{\pi\sigma}[i]\ R\ \tau_{\pi'\sigma}[i] \tag{4.9}$$

Proof. We prove Equation (4.9) by induction on i. The base follows from $\tau_{\pi\sigma}[0] = \tau_{\pi'\sigma}[0] = (\sigma, 1)$ and the assumption that R is reflexive. Assume as inductive hypothesis that $\tau_{\pi\sigma}[i]\ R\ \tau_{\pi'\sigma}[i]$ for any $i < |\tau_{\pi\sigma}|$. Since $|\tau_{\pi\sigma}| > i$ then $\tau_{\pi\sigma}[i] \Rightarrow_\pi \tau_{\pi\sigma}[i+1]$ by Definition 14. It follows by Definition 23 that $\tau_{\pi\sigma}[i+1]\ R\ \tau_{\pi'\sigma}[i+1]$.

To prove Equation (4.8), assume by contradiction that $|\tau_{\pi\sigma}| \neq |\tau_{\pi'\sigma}|$, e.g., $|\tau_{\pi\sigma}| > |\tau_{\pi'\sigma}| = k$. Since $|\tau_{\pi\sigma}| > k$ then $\tau_{\pi\sigma}[k]\ R\ \tau_{\pi'\sigma}[k]$ by Equation (4.9) and $\tau_{\pi\sigma}[k] \Rightarrow_\pi \tau_{\pi\sigma}[k+1]$ by Definition 14. It follows by Definition 23 that $\tau_{\pi'\sigma}[k] \Rightarrow_{\pi'} \tau_{\pi'\sigma}[k+1]$. Hence $|\tau_{\pi'\sigma}| > k$, contradicting the initial assumption. The proof for the case $|\tau_{\pi'\sigma}| > |\tau_{\pi\sigma}|$ is analogous. □

Definition 24 (Partial State Equivalence) *For any function $A : \mathbb{N} \to 2^{Var}$, the* partial state equivalence *relation $R_A \subseteq State \times State$ is defined as:*

$$R_A \triangleq \{(s, s') \in State \times State \mid \\ s = (\sigma, l) \ \wedge\ s' = (\sigma', l) \ \wedge\ \sigma|_{A(l)} = \sigma'|_{A(l)}\}.$$

Relation R_A is clearly reflexive, symmetric, and transitive.

Definition 25 (Live-Variable Bisimilar Programs) *π and π' are* live-variable bisimilar *(LVB) if R_A is a bisimulation relation between them, where $A = l \mapsto$ $\texttt{live}(\pi, l) \cap \texttt{live}(\pi', l)$ is the function that yields for each program point l the set of variables that are live at l in both π and π'.*

One consequence of Definition 24, which simplifies our formal discussion, is the following:

Lemma 3 *If π and π' are live-variable bisimilar, then for any σ, corresponding states in program traces $\tau_{\pi\sigma}$ and $\tau_{\pi'\sigma}$ are located at the same program points: $\forall i : \tau_{\pi\sigma}[i] = (\sigma_i, l_i) \wedge \tau_{\pi'\sigma}[i] = (\sigma'_i, l'_i) \Longrightarrow l_i = l'_i$.*

Proof. Straightforward by Lemma 2 and Definition 24. □

Corollary 1 *If π and π' are live-variable bisimilar, then they have the same size: $\pi = \langle I_1, \ldots, I_n \rangle \wedge \pi' = \langle I'_1, \ldots, I'_{n'} \rangle \Longrightarrow n = n'$.*

Proof. By Lemma 2, Lemma 3, and Equation (4.7), for any finite trace $\tau_{\pi\sigma}$ it holds $\tau_{\pi\sigma}[|\tau_{\pi\sigma}|] = (-, n+1)$ and $\tau_{\pi'\sigma}[|\tau_{\pi'\sigma}|] = (-, n+1)$. Hence both π and π' contain n instructions. □

We can now prove that `build_comp` generates correct OSR compensation code under the live-variable bisimilarity assumption.

Lemma 4 (Correctness of Algorithm `build_comp`) *Let π and π' be live-variable bisimilar programs. For each initial store $\sigma \in \Sigma$ it holds:*

$$\forall i \in dom(\tau_{\pi\sigma}) : \chi \neq undef \implies \\ [\![\chi]\!](\sigma_i|_{\texttt{live}(\pi, l_i)}) = \sigma'_i|_{\texttt{live}(\pi', l_i)}$$

where $(\sigma_i, l_i) = \tau_{\pi\sigma}[i]$, $(\sigma'_i, l_i) = \tau_{\pi'\sigma}[i]$, and $\chi = \texttt{build\_comp}(\pi, l_i, \pi', l_i)$.

Proof. The correctness of `build_comp` relies on the ability of `reconstruct` to produce compensation code for each variable that is live at OSR destination, but not at origin. Algorithm `reconstruct`$(\mathtt{x}, \pi, l, \pi', l', l'')$ aims to create a sequence of instructions that assigns `x` with the value that it would have assumed at l'' in π', using as input the values of live variable at l in π.

We proceed by induction on the recursive calls of `reconstruct`. For the algorithm to succeed, there must be a unique definition `x:=e` at some point $\hat{l}$ that dominates l'', otherwise *undef* is thrown (see Figure 4.12). The base case happens when either:

1. `e` has no free variables (line 6), hence the compensation code for `x` is just `x:=e` (line 9);
2. the definition at $\hat{l}$ reaches both l'' and l' (lines 1, 4) and `x` is live at both origin and destination (line 4), hence, since π and π' are live-variable bisimilar and `x` has the same value at l and l', then no compensation code for `x` is needed as the value of `x` at l is the same that we would have had at l'';
3. $\hat{l}$ has already been visited, so compensation code for `x` has already been created.

Assume by inductive hypothesis that the recursive calls of `reconstruct` have added to χ the code to assign each free variable y of e with the value they would have assumed at $\hat{l}$ (line 7). Then the value of `x` that we would have had at $\hat{l}$ is determined by `x` := `e`, which is appended to χ (line 9). □

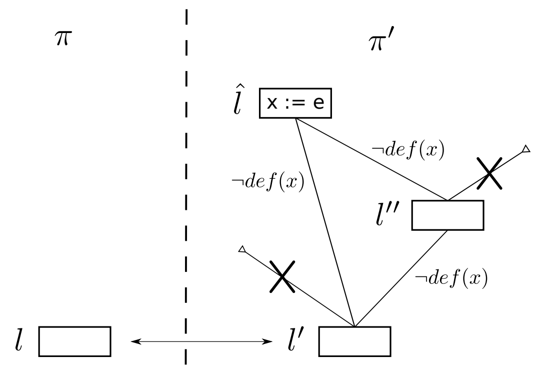


Figure 4.12. Algorithm `reconstruct` identifies an assignment `x` := `e` at $\hat{l}$ that reaches both l' and l'', and no other definition of `x` is possible.

Definition 26 (Live-Variable Equivalent Transformation) *A program transformation T is* live-variable equivalent *(LVE) if for any program π, π and $[\![T]\!](\pi)$ are live-variable bisimilar.*

We can finally establish the correctness of `OSR_trans`, which follows directly by Lemmas 3 and 4 and Corollary 1:

Theorem 2 *For any program π and live-variable equivalent transformation T, if* `apply`$(\pi, T) \triangleq (\pi', \Delta_I, \Delta_I)$ *where $\pi' = [\![T]\!](\pi)$ and $\Delta_I : [1, |\pi|] \to [1, |\pi|]$ is the identity mapping between program points, then* `OSR_trans`$(\pi, T) = (\pi', \mu_{\pi\pi'}, \mu_{\pi'\pi})$ *yields a strict OSR mapping $\mu_{\pi\pi'}$ between π and π' and a strict OSR mapping $\mu_{\pi'\pi}$ between π' and π.*

Discussion. We remark that the assumption of an identity mapping between program points, which is a necessary condition of live-variable bisimilarity, is without loss of generality as it can always be enforced by padding programs with `skip` statements. For instance, the Hoist transformation of Figure 4.11, which we prove to be LVE in **XXX**, replaces the hoisted instruction with a `skip`, and expects a `skip` to already exist at the point where it is moved. As we discuss in **XXX**, this is not required in a real compiler, as a transformation pass can be instrumented to capture code modifications that require updating the mapping.

4.2.3.4 Examples of LVE Transformations

In this section, we show that classic compiler optimizations such as constant propagation, dead code elimination, and code hoisting as defined in Figure 4.11 are all examples of live-variable equivalent transformations. Hence, they are provably correct building blocks of an OSR-aware compilation toolchain based on algorithm `OSR_trans`. These optimizations are representatives of a broad class of transformations that insert, delete, and modify instructions. Further optimizations, which we do not formally discuss in this paper, are evaluated in **XXX**.

Theorem 3 *Transformations CP, DCE, and Hoist of Figure 4.11 are live-variable equivalent.*

Proof. In [63], CP, DCE, and Hoist are proven correct, each using a different bisimulation relation R. For CP, R is simply the identity relation, hence $A(l) = Val \supseteq \texttt{live}(\pi, l) \cap \texttt{live}(\pi', l)$ in Definition 24.

For the other two transformations, R is piecewise-defined on the indices of the traces. For any initial store $\sigma \in \Sigma$, let $\tau_{\pi\sigma}[i] = (\sigma_i, l_i)$, $\tau_{\pi'\sigma}[i] = (\sigma'_i, l'_i)$, and t be the index of the final state in both traces (note that $|\tau_{\pi\sigma}| = |\tau_{\pi'\sigma}|$ from Lemma 2). Let also θ be a substitution that bounds free meta-variables with concrete program objects so that a rule's side-condition is satisfied.

For DCE, R is the identity relation before the eliminated assignment $\texttt{x} := \texttt{e}$, and $A(l) = Val \setminus \{\theta(\texttt{x})\} = \texttt{live}(\pi, l) \cap \texttt{live}(\pi', l)$ after it. R is a bisimulation such that $\forall i \in [1, t]\ l_i = l'_i$ and both the following conditions hold:

1. $[\forall j, j < i \Rightarrow l_j \neq \theta(p)] \Rightarrow \sigma_i = \sigma'_i$ and
2. $[\exists j, j \leq i \wedge l_j = \theta(p)] \Rightarrow \sigma_i \setminus \texttt{x} = \sigma'_i \setminus \texttt{x}$

where p is the meta-variable for the eliminated assignment in π', and $\sigma \setminus \texttt{x}$ is a syntactic sugar for $\sigma|_{D(\sigma)}$, where $D(\sigma) = \{v \in Var \mid v \neq \texttt{x} \ \wedge\ \sigma(v) \neq \bot\}$ is the set of all the variable identifiers other than x currently defined in σ.

For Hoist, R is the identity relation before $\theta(p)$ and after $\theta(q)$ (see Figure 4.11), and $A(l) = Val \setminus \{\theta(\texttt{x})\} = \texttt{live}(\pi, l) \cap \texttt{live}(\pi', l)$ between them. Formally, we have that $\forall i \in [1, t]\ l_i = l'_i$ and one of the following cases holds:

1. $\sigma_t = \sigma'_t \ \wedge\ \forall i\ [0 \leq i < t \ \Rightarrow\ l_i \notin \{\theta(p),\ \theta(q)\}]$
2. $\sigma_t = \sigma'_t \ \wedge\ \exists i\ [0 \leq i < t \ \wedge\ l_i = \theta(q) \ \wedge\ \sigma_i = \sigma'_i \ \wedge$
   $\forall j\ (i < j < t \ \Rightarrow\ l_j \notin \{\theta(p),\ \theta(q)\})]$

3. $\exists i\ [0 \leq i < t\ \wedge\ l_i = \theta(p)\ \wedge\ (\sigma_t \setminus \mathtt{x} = \sigma'_t \setminus \mathtt{x})\ \wedge\ (\sigma_i \setminus \mathtt{x} = \sigma'_i \setminus \mathtt{x}) \wedge$
$\forall j\ (i < j < t\ \Rightarrow\ l_j \notin \{\theta(p),\ \theta(q)\}]$

Case 1 applies before $\theta(p)$ is reached in the trace. Case 3 applies after $\theta(p)$ has been reached, but $\theta(q)$ has not. Finally, case 2 applies after $\theta(q)$ has been reached. □

4.2.3.5 Composing Multiple Transformation Passes

In this section, we show that OSR mappings can be composed, allowing several optimization passes to be applied to a program using algorithm `OSR_trans`. The first ingredient is *program composition*, defined as follows:

Definition 27 (Program composition) *We say that two programs $\pi, \pi' \in Prog$ with $\pi = \langle I_1, \ldots, I_n\rangle$ and $\pi' = \langle I'_1, \ldots, I'_{n'}\rangle$ are* composable *if $I_n =$ **out** $v_1, \ldots, v_k$ and $I'_1 =$ **in** $v'_1, \ldots, v'_{k'}$ with $\{v'_1, \ldots, v'_{k'}\} \subseteq \{v_1, \ldots, v_k\}$. For any pair of composable programs π, π', we define $\pi \circ \pi' = \langle I_1, \ldots, I_{n-1}, \hat{I'}_2, \ldots, \hat{I'}_{n'}\rangle$, where $\forall i \in [1, n']$, $\hat{I'}_i$ is obtained from I'_i by relocating each* `goto` *target m with $m + n - 2$.*

Lemma 5 (Semantics of program composition) *Let $\pi, \pi' \in Prog$ be any pair of composable programs, then $\forall \sigma \in \Sigma$, $[\![\pi \circ \pi']\!](\sigma) = [\![\pi']\!]\,([\![\pi]\!](\sigma))$.*

Proof. Straightforward by Definitions 12 and 27. □

We show how to define a composition of OSR mappings and we prove that it yields a valid OSR mapping.

Lemma 6 (Mapping Composition) *Let $\pi, \pi', \pi'' \in Prog$, let $\mu_{\pi\pi'}$ and $\mu_{\pi'\pi''}$ be OSR mappings as in Definition 22, and let $\mu_{\pi\pi'} \circ \mu_{\pi'\pi''}$ be a* composition of mappings *defined as follows:*

$$\forall l \in dom(\mu_{\pi\pi'})\ s.t.\ \mu_{\pi\pi'}(l) = (l', \chi) \wedge l' \in dom(\mu_{\pi'\pi''}) :$$
$$\mu_{\pi'\pi''}(l') = (l'', \chi') \implies (\mu_{\pi\pi'} \circ \mu_{\pi'\pi''})(l) = (l'', \chi \circ \chi')$$

Then $\mu_{\pi\pi'} \circ \mu_{\pi'\pi''}$ is an OSR mapping from π to π''.

Proof. Let $\mu_{\pi\pi''} = \mu_{\pi\pi'} \circ \mu_{\pi'\pi''}$. By Definition 22, it holds:

$$\forall \sigma \in \Sigma, \forall s_i = (\sigma_i, l_i) \in \tau_{\pi\sigma} \text{ s.t. } l_i \in dom(\mu_{\pi\pi''}),$$
$$\exists \sigma', \sigma'' \in \Sigma,\ \exists s_j = (\sigma_j, l_j) \in \tau_{\pi'\sigma'},$$
$$\exists s_k = (\sigma_k, l_k) \in \tau_{\pi''\sigma''} \text{ s.t. } \mu_{\pi\pi''}(l_i) = (l_k, \chi \circ \chi')\ \wedge$$
$$[\![\chi \circ \chi']\!](\sigma_i|_{\mathtt{live}(\pi, l_i)}) = \text{[by Lemma 5]}$$
$$[\![\chi']\!]([\![\chi]\!](\sigma_i|_{\mathtt{live}(\pi, l_i)})) = [\![\chi']\!](\sigma_j|_{\mathtt{live}(\pi', l_j)}) = \sigma_k|_{\mathtt{live}(\pi'', l_k)}$$

Hence, $\mu_{\pi\pi'} \circ \mu_{\pi'\pi''}$ is an OSR mapping from π to π''. □

Corollary 2 *Let $\pi, \pi', \pi'' \in Prog$, let $\mu_{\pi\pi'}$ and $\mu_{\pi'\pi''}$ be strict OSR mappings as in Definition 22. Then $\mu_{\pi\pi'} \circ \mu_{\pi'\pi''}$ is a strict OSR mapping from π to π''.*

Proof. Straightforward by Definition 22 and Lemma 6. □

Based on Lemma 6, we can easily prove by induction the correctness of the multi-pass transformation algorithm of Algorithm 10, which takes a program π and a list of program transformations, and applies them to π, producing a bidirectional OSR mapping $\mu_{\pi\pi''}, \mu_{\pi''\pi}$ between π and the resulting program π''.

Input: Program π, list of program transformations L.
Output: Program $\hat{\pi}$, mappings $\mu_{\pi\hat{\pi}}$ and $\mu_{\hat{\pi}\pi}$.

```
algorithm do_passes($\pi$, $T :: L$)→($\pi''$, $\mu_{\pi\pi''}$, $\mu_{\pi''\pi}$):
1  $(\pi', \mu_{\pi\pi'}, \mu_{\pi'\pi}) \leftarrow$ OSR_trans($\pi, T$)
2  if $L = Nil$ then return $(\pi', \mu_{\pi\pi'}, \mu_{\pi'\pi})$
3  $(\pi'', \mu_{\pi'\pi''}, \mu_{\pi''\pi'}) \leftarrow$ do_passes($\pi', L$)
4  return $(\pi'', \mu_{\pi\pi'} \circ \mu_{\pi'\pi''}, \mu_{\pi''\pi'} \circ \mu_{\pi'\pi})$
```

Algorithm 10: OSR-aware multi-pass program transformations.

4.2.4 Multi-Version Programs

In this section, we propose a general OSR model where computations are described by a *multi-version program*, which consists of different versions of a program along with OSR mappings that allow execution to be transferred between them.

Definition 28 (Multi-Version Program) *A multi-version program is modeled by an edge-labeled graph* $\Pi = (\mathcal{V}, \mathcal{E}, \mathcal{M})$ *where* $\mathcal{V} = \{\pi_1, \pi_2, \ldots, \pi_r\}$ *is a set of program versions,* $\mathcal{E} \subseteq \Pi^2$ *is a set of edges such that* (π_p, π_q) *indicates that an OSR transition can be fired from some point of* π_p *to* π_q*, and* $\mathcal{M} : \mathcal{E} \to OSRMap$ *labels each edge* $(\pi, \pi') \in \mathcal{E}$ *with an OSR mapping from* π *to* π'.

4.2.4.1 Semantics

The state of a multi-version program is similar to the state of a program (Definition 10), but it also includes the index of the currently executed program version:

Definition 29 (Multi-Version Program State) *The* state *of a multi-version program* $\Pi = (\mathcal{V}, \mathcal{E}, \mathcal{M})$ *is described by a triple* (p, σ, l)*, where* $p \in [1, |\mathcal{V}|]$ *is the index of a program version,* σ *is a memory store, and* $l \in [1, |\pi_p|]$ *is the point of the next instruction to be executed in* π_p*. The* initial state *from a store* σ *is* $(1, \sigma, 1)$*, i.e., computations start at* π_1*. We denote by* $MState = \mathbb{N} \times \Sigma \times \mathbb{N}$ *the set of all possible multi-version program states.*

The execution semantics of a multi-version program is described by the following transition relation:

Definition 30 (Multi-Version Big-Step Transitions) *For any multi-version program* Π*, relation* $\Rightarrow_\Pi \subseteq MState \times MState$ *is defined as follows:*

$$\begin{array}{ll} (Norm) & \dfrac{(\sigma, l) \Rightarrow_{\pi_p} (\sigma', l')}{(p, \sigma, l) \Rightarrow_\Pi (p, \sigma', l')} \\[2ex] (OSR) & \dfrac{(\pi_p, \pi_q) \in \mathcal{E} \ \wedge \ (l', \chi) = \mathcal{M}(\pi_p, \pi_q)(l) \ \wedge \ \sigma' = [\![\chi]\!](\sigma)}{(p, \sigma, l) \Rightarrow_\Pi (q, \sigma', l')} \end{array} \tag{4.10}$$

The meaning is that at any time, execution can either continue in the current program version (Norm rule), or an OSR transition – if possible at the current point – can direct the control to another program version (OSR rule). The choice is

non-deterministic, i.e., an oracle can tell the execution engine which rule to apply. In practice, the choice may be based for instance on profile data gathered by the runtime system: a common strategy is to dynamically "OSR" to the available version with the best expected performance on the actual workload. Notice that, since $\Rightarrow_\Pi$ may be non-deterministic, in general there may be different final stores for the same initial store. However, we are only interested in multi-version programs that deterministically yield a unique result, which guarantees semantic transparency of OSR transitions.
To characterize the execution behavior of a multi-version program, we consider the system of traces of an execution transition system that start from a given initial state.

Definition 31 (Trace System of Multi-Version Program) *The system of traces $\mathcal{T}_{\Pi,\sigma}$ contains all traces τ of transition system $(MState, \Rightarrow_\Pi)$ such that $\tau[0] = (1, \sigma, 1)$.*

Definition 32 (Deterministic Multi-Version Program) *A multi-version program Π is deterministic iff $\forall \sigma \in \Sigma$, either all traces in $\mathcal{T}_{\Pi,\sigma}$ are infinite, or they all lead to the same store, i.e.:*

$$\begin{gathered}\forall \tau, \tau' \in \mathcal{T}_{\Pi,\sigma} : \quad (|\tau| = \infty \iff |\tau'| = \infty) \wedge \\ (|\tau| < \infty \implies \exists\, p, p', l, l' \in \mathbb{N}, \sigma, \sigma' \in \Sigma : \\ \tau[|\tau|] = (p, \sigma, l) \ \wedge\ \tau'[|\tau'|] = (p', \sigma', l') \ \wedge\ \sigma = \sigma').\end{gathered}$$

The meaning of a deterministic multi-version program can be defined as follows:

Definition 33 (Multi-Version Semantic Function) *The semantic function $[\![\Pi]\!] : \Sigma \to \Sigma$ of a deterministic multi-version program Π is defined as:*

$$\forall \sigma \in \Sigma : \quad [\![\Pi]\!](\sigma) = \sigma' \quad \iff \quad (1, \sigma, 1) \Rightarrow_\Pi^* (p, \sigma', |\pi_p| + 1)$$

where $\Rightarrow_\Pi^*$ is the transitive closure of $\Rightarrow_\Pi$.

4.2.4.2 Generation Algorithm and Correctness

A natural way to generate a multi-version program consists in starting from a base program π_1 and constructing a tree of different versions, where each version is derived from its parent by applying one or more transformations. Using this approach and procedure `do_passes` described in Section 4.2.3.5, it is straightforward to construct a multi-version program $\Pi = (\mathcal{V}, \mathcal{E}, \mathcal{M})$ such that:

$$\begin{aligned}(\pi_p, \pi_q) \in \mathcal{E} \quad \iff \quad \exists L :\ & \texttt{do\_passes}(\pi_p, L) = (\pi_q, \mu, \mu') \ \wedge\ \mathcal{M}(\pi_p, \pi_q) = \mu \ \vee \\ & \texttt{do\_passes}(\pi_q, L) = (\pi_p, \mu, \mu') \ \wedge\ \mathcal{M}(\pi_p, \pi_q) = \mu'\end{aligned}$$

To prove the correctness of this approach, we introduce a preliminary lemma and then use it to prove that a multi-version program built in this way is deterministic.

Lemma 7 *Let $\tau \in \mathcal{T}_{\Pi,\sigma}$ be an execution trace in the system of the traces for the multi-version program $\Pi = (\mathcal{V}, \mathcal{E}, \mathcal{M})$ constructed using* `do_passes` *and LVE transformations, and let $\omega_1, \ldots, \omega_k$ be the indices of τ where an OSR transition has just occurred, with $\tau[\omega_i] = (p_{\omega_i}, \sigma_{\omega_i}, l_{\omega_i})$. Then $\forall i \in [1,k]$ there exists a state $(\hat{\sigma}_i, \hat{l}_i)$ in the trace of $\pi_{p_{\omega_i}}$ starting from the initial store σ such that $\hat{l}_i = l_{\omega_i}$ and $\hat{\sigma}_i|_{\texttt{live}(\pi_{p_{\omega_i}}, \hat{l}_i)} = \sigma_{\omega_i}|_{\texttt{live}(\pi_{p_{\omega_i}}, \hat{l}_i)}$.*

Proof. To simplify the notation we introduce:

$$\hat{\pi}_i = \begin{cases} \pi_1 & \text{if } i = 0 \\ \pi_{p_{\omega_i}} & \text{if } i \in [1,k] \end{cases}$$

From Equation (4.10) we can write that $\tau[\omega_i] = (p_{\omega_i}, \sigma_{\omega_i}, l_{\omega_i - 1})$ has been obtained from $\tau[\omega_i - 1] = (p_{\omega_i - 1}, \sigma_{\omega_i - 1}, l_{\omega_i - 1})$ with $\sigma_{\omega_i} = [\![\chi_{\omega_i - 1}]\!](\sigma_{\omega_i})$. For each OSR transition $\hat{\pi}_i$ has been obtained from $\hat{\pi}_{i-1}$ using `do_passes` for some sequence L of LVE transformations. Indeed, in order for Equation (4.10) to apply:

$$(\hat{\pi}_{i-1}, \hat{\pi}_i) \in \mathcal{E} \;\wedge\; \exists L : \texttt{do\_passes}(\hat{\pi}_{i-1}, L - 1) = (\hat{\pi}_i, \mu_{\hat{\pi}_{i-1}\hat{\pi}_i}, {\mu'}_{\hat{\pi}_i\hat{\pi}_{i-1}}) \;\wedge$$
$$M(\hat{\pi}_{i-1}, \hat{\pi}_i) = \mu_{\hat{\pi}_{i-1}\hat{\pi}_i}$$

When the OSR step is performed we thus have:

$$M(\hat{\pi}_{i-1}, \hat{\pi}_i)(l_{\omega_i - 1}) = \mu_{\hat{\pi}_{i-1}\hat{\pi}_i}(l_{\omega_i - 1}) = (l_{\omega_i}, \chi_{\omega_i - 1})$$

By Theorem 2 function $\mu_{\hat{\pi}_{i-1}\hat{\pi}_i}$ provides a strict OSR mapping between $\hat{\pi}_{i-1}$ and $\hat{\pi}_i$, as all LVE transformations in L are composed into a strict mapping (Corollary 2). Note also that since Δ_I is being used to map OSR program points between $\hat{\pi}_{i-1}$ and $\hat{\pi}_i$, it follows that $l_{\omega_i} = l_{\omega_i - 1}$ $\forall i \in [1,k]$. We now prove our claim by induction on i.

Base step. When $i = 1$, we know that no OSR transition has been performed till $l_{\omega_1 - 1}$ and thus $\hat{\pi}_0$ has been executing all the time. Then we can write:

$$(1, \sigma, 1) \Rightarrow^*_{\Pi} (1, \sigma_{\omega_1 - 1}, l_{\omega_1 - 1}) \Longleftrightarrow (\sigma, 1) \Rightarrow^*_{\hat{\pi}_0} (\sigma_{\omega_1 - 1}, l_{\omega_1 - 1})$$

Trivially, $(\sigma_{\omega_1 - 1}, l_{\omega_1 - 1}) \in \tau_{\hat{\pi}_0\sigma}$. We can thus infer from Definition 22:

$$\exists s_j = (\sigma_j, l_j) \in \tau_{\hat{\pi}_1\sigma} \text{ s.t. } \mu_{\hat{\pi}_0\hat{\pi}_1}(l_{\omega_1 - 1}) = (l_j, \chi) \;\wedge$$
$$[\![\chi]\!](\sigma_{\omega_1 - 1}|_{\texttt{live}(\hat{\pi}_0, l_{\omega_1 - 1})}) = \sigma_j|_{\texttt{live}(\hat{\pi}_1, l_j)}$$

From the definition of $\mu_{\hat{\pi}_0\hat{\pi}_1}$ it follows that $\chi = \chi_{\omega_1 - 1}$ and $l_j = l_{\omega_1} = l_{\omega_1 - 1}$. To prove the claim we need to show that:

$$\sigma_j|_{\texttt{live}(\hat{\pi}_1, l_{\omega_1})} = \sigma_{\omega_1}|_{\texttt{live}(\hat{\pi}_1, l_{\omega_1})}$$

which follows directly from Lemmas 4 and 6.

Step case. As inductive hypothesis we assume that $\exists(\hat{\sigma}_{k-1}, \hat{l}_{k-1}) \in \tau_{\hat{\pi}_{k-1}\sigma}$ s.t.:

$$\hat{l}_{k-1} = l_{\omega_{k-1}} \ \wedge \ \hat{\sigma}_{k-1}|_{\texttt{live}(\hat{\pi}_{k-1}, \hat{l}_{k-1})} = \sigma_{\omega_{k-1}}|_{\texttt{live}(\hat{\pi}_{k-1}, \hat{l}_{k-1})}$$

Since no OSR is performed between $\tau[\omega_{k-1}]$ and $\tau[\omega_k - 1]$ we can write:

$$\begin{gathered}(\hat{\sigma}_{k-1}, l_{\omega_{k-1}}) \Rightarrow^*_{\hat{\pi}_{k-1}} \cdots \Rightarrow^*_{\hat{\pi}_{k-1}} (\tilde{\sigma}, l_{\omega_k - 1}) \Longleftrightarrow \\ (\sigma_{\omega_{k-1}}, l_{\omega_{k-1}}) \Rightarrow^*_{\hat{\pi}_{k-1}} \cdots \Rightarrow^*_{\hat{\pi}_{k-1}} (\sigma_{\omega_k - 1}, l_{\omega_k - 1})\end{gathered}$$

in the same number of steps, where $\tilde{\sigma}|_{\texttt{live}(\hat{\pi}_{k-1}, l_{\omega_k - 1})} = \sigma_{\omega_k - 1}|_{\texttt{live}(\hat{\pi}_{k-1}, l_{\omega_k - 1})}$ by Lemma 1. Since $(\tilde{\sigma}, l_{\omega_k - 1}) \in \tau_{\hat{\pi}_{k-1}\sigma}$ by the strictness of the OSR mapping $\mu_{\hat{\pi}_{k-1}\hat{\pi}_k}$:

$$\begin{aligned}\exists s_j = (\sigma_j, l_j) \in \tau_{\hat{\pi}_k\sigma} \text{ s.t. } \mu_{\hat{\pi}_{k-1}\hat{\pi}_k}(l_{\omega_k - 1}) = (l_j, \chi) \ \wedge \\ [\![\chi]\!](\tilde{\sigma}|_{\texttt{live}(\hat{\pi}_{k-1}, l_{\omega_k - 1})}) = \sigma_j|_{\texttt{live}(\hat{\pi}_k, l_j)}\end{aligned}$$

From the definition of $\mu_{\hat{\pi}_{k-1}\hat{\pi}_k}$ it follows that $\chi = \chi_{\omega_k - 1}$ and $l_j = l_{\omega_k} = l_{\omega_k - 1}$. By Lemmas 4 and 6 we thus prove:

$$\begin{aligned}\sigma_j|_{\texttt{live}(\hat{\pi}_k, l_{\omega_k})} &= [\![\chi_{\omega_k - 1}]\!](\tilde{\sigma}|_{\texttt{live}(\hat{\pi}_{k-1}, l_{\omega_k - 1})}) \\ &= [\![\chi_{\omega_k - 1}]\!](\sigma_{\omega_k - 1}|_{\texttt{live}(\hat{\pi}_{k-1}, l_{\omega_k - 1})}) \\ &= \sigma_k|_{\texttt{live}(\hat{\pi}_k, l_{\omega_k})})\end{aligned}$$

□

Theorem 4 (Multi-Version Program Determinism) *Let* $\Pi = (\mathcal{V}, \mathcal{E}, \mathcal{M})$ *be a multi-version program constructed using* `do_passes` *and live-variable equivalent transformations. Then* Π *is deterministic.*

Proof. To prove that Π is deterministic, we need to show that, given any initial store σ on which $\pi_1 \in \Pi$ terminates on some final state $\sigma' = [\![\pi_1]\!](\sigma)$, any execution trace $\tau \in \mathcal{T}_{\Pi,\sigma}$ terminates with σ'.

Let $\omega_1, \ldots, \omega_k$ be the indices of τ where an OSR transition has just occurred, i.e., for any $i \in [1, k]$, state $\tau[\omega_i]$ is obtained from $\tau[\omega_i - 1]$ by applying compensation code $\chi_{\omega_i - 1}$ on store $\sigma_{\omega_i - 1}$, which yields a store σ_{ω_i}. The transition leads from a point $l_{\omega_i - 1}$ in version $\pi_{p_{\omega_i - 1}}$ to a point $l_{\omega_i} = l_{\omega_i - 1}$ in version $\pi_{p_{\omega_i}}$ in Π.

By Lemma 7, $\forall i \in [1, k]$ there exists a state $(\hat{\sigma}_i, \hat{l}_i)$ in the trace of $\hat{\pi}_i = \pi_{p_{\omega_i}}$ starting from the initial store σ such that $\hat{l}_i = l_{\omega_i}$ and $\hat{\sigma}_i|_{\texttt{live}(\hat{\pi}_i, \hat{l}_i)} = \sigma_{\omega_i}|_{\texttt{live}(\hat{\pi}_i, \hat{l}_i)}$. Hence, since no OSR is fired after ω_k, by Equation (4.10) it holds:

$$(\hat{\pi}_k, \sigma_{\omega_k}, l_{\omega_k}) \Rightarrow^*_{\Pi} (\hat{\pi}_k, \sigma', |\hat{\pi}_k| + 1) \Longleftrightarrow (\sigma_{\omega_k}, l_{\omega_k}) \Rightarrow^*_{\hat{\pi}_k} (\sigma', |\hat{\pi}_k| + 1)$$

We can then apply Lemmas 1 and 7 to write:

$$\begin{gathered}(\sigma_{\omega_k}, l_{\omega_k}) \Rightarrow^*_{\hat{\pi}_k} (\sigma', |\hat{\pi}_k| + 1) \Longleftrightarrow \\ (\sigma_{\omega_k}|_{\texttt{live}(\hat{\pi}_k, l_{\omega_k})}, l_{\omega_k}) \Rightarrow^*_{\hat{\pi}_k} (\sigma', |\hat{\pi}_k| + 1) \Longleftrightarrow \\ (\hat{\sigma}_k|_{\texttt{live}(\hat{\pi}_k, \hat{l}_k)}, \hat{l}_k) \Rightarrow^*_{\hat{\pi}_k} (\sigma', |\hat{\pi}_k| + 1)\end{gathered}$$

As $(\hat{\sigma}_k, \hat{l}_k) \in \tau_{\hat{\pi}_k\sigma}$, by Lemma 1 necessarily $\sigma' = [\![\hat{\pi}_k]\!](\sigma)$. Given that all programs in Π are semantically equivalent, we can conclude that $[\![\Pi]\!](\sigma) = \sigma' = [\![\hat{\pi}_k]\!](\sigma) = [\![\pi_1]\!](\sigma)$.

□

4.2.5 LLVM Implementation

4.2.6 Discussion

Chapter 5

Experimental Evaluation

In this chapter we illustrate experimental studies that we have performed for the techniques described in Chapters 3 and 4. Our techniques have been implemented in production systems and typically evaluated against industry-strength benchmarks. For performance metrics, reported figures have been obtained by performing multiple runs in a Linux system with negligible background activity, and we also show confidence intervals stated at 95% confidence level where possible.

In the first part of the chapter, we evaluate our space-efficient inter-procedural technique for context-sensitive profiling. In our analysis, we take into account a large collection of prominent interactive Linux applications and benchmarks from popular suites. Results collected for a number of accuracy and space usage metrics reinforce the theoretical prediction that the HCCT achieves a similar precision as the CCT in a space that is several orders of magnitude smaller, and roughly proportional to the number of hot contexts. Our implementation is casted in a full-fledged infrastructure that we developed for profiling multi-threaded Linux C/C++ applications, and ships as a plugin for the GNU Compiler Collection. We also discuss how we integrated our technique with static bursting, resulting in faster running times without substantially affecting accuracy: we incur a slowdown competitive with the `gprof` call-graph profiler, while collecting finer-grained program profiles.

In the second part, we discuss an implementation in the Jikes RVM of our intra-procedural technique for multi-iteration path profiling. We present a broad experimental study on a large suite of prominent Java benchmarks, showing that our profiler can collect profiles that would have been too costly to gather using previous multi-iteration techniques. The key to the efficiency of our approach is to replace costly hash table accesses, which are required by the Ball-Larus algorithm to maintain path counters for larger programs, with substantially faster operations on trees. We then study structural properties of path profiles that span multiple iterations for several representative benchmarks, and discuss memory footprints of the k-SF and k-IPF data structures for increasing values of k.

Finally, we present an extensive experimental evaluation of our OSR techniques for continuous program optimization. We analyze the performance of OSRKit in the TinyVM proof-of-concept virtual machine that we developed in LLVM. Our goal is to address a number of typical concerns of VM builders, measuring, e.g., the impact of having an OSR point in a hot code portion, and the actual cost of performing an

OSR transition. We also measure the time required by OSRKit to insert an OSR point and to create a stub or a continuation function. We then **XXX**

5.1 HCCT Construction and Accuracy

In this section, we present an extensive experimental study of our data streaming-based methodology for context-sensitive profiling. We implemented several variants of context-sensitive profilers and we analyzed their performance and the accuracy of the produced (ϕ, ε)-HCCT with respect to a number of metrics and using many different parameter settings. Besides the exhaustive approach, where each routine call and return is instrumented, we integrate our solution with previous techniques aimed at reducing time overhead: we focus in particular on static bursting [113], which offers convenient time-accuracy tradeoffs. The experimental analysis not only confirms, but reinforces the theoretical prediction: the (ϕ, ε)-HCCT represents the hot portions of the full CCT very well using only an extremely small percentage of the space required by the entire CCT: all the hottest calling contexts are always identified correctly, their counters are very accurate, and the number of false positives is rather small. With bursting, the running time overhead can be kept under control without affecting accuracy in a substantial way.

5.1.1 Implementation

Compiler Plugin. The `gcc` compiler provides an instrumentation infrastructure to emit calls to analysis routines at the beginning and at the end of each function, passing as arguments the address of the current function and its calling site. On top of these two primitives, we have built a full-fledged infrastructure for context sensitive profiling of multi-threaded Linux C/C++ applications that ships as a plugin[1] for the GNU Compiler Collection.

Our plugin provides native support for techniques aimed at reducing runtime overhead, such as sampling and bursting, and does not require modifications to the existing `gcc` installation or to the program to be analyzed (except for its `Makefile`). When a program is compiled, instrumentation is injected into the code by the compiler and the executable is eventually linked against a generic profiling library named `libhcct`. When a user wants to analyze the behavior of an instrumented program, it is possible to switch between different techniques – including the canonical CCT construction – or parameter settings with no need to further recompile the code.

Data Structures. We use a first-child, next-sibling representation for calling context tree nodes. Each MCCT node also contains a pointer to its parent, the routine ID, the call site, and the performance metric. The first-child, next-sibling representation is space-efficient and still guarantees that the children of each node can be explored in time proportional to their number. According to our experiments with several benchmarks, the average number of scanned children is a small constant around 2-3, so this representation turns out to be convenient also for checking whether a routine ID already appears among the children of a node. The parent field,

[1] Source code and documentation are available at: https://github.com/dcdelia/hcct

which is needed to perform tree pruning efficiently (see Algorithm 1 in Section 3.1.3), is not required in CCT nodes. As a routine ID, we simply use the routine address. Overall, CCT and MCCT nodes require 20 and 24 bytes, respectively, on 32 bit architectures. Using a simple bit packing technique [95], we also encode in one of the pointer fields a Boolean flag that tells if the calling context associated with the node is monitored in the streaming data structure M, without increasing the number of bytes per node. To improve time and space efficiency, we allocate nodes through a custom, page-based allocator, which maintains blocks of fixed size. Any additional algorithm-specific information needed to maintain the heavy hitters is stored as trailing fields within MCCT nodes.

Integration with Static Bursting. Static bursting [113] is a profiling technique that combines the advantages of sampling-based and exhausting profiling mechanisms. As in sampling-based solutions, a bursting profiler lets a program run unhindered between sampling points, and performs stack walking to determine the current calling context when a sampling point is reached. Rather than incrementing the counter for the corresponding node (which may not reflect an actual function call and thus drive to misleading results), a bursting profiler performs exhaustive instrumentation on the sequence (i.e., burst) of call/return events collected in an interval whose length we refer to as *burst length*. Further refinement of static bursting are possible, e.g., analysis overhead can be further reduced by selectively disabling bursts for previously sampled calling-contexts and then probabilistically re-enabling them [113]. In our setting, driven by the shadow stack maintained by the profiling infrastructure, we update our cursor pointer by walking down the tree from its root. Missing nodes are initialized and added to the tree during the walk. The execution stream we observe is thus partitioned into bursts and sequences that are transparent to profiling.

Other Software. As part of our infrastructure, we have developed two additional pieces of software that might be of independent interest: a library for resolving addresses to symbols, and a set of tools for the analysis and comparison of CCTs from distinct executions. In general, even for deterministic benchmarks it might not be trivial to line up nodes from two executions, as due to technical aspects such as address space randomization and dynamic loading of libraries program addresses can change. In some cases it is not always possible to resolve addresses offline up to a source-file line-number granularity, but the available information is only partial (e.g., we know only the source file where the method is defined). If this happens for two or more sibling nodes that have identical frequency counters, lining them up with tree nodes from another execution requires a similarity analysis of their spanned subtrees. We observed similar scenarios frequently in our experiments, both for hot and cold calling contexts. Since spanned subtrees for CCT nodes can be large, rapid and accurate heuristics are required to summarize the subtrees and compute their similarity; accuracy of heuristics is even more crucial when comparing a CCT with a HCCT, as spanned subtrees in the latter might have been partially or entirely pruned. Using combinatorial techniques and ad-hoc heuristics based on topological properties of the trees, we were able to quickly (i.e., in a few minutes) reconstruct for all our experiments a full and accurate mapping between pairs of different trees.

5.1.2 Experimental Setup

In this section we present the details of our experimental methodology, focusing on benchmarks and accuracy metrics, and we describe how the parameters of the streaming algorithms can be tuned.

Benchmarks

Tests were performed on a variety of large-scale Linux applications and on a set benchmarks drawn from the `Phoronix PTS` and the `SPEC CPU2006` test suites. To ensure deterministic replay of the execution of the interactive applications, we used the PIN dynamic instrumentation framework [72] to record timestamped execution traces for typical usage sessions of appoximately fifteen minutes.

Interactive applications include graphics programs (`inkscape` and `gimp`), a hexadecimal file viewer (`ghex2`), audio players/editors (`amarok` and `audacity`), an archiver (`ark`), an Internet browser (`firefox`), an HTML editor (`quanta`), a chat program (`pidgin`), the OpenOffice suite for word processing (`oowriter`), spreadsheets (`oocalc`), and drawing (`ooimpress`).

Non-interactive benchmarks include a cryptographic library (`botan`), a 2D graphics library (`cairo-perf-trace`), advanced chess engines (`crafty` and `sjeng`), the Connect Four (`fhourstones`) and Go (`gobmk`) games, and two 3D games run in demo mode (PlanetPenguin Racer in the `ice-labyrinth` and `mount-herring` scenarios, and SuperTuxKart on the `overworld` and `scotland` tracks).

Statistical information about test sets are reported in Table 3.1 (page 7): even short sessions result in CCTs consisting of tens of millions of calling contexts, whereas the call graph has only a few thousand nodes. We also observe that the number of distinct call sites is roughly one order of magnitude larger than the call graph.

Metrics

Besides memory usage and time consumption of our profiler, we test the accuracy of the (ϕ, ε)-HCCT according to a variety of metrics.

1. Degree of overlap [6, 7, 113] measures the completeness of the (ϕ, ε)-HCCT with respect to the full CCT:

$$overlap((\phi,\varepsilon)\text{-HCCT}, \text{CCT}) = \frac{1}{N} \sum_{\text{arcs } e \in (\phi,\varepsilon)\text{-HCCT}} w(e)$$

   where N is the total number of routine activations (corresponding to the CCT total weight) and $w(e)$ is the true frequency of the target node of arc e in the CCT.

2. Hot edge coverage [113] measures the percentage of CCT hot edges covered by the (ϕ, ε)-HCCT, using an edge-weight threshold $\tau \in [0, 1]$ to determine hotness. Since (ϕ, ε)-HCCT$\subseteq$CCT, hot edge coverage can be defined as follows:

$$cover((\phi,\varepsilon)\text{-HCCT}, \text{CCT}, \tau) = \frac{|\{e \in (\phi,\varepsilon)\text{-HCCT}\colon w(e) \geq \tau \cdot w_{max}\}|}{|\{e \in \text{CCT}\colon w(e) \geq \tau \cdot w_{max}\}|}$$

   where w_{max} is the weight of the hottest CCT arc.

3. Maximum hotness of uncovered calling contexts, where a context is uncovered if is not included in the (ϕ, ε)-HCCT:

$$maxUncov((\phi,\varepsilon)\text{-HCCT}, \text{CCT}) = \max_{e \in \text{CCT} \setminus (\phi,\varepsilon)\text{-HCCT}} \frac{w(e)}{w_{max}} \times 100$$

   Average hotness of uncovered contexts is defined similarly.

4. Number of false positives, i.e., $|A \setminus H|$: the smaller this number, the better the (ϕ, ε)-HCCT approximates the exact HCCT obtained from CCT pruning.

5. Maximum counter error, i.e., maximum error in the frequency counters of (ϕ, ε)-HCCT nodes with respect to their true value in the full CCT:

$$maxError((\phi,\varepsilon)\text{-HCCT}) = \max_{e \in (\phi,\varepsilon)\text{-HCCT}} \frac{|w(e) - \widetilde{w}(e)|}{w(e)} \times 100$$

   where $w(e)$ and $\widetilde{w}(e)$ are the true and the estimated frequency of context e, respectively. Average counter error is defined similarly.

We remark that an accurate solution should maximize (1) and (2), and minimize the remaining metrics.

Parameter Tuning

Before describing our experimental findings, we discuss how to choose parameters ϕ and ε to be provided as input to the streaming algorithms. According to the theoretical analysis, an accurate choice of ϕ and ε might greatly affect the space used by the algorithms and the accuracy of the solution. In our study we considered many different choices of ϕ and ε across rather heterogeneous sets of benchmarks and execution traces, always obtaining similar results that we summarize below.

A rule of thumb about ϕ and ε validated by previous experimental studies [33] suggests that it is sufficient to choose $\varepsilon = \phi/10$ in order to obtain high counter accuracy and a small number of false positives.

We found this choice overly pessimistic in our scenario: the extremely skewed cumulative distribution of calling context frequencies shown in Figure 3.1 (page 8) makes it possible to use much larger values of ε without sacrificing accuracy. This yields substantial benefits on the space usage, which is roughly proportional to $1/\varepsilon$. Unless otherwise stated, in all our experiments we used $\varepsilon = \phi/5$.

Let us now consider the choice of ϕ: ϕ is the hotness threshold with respect to the stream length N, i.e., to the number of routine enter events. However, N is unknown *a priori* during profiling, and thus choosing ϕ appropriately may appear to be difficult: too-large values might result in returning very few hot calling contexts (even no context at all in some extreme cases), while too-small values might result in using too much space and returning too many contexts without being able to discriminate accurately which of them are actually hot. Our experiments suggest that an appropriate choice of ϕ is mostly independent of the specific benchmark and of the stream length: as shown in Table 5.1, different benchmarks have HCCT sizes of the same order of magnitude when using the same ϕ threshold (results for omitted benchmarks are similar). This is a consequence of the skewness of context frequency distribution, and greatly simplifies the choice of ϕ in practice. Unless otherwise stated, in our experiments we used $\phi = 10^{-4}$, which corresponds to mining roughly the hottest 1,000 calling contexts independently of the benchmark.

Benchmark	HCCT nodes $\phi = 10^{-3}$	HCCT nodes $\phi = 10^{-5}$	HCCT nodes $\phi = 10^{-7}$	CCT nodes
audacity	112	9 181	233 362	13 131 115
dolphin	97	14 563	978 544	11 667 974
gimp	96	15 330	963 708	26 107 261
ice-labyrinth	93	9 413	529 945	2 160 052
inkscape	80	16 713	830 191	13 896 175
oocalc	136	13 414	1 339 752	48 310 585
quanta	94	13 881	812 098	27 426 654

Table 5.1. Typical thresholds for calling context frequencies.

Platform

Experiments were performed on a 2.53GHz Intel Core2 Duo T9400 with 128KB of L1 data cache, 6MB of L2 cache, and 4 GB of main memory DDR3 1066, running Ubuntu 12.04, Linux Kernel 3.5.0, gcc 4.7.2, 32 bit. Performance measurements were collected with negligible background activity, running multiple trials for each benchmark/tool combination and reporting confidence intervals stated at 95% confidence level.

5.1.3 Memory Usage

We first evaluate how much space can be saved by our approach, reporting the size of the MCCT constructed by the SS algorithm compared to the size of the full CCT as a function of the hotness threshold ϕ. Figure 5.1 shows the results for a subset of our benchmarks. Notice that the size of the MCCT, hence the space used by the algorithm, decreases with ϕ. For values of $\phi \geq 10^{-4}$, i.e., contexts that appear at least 0.01% of the times, space usage remains less than 1% than the CCT size for most benchmarks, with a worst case of about 4.1% over all our experiments.

As a second experiment, we study the actual memory footprint of our profilers considering both SS and the combination of SS with static bursting. Figure 5.2 plots the peak memory usage of our profilers as a percentage of the full CCT. We recall that during the computation we store the minimal subtree MCCT of the CCT spanning all monitored contexts. This subtree is eventually pruned to obtain the (ϕ, ε)-HCCT (Sections 3.1.2 and 3.1.3). The peak memory usage is proportional to the number of MCCT nodes, which is typically much larger than the actual number of hot contexts obtained after pruning.

Quite surprisingly, static bursting also improves space usage. This depends on the fact that sampling reduces the variance of calling context frequencies: MCCT cold nodes that have a hot descendant are more likely to become hot when sampling

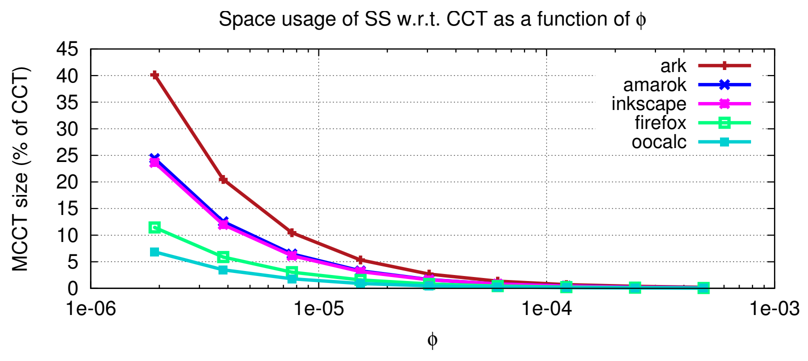


Figure 5.1. Space usage as a function of the hotness threshold ϕ..

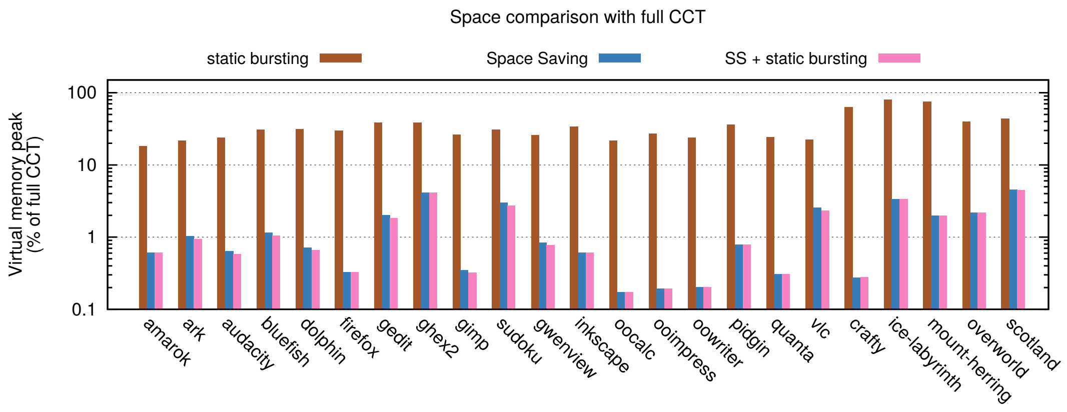


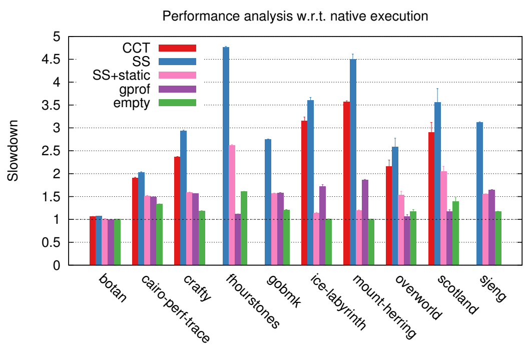
Figure 5.2. Space analysis of static bursting (left), SS (center), and SS combined with static bursting (right) with sampling interval 2 msec and burst length 0.2 msec.

is active, and monitoring these nodes reduces the total MCCT size. The histogram also shows that static bursting alone (i.e., without streaming) is not sufficient to substantially reduce space: in addition to hot contexts, a large fraction of cold contexts is also sampled and included in the CCT. We also observed that the larger the applications, the larger the space reduction of our approach over bursting alone.

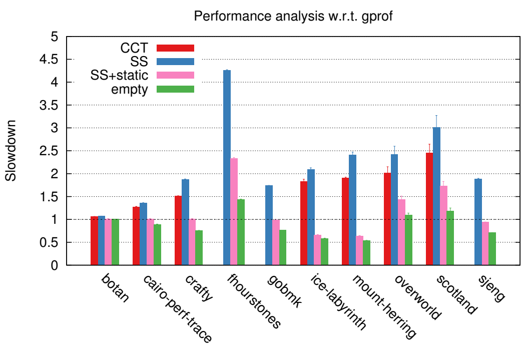
Since the average node degree is a small constant, cold HCCT nodes are typically a fraction of the total number of nodes, as shown in Figure 5.7 for $\phi = 10^{-4}$ (page 72). In our experiments we observed that this fraction strongly depends on the hotness threshold ϕ, and in particular decreases with ϕ: cold nodes that have a hot descendant are indeed more likely to become hot when ϕ is smaller.

5.1.4 Time Overhead

We now discuss the time overhead of our approach, both alone and in combination with static bursting. We compare to native execution, to the widely-used call-graph profiler `gprof` [50], and to the canonical CCT construction. To assess the instrumentation overhead, we also compare to *empty* instrumentation (i.e., when no analysis is performed).



(a)



(b)

Figure 5.3. Runtime analysis for CCT and (ϕ, ε)-HCCT construction compared to (a) native executions and (b) executions under `gprof`. The *empty* bars measure the cost of instrumenting function calls and returns. SS with stating bursting has been executed with sampling interval 20 msec and burst length 2 msec.

Figure 5.3(a) shows the overheads of the different profilers normalized against the performance of a native execution. The average slowdown for the CCT construction is 2.45×, with a peak of 3.56× for `mount-herring`. Note that data for benchmarks `fhourstones`, `gobmk` and `sjeng` are not reported for the CCT profiler as it ran out of memory. We observe that the construction of the (ϕ, ε)-HCCT incurs an average slowdown of 2.9× (3.09× considering also OOM benchmarks) and is 16.28% slower than the CCT profiler, with a peak of 26.08% for `mount-herring`. Given the previously discussed memory usage reduction, this represents an interesting space-time trade-off.

The integration with static bursting reduces the average overhead of our approach to 1.58× for the whole set of benchmarks, which is not far from the 1.21× slowdown introduced by the `gcc` instrumentation itself. We observe a peak of 2.62× on the benchmark `fhourstones`: we believe this is due to the particular structure of its source code, which contains very frequently invoked tiny functions that could be replaced with macros.

In Figure 5.3(b) we have normalized the runtime overheads against an execution under `gprof`. The combination of our approach with static bursting is very effective, as it is on average 18% (5.16% if we exclude `fhourstones`) slower than `gprof`. On 5 out of 10 benchmarks, the two tools achieve nearly-identical slowdowns. We observe 2.34×, 1.44× and 1.74× slowdowns on `fhourstones`, `overworld`, and `scotland`, respectively. Notice that for all these benchmarks the cost of the instrumentation inserted by `gcc` is already greater than the slowdown introduced by `gprof`. On the other hand, we observe appreciable speedups on `ice-labyrinth` and `mount-herring`, for which SS combined with static bursting is 1.51× and 1.55× faster than `gprof`, respectively.

5.1.5 Accuracy

Exact HCCT. We first discuss the accuracy of the exact HCCT with respect to the full CCT. Since the HCCT is a subtree of the (ϕ, ε)-HCCT computed by our algorithms, the results described in this section apply to the (ϕ, ε)-HCCT as well: the values of degree of overlap and hot edge coverage on the HCCT are a lower bound to the corresponding values in the (ϕ, ε)-HCCT, while the frequency of uncovered contexts is an upper bound.

It is not difficult to see that the cumulative distribution of calling context frequencies shown in Figure 3.1 (page 8) corresponds exactly to the degree of overlap with the full CCT. This distribution roughly satisfies the 10% - 90% rule: hence, with only 10% of hot contexts, we have a degree of overlap around 90% on all benchmarks.

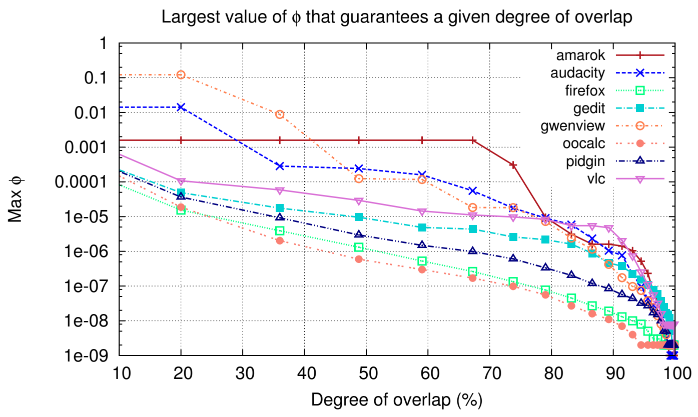


Figure 5.4. Relation between ϕ and degree of overlap between the exact HCCT and the full CCT on a representative subset of benchmarks.

Figure 5.4 illustrates the relation between degree of overlap and hotness threshold, plotting the value $\widetilde{\phi}$ of the largest hotness threshold for which a given degree of overlap d can be achieved: using any $\phi \leq \widetilde{\phi}$, the achieved degree of overlap will be larger than or equal to d. The value of $\widetilde{\phi}$ decreases as d increases: if we want to achieve a larger degree of overlap, we must include in the HCCT a larger number of nodes, which corresponds to choosing a smaller hotness threshold. However, when computing the (ϕ, ε)-HCCT, the value of ϕ indirectly affects the space used by the algorithm and in practice cannot be too small (see Section 5.1.2). By analyzing hot edge coverage and uncovered frequency, we show that even when the degree of overlap is not particularly large, the HCCT and the (ϕ, ε)-HCCT are nevertheless good approximations of the full CCT.

As shown in Figures 5.1 and 5.5, for values of ϕ as small as 10^{-4} the space usage is less than 1% of the full CCT, while guaranteeing 100% coverage for all edges with hotness at least 10% on most benchmarks. Smaller values of ϕ increase space and improve the degree of overlap, but are unlikely to be interesting in applications that require mining hot calling contexts.

Notice that $\phi = 10^{-4}$ yields a degree of overlap as small as 10% on two of the less skewed benchmarks (`oocalc` and `firefox`), which seems to be a bad scenario. However, Figure 5.6 analyzes how the remaining 90% of the total CCT weight is distributed among uncovered contexts: the average frequency of uncovered contexts is about 0.01% of the frequency of the hottest context, and the maximum frequency is typically less than 10%. This suggests that uncovered contexts are likely to be uninteresting with respect to the hottest contexts, and that the distribution of calling context frequencies obeys a “long-tail, heavy-tail” phenomenon: the CCT contains a huge number of calling contexts that rarely get executed, but overall these low-frequency contexts account for a significant fraction of the total CCT weight.

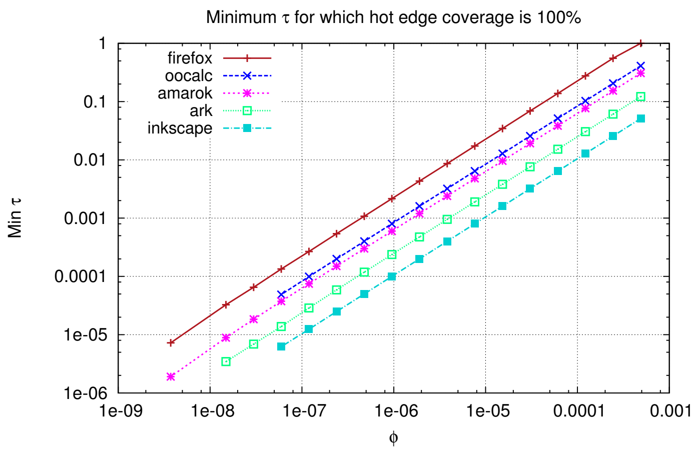


Figure 5.5. Hot edge coverage of the exact HCCT: relation between ϕ and edge–weight threshold τ on a representative subset of benchmarks.

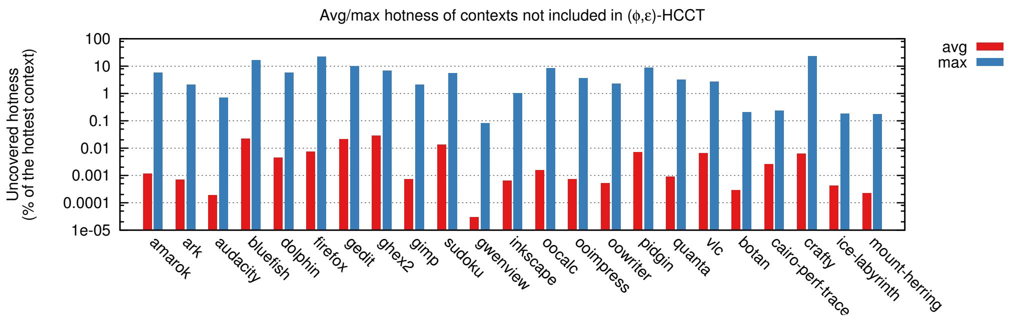


Figure 5.6. Maximum and average hotness of calling contexts not included in the (ϕ, ε)-HCCT.

Figure 5.5 confirms this intuition, showing that the HCCT represents the hot portions of the full CCT remarkably well even for values of ϕ for which the degree of overlap may be small. The figure plots, as a function of ϕ, the smallest value $\tilde{\tau}$ of the hotness threshold τ for which hot edge coverage of the HCCT is 100%. Results are shown only on some of the less skewed, and thus more difficult, benchmarks. Note that $\tilde{\tau}$ is directly proportional to and roughly one order of magnitude larger than ϕ. This is because the HCCT contains all contexts with frequency $\geq \lfloor \phi N \rfloor$, and always contains the hottest context, which has weight w_{max} as in the definition of hot edge coverage in Section 5.1.2. Hence, the hot edge coverage is 100% as long as $\lfloor \phi N \rfloor \geq \tau \cdot w_{max}$, which yields $\tilde{\tau} = \lfloor \phi N \rfloor / w_{max}$. The experiment shows that 100% hot edge coverage is always obtained for $\tau \geq 0.01$. As a frame of comparison, notice that the τ thresholds used in [113] to analyze hot edge coverage are always larger than 0.05, and for those values we always guarantee total coverage.

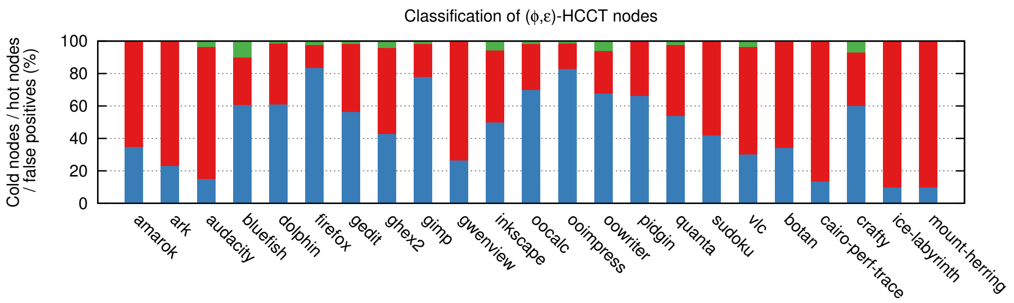


Figure 5.7. Partition of (ϕ, ε)-HCCT nodes into: cold (bottom bar), hot (middle bar), and false positives (top bar).

(ϕ,ε)-HCCT. We now discuss the accuracy of the (ϕ, ε)-HCCT compared to the exact HCCT. Figure 5.7 shows the percentages of cold nodes, true hots, and false positives in the (ϕ, ε)-HCCT using $\phi = 10^{-4}$ and $\varepsilon = \phi/5$. We observe that SS includes in the tree only very few false positives: less than 10% of the total number of tree nodes in the worst case, and between 0% and 5% for the large majority of the benchmarks. The percentage of cold nodes strictly depends on the characteristics of the single benchmark, and is not remarkably influenced by the number of false positives, which is small.

An interesting feature of our approach is that counter estimates are very close to the true frequencies, as shown in Figure 5.8. A comparison between average and maximum errors suggests that just a few nodes are appreciably overestimated. The average counter error computed for hot contexts is actually greater than 4% only for `crafty`, and smaller than 2% for the majority of the benchmarks.

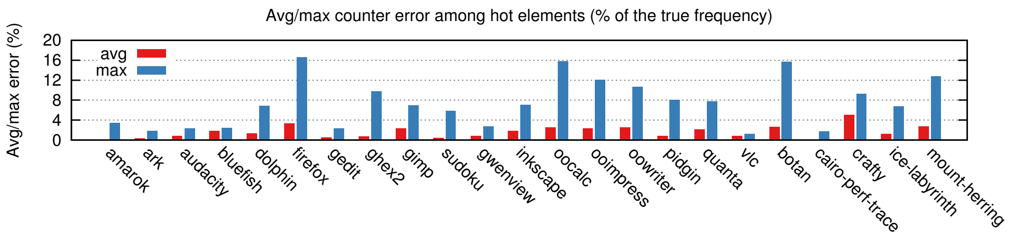
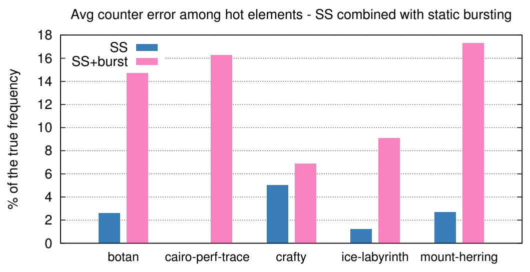


Figure 5.8. Accuracy of calling-context frequencies measured on hot contexts included in the (ϕ, ε)-HCCT.

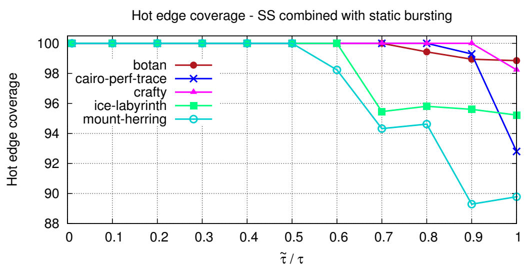
It is worth noticing that, when integrating SS with sampling-based approaches, the theoretical guarantees of the algorithm apply only to the stream of sampled events, and not to the full stream of routine calls and returns from the execution. For this reason, we analyze the impact of static bursting on the quality of the solution.

Figure 5.9(a) shows the average counter error among hot calling contexts. In order to compare them with the exact frequencies from the corresponding CCTs, we adjusted the counters by dividing them by the fraction of sampled events in the stream of function calls and returns. Note that, if the stream is uniformly distributed in time, this fraction is equal to the ratio between burst length and sampling interval. While processing roughly only one tenth of the whole stream, we observe that the average counter error ranges from 6.89% to 17.31%. An analysis of the results from the integration of static bursting with the canonical CCT construction shows very similar numbers, thus suggesting that SS does not degrade the quality of the solution.

The adoption of sampling-based techniques may cause the algorithm to miss some of the contexts with frequency very close to $\lfloor \phi N \rfloor$, leading to some false negatives. However, our analysis of hot edge coverage reported in Figure 5.9(b) shows that static bursting does not appreciably degrade the quality of the solution. Given the smallest $\widetilde{\tau}$ value for which SS guarantees 100% coverage, in all of our experiments we achieve 100% coverge for any $\tau \geq 2\widetilde{\tau}$ (i.e., when $\widetilde{\tau}/\tau \leq 0.5$), and only in one case (`mount-herring` benchmark) hot edge coverage drops below 90%.



(a)



(b)

Figure 5.9. (a) Accuracy of frequencies measured on hot calling contexts when SS is combined with static bursting. (b) Hot edge coverage for decreasing values of the hotness threshold τ. As pointed out in Section 5.1.5, $\widetilde{\tau}$ is the minimum threshold for which SS guarantees 100% coverage when static bursting is disabled. We chose a representative subset of benchmarks for both charts; sampling interval has been set to 20 msec and burst length to 2 msec.

5.2 Multi-iteration Path Profiling

In this section, we discuss and evaluate an implementation, which we call k-BLPP, of our approach to multi-iteration path profiling in the Jikes Research Virtual Machine (RVM) [3]. Our code is publicly available in the Jikes RVM Research Archive[2] and has been endorsed by the OOPSLA 2013 Artifact Evaluation Committee. The goal of our experimental study is to assess the performance of our profiler compared to previous approaches and to study properties of path profiles that span multiple iterations for several representative benchmarks. The results indicate that our technique can profile paths that extend across many loop iterations in a time comparable with acyclic path profiling on a large variety of industry-strength benchmarks.

5.2.1 Implementation

Adaptive Compilation. The Jikes RVM is a high-performance *meta-circular* virtual machine: unlike most other JVMs, it is written in Java. The Jikes RVM does

[2] `http://sourceforge.net/p/jikesrvm/research-archive/41/`.

not include an interpreter: all bytecode must be first translated into native machine code. The unit of compilation is the method, and methods are compiled lazily by a fast non-optimizing compiler – the so-called *baseline* compiler – when they are first invoked by the program. As execution continues, the Adaptive Optimization System monitors program execution to detect program hot spots and selectively recompiles them with three increasing levels of optimization. This approach is typical of modern production JVMs, which rely on some variant of selective optimizing compilation to target the subset of the hottest program methods where they are expected to yield the most benefits.

Recompilation is performed by the *optimizing* compiler, that generates higher-quality code but at a significantly larger cost than the baseline compiler. Since Jikes RVM quickly recompiles frequently executed methods, we implemented k-BLPPin the optimizing compiler only.

Adding Instrumentation. As discussed in Section 3.2.2, the Ball-Larus tracing technique requires instrumenting CFG edges so that when an edge is traversed, the probe value is incremented by a quantity computed by the path numbering algorithm on the DAG obtained by transforming back edges in the CFG.

k-BLPP adds instrumentation to hot methods in three passes:

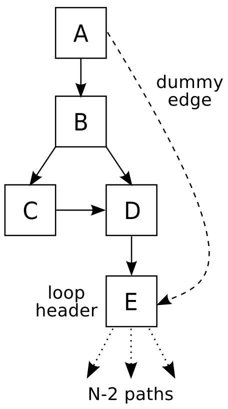
1. building the DAG representation;
2. assigning values to edges;
3. adding instrumentation to edges.

k-BLPP adopts the *smart path numbering* algorithm proposed by Bond and McKinley [21] to improve performance by placing instrumentation on cold edges. In particular, line 6 of the canonical Ball-Larus path numbering algorithm shown in Algorithm 3 (page 20) is modified such that outgoing edges are picked in decreasing order of execution frequency. For each basic block edges are sorted using existing edge profiling information collected by the baseline compiler: we can thus assign zero to the hottest hedge, so that k-BLPP will not place any instrumentation on it.

During compilation, the Jikes RVM introduces *yield points*, which are program points where the running thread determines if it should yield to another thread. Since JVMs need to gain control of threads quickly, compilers insert yield points in method prologues, loop headers, and method epilogues. We modified the optimizing compiler to also store the path profiling probe on loop headers and method epilogues. Ending paths at loop headers rather than back edges causes a path that traverse a header to be split into two paths: this difference from canonical Ball-Larus path profiling is minor because it only affects the first path through a loop [20].

Note that optimizing compilers do not insert yield points in a method when either it does not contain branches (hence its profile is trivial) or it is marked as uninterruptible. The second case occurs in internal Jikes RVM methods only; the compiler occasionally inlines such a method into an application method, and this might result in a loss of information only when the execution reaches a loop header contained in the inlined method. However, according to [20], this loss of information appears to be negligible.

Path Profiling. To make fair performance comparisons with state-of-the-art previous profilers, we built our code on top of the BLPP profiler developed by Bond [20, 59], which provides an efficient implementation of the Ball-Larus acyclic-path profiling technique. The k-SF construction algorithm described in Section 3.2.3 is implemented using a standard first-child, next-sibling representation for nodes. This representation is very space-efficient, as it requires only two pointers per node: one to its leftmost child and the other to its right nearest sibling.



BL path	ID
ABDE	0
ABCDE	1
AExxx	2
AEyyy	3
...	...
AEzzz	N-1

Figure 5.10. Routine with an initial branch before the first cycle.

Tree roots are stored and accessed through an efficient implementation[3] of a hash map, using the pair represented by the Ball-Larus path ID and the unique identifier associated to the current routine (i.e., the compiled method ID) as key. Note that this map is typically smaller than a map required by a traditional BLPP profiler, since tree roots represent only a fraction of the distinct path IDs encountered during the execution. Consider, for instance, the example shown in Figure 5.10: this control flow graph has N acylic paths after backedges have been removed. Since cyclic paths are truncated on loop headers, only path IDs 0 and 1 can appear after the special marker $*$ in the stream, thus leading to the creation of an entry in the hash map. Additional entries might be created when a new tree is added to the k-SF (line 10 of the streaming algorithm shown in Algorithm 4 on page 25); however, experimental results show that the number of tree roots is usually small, while N increases with the complexity (i.e., number of branches and loops) of the routine.

5.2.2 Experimental Setup

In this section we illustrate the details of our experimental methodology, focusing on benchmarks, performance and topological metrics, and compared profiling techniques.

Benchmarks

We evaluated k-BLPP against a variety of prominent benchmarks drawn from three suites. The `DaCapo` suite [14] consists of a set of open source, real-world applications

[3]`HashMapRVM` is a stripped-down implementation of the `HashMap` data structure used by core parts of the Jikes RVM runtime and by Bond's BLPP path profiler.

with non-trivial memory loads. We use the superset of all benchmarks from `DaCapo` releases 2006-10-MR2 and 9.12 that can run successfully in the Jikes RVM, using the largest available workload for each benchmark. In particular, `avrora`, `jython`, `luindex`, `sunflow`, and `xalan` are taken from the 9.12 release, while `chart`, `eclipse`, and `hsqldb` are from the 2006-10-MR2 release.

The `SPEC JVM2008` suite focuses on the performance of the hardware processor and memory subsystem when executing common general purpose application computations. Benchmarks from the suite that can run successfully[4] on the Jikes RVM include: `compiler.compiler`, `compress`, `mpegaudio`, and `scimark.{montecarlo, sor.large, sparse.large}`.

Finally, we chose two memory-intensive benchmarks (`heapsort` and `md`) from the `Java Grande` 2.0 suite [24] to further evaluate the performance of k-BLPP.

Metrics

We considered a variety of metrics, including wall-clock time, number of operations per second performed by the profiled program, number of hash table operations, data structure size (e.g., number of hash table items for BLPP and number of k-SF nodes for k-BLPP), and statistics such as average node degree of the k-SF and the k-IPF and average depth of k-IPF leaves. To interpret our results, we also “profiled our profiler” by collecting hardware performance counters with `perf` [84], including L1 and L2 cache miss rate, branch mispredictions, and cycles per instruction (CPI).

Compared Codes

In our experiments, we analyzed the native (uninstrumented) version of each benchmark and its instrumented counterparts, comparing k-BLPP for different values of k (2, 3, 4, 6, 8, 11, 16) with the BLPP profiler developed by Bond [59] for Ball-Larus acyclic-path profiling. We upgraded the original tool by Bond to take advantage of native threading support introduced in later Jikes RVM releases; the code is structured as in Figure 3.4 (page 19), except that it does not produce any intermediate stream, but it directly performs `count[r]++`.

Platform

Experiments were performed on a 2.53GHz Intel Core2 Duo T9400 with 128KB of L1 data cache, 6MB of L2 cache, and 4 GB of main memory DDR3 1066, running Ubuntu 12.10, Linux Kernel 3.5.0, 32 bit. We ran all of the benchmarks on Jikes RVM 3.1.3 (default `production` build) using a single core and a maximum heap size equal to half of the amount of physical memory. For each benchmark/profiler combination, we performed 10 trials, each preceded by a warmup execution, and computed the arithmetic mean. Performance measurements were collected on a machine with negligible background activity. We report confidence intervals stated at 95% confidence level.

[4]Due to limitations of the GNU classpath, only a small number of them are supported.

5.2.3 Time Overhead

In Figure 5.11 we report for each benchmark the profiling overhead of k-BLPP relative to BLPP. The chart shows that for 12 out of 16 benchmarks the overhead decreases for increasing values of k, providing up to almost 45% improvements over BLPP. This is explained by the fact that hash table accesses are performed by `process_bl_path_id` every $k-1$ items read from the input stream between two consecutive routine entry events (lines 8 and 10 in Algorithm 4on page 25). As a consequence, the number of hash table operations for each routine call is $O(1+N/(k-1))$, where N is the total length of the path taken during the invocation.

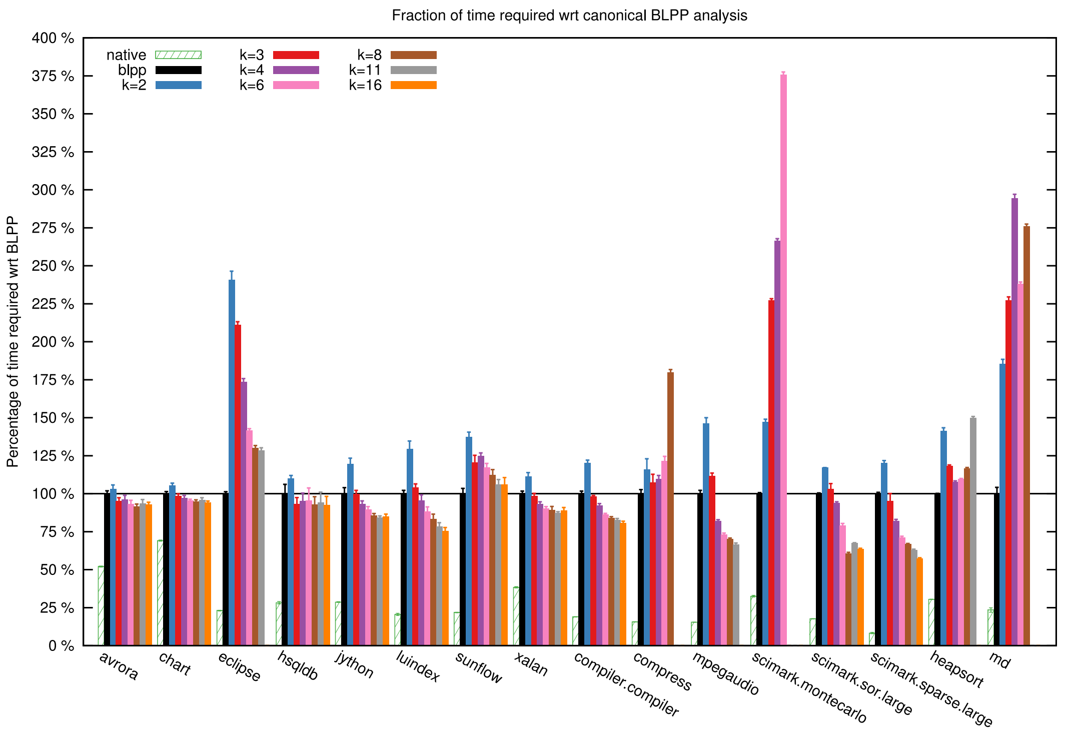


Figure 5.11. Performance of k-BLPP relative to BLPP.

In Figure 5.12 we report the measured number of hash table accesses for our experiments, which decreases as predicted on all benchmarks with intense loop iteration activity. Notice that, not only does k-BLPP perform fewer hash table operations, but since only a subset of BL path IDs are inserted, the table is also smaller yielding further performance improvements. For codes such as `avrora` and `hsqldb`, which perform on average a small number of iterations, increasing k beyond this number does not yield any benefit.

On `eclipse`, k-BLPP gets faster as k increases, but differently from all other benchmarks in this class, it remains slower than BLPP by at least 25%. The reason is that, due to structural properties of the benchmark, the average number of node scans at lines 13 and 21 of `process_bl_path_id` is rather high (58.8 for $k = 2$ down to 10.3 for $k = 16$). In contrast, the average degree of internal nodes of the k-SF is small (2.6 for $k = 2$ decreasing to 1.3 for $k = 16$), hence there is intense activity on

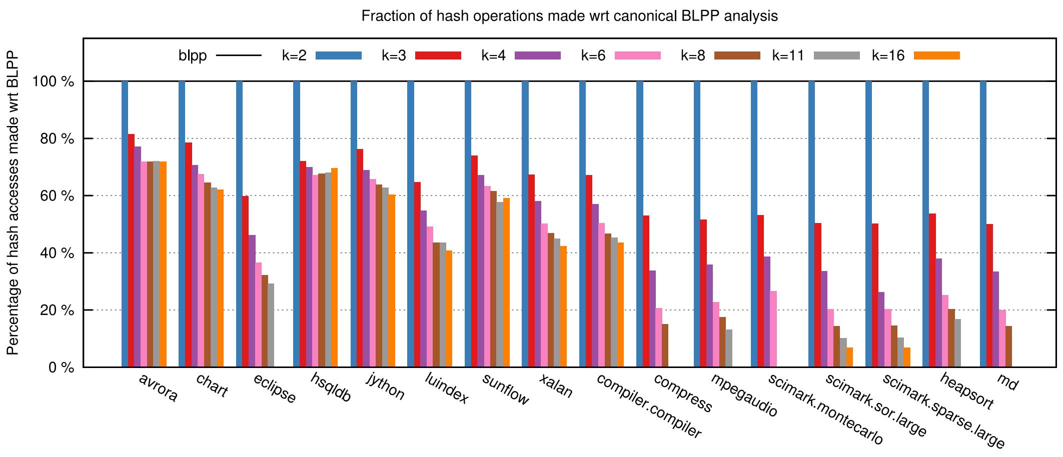


Figure 5.12. Number of hash table operations performed by k-BLPP relative to BLPP.

nodes with a high number of siblings. No other benchmark exhibited this extreme behavior. We expect that a more efficient implementation of `process_bl_path_id`, e.g., by adaptively moving hot children to the front of the list, could reduce the scanning overhead for this kind of worst-case benchmarks as well.

Benchmarks `compress`, `scimark.montecarlo`, `heapsort`, and `md` made an exception to the general trend we observed, with performance overhead increasing, rather than decreasing, with k. To justify this behavior, we collected and analyzed several hardware performance counters and noticed that on these benchmarks our k-BLPP implementation suffers from increased CPI for higher values of k.

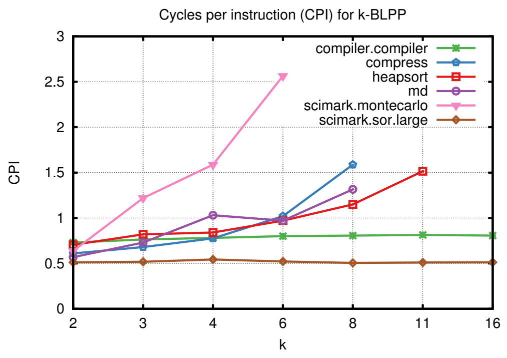


Figure 5.13. Hardware performance counters for k-BLPP: cycles per instruction (CPI).

Figure 5.13 shows this phenomenon, comparing the four outliers with other benchmarks in our suite. By analyzing L1 and L2 cache miss rates, reported in Figure 5.14 (a) and Figure 5.14 (b), we noticed that performance degrades due to poor memory access locality. We believe this to be an issue of our current implementation of k-BLPP, in which we did not make any effort aimed at improving cache efficiency in accessing the k-SF, rather than a limitation of the general approach we propose.

Indeed, as nodes may be unpredictably scattered in memory due to the linked structure of the forest, pathological situations may arise where node scanning incurs several cache misses.

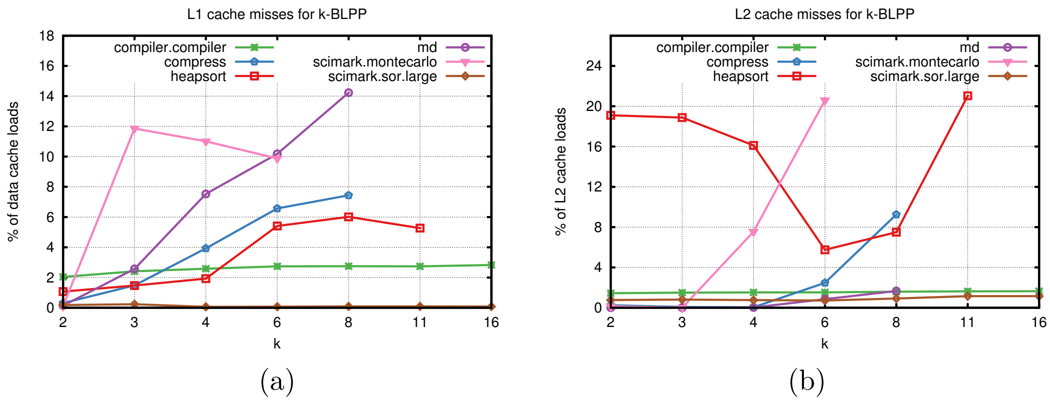


Figure 5.14. Hardware performance counters for k-BLPP: (a) L1 and (b) L2 cache miss rates.

Notice that, since we never delete either entries from the hash table or nodes from the k-SF, our implementation does not place any additional burden on the garbage collector. The profiler causes memory release operations only when a thread terminates, dumping all of its data structures at once.

5.2.4 Memory Usage and Structural Properties

Figure 5.15 compares the space requirements of BLPP and k-BLPP for different values of k. The chart reports the total number of items stored in the hash table by BLPP and the number of nodes in the k-SF.

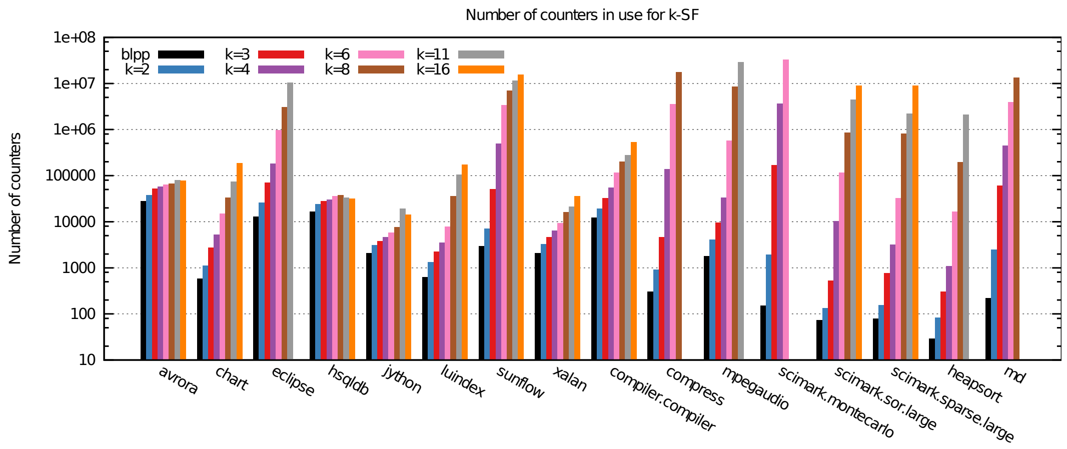


Figure 5.15. Space requirements: number of hash table entries in BLPP and number of nodes in the k-SF.

Since both BLPP and k-BLPP exhaustively encode exact counters for all distinct taken paths of bounded length, space depends on intrinsic structural properties of the benchmark. Programs with intense loop iteration activity are characterized by substantially higher space requirements by k-BLPP, which collects profiles containing up to several millions of paths. Notice that on some benchmarks we ran out of memory for large values of k, hence some bars in the charts we report in this section are missing. In Figure 5.16 we report the number of nodes in the k-IPF, which corresponds to the number of paths profiled by k-BLPP. Notice that, since a path may be represented more than once in the k-SF, the k-IPF represents a more compact version of the k-SF.

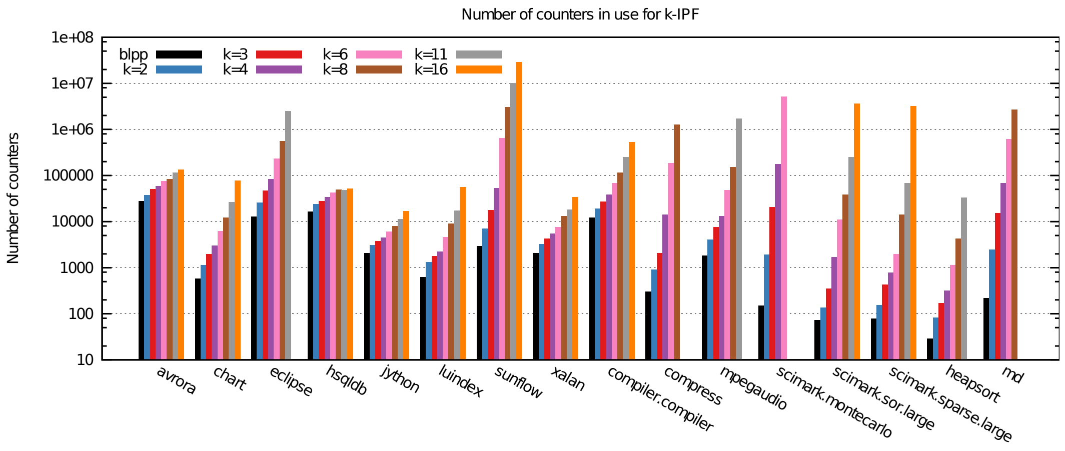


Figure 5.16. Number of paths profiled by BLPP and k-BLPP.

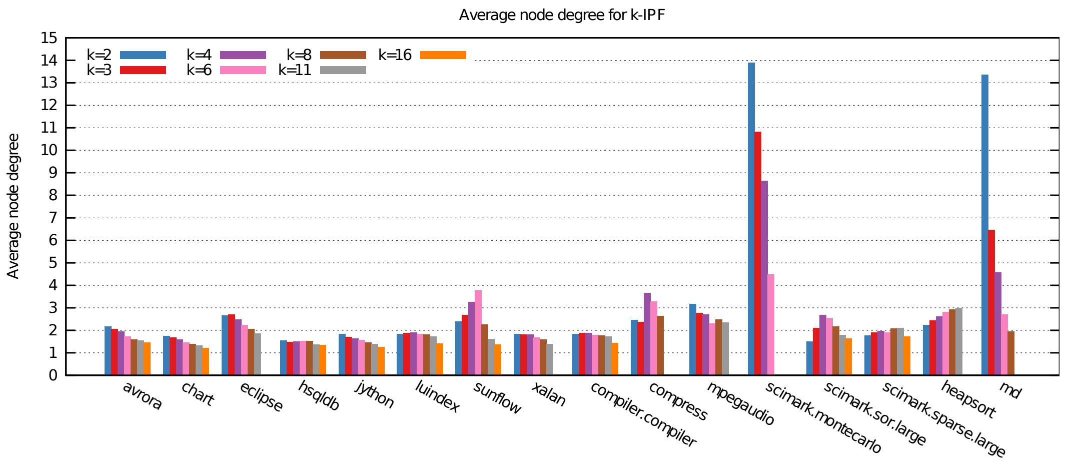


Figure 5.17. Average degree of k-IPF internal nodes.

As a final experiment, we measured structural properties of the k-IPF such as average degree of internal nodes (Figure 5.17) and the average leaf depth (Figure 5.18). Our tests reveal that the average node degree generally decreases with k, showing that similar patterns tend to appear frequently across different iterations. Some benchmarks, however, such as `sunflow` and `heapsort` exhibit a larger variety of path ramifications, witnessed by increasing node degrees at deeper levels of the k-IPF. The average leaf depth allows us to characterize the loop iteration activity of different benchmarks. Notice that for some benchmarks, such as `avrora` and `hsqldb`, most cycles consist of a small number of iterations: hence, by increasing k beyond this number, k-BLPP does not collect any additional useful information.

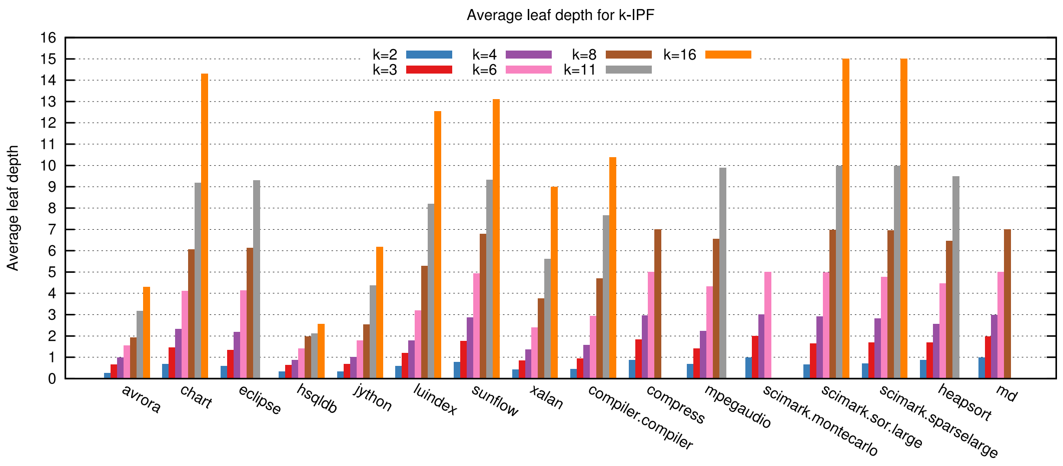


Figure 5.18. Average depth of k-IPF leaves.

5.3 OSRKit

In this section we present a preliminar experimental study of OSRKit. In particular, we aim at addressing the following questions:

Q1 How much does a never-firing OSR point impact code quality? What kind of slowdown should we expect?

Q2 What is the run-time overhead of an OSR transition, for instance to a clone of the running function?

Q3 What is the overhead of OSRKit for inserting OSR points and creating a stub or a continuation function?

XXX

5.3.1 Experimental Setup

Benchmarks

We address questions Q1-Q3 by analyzing the performance of OSRKit on a selection of the `shootout` benchmarks, also known as the Computer Language Benchmarks Game [48], running in TinyVM. In particular, we focus on single-threaded benchmarks that do not rely on external libraries to perform their core computations. Benchmarks and their description are reported in Table 5.2; four of them (`b-trees`, `mbrot`, `n-body` and `sp-norm`) are evaluated against two workloads of different size.

Benchmark	Description
b-trees	Adaptation of a GC bench for binary trees
fannkuch	Fannkuch benchmark on permutations
fasta	Generation of DNA sequences
fasta-redux	Generation of DNA sequences (with lookup table)
mbrot	Mandelbrot set generation
n-body	N-body simulation of Jovian planets
rev-comp	Reverse-complement of DNA sequences
sp-norm	Eigenvalue calculation with power method

Table 5.2. Description of the `shootout` benchmarks.

We generate the IR modules for our experiments with `clang` starting from the C version of the `shootout` suite. To cover scenarios where OSR machinery is inserted in programs with different optimization levels, we consider two versions: 1) *unoptimized*, where the only LLVM optimization we perform is `mem2reg` to promote stack references to registers and construct the SSA form; 2) *optimized*, where we apply `opt -O1` to the unoptimized version.

Environment

TinyVM supports interactive invocations of functions and it can compile LLVM IR either generated at run-time or loaded from disk. The main design goal behind TinyVM is the creation of an interactive environment for IR manipulation and JIT-compilation of functions: for instance, it allows the user to insert OSR points in loaded functions, run optimization passes on them or display their CFGs, repeatedly invoke a function for a specified amount of times and so on.

TinyVM supports dynamic library loading and linking, and comes with a helper component for MCJIT that simplifies tasks such as handling multiple IR modules, symbol resolution in the presence of multiple versions of a function, and tracking native code and other machine-level generated object such as Stackmaps. TinyVM is thus an ideal playground to exercise our OSR technique.

Platform

Experiments were performed on an octa-core 2.3Ghz Intel Xeon E5-4610 v2 with 256+256KB of L1 cache, 2MB of L2 cache, 16MB of shared L3 cache, and 128 GB of DDR3 main memory, running Debian Wheezy 7, Linux kernel 3.2.0, LLVM 3.6.2 (Release build, compiled using gcc 4.7.2), 64 bit. For each benchmark we analyze CPU time performing 10 trials preceded by an initial warm-up iteration; reported confidence intervals are stated at 95% confidence level.

5.3.2 Impact on Code Quality

In order to measure how much a never-firing OSR point might impact code quality (Q1), we analyzed the source-code structure of each benchmark and profiled its run-time behavior to identify performance-critical sections for OSR point insertion.

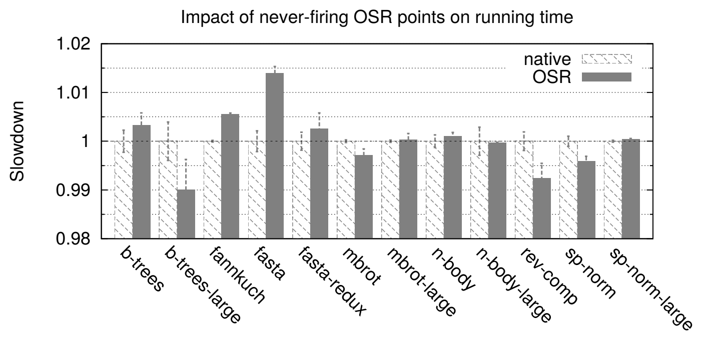


Figure 5.19. Q1: Impact on running time of never-firing OSR points inserted inside hot code portions (unoptimized code).

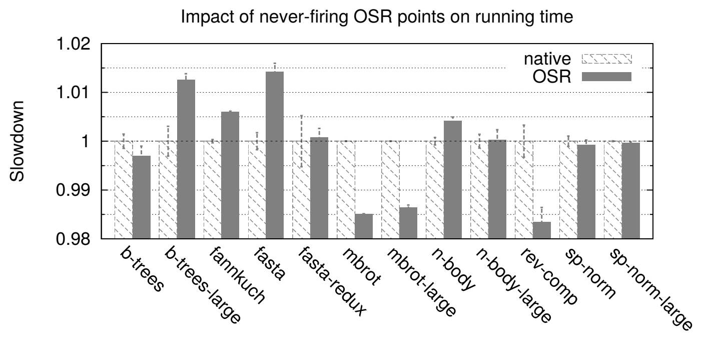


Figure 5.20. Q1: Impact on running time of never-firing OSR points inserted inside hot code portions (optimized code).

The distinction between open and resolved OSR points is nearly irrelevant in this context: we choose to focus on open OSR points, passing `null` as the `val` argument for the stub (see **XXX**).

For iterative benchmarks, we insert an OSR point in the body of their hottest loops. We classify a loop as hottest when its body is executed for a very high cumulative number of iterations (e.g., from millions up to billions) and it either calls the method with the highest *self* time in the program, or it performs the most computational-intensive operations for the program in its own body.

These loops are natural candidates for OSR point insertion: for instance, the Jikes RVM inserts yield points on backward branches to trigger operations such as method recompilation through OSR and thread preemption for garbage collection. In the `shootout` benchmarks, the number of such loops is typically 1 (2 for `spectral-norm`).

For recursive benchmarks, we insert an OSR point in the body of the method that accounts for the largest *self* execution time in the program. Such an OSR point might be useful to trigger recompilation of the code at a higher degree of optimization, enabling for instance multiple levels of inlining for non-tail-recursive functions. The only analyzed benchmark showing a recursive pattern is `b-trees`.

Results for the unoptimized and optimized versions of the benchmarks are reported in Figure 5.19 and Figure 5.20, respectively. For both scenarios we observe that the overhead is very small, i.e., less than 1% for most benchmarks and less than 2% in the worst case. For some benchmarks, code might run slightly faster after OSR point insertion due to instruction cache effects. The number of times the OSR condition is checked for each benchmark is reported in Table 5.3.

5.3.3 Overhead of OSR Transitions

Table 5.3 reports estimates of the average cost of performing an OSR transition to a clone of the running function (Q2). For each benchmark we compute the time difference between the scenarios in which an always-firing and a never-firing resolved OSR point is inserted in the code, respectively; we then normalize this difference against the number of fired OSR transitions.

		Unoptimized code		*Optimized code*	
Benchmark	Fired OSRs (M)	Live values	Avg time (ns)	Live values	Avg time (ns)
b-trees	605	2	1.731	3	0.974
b-trees-large	2 690	2	1.749	3	1.423
fannkuch	399	0	1.793	0	0.621
fasta	400	2	2.335	2	2.699
fasta-redux	400	4	2.306	4	2.269
mbrot	256	15	5.016	15	3.628
mbrot-large	1 024	15	5.268	15	4.637
n-body	50	3	2.952	3	6.929
n-body-large	500	3	2.953	3	6.953
rev-comp	6	8	-10.158	8	8.267
sp-norm	1 210	2	0.772	2	-0.030
sp-norm-large	19 360	2	0.778	2	-0.003

Table 5.3. Cost of OSR transitions to the same function. For each benchmark we report the number of fired OSR transitions (rounded to millions), the number of live values passed at the OSR point, and the average time for a transition.

Hot code portions for OSR point insertion have been identified as in the Q1 experiments for code quality. Depending on the characteristics of the hot loop, we either transform its body into a separate function and instrument its entrypoint, or, when the loop calls a method with a high self time, we insert an OSR point at the beginning of that method.

Normalized differences reported in the table represent a reasonable estimate of the average cost of firing an OSR transition. Reported numbers are in the order of nanoseconds, and might be negative due to instruction cache effects. We remark that for this experiment slicing the loop body is preferable to inserting an OSR point in it, as the continuation function should fire an OSR itself at the very next loop iteration and so on, possibly leading to an undesired stack growth.

5.3.4 OSR Machinery Generation

We now discuss the overhead of the OSRKit library for inserting OSR machinery in the IR of a function (Q3). Table 5.4 reports for each benchmark the number of IR instructions in the instrumented function and the time spent in the IR manipulation. Locations for OSR points are chosen as in the Q1 experiments, and the target function is a clone of the source function.

		Open OSR (μs)		*Resolved OSR* (μs)		
		Insert	Gen.	Insert	Generate f'$_{to}$	
Benchmark	|IR|	point	stub	point	Total	Avg/inst
b-trees	13	15.40	28.32	14.31	76.13	5.86
fannkuch	50	14.16	18.66	12.84	208.03	4.16
fasta	38	12.93	27.07	13.01	250.39	6.59
fasta-redux	55	13.79	23.44	9.32	258.36	4.70
mbrot	77	15.96	27.39	15.30	384.61	4.99
n-body	19	14.31	19.73	11.58	88.73	4.67
rev-comp	145	16.31	39.99	13.90	810.84	5.59
sp-norm	28	15.31	27.50	12.41	154.54	5.52

Table 5.4. Q3: OSR machinery insertion in optimized code. Time measurements are expressed in microseconds. Results for unoptimized code are very similar and thus not reported.

For open OSR points, we report the time spent in inserting the OSR point in the function and in generating the stub; both operations do not depend on the size of the function. For resolved OSR points, we report the time spent in inserting the OSR point and in generating the f'$_{to}$ function.

Not surprisingly, constructing a continuation function takes longer than the other operations (i.e., up to 1 ms vs. 20-40 us), as it involves cloning and manipulating the body of the target function and thus depends on its size: Table 5.4 hence comes with an additional column in which time is normalized against the number of IR instructions in the target function.

5.3.5 Discussion

Experimental results presented in the previous sections suggest that inserting an OSR point is unlikely to degrade the quality of generated code (Q1). The time required to fire an OSR transition is negligible (i.e., order of nanoseconds, Q2), while the cost of OSR-point insertion and of generating a continuation function - either when inserting a resolved OSR point, or from the callback method invoked at an open OSR transition - is likely to be dominated by the cost of its compilation (Q3).

For a front-end, the choice whether to insert an OSR point into a function for dynamic optimization merely depends on the trade-off between the expected benefits in terms of execution time and the overheads from generating on optimized version of the function and eventually JIT-compiling it; compared to these two operations, the cost of OSR-related operations is negligible. **XXX**

5.4 On-Stack Replacement à la Carte

In this section we present and evaluate an implementation in LLVM of the techniques described in Section 4.2.3. In particular, we discuss how to deal with the presence of memory `load` and `store` instructions, and how to implement algorithms `apply` and `build_comp` in a real compiler. We then investigate whether in the presence of a number of common compiler optimizations, `build_comp` can "offer" an extensive menu of possible program points where OSR can safely occur, generating the possibly required compensation code in an automated fashion.

5.4.1 Implementation

The LLVM compiler infrastructure is designed to support transparent, life-long program analysis and transformation for arbitrary programs [68]. LLVM is widely used to efficiently compile static languages (e.g., C, C++, Objective C/C++) and, as we have seen in Section 4.1.2, as a JIT compiler for a variety of dynamic languages.

The core of LLVM is its low-level intermediate representation (IR): a front-end for a high-level language can compile a program's source code to LLVM IR; platform-independent optimization passes then manipulate the IR, and a back-end eventually compiles IR to native code, performing architecture-specific further optimizations. Front-end authors can thus benefit from LLVM's shared extensive optimization pipeline to generate better code for their language.

LLVM provides an infinite set of typed *virtual registers* that can hold primitive types. Virtual registers are in SSA (Static Single Assignment) form [35], and values can be transferred between registers and memory solely via `load` and `store` operations. Virtual registers are uniquely assigned by expressions defined on incoming registers. When a program variable might have different values at different points in the control flow, the SSA form requires the insertion of ϕ-nodes to merge multiple incoming virtual registers into a new one. Note also that in the LLVM programming practice, virtual registers correspond to the instructions assigning to them.

Supporting `load` and `store` Instructions

A `store` instruction writes the content of a virtual register to a given address. For live-variable bisimilar versions of a program, a sufficient condition for which the associated multi-program is deterministic is that `store` instructions are executed at the same program point in all versions.

A `load` instruction assigns to a virtual register the value read from a given address and can be treated as a special case of variable assignment in our setting. This ensures that, under the above assumption for `store` instructions, given a program location and two program versions, if a live virtual register from one version corresponds to a live register in the other version according to $OSRMap$, then both `load` instructions would have yielded the same value.

Compiler optimizations presented in this section do not move by default `store` instructions arounds. However, enforcing the `store` hypothesis described above might be restrictive in a different optimization setting: for this reason, we describe a possible extension of our approach to deal with the issue. We can model a hoisted or sunk `store` instruction as special assignment to a variable that is assumed to be live at each program location reachable in the CFG between the original location and the insertion point. For the sinking case, when performing an OSR at one of the affected points:

- from the less to the more optimized version of the function, no compensation code is required, and repeating the sunk `store` will simply be redundant;
- from the more to the less optimized version, we realign the memory state by executing the sunk `store` (not reached yet in the current version).

The hoisting case - although `store`(s) are typically only sunk - is the converse.

Tracking Optimizations

Without loss of generality, we can capture the effects of a live-variable equivalent program transformation in terms of six primitive actions:

- `add`$(inst, loc)$: insert a new instruction *inst* at location *loc*;
- `delete`(loc): delete the instruction at *loc*;
- `hoist`$(loc, newLoc)$: hoist an instruction from *loc* to *newLoc*;
- `sink`$(loc, newLoc)$: sink an instruction from *loc* to *newLoc*;
- `replace_operand`$(inst, old\_op, new\_op)$: replace an operand *old_op* for a a given instruction with another operand *new_op*;
- `replace_all`(old_op, new_op): replace all uses in the code of an operand with another operand.

Algorithm `apply` takes as input a function and an optimization, clones the function, optimizes the clone, and finally constructs an OSR mapping between the two versions by processing the history of applied actions. Existing LLVM passes do not need to be rewritten, as we simply instrument them at places where a primitive action is performed.

Tracking actions of the first four kinds is essential in order to maintain mappings between program points from different versions. While programs expressed in the language from Section 4.2.1 are padded by an oracle with `skip` instructions for optimizations, a mapping between LLVM instruction locations for two versions should be explicitly maintained.

The OSR mapping (Section 4.2.3.1) for LLVM programs is defined as a mapping between virtual registers. For each `replace_all`(O, N) operation we can update the OSR mapping as follows. When all uses of O are replaced with N, O becomes trivially dead: as in a LVE transformation N and O yield the same result, any virtual register O' in $OSRMap$ pointing to O can be updated to point to N. This is useful for deoptimization, as our experiments suggest that a variable in an optimized program often holds the value of more than one variable in the unoptimized code.

In our experience, to make an LLVM pass OSR-aware we usually needed to insert 5-15 tracking primitive actions, while the hardest part was clearly understanding what each LLVM pass does. Readers familiar with LLVM may notice that most primitive actions mirror typical manipulation utilities used in optimization passes (e.g., `replace_all` is equivalent to LLVM's widely employed `RAUW`).

Implementing `build_comp`

In this section, we discuss the implications of implementing the `build_comp` algorithm presented in Algorithm 7 (page 50) for a program written in SSA form. While this form guarantees that the reaching definition for a variable is unique at any point it dominates, `reconstruct` gives up when attempting to reconstruct an assignment made through a ϕ function. We also conservatively prevent it from inserting `load` instructions in the compensation code.

Compared to the abstract model described in Section 4.2.1, the particular form of IR code generated by LLVM may limit in our context the effectiveness of algorithm `reconstruct`(Algorithm 9 on page 52). For this reason, in our experiments in LLVM we consider four versions of `reconstruct`, each one extending the previous one.

We denote by P the pool of variables at OSR source that are used to reconstruct assignments:

1. The first version, which we will refer to as *live*, is the base version of Algorithm 9 that uses as P only variables that are live at the OSR source.

2. An enhanced version $live_{(e)}$ exploits some features of LLVM IR. In particular, this version can recursively reconstruct a ϕ-assignment that merges together the same value for all CFG paths[5], and includes in P also non-live function arguments, as they cannot be modified in the IR.

3. A third version, which we call *alias*, can also exploit implicit aliasing information deriving from a `replace_all`(O, N). Let O' and N' be the corresponding variables according to the OSR mapping for O and N, respectively: we can add $\mathtt{N} := \mathtt{O}'$ to the compensation code required to reconstruct N when N' is not live at the source location, but O' is.

4. Finally, the fourth version *avail* includes in P also those variables that are not live at the source location, but contain available values that `reconstruct` can directly assign to the instruction operand (line 7) or assignment (line 8) being reconstructed. We exploit the uniqueness of reaching definitions in the SSA form to efficiently identify such variables.

5.4.2 Experimental Setup

Benchmarks and Environment

We implemented our technique in TinyVM, introducing a number of features to:

- clone a function f_{base} and apply a sequence of OSR-aware optimization passes, thus generating an optimized version f_{opt};

- construct and compose OSR mappings for the applied transformations;

- for each feasible OSR point in f_{base}/f_{opt}, invoke OSRKit to materialize the compensation code χ produced by `reconstruct` into a sequence of IR instructions for the OSR entry block of f'_{opt}/f'_{base} (Section 4.1.1).

[5]Compilers can place ϕ-nodes at loop exits for values that are live across the loop boundary, constructing the *Loop-Closed* SSA (LCSSA) form.

		Optimizations						Utilities	
Suite	Benchmark	*ADCE*	*CP*	*CSE*	*SCCP*	*LICM*	*Sink*	*LS*	*LCSSA*
SPEC	bzip2			✓		✓	✓		✓
	h264ref	✓		✓		✓	✓	✓	✓
	hmmer			✓		✓	✓		✓
	namd	✓	✓	✓	✓	✓	✓		✓
	perlbench	✓		✓		✓	✓	✓	✓
	sjeng			✓	✓	✓	✓		✓
	soplex			✓	✓	✓	✓		
PTS	bullet		✓	✓		✓	✓		✓
	dcraw			✓		✓	✓	✓	✓
	ffmpeg	✓	✓	✓		✓	✓	✓	✓
	fhourstones		✓	✓		✓		✓	✓
	vp8			✓		✓	✓		✓

Table 5.5. Optimizations and utility passes effective on the hottest function of each benchmark. Optimization passes have been applied in the same order (left-to-right) as they apper in the table. Utility passes *LC* and *LCSSA* are pre-requisites of *LICM*.

We instrumented a number of standard LLVM optimization passes, including *aggressive dead code elimination* (ADCE), *constant propagation* (CP), *common subexpression elimination* (CSE), *loop-invariant code motion* (LICM), *sparse conditional constant propagation* (SCCP), and *code sinking* (Sink). We also instrumented a number of utility passes required by LICM, such as *natural loop canonicalization* (LC) and *LCSSA-form construction* (LCSSA). Optimizations performed by the back-end (e.g., instruction scheduling, register allocation, peephole optimizations) do not require instrumentation as we operate at the IR level.

We evaluated our technique on the `SPEC CPU2006` [55] and the `Phoronix PTS` [85] benchmarking suites, reporting data for a subset of their C/C++ benchmarks. We profiled each benchmark to identify the hottest method and generated the IR for it using `clang` with no optimization enabled other than `mem2reg`. Starting from this version of the IR, which we will refer to as *base*, we generated an *opt* version by applying all our instrumented LLVM optimizations.

The list of benchmarks and transformations that are effective on their hottest method is reported in Table 5.5. Numbers reported in Table 5.6 for the IR manipulations performed by the transformations suggest that, while the *opt* version is typically shorter than its *base* counterpart, it might have a larger number of ϕ-nodes (most of them are inserted during the LCSSA-form construction). We observed that SCCP was able to eliminate a large number of unreachable blocks for `ffmpeg`, while for the remaining benchmarks the majority of instruction deletions are performed by CSE, which replaces all the uses of these instructions in the rest of the code with uses of equivalent available instructions.

Platform

Experiments were performed on a machine equipped with an Intel Core i7-3632QM processor, running Ubuntu 14.10, LLVM 3.6.2 (Release build), 64 bit.

		base		opt	
Benchmark	Function	$\|\pi\|$	$\|\phi\|$	$\|\pi\|$	$\|\phi\|$
bzip2	mainSort	657	32	596	44
h264ref	SetupFastFullPelSearch	671	28	576	36
hmmer	P7Viterbi	568	6	383	8
namd	ComputeNonbondedUtil::calc_pair_energy_fullelect	1737	159	1636	224
perlbench	S_regmatch	5574	305	5001	355
sjeng	std_eval	1940	93	1540	105
soplex	SPxSteepPR::entered4X	195	2	154	2
bullet	btGjkPairDetector::getClosestPointsNonVirtual	587	24	553	42
dcraw	vng_interpolate	590	37	545	49
ffmpeg	decode_cabac_residual_internal	618	34	462	40
fhourstones	ab	288	29	284	39
vp8	vp8_full_search_sadx8	334	41	299	60

Benchmark	Added	Deleted	Hoisted	Sunk	$RAUW_I$	$RAUW_C$	$RAUW_A$
bzip2	16	77	12	3	71	0	2
h264ref	9	105	4	21	102	0	0
hmmer	2	187	13	1	187	0	0
namd	68	169	36	73	145	17	0
perlbench	86	667	96	28	627	0	0
sjeng	13	413	20	34	412	1	0
soplex	0	41	2	4	41	0	0
bullet	26	60	37	3	51	1	0
dcraw	13	58	25	6	58	0	0
ffmpeg	11	168	9	17	52	51	0
fhourstones	14	20	3	0	14	2	0
vp8	19	54	17	34	54	0	0

Table 5.6. Details on the IR manipulations on the hottest function of each benchmark. For each function we report the number of instructions $|\pi|$ ($|\phi|$ of which represent ϕ-nodes) for both the *base* and the *opt* version. We then report the number of primitive actions for code manipulations tracked across the applied transformations. $RAUW_{\{A,C,I\}}$ is used to indicate `replace_all`(O, N) actions performed for some N having Argument, Constant, or Instruction type in LLVM.

5.4.3 OSR to Optimized Version

Figure 5.21 shows the fraction of program points that are feasible for an OSR from *base* to *opt* depending on the version of `reconstruct` being used.

Locations that can fire an OSR with no need of a compensation code (i.e., $\chi = \langle\rangle$) account for a limited fraction of all the potential OSR points (less than 10% for most benchmarks). This suggests that optimizations can significantly modify a program's live state across program locations.

We observe that the *live* version of `reconstruct` performs well on some benchmarks (e.g., `perlbench`, `bullet`, `dcraw`) and poorly on others (e.g., `h264ref`, `namd`).

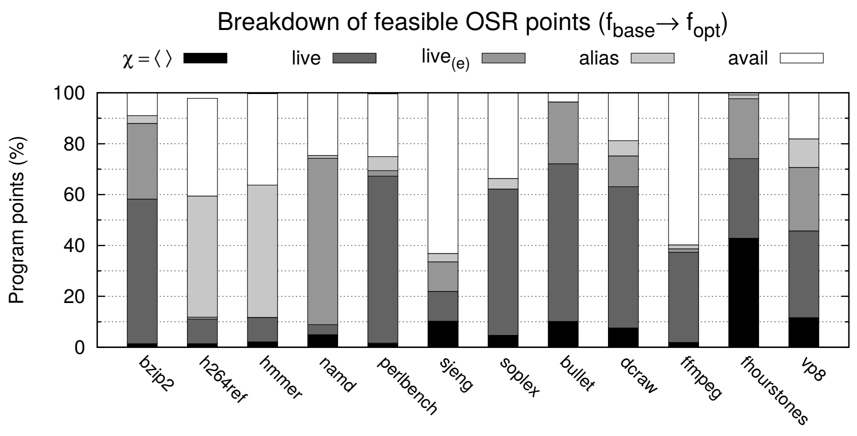


Figure 5.21. Fraction of program points that are OSR-feasible (from *base* to *opt*).

The enhancements introduced in the $live_{(e)}$ version are effective for some benchmarks (e.g., `namd`, `sjeng`), while aliasing information exploited in the *alias* version increases the number of feasible OSR points for all benchmarks. For 9 out of 12 of them, it is possible in fact to build a compensation code using only live variables at the OSR source for more than 60% of potential OSR points.

When in the *avail* version `reconstruct` is allowed to extend the liveness range of an "available" variable, the percentage of feasible OSR points grows to nearly 100%. We observed for `bullet` that a specific ϕ-node needs to be reconstructed at nearly 20% of feasible OSR points: this node takes as incoming values a number of ϕ-nodes that in turn all yield the same value. While LLVM's built-in method for detecting trivially constant ϕ-nodes does not cover this case, our recursive heuristic introduced in the $live_{(e)}$ version is able to identify the value and use it directly.

In Table 5.7 we report the average and peak size of the compensation code χ generated by the $live_{(e)}$, *alias*, and *avail* variants of `reconstruct` across feasible OSR points. Figures for *live* are not reported as they would not add much to the discussion. Note also that average values have been calculated for different sets of program points, although the set of a version includes the set of the previous version.

The assignment step of `reconstruct` (line 9 in Algorithm 9, page 52) generates an average number of instructions typically smaller than 20, with the notable exception of `perlbench`. Observe that `perlbench`'s hottest function `S_regmatch` is highly amenable to CSE: we found out that no less than 583 out of its 667 deleted instructions (thus about 10% of the *base* function - see Table 5.6) are removed by this optimization, and we believe that local CSE would shrink the OSR entry block of the continuation function f' as well. However, we would like to remark that the size of ϕ is unlikely to affect the performance of f' for a hot method, as compensation code will be located at the beginning of the function and executed only once.

The last two columns of Table 5.7 report the average and peak number of variables that are not live at the source location, but for which the *avail* version of `reconstruct` would artificially extend liveness to support OSR at more program points (i.e., those represented by the white portions of the bars in Figure 5.21). We observe that the average number of values to spill on the stack is less than 3 for 9

	$\|\chi\| \leftarrow live_{(e)}$		$\|\chi\| \leftarrow alias$		$\|\chi\| \leftarrow avail$		$\|K_{avail}\|$	
Benchmark	Avg	Max	Avg	Max	Avg	Max	Avg	Max
bzip2	4.29	14	4.3	14	4.73	13	3.6	8
h264ref	1.94	2	2.9	5	3.37	5	1.02	2
hmmer	3.3	5	16.11	23	16.63	24	4.02	7
namd	18.48	28	18.61	28	17.82	28	3.38	6
perlbench	46.29	57	46.12	57	45.82	57	1.24	12
sjeng	9.51	21	9.72	21	18.52	32	4.2	12
soplex	5.08	7	5.02	7	4.38	7	2.34	4
bullet	16.79	46	16.69	46	15.93	46	6.15	17
dcraw	7.72	15	7.6	15	7.32	15	1.97	7
ffmpeg	5.22	8	5.05	8	4.03	8	1.85	3
fhourstones	4.64	6	4.5	6	4.98	6	1.7	2
vp8	9.6	16	10.51	16	10.13	17	2.35	6
Mean	**11.07**	18.75	**12.26**	20.50	**12.81**	21.50	**2.82**	7.17

Table 5.7. Average and peak size $|\chi|$ of the compensation code generated by the $live_{(e)}$, *alias*, and *avail* versions of algorithm `reconstruct`. $|K_{avail}|$ is the size of the set of variables that we should artificially keep alive in order to make program points represented by white bars in Figure 5.21 feasible for an OSR from *base* to *opt*.

out of 12 benchmarks, with a maximum of 6.16 for `bullet`. *avail* by default will extend the liveness of an available value only if it is not possible to reconstruct it: this was implemented using a simple backtracking algorithm.

5.4.4 OSR to Base Version

Figure 5.22 reports the fraction of OSR points eligible for *opt* to *base* deoptimization. We observe that the fraction of locations that can fire an OSR with an empty χ varies significantly from benchmark to benchmark, suggesting a dependence on the structure of the original program.

For 9 out of 12 benchmarks, compensation code can be built using only live

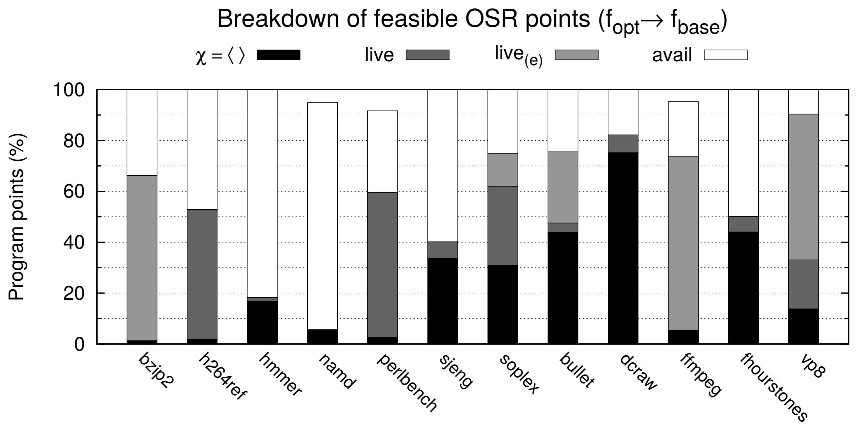


Figure 5.22. Fraction of program points that are OSR-feasible (from *opt* to *base*).

variables for more than 50% of potential OSR points. When the *avail* version is used, the percentage of feasible OSR points is greater than 90% on all benchmarks and nearly 100% for 9 out of 12 of them.

Results for the *alias* version of `reconstruct` are not reported, as they do not improve those for $live_{(e)}$. Indeed, aliasing information is useful when a variable to set at destination is aliased by multiple variables at source, which we do not expect to happen in an optimized code.

	$\lvert\chi\rvert \leftarrow live_{(e)}$		$\lvert\chi\rvert \leftarrow avail$		$\lvert K_{avail}\rvert$	
Benchmark	Avg	Max	Avg	Max	Avg	Max
bzip2	1.55	4	1.77	4	1.47	4
h264ref	4.46	9	2.82	9	1.45	7
hmmer	1	1	1	1	1.02	2
namd	1.5	2	5.93	15	4.74	18
perlbench	4.09	12	4.22	12	1.37	11
sjeng	1.29	2	1.67	11	4.09	14
soplex	3.3	4	3.3	4	1.00	1
bullet	1	1	1.26	3	1.14	2
dcraw	1.68	2	3.84	6	4.06	8
ffmpeg	1.94	5	1.95	6	1.08	4
fhourstones	0	0	1.12	4	1.42	4
vp8	5.74	13	5.51	13	1.18	5
Mean	**2.30**	4.58	**2.87**	7.33	**2.00**	6.67

Table 5.8. Average and peak size $|\chi|$ of the compensation code generated by the $live_{(e)}$ and *avail* versions of algorithm `reconstruct`. $|K_{avail}|$ is the size of the set of variables that we should artificially keep alive in order to make program points represented by white bars in Figure 5.22 feasible for an OSR from *opt* to *base*.

In Table 5.8 we report the average and peak size of the compensation code χ generated by the $live_{(e)}$ and *avail* variants of `reconstruct` across feasible OSR points, along with the average and peak number of available variables for which *avail* artificially extends liveness to support OSR at program points represented by white bars in Figure 5.22. We observe that, compare to the *base*-to-*opt* case, the size of the compensation code is much smaller, suggesting that shorter portions of executions need to be reconstructed when OSR-ing to less optimized code.

Note that the 0 values reported for `fhourstones` in the $live_{(e)}$ scenario do not imply that state compensation is not needed. In fact, the algorithm detected that each variable v to assign at the OSR landing pad for which no live counterpart was available at the source location, could be initialized with the value of either a (non-live) function argument or some live variable when v is a constant ϕ-node. In LLVM IR assignments of the form $\mathtt{x} := \mathtt{y}$ are not allowed, since all uses of `x` can simply be replaced with uses of `y`: for this reason, a $\mathtt{RAUW}(\mathtt{x}, \mathtt{y})$ operation is performed on the body of the continuation function f', where `y` is a live value transferred as argument for f', and no instruction is added to the OSR entry block of f'.

Chapter 6

Case Studies

XXX

6.1 Multi-iteration Path Profiling

In this section we consider examples of applications where k-iteration path profiling can reveal optimization opportunities or help developers comprehend relevant properties of a piece of software by identifying structured execution patterns that would be missed by an acyclic-path profiler. Our discussion is based on idealized examples found in real programs of the kind of behavior that can be exploited using multi-iteration path profiles. Since our methodology can be applied to different languages, we addressed both Java and C applications[1].

6.1.1 Masked Convolution Filters in Image Processing

As a first example, we consider a classic application of convolution filters to image processing, addressing the problem of masked filtering that arises when the user applies a transformation to a collection of arbitrary-shaped subregions of the input image. A common scenario is face anonymization, illustrated in the example of Figure 6.3. The case study discussed in this section shows that k-iteration path profiling with large values of k can identify regular patterns spanning multiple loop iterations that can be effectively exploited to speed up the code.

Figure 6.1 shows a C implementation of a masked image filtering algorithm based on a 5×5 convolution matrix[2]. The function takes as input a grayscale input image (8-bit depth) and a black and white mask image that specifies the regions of the image to be filtered. Figure 6.3 shows a sample input image (top), a mask image (center), and the output image (bottom) generated by the code of Figure 6.1 by applying a blur filter to the regions of the input image specified by the mask. Notice that the filter code iterates over all pixels of the input image and for each pixel checks if the corresponding mask is black (zero) or white (non-zero, i.e., 255). If

[1]We profiled Java programs using the k-BLPP tool described in **XXX** and C programs with manual source code instrumentation based on a C implementation of the algorithms and data structures of Section**XXX**, available at `http://www.dis.uniroma1.it/~demetres/kstream/`.

[2]The source code of our example is provided at `http://www.dis.uniroma1.it/~demetres/kstream/`.

```
#define NEIGHBOR(m,i,dy,dx,w)                  \
    (*((m)+(i)+(dy)*(w)+(dx)))

#define CONVOLUTION(i) do {                    \
    val  = NEIGHBOR(img_in, (i),               \
                    -2, -2, cols)*filter[0];   \
    val += NEIGHBOR(img_in, (i),               \
                    -2, -1, cols)*filter[1];   \
    ...
    val += NEIGHBOR(img_in, (i),
                    +2, +2, cols)*filter[24]; \
    val = val*factor+bias;                     \
    img_out[i] = (unsigned char)               \
       (val < 0 ? 0 : val > 255 ? 255 : val); \
} while(0)

void filter_conv(unsigned char* img_in,
                 unsigned char* img_out,
                 unsigned char* mask,
                 char filter[25],
                 double factor, double bias,
                 int rows, int cols) {
    int val;
    long n = rows*cols, i;

    for (i = 0; i < n; i++)
        if (mask[i]) img_out[i] = img_in[i];
        else CONVOLUTION(i);
}
```

Figure 6.1. Masked image filtering code based on a convolution matrix.

the mask is white, the original grayscale value is copied from the input image to the output image; otherwise, the grayscale value of the output pixel is computed by applying the convolution kernel to the neighborhood of the current pixel in the input image. To avoid expensive boundary checks in the convolution operation, the mask is preliminarily cropped so that all values near the border are white (this operation takes negligible time).

Figure 6.2 shows a portion of the 10-IPF forest containing the trees rooted at the BL path IDs that correspond to: the path entering the loop (ID=0), the copy branch taken in the loop body when the mask is non-zero (ID=1), and the convolution branch taken in the loop body when the mask is zero (ID=2). The 10-IPF was generated on the workload of Figure 6.3 and was pruned by removing all nodes whose counters are less than 0.01% of the counter of their parents or less than 0.01% of the counter of their roots. For each node v in the forest, if v has a counter that is $X\%$ of the counter of its parent and is $Y\%$ of the counter of the root, then the edge leading to v is labeled with "$X\%(Y\%)$". A visual analysis of the forest shows that:

- the copy branch (1) is more frequent than the convolution branch (2);
- 98.9% of the times a copy branch (1) is taken, it is repeated consecutively at least 10 times, and only 0.1% of the times is immediately followed by a

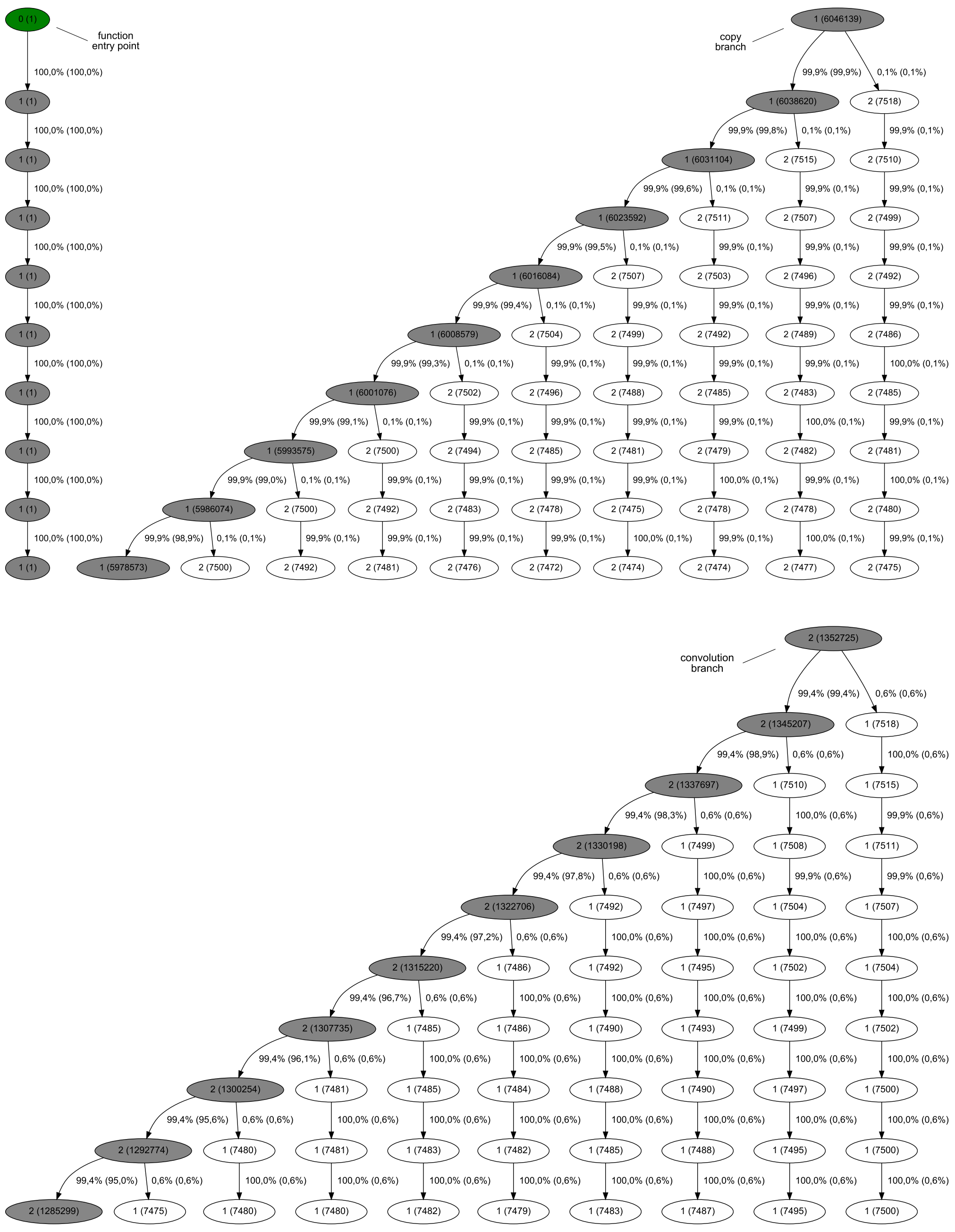


Figure 6.2. 10-IPF forest of the code of Figure 6.1 on the workload of Figure 6.3.

convolution branch;

- 95% of the times a convolution branch (2) is taken, it is repeated consecutively at least 10 times, and only 0.6% of the times is immediately followed by a copy branch.

This entails that both the copy and the convolution operations are repeated along long consecutive runs. The above properties are typical of masks used in face anonymization and other common image manipulations based on user-defined se-

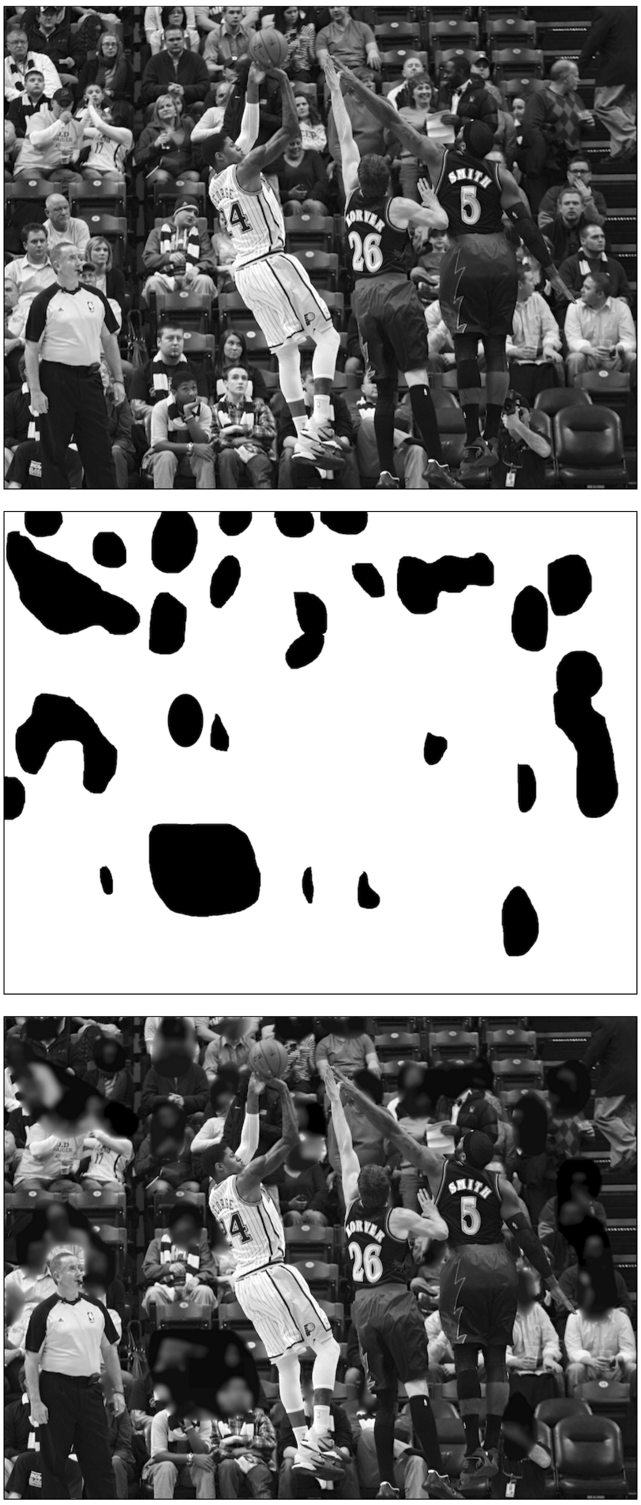


Figure 6.3. Masked blur filter example: original 3114 × 2376 image (top), filter mask (center), filtered image (bottom).

lections of portions of the image. The collected profiles suggest that consecutive iterations of the same branches may be selectively unrolled as shown in Figure 6.4. Each iteration of the outer loop, designed for a 64-bit platform, works on 8 (rather than 1) pixels at a time. Three cases are possible:

1. the next 8 mask entries are all 255 (white): the 8 corresponding input pixel values are copied to the output image at once with a single assignment instruction;
2. the next 8 mask entries are all 0 (black): the kernel is applied sequentially to each of the next 8 input pixels;
3. the next 8 mask entries are mixed: an inner loop performs either copy or convolution on the corresponding pixels.

```
for (i = 0; i < n-7; i += 8) {

   if (*(long*)(mask+i) == 0xFFFFFFFFFFFFFFFF)
      *(long*)(img_out+i) = *(long*)(img_in+i);

   else if (*(long*)(mask+i) == 0) {
      CONVOLUTION(i);
      CONVOLUTION(i+1);
      CONVOLUTION(i+2);
      CONVOLUTION(i+3);
      CONVOLUTION(i+4);
      CONVOLUTION(i+5);
      CONVOLUTION(i+6);
      CONVOLUTION(i+7);
   }

   else for (j = i; j < i+8; j++)
      if (mask[j]) img_out[j] = img_in[j];
      else CONVOLUTION(j);
}
```

Figure 6.4. Optimized 64-bit version of the loop of Figure 6.1.

Performance Analysis

To assess the benefits of the optimization performed in Figure 6.4, we conducted several tests on recent commodity platforms (Intel Core 2 Duo, Intel Core i7, Linux and MacOS X, 32 and 64 bits, `gcc` -O3), considering a variety of sample images and masks with regions of different sizes and shapes. We obtained non-negligible speedups on all our tests, with a peak of about 21% on the workload of Figure 6.3 (3114×2376 pixels) and about 30% on a larger 9265×7549 image with a memory footprint of about 200 MB. In general, the higher the white entries in the mask, the faster the code, with larger speedups on more recent machines. As we expected, for entirely black masks the speedup was instead barely noticeable: this is due to the fact that the convolution operations are computationally demanding and tend to hide the benefits of loop unrolling.

Discussion

The example discussed in this section shows that both the ability to profile paths across multiple iterations, and the possibility to handle large values of k played a crucial role in optimizing the code. Indeed, acyclic-path profiling would count the number of times each branch is taken, but would not reveal that they appear in consecutive runs. Moreover, previous multi-iteration approaches that only handle very small values of k would not capture the long runs that make the proposed optimization effective.

For the example of Figure 6.3, an acyclic-path profile would indicate that the copy branch is taken 81.7% of the times, but not how branches are interleaved. From this information, we would be able to deduce that the average length of a sequence of consecutive white values in the mask is ≥ 4. Our profile shows that, 98.9% of the times the actual length is at least 10, fully justifying our optimization that copies 8 bytes at a time. The advantage of k-iteration path profiling increases for masks with a more balanced ratio between white and black pixels: for a 50-50 ratio, an acyclic-path profile would indicate that the average length of consecutive white/black runs is ≥ 1, yielding no useful information for loop unrolling purposes.

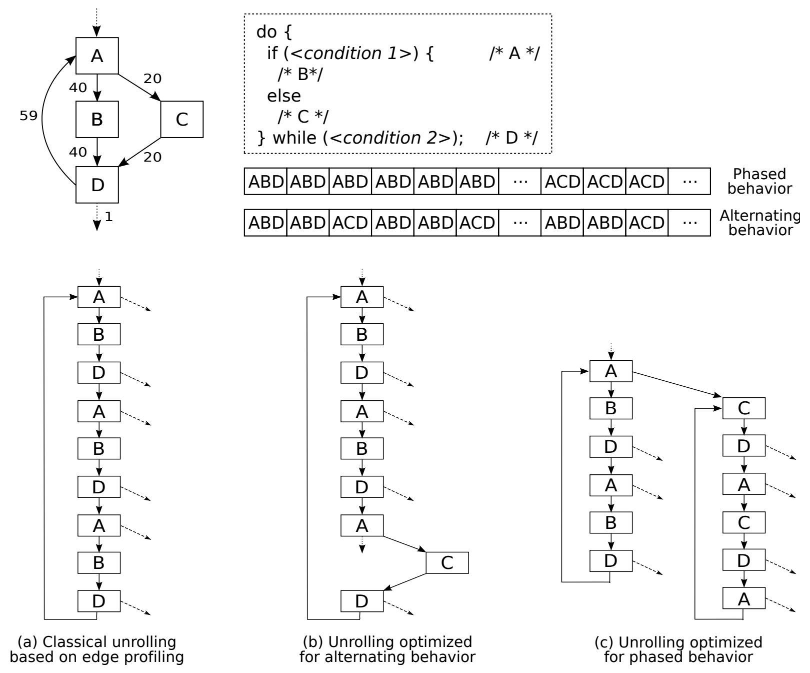
The execution pattern where the same branches are repeatedly taken over consecutive loop iterations is common to several other applications, which may benefit from optimizations that take advantage of long repeated runs. For instance, the `LBM_performStreamCollide` function of the `lbm` benchmark from the `SPEC CPU2006` suite iterates over a 3D domain, simulating incompressible fluid dynamics based on the Lattice Boltzmann Method. An input geometry file specifies obstacles that determine a steady state solution. The loop contains branches that depend upon the currently scanned cell, which alternates between obstacles and void regions of the domain, producing a k-IPF similar to that of Figure 6.2 on typical workloads.

6.1.2 Instruction Scheduling

Young and Smith [108] have shown that path profiles spanning multiple loop iterations can be used to improve the construction of superblocks in trace schedulers.

Global instruction scheduling groups and orders the instructions of a program in order to match the hardware resource constraints when they are fetched. In particular, trace schedulers rely on the identification of traces (i.e., sequences of basic blocks) that are frequently executed. These traces are then extended by appending extra copies of likely successors blocks, in order to form a larger pool of instructions for reordering. A trace that is likely to complete is clearly preferable, since instructions moved before an early exit point are wasted work.

Superblocks are defined as sequences of basic blocks with a single entry point and multiple exit points; they are useful for maintaining the original program semantics during a global code motion. Superblock formation is usually driven by edge profiles: however, path profiles usually provide better information to determine which traces are worthwhile to enlarge (i.e., those for which execution reaches the ending block most of the times). Figure 6.5 shows how superblock construction may benefit from path profiling information for two different behaviors, characterized by the same edge profile, of a `do` ... `while` loop.



```
do {
  if (<condition 1>) {          /* A */
    /* B*/
  else
    /* C */
} while (<condition 2>);    /* D */
```

40
20
59
40
20
1
ABD ABD ABD ABD ABD ABD ··· ACD ACD ACD ···
Phased behavior
ABD ABD ACD ABD ABD ACD ··· ABD ABD ACD ···
Alternating behavior
(a) Classical unrolling based on edge profiling
(b) Unrolling optimized for alternating behavior
(c) Unrolling optimized for phased behavior

Figure 6.5. Superblock construction using cyclic-path profiles.

Path profiling techniques that do not span multiple loop iterations chop execution traces into pieces separated at back edges, hence the authors collect execution frequencies for *general paths* [109], which contain any contiguous sequences of CFG edges up to a limiting path length; they use a path length of 15 branches in the experiments.

Example

Phased and alternating behaviors as in Figure 6.5 are quite common among many applications, thus offering interesting optimization opportunities. For instance, the convolution filter discussed in the previous section is a clear example of phased behavior. An alternating behavior is shown by the `checkTaskTag` method of class `Scanner` in the `org.eclipse.jdt.internal.compiler.parser` package of the `eclipse` benchmark included in the `DaCapo` release 2006-10-MR2. In Figure 6.6 we show a subtree of the 11-IPF generated for this method; in the subtree, we pruned all nodes with counters less than 10% of the counter of the root. Notice that, after executing the BL path with ID 38, in 66% of the executions the program continues with 86, and in 28% of the executions with BL path 87. When 86 follows 38, in 100% of the executions the control flow takes the path $\langle 86, 86, 86, 755\rangle$, which spans four loop iterations and may be successfully unrolled to perform instruction scheduling. Interestingly, sequence $\langle 38, 86, 86, 86, 755, 38, 86, 86, 86, 755, 38\rangle$ of 11 BL path IDs, highlighted in Figure 6.6, accounts for more than 50% of all executions of the first BL path in the sequence, showing that sequence $\langle 38, 86, 86, 86, 755\rangle$ is likely to be repeated consecutively more than once.

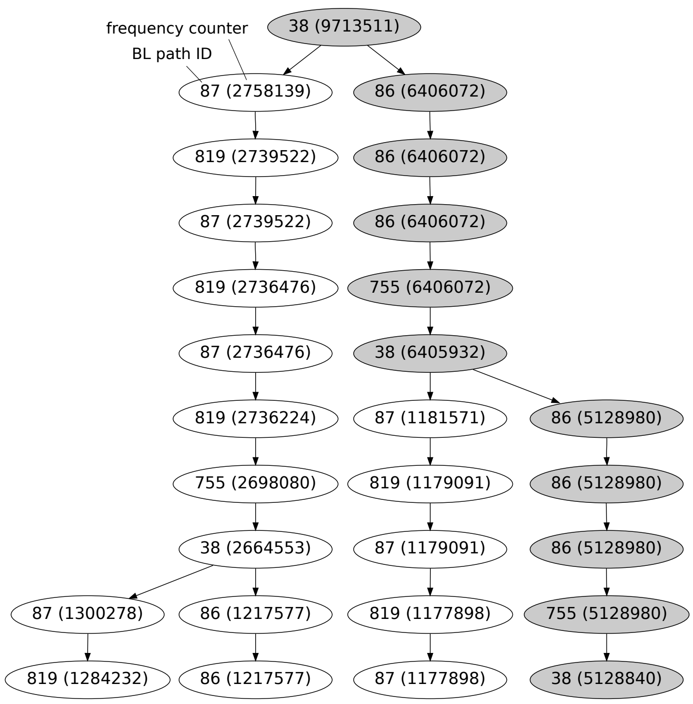


Figure 6.6. Subtree of the 11-IPF of method `org.eclipse.jdt.internal.compiler.parser.Scanner.checkTaskTag` taken from release 2006-10-MR2 of the `DaCapo` benchmark suite.

Discussion

The work presented in [108] focused on assessing the benefits of using general paths for global instruction scheduling, rather than on how to profile them. As we have seen in Section **XXX**, compared to our approach the technique proposed by Young [109] for profiling general paths scales poorly for increasing path lengths both in terms of space usage and running time. We believe that our method, by substantially reducing the overhead of cyclic-path profiling, has the potential to provide a useful ingredient for making profile-guided global instruction scheduling more efficient in modern compilers. **XXX** JIT? trace stitching?

6.2 Optimizing Higher-Order Functions in MATLAB

In this section we show how OSRKit can be used in a production VM to implement aggressive optimizations for dynamic languages; in particular, we focus on MATLAB, a popular dynamic language for scientific and numerical programming.

Introduced in the late 1970s mainly as a scripting language for performing computations through efficient libraries, MATLAB has evolved over the years into a more complex programming language with support for high-level features such as functions, packages and object orientation. A popular feature of the language is the `feval` construct, a built-in higher-order function that applies the function passed as first parameter to the remaining arguments (e.g., `feval(g,x,y)` computes `g(x,y)`).

This feature is heavily used in many classes of numerical computations, such as iterative methods for approximate solutions of an ordinary differential equation (ODE) and simulated annealing heuristics to locate a good approximation to the global optimum of a function in a large search space.

A previous study by Lameed and Hendren [67] shows that the overhead of an `feval` call is significantly higher than a direct call, especially in JIT-based execution environments such as McVM [29] and the proprietary MATLAB JIT accelerator by Mathworks. In fact, the presence of an `feval` instruction can disrupt the results of intra- and inter-procedural level for type and array shape inference analyses, which are key factors for efficient code generation. Furthermore, since `feval` invocations typically require a fallback to an interpreter, parameters passed to an `feval` are typically boxed to make them more generic.

Our case study presents a novel technique for optimizing `feval` in the McVM virtual machine, a complex research project developed at McGill University. McVM is publicly available [102] and includes: a front-end for lowering MATLAB programs to an intermediate representation called IIR that captures the high-level features of the language; an interpreter for running MATLAB functions and scripts in IIR format; a manager component to perform analyses on IIR; a JIT compiler based on LLVM for generating native code for a function, lowering McVM IIR to LLVM IR; a set of helper components to perform fast vector and matrix operations using optimized libraries such as ATLAS, BLAS and LAPACK.

McVM implements a function versioning mechanism based on type specialization, which is the main driver for generating efficient code [29]: for each IIR representation of a function, different IR versions are generated according to the types of the arguments at each call site. The number of generated versions per function is on average small (i.e., less than two), as in most cases functions are always called with the same argument types.

6.2.1 Current Approaches

Lameed and Hendren [67] proposed two dynamic techniques for optimizing `feval` instructions in McVM: *JIT-based* and *OSR-based* specialization. Both attempt to optimize a function f that contains instructions of the form `feval`$(g, ...)$, leveraging information about g and the type of its arguments observed at run-time. The optimization produces a specialized version f' where `feval`$(g, x, y, z, ...)$ instructions are replaced with direct calls of the form $g(x, y, z, ...)$.

The two approaches differ in the points where code specialization is performed. In JIT-based specialization, f' is generated when f is called. In contrast, the OSR-based method interrupts f as it executes, generates a specialized version f', and resumes from it.

Another technical difference, which has substantial performance implications, is the representation level at which optimization occurs in the two approaches. When a function f is first compiled from MATLAB to IIR, and then from IIR to IR, the functions it calls via `feval` are unknown and the type inference engine is unable to infer the types of their returned values. Hence, these values must be kept boxed in heap-allocated objects and handled with slow generic instructions in the IR representation of f (suitable for handling different types).

The JIT method works on the IIR representation of f and can resort to the full power of type analysis to infer the types of the returned values of g, turning the slow generic instructions of f into fast type-specialized instructions in f'. When g is one of the parameters of f, each call to f can be redirected to a dispatcher that evaluates at run-time the value of the argument to use for the `feval` and executes either a previously compiled cached code or generates and JIT-compiles a version of the function optimized for the current value.

On the other hand, OSR-based specialization operates on the IR representation of f, which prevents the optimizer from exploiting type inference. As a consequence, for f' to be sound, the direct call to g must be guarded by a condition that checks if the type of its parameters remain the same as observed at the time when f was interrupted. If the guard fails, or the `feval` target g changes, the code falls back to executing the original `feval` instruction.

JIT-based specialization is substantially faster than OSR-based specialization due to the benefits of type inference, but is less general as it only works if the `feval` argument g is one of the parameters of f. JIT-based specialization thus cannot be applied to scenarios where, e.g.:

- f is an `inline` or an anonymous function defined in g;
- f is the return value from a previous call in g to another function;
- f is retrieved from a data structure [67];
- f is a constant string containing the name of a user-defined function (a typical misuse of `feval` according to [88]).

6.2.2 A New Approach

In this section, we present a new approach that combines the flexibility of OSR-based specialization with the efficiency of the JIT-based method, answering an open question raised by Lameed and Hendren [67].

The key idea is to lift the f-to-f' optimization performed by the OSR-based specialization from IR to IIR level. This makes it possible to perform type inference in f', generating a much more efficient code. The main technical challenge of this idea is that the program's state in f at the OSR point may be incompatible with the state of f' from which execution continues. Indeed, some variables may be boxed in f and unboxed in f'. Hence, compensation code is needed to adjust the state by performing live variable unboxing during the OSR.

Implementation

We implemented our approach in McVM[3], extending it with four main components:

1. An analysis pass to identify optimization opportunities for `feval` instructions in the IIR of a function.
2. An extension for the IIR compiler to track the *variable map* between IIR and IR objects at `feval` sites.

[3]As a by-product of our project, we ported the MATLAB McVM virtual machine from the LLVM legacy JIT to the new MCJIT toolkit. Our code is available at https://github.com/dcdelia/mcvm.

3. An OSR inserter based on OSRKit to inject open OSR points in the IR for IIR locations annotated during the analysis pass.
4. An `feval` optimizer triggered at OSR points, which uses:
   (a) a profile-driven IIR generator to replace `feval` calls with direct calls;
   (b) a helper component to lower the optimized IIR function to IR and construct a state mapping;
   (c) a code caching mechanism to handle the compilation of the continuation functions.

We remark that our implementation heavily depends on OSRKit's ability to handle compensation code.

Analysis Pass. The analysis pass, which is fully integrated in McVM's analysis manager, groups `feval` instructions whose first argument is reached by the same definition, and for each group marks for instrumentation only those instructions that are not dominated by others, so that the function can be optimized as early as possible at run-time. It also determines whether the value of the argument can change across two executions of the same `feval`, and a run-time guard must thus be inserted during the optimization phase.

IIR Compiler Extension. The extension operates when the IIR compiler processes an annotated `feval` instruction. It builds a *variable map* between IIR and IR objects, and keeps track of the `llvm::BasicBlock*` b created for the `feval` in the IR code and of the `llvm::Value*` object g used as its first argument.

OSR Inserter. The OSR inserter uses the b and g objects collected by the compiler extension as basic block and `val` argument in the open-OSR stub that invokes the `feval` optimizer, respectively.

Optimizer. The core of our optimization pipeline is the optimizer module, which is called as `gen` function in the open OSR stub (see **XXX**) created by the OSR inserter. It receives the IR version f^{IR} of function f, the basic block of f^{IR} where the OSR was fired, and the native code address of the `feval` target function g. As a first step, the optimizer looks up the IR code of g by its address and checks whether a previously compiled version of f specialized with g was previously cached. If not, a new function f^{IIR}_{opt} is generated by cloning the IIR representation f^{IIR} of f and by replacing all `feval` calls to g in f^{IIR}_{opt} with direct calls.

As a next step, the optimizer asks the IIR compiler to lower f^{IIR}_{opt} to f^{IR}_{opt}. During the process, the compiler stores the variable map between IIR and IR objects at the direct call replacing the `feval` instruction that triggered the OSR.

Using this map and the one stored during the lowering of f^{IIR}, the optimizer constructs a state mapping between f^{IR} and f^{IR}_{opt}. In particular, for each value in f^{IR}_{opt} live at the continuation block we determine whether we can assign to it a live value passed at the OSR point, or a compensation code is required to set its value.

Notice that, since the type inference engine yields more accurate results for f_{opt}^{IIR} compared to f^{IIR}, the IIR compiler can in turn generate efficient specialized IR code for representing and manipulating IIR variables, and compensation code is typically required to unbox or downcast some of the live values passed at the OSR point, or to materialize as an IR object an IIR variable previously accessed through `get/set` methods from McVM's environment.

Once a state mapping has been constructed, the optimizer asks OSRKit to generate the continuation function for the OSR transition and then executes it, also storing the address of the compiled function in the internal code cache.

```
define void @odeEuler_OSR(
  i64 %0, i64 %1, i8* %2, i8* %3, i8* %4,
  i64 %5, i8* %6, double %7,
  { i8*, i8*, i64 }* %8, i8* %9) {

osr.entry:
  %castUNKtoMF64 = call double
      @"MatrixF64Obj::getScalarVal"(i8* %2)
  %castUNKtoMF64_2 = call double
      @"MatrixF64Obj::getScalarVal"(i8* %4)
  %envLookupFory = call i8*
      @"Environment::lookup"(i8* %9, i8* inttoptr
      (i64 32152960 to i8*))
  %10 = alloca [16 x i8]
  %11 = alloca [24 x i8]
  br label %31
```

Figure 6.7. Compensation code for `odeEuler` benchmark. McVM-specific instructions are highlighted in grey.

An example of compensation code is reported in Figure 6.7. In order to correctly resume the execution at the first instruction in basic block `%31`, the entrypoint of `odeEuler`'s continuation function executes a sequence of instructions that: 1) convert to `double` two live variables – i.e., function arguments `%2` and `%4` – that are represented as boxed values in the unoptimized function, 2) look up in McVM's environment at `%9` the pointer to the object instantiated for the symbol description stored at address `0x32152960`, and 3) allocate on the stack two buffers of 16 and 24 bytes, respectively.

6.2.3 Performance Analysis

We now analyze the impact of our optimization technique for `feval` on the running time of a few numeric benchmarks, namely `odeEuler`, `odeMidpt`, `odeRK4`, and `sim_anl`. The first three benchmarks [89] solve an ordinary differential equation for heat treating simulation using the Euler, midpoint, and Range-Kutta method, respectively; the last benchmark minimizes the six-hump camelback function with the method of simulated annealing [34].

We report the speed-ups enabled by our technique in Table 6.1, using the running times for McVM's `feval` default dispatcher as baseline. As the dispatcher typically JIT-compiles the invoked function, we also analyzed running times when the dispatcher calls a previously compiled function. In the last column, we show speed-ups from a modified version of the benchmarks in which each `feval` call is replaced by hand with a direct call to the function in use for the specific benchmark.

Unfortunately, we are unable to compute direct performance metrics for the solution by Lameed and Hendren since its source code has not been released. Figures in their paper [67] show that for the very same MATLAB programs the speed-up of the OSR-based approach is on average within 30.1% of the speed-up of hand-coded optimization (ranging from 9.2% to 73.9%); for the JIT-based approach, the average grows to 84.7% (ranging from 75.7% to 96.5%).

Benchmark	Base (cached)	Optimized (JIT)	Optimized (cached)	Direct (by hand)
odeEuler	1.046	2.796	2.800	2.828
odeMidpt	1.014	2.645	2.660	2.685
odeRK4	1.005	2.490	2.582	2.647
sim_anl	1.009	1.564	1.606	1.612

Table 6.1. Q4: Speedup comparison for `feval` optimization.

Our optimization technique yields speed-ups that are very close to the upper bound given from by-hand optimization; in the *worst case* (`odeRK4` benchmark), we observe a 94.1% when the optimized code is generated on the fly, which becomes 97.5% when a cached version is available. Compared to their OSR-based approach, the compensation entry block is a key driver of improved performance, as the benefits from a better type-specialized whole function body outweigh those from performing a direct call using boxed arguments and return values in place of the original `feval`.

6.2.4 Discussion

The ideas presented in this section advance the state of the art of `feval` optimization in MATLAB runtimes. Similarly to OSR-based specialization, we do not place restrictions on the functions that can be optimized. On the other hand, we work at IIR (rather than IR) level as in JIT-based specialization, which allows us to perform type inference on the code with direct calls. Working at IIR level eliminates the two main sources of inefficiency of OSR-based specialization:

1. we can replace generic instructions with specialized instructions, and
2. the types of g's arguments do not need to be cached or guarded as they are statically inferred.

These observations are confirmed in practice by experiments on a number of typical benchmarks from the MATLAB community. **XXX**

6.3 Source-level Debugging of Optimized Code

A *source-level* (or *symbolic*) *debugger* is a program development tool that allows a programmer to monitor an executing program at the source-language level. Interactive mechanisms are typically provided to the user to halt/resume the execution at *breakpoints*, and to inspect the state of the program in terms of its source language.

The importance of the design and use of these tools was already clear in the '60s [42]. In a production environment it is desirable to use optimizations, and bugs can surface when optimizations are enabled, as the debuggable translation of a program may hide them, or because differences in timing behavior may cause the appearance of bugs due to race conditions [1]. Also, optimizations may be absolutely necessary to executed a program - for example, because of memory limitations, efficiency reasons, or other platform-specific constraints.

As pointed out by Hennessy in his seminal paper from 1982 [54], a classic conflict exists between the application of optimization techniques and the ability to debug a program symbolically. A debugger provides the user with the illusion that the source program is executing one statement at a time. On the other hand, optimizations preserve the semantic equivalence between optimized and unoptimized code, but normally alter the structure or intermediate results of a program.

Two problems surface when trying to symbolically debug optimized code [2, 58]. First, the debugger must determine the position in the optimized code that corresponds to the breakpoint in the source code (*code location* problem). Second, the user expects to see the values of source variables at a breakpoint in a manner consistent with the source code, even though the optimizer might have deleted or reordered instructions, or values might have been overwritten as a consequence of the register allocator's choices (*data location* problem).

Thus, when attempting to debug optimized programs, debuggers may give misleading information about the value of variables at breakpoints. Hence, the programmer has the difficult task of attempting to unravel the optimized code and determine what values the variables should have [54].

In general, there are two ways for a symbolic debugger to present meaningful information about the debugged optimized program [106]. It provides *expected behavior* of the program if it hides the effect of the optimizations from the user and presents the program state consistent with what the user expects from the unoptimized code. It provides *truthful behavior* if it makes the user aware of the effects of optimizations and warns of possibly surprising outcomes.

Adl-Tabatabai observes in his PhD thesis that constraining optimizations or adding machinery during compilation to aid debugging do not solve the problem of debugging the optimized translation of a program, as the user debugs suboptimal code [1]. Source-level debuggers thus need to implement techniques to recover expected behavior when possible, without relying on intrusive compiler extensions.

6.3.1 Using `build_comp` for State Recovery

On-Stack Replacement has been pioneered in implementations of the SELF programming language to provide expected behavior with globally optimized code [57]. OSR shields the debugger from the effects of optimizations by dynamically deoptimizing code on demand. Debugging information is supplied by the compiler at discrete *interrupt points*, which act as a barrier for optimizations, letting the compiler run unhindered between them. Starting from the observation that our algorithms for generating OSR mappings (Section 4.2.3.1) do not place barriers for live-variable equivalent transformations, we investigated whether they could encode useful information for expected-behavior recovery in a source-level debugger.

As in most recent works on optimized code debugging, we focus on identifying and recovering scalar source variables in the presence of global optimizations. In LLVM, debugging information is inserted by the front-end as *metadata* attached to global variables, single instructions, functions or entire IR modules. Debugging metadata are transparent to optimization passes, do not prevent optimizations from happening, and are designed to be agnostic about the target debugging information representation (e.g., DWARF, stabs). Two intrinsics are used to associate IR virtual registers with source-level variables:

- `llvm.dbg.declare` typically associates a source variable with an `alloca`[4] buffer;
- `llvm.dbg.value` informs that a source variable is being set to the value held by the virtual register.

We extended TinyVM to reconstruct this mapping and also to indentify which program locations in the unoptimized IR version f_{base} correspond to source-level locations for a function (as they would correspond to a possible user breakpoint location). An OSR mapping is then generated when OSR-aware transformation passes are applied to f_{base} to generate the optimized version f_{opt}. For each location in f_{opt} that might correspond to (i.e., have as OSR landing pad) a source-level location in f_{base}, we determine which variables live at destination are live also at source (and thus yield the same value), and which instead need to be reconstructed. We thus rely on the SSA form to identify which assignment(s) should be made in `reconstruct`, as distinct assignments to source-level variables are represented by distinct IR objects. ϕ-nodes at control-flow merge points of course can not be reconstructed, but our preliminary experimental investigation suggests that this might not be a huge issue in practice.

6.3.2 The SPEC CPU2006 Benchmarks

To capture a variety of programming patterns and styles from applications with different sizes, we have analyzed each method for each C benchmark in the SPEC CPU2006 suite, applying the same sequence of OSR-aware optimization passes used in Section 5.4.2 to the baseline IR version obtained with `clang −O0` and then processed with `mem2reg`. Table 6.2 reports for each benchmark the code size, the total number of functions in it, the number of functions amenable to optimization and, in turn, how many optimized functions report “endangered” user variables from a source-level debugger’s perspective.

We observe that the fraction of functions that do not benefit from optimizations (i.e., $1 - |F_{opt}|/F_{tot}|$) ranges from one tenth to one third of the total number of functions. For the optimized functions, the fraction of those that belong to F_{end} - defined as the set of functions that require recovery of the expected behavior - ranges from 0.11 (`libquantum`) to 0.54 (`gobmk`).

Table 6.3 reports figures that we have collected for functions in F_{end}. We observe that on average, more than one in every four program points there is at least a user

[4]`alloca` is used to allocate space on the stack of the current function, to be automatically released when the function returns. Front-ends are not required to generate code in SSA form, but they can manipulate local variables created with `alloca` using `load` and `store` instruction. Then the SSA form can be constructed using `mem2reg`.

		Functions					
		Total	Optimized		Endangered		
Benchmark	LOC	$\|F_{tot}\|$	$\|F_{opt}\|$	$\frac{\|F_{opt}\|}{\|F_{tot}\|}$	$\|F_{end}\|$	$\frac{\|F_{end}\|}{\|F_{tot}\|}$	$\frac{\|F_{end}\|}{\|F_{opt}\|}$
bzip2	8 293	100	66	0.66	24	0.24	0.36
gcc	521 078	5 577	3 884	0.70	1 149	0.21	0.30
gobmk	197 215	2 523	1 664	0.66	893	0.35	0.54
h264ref	51 578	590	466	0.79	163	0.28	0.35
hmmer	35 992	538	429	0.80	80	0.15	0.19
lbm	1 155	19	17	0.89	2	0.11	0.12
libquantum	4 358	115	85	0.74	9	0.08	0.11
mcf	2 658	24	21	0.88	11	0.46	0.52
milc	15 042	235	157	0.67	34	0.14	0.22
perlbench	155 418	1 870	1 286	0.69	593	0.32	0.46
sjeng	13 847	144	113	0.78	31	0.22	0.27
sphinx3	25 090	369	275	0.75	76	0.21	0.28

Table 6.2. Characteristics of the C benchmarks from the `SPEC CPU2006` suite.

	Fraction of affected program points		Endangered user vars per affected point		
Benchmark	Avg_w	Avg_u	Avg	σ	Max
bzip2	0.17	0.12	1.22	0.55	5
gcc	0.25	0.22	1.13	0.31	14
gobmk	0.40	0.29	1.48	0.72	9
h264ref	0.45	0.55	1.69	1.23	14
hmmer	0.17	0.22	1.13	0.37	5
lbm	0.30	0.51	1.97	1.37	3
libquantum	0.13	0.10	1.06	0.17	2
mcf	0.35	0.32	1.00	0.00	1
milc	0.24	0.21	1.14	0.29	3
perlbench	0.37	0.35	1.16	0.36	8
sjeng	0.26	0.20	1.24	0.42	3
sphinx3	0.29	0.31	1.19	0.44	6
Mean	0.28	0.28	1.28	0.52	6.08

Table 6.3. Fraction of program points with endangered user variables, and number of affected variables. The second and third column report weighted Avg_g and unweighted Avg_u average, respectively, of the fraction of such points for functions in F_{end}. We use the number of IR instructions in the unoptimized code as weight for computing Avg_w, and consider only IR program points corresponding to source-level locations. We then show mean, std deviation, and peak number of endangered variables at such points.

variable whose source-level value might not be reported correctly by a debugger. For most functions in the benchmarks, the average number of affected user variables at such points ranges between 1 and 2, although for some benchmarks we observe high peak values at specific points (e.g., 9 for `gobmk` and 14 for `gcc` and `h264ref`).

To investigate possible correlations between the size of a function and the number of user variables affected by source-level debugging issues, we analyzed the corpus

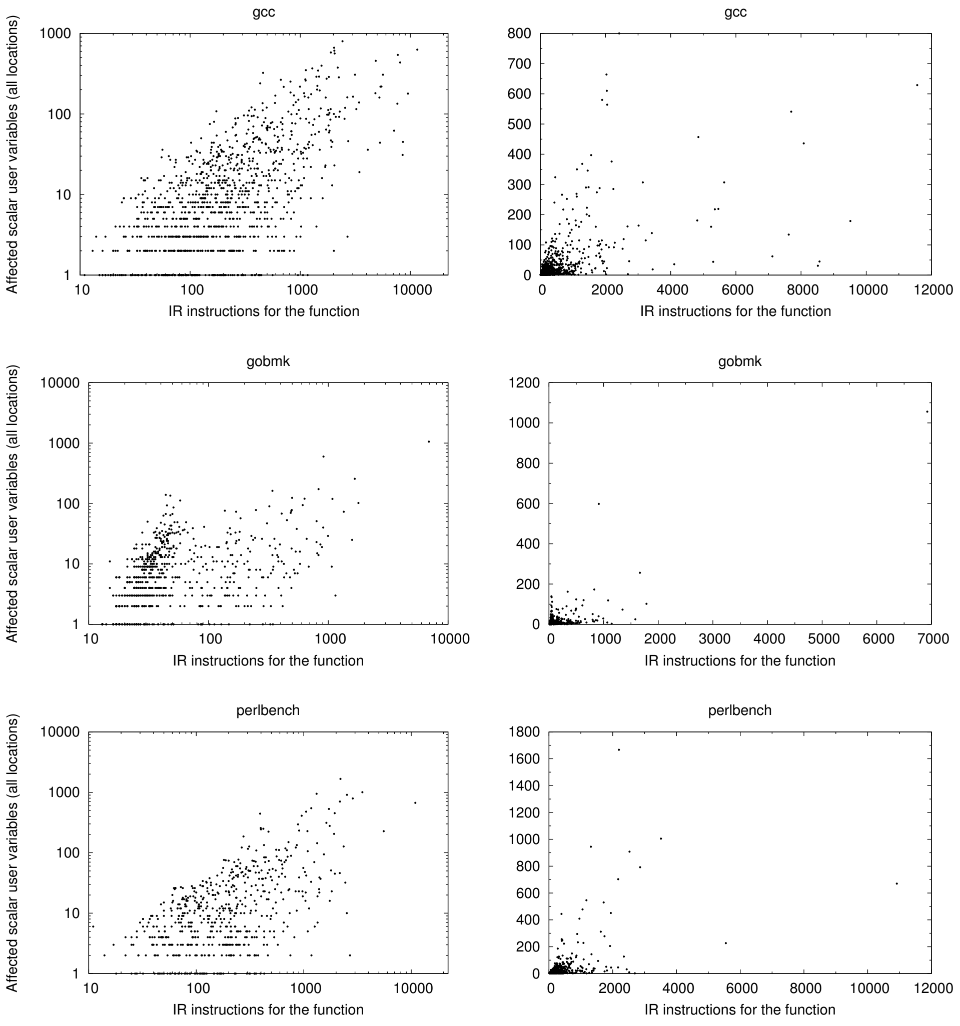


Figure 6.8. Scatter plot of the total number of scalar user variables that are endangered by optimizations across program points. The position on the horizontal axis is determined by the number of instructions in each function's unoptimized version. For each selected benchmark we report both a log-log (left) and a linear (right) plot.

of functions for the three largest benchmarks in our suite, namely `gcc`, `gobmk`, and `perlbench`. Figure 6.9 shows scatter plots in which each point represents a function: the horiziontal position is given by the number of IR instructions in the unoptimized code, while the vertical position by the sum of the number of endangered user variables across program points corresponding to source-level locations.

The log-log plots for `gcc` may suggest a trendline such that larger functions would typically have a large number of affected variables. However, this trend is less pronounced in `perlbench`, and nearly absent from `gobmk`. Linear plots should provide a better visualization of what happens for larger functions and for functions with a higher total number of affected variables. We can safely conclude that, although larger functions might be more prone to source-level debugging issues, these issues frequently arise for smaller functions as well.

6.3.3 Experimental Results

We have evaluated the ability of `build_comp` to correctly reconstruct the source-level expected value for all the endangered user variables in the `SPEC CPU2006` experiments. For each function, we measured the *average recoverability ratio*, defined as the average across all program points corresponding to source-level locations of the ratio between recoverable and endangered user variables for a specific point. Two versions of `reconstruct` can be useful in this setting: $live_{(e)}$ and *avail* (Section 5.4.1).

$live_{(e)}$ can be used in a debugger that can evaluate expressions over the current program state, such as `gdb` or LLDB[5]. In fact, this version of `reconstruct` needs only to access the live state of the optimized program at the breakpoint.

avail can be integrated in a debugger using *invisible* breakpoints to spill a number of non-live available values before they are overwritten. Invisible breakpoints are indeed largely employed in source-level debuggers (e.g., [110, 106, 58]). Using spilled values and the current live state, expected values for endangered user variables can be reconstructed as for $live_{(e)}$. Alternatively, in a virtual machine with a JIT compiler and an integrated debugger, the runtime might decide to recompile a function when the user inserts a breakpoint in it, artificially extending the liveness range for the values that will be needed by `build_comp`.

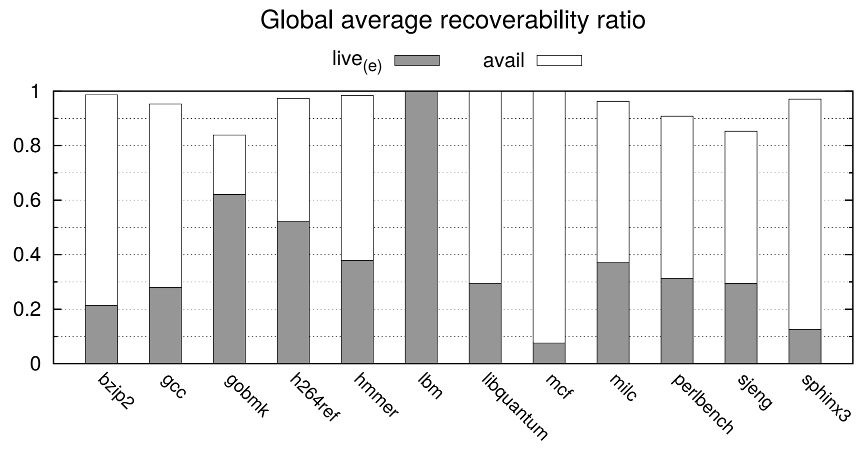


Figure 6.9. Global average recoverability ratio, defined as weighted average of each function's average recoverability ratio. We used the number of LLVM IR instructions in the unoptimized function version as weight.

Figure 6.9 shows for each benchmark the global average recoverability ratio achieved by $live_{(e)}$ and *avail* on the set of affected functions F_{end}. To compute the global average, the average recoverability ratio for each function has been weighted using the size of the unoptimized function as weight. We observe that *avail* performs particularly well on all benchmarks, with a global ratio higher than 0.95 for half of the benchmarks, and higher than 0.9 for 10 out of 12 benchmarks. In the worst case

[5]As LLDB is tightly coupled with the rest of the LLVM infrastructure, it can also utilize its JIT to run and evaluate arbitrary code. `gdb` can typically evaluate complex expressions as well.

(gobmk), we observe a global ratio slightly higher than 0.83. Results thus suggest that `build_comp` can be effective in recovering a vast majority of expected values for endangered source-level variables.

	bzip2	gcc	gobmk	h264ref	hmmer	lbm	libquantum	mcf	milc	perlbench	sjeng	sphinx3	Mean
$frac$	0.71	0.72	0.16	0.71	0.70	-	0.67	1.00	0.76	0.66	0.77	0.72	0.69
avg	3.24	2.77	2.31	4.90	2.79	-	3.00	1.82	2.19	4.76	1.88	2.31	2.91
σ	3.38	5.12	2.22	9.23	2.33	-	3.46	0.87	1.94	4.94	1.12	2.08	3.34

Table 6.4. Available values to preserve when using *avail*. For functions that require to preserve at least one value, we report the fraction $frac$ of $|F_{end}|$ they cumulatively account for, the average number avg of values to preserve across such functions, and the associated standard deviation σ.

To estimate how many values should be preserved - through either invisible breakpoints or recompilation - to use *avail* in a debugger, we collected for each function the “keep” set of non-live available values to save to support deoptimization across all program points corresponding to source-level locations. We then computed the mean and the standard deviation for the size of this set on all functions in F_{end}. Figures reported in Table 6.4 show that typically a third of the functions in F_{end} do not require any values to be preserved. For the remaining functions, on average 2.91 values need to be preserved, with a peak of 4.90 observed for `h264ref`.

Observe that values in the keep set do not necessarily neeed to be preserved simultaneously at all program points same points: indeed, some of them might be required only in certain regions of a function. In the debugging practice, what typically happens is that values are saved using an invisible breakpoint before they are overwritten, and deleted as soon as they are no longer needed [58]. For the recompilation-based approach, on the other hand, numbers reported in Table 6.4 should be interpreted as a pessimistic upper-bound for register pressure increase.

6.3.4 Comparison with Related Work

In the previous sections we have seen that the techniques described in Section 4.2.3.1 for constructing OSR mapping can be useful to reconstruct expected behavior in source-level debuggers. We now discuss the connections of this approach with previous works.

In the debugging literature, we are aware of only one work that supports full source-level debugging. TARDIS [11] is a time-traveling debugger for managed runtimes that takes snapshots of the program state at a regular basis, and lets the unoptimized code run after a snapshot has been restored to answer queries. Our solution is different in the spirit, as we tackle the problem from the performance-preserving end of the spectrum [1], and in some ways more general, as it can be applied to the debugging of statically compiled languages such as C.

The debugging framework proposed by Wu *et al.* [106] selectively takes control of the optimized program execution by inserting breakpoints of four kinds, and then performs a forward recovery process in a complex emulator that executes instructions from the optimized program mimicking their execution order at source level. Their emulation scheme however cannot report values whose reportability is path-sensitive. The FULLDOC debugger by Jaramillo *et al.* [58] makes a step further, as it is able to provide truthful behavior for deleted values, and expected behavior for the other values. The authors remark that FULLDOC can be integrated with techniques for reconstructing deleted values, and `build_comp` might be an ideal candidate for it.

In his seminal paper [54], Hennessy presented algorithms for recovering values in locally optimized code, with weaker extensions to globally optimized code. These algorithms, however, can only work with operand values that are user variables coming from memory, as they ignore compiler temporaries or registers. Also because the assumptions made by Hennessy need to be revised due to the advances in compiler and debugging technology [32], they have not been implemented in real debuggers. Adl-Tabatabai in his PhD thesis [1] presents algorithms for recovering values in the presence of local and global optimizations. In particular, the algorithms for global optimizations identifies compiler temporaries introduced by optimizations that alias endangered source variables. This idea is captured by `build_comp` which is able to exploit the OSR mapping information, along with additional facts recorded during IR manipulations (Section 5.4.1), to reconstruct portions of program state by recursively executing instructions from the original program.

Chapter 7

Conclusions and Future Work

Bibliography

[1] A.-R. Adl-Tabatabai. *Source-Level Debugging of Globally Optimized Code*. PhD thesis, Pittsburgh, PA, USA, 1996.

[2] A.-R. Adl-Tabatabai and T. Gross. Source-level Debugging of Scalar Optimized Code. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 33–43, New York, NY, USA, 1996. ACM. ISBN 0-89791-795-2. doi: 10.1145/231379.231388. URL http://doi.acm.org/10.1145/231379.231388.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, Jan. 2000. ISSN 0018-8670. doi: 10.1147/sj.391.0211. URL http://dx.doi.org/10.1147/sj.391.0211.

[4] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 85–96, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. doi: 10.1145/258915.258924. URL http://doi.acm.org/10.1145/258915.258924.

[5] T. Apiwattanapong and M. J. Harrold. Selective Path Profiling. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '02, pages 35–42, New York, NY, USA, 2002. ACM. ISBN 1-58113-479-7. doi: 10.1145/586094.586104. URL http://doi.acm.org/10.1145/586094.586104.

[6] M. Arnold and B. G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: 10.1145/378795.378832. URL http://doi.acm.org/10.1145/378795.378832.

[7] M. Arnold and P. F. Sweeney. Approximating the Calling Context Tree Via Sampling. Technical Report RC 21789, IBM Research, 2000.

[8] G. Ausiello, C. Demetrescu, I. Finocchi, and D. Firmani. k-Calling Context Profiling. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 867–878, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384679. URL http://doi.acm.org/10.1145/2384616.2384679.

[9] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349303.

[10] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7641-8. doi: 10.1109/micro.1996.566449. URL http://dl.acm.org/citation.cfm?id=243846.243857.

[11] E. T. Barr and M. Marron. TARDIS: Affordable Time-travel Debugging in Managed Runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 67–82, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660209. URL http://doi.acm.org/10.1145/2660193.2660209.

[12] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A Trace-based JIT Compiler for CIL. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 708–725, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869517. URL http://doi.acm.org/10.1145/1869459.1869517.

[13] A. R. Bernat and B. P. Miller. Incremental Call-Path Profiling. *Concurrency and Computation: Practice and Experience*, 19(11):1533–1547, Aug. 2007. ISSN 1532-0626. doi: 10.1002/cpe.v19:11.

[14] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167488. URL http://doi.acm.org/10.1145/1167473.1167488.

[15] R. Bodík, R. Gupta, and M. L. Soffa. Interprocedural Conditional Branch Elimination. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pages 146–158, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. doi: 10.1145/258915.258929.

[16] R. Bodík, R. Gupta, and M. L. Soffa. Complete Removal of Redundant Expressions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 1–14, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: 10.1145/277650.277653. URL http://doi.acm.org/10.1145/277650.277653.

[17] R. Bodík, R. Gupta, and M. L. Soffa. Load-Reuse Analysis: Design and Evaluation. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 64–76, New York, NY, USA, 1999. ACM. ISBN 1-58113-094-5. doi: 10.1145/301618.301643.

[18] R. Bodík, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 321–333, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349342. URL http://doi.acm.org/10.1145/349299.349342.

[19] R. Bodík, R. Gupta, and M. L. Soffa. Complete Removal of Redundant Expressions. *SIGPLAN Notices*, 39(4):596–611, Apr. 2004. ISSN 0362-1340. doi: 10.1145/989393.989453.

[20] M. D. Bond and K. S. McKinley. Continuous Path and Edge Profiling. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 130–140, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2440-0. doi: 10.1109/MICRO.2005.16. URL http://dx.doi.org/10.1109/MICRO.2005.16.

[21] M. D. Bond and K. S. McKinley. Practical Path Profiling for Dynamic Optimizers. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 205–216, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. doi: 10.1109/CGO.2005.28. URL http://dx.doi.org/10.1109/CGO.2005.28.

[22] M. D. Bond and K. S. McKinley. Probabilistic Calling Context. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 97–112, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297035. URL http://doi.acm.org/10.1145/1297027.1297035.

[23] M. D. Bond, G. Z. Baker, and S. Z. Guyer. Breadcrumbs: Efficient Context Sensitivity for Dynamic Bug Detection Analyses. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 13–24, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806599. URL http://doi.acm.org/10.1145/1806596.1806599.

[24] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Methodology for Benchmarking Java Grande Applications. In *Proceedings of the ACM 1999 Conference on Java Grande*, JAVA '99, pages 81–88, New York,

NY, USA, 1999. ACM. ISBN 1-58113-161-5. doi: 10.1145/304065.304103. URL http://doi.acm.org/10.1145/304065.304103.

[25] C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, 1992. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.30.1652&rep=rep1&type=pdf.

[26] C. Chambers and D. Ungar. Making Pure Object-oriented Languages Practical. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 1–15, New York, NY, USA, 1991. ACM. ISBN 0-201-55417-8. doi: 10.1145/117954.117955. URL http://doi.acm.org/10.1145/117954.117955.

[27] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W.-m. W. Hwu. Profile-guided Automatic Inline Expansion for C Programs. *Software: Practice and Experience*, 22(5):349–369, May 1992. ISSN 0038-0644. doi: 10.1002/spe.4380220502. URL http://dx.doi.org/10.1002/spe.4380220502.

[28] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ICALP '02, pages 693–703, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43864-5. doi: 10.1007/3-540-45465-9_59. URL http://dl.acm.org/citation.cfm?id=646255.684566.

[29] M. Chevalier-Boisvert, L. Hendren, and C. Verbrugge. Optimizing MATLAB Through Just-in-time Specialization. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 46–65, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11969-7, 978-3-642-11969-9. doi: 10.1007/978-3-642-11970-5_4. URL http://dx.doi.org/10.1007/978-3-642-11970-5_4.

[30] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, Apr. 1986. ISSN 0164-0925. doi: 10.1145/5397.5399.

[31] K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator Strength Reduction. *ACM Transactions on Programming Languages and Systems*, 23(5):603–625, Sept. 2001. ISSN 0164-0925. doi: 10.1145/504709.504710.

[32] M. Copperman and C. E. McDowell. A Further Note on Hennessy's &Ldquo;Symbolic Debugging of Optimized Code&Rdquo;. *ACM Transactions Programming Languages and Systems*, 15(2):357–365, Apr. 1993. ISSN 0164-0925. doi: 10.1145/169701.214526. URL http://doi.acm.org/10.1145/169701.214526.

[33] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. *Proceedings of the VLDB Endowment*, 1(2):1530–1541, Aug. 2008. ISSN 2150-

8097. doi: 10.14778/1454159.1454225. URL http://dx.doi.org/10.14778/1454159.1454225.

[34] H. Corte. Simulated Annealing Optimization. http://www.mathworks.com/matlabcentral/fileexchange/33109-simulated-annealing-optimization. Accessed: 2016-03-26.

[35] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4): 451–490, Oct. 1991. ISSN 0164-0925. doi: 10.1145/115372.115320.

[36] D. C. D'Elia and C. Demetrescu. Ball-Larus Path Profiling Across Multiple Loop Iterations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 373–390, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509521. URL http://doi.acm.org/10.1145/2509136.2509521.

[37] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining hot calling contexts in small space. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 516–527, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993559. URL http://doi.acm.org/10.1145/1993498.1993559.

[38] C. Demetrescu and I. Finocchi. Algorithms for Data Streams. In *Handbook of Applied Algorithms: Solving Scientific, Engineering, and Practical Problems*, volume 241. John Wiley and Sons, 2007.

[39] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM. ISBN 0-89791-125-3. doi: 10.1145/800017.800542. URL http://doi.acm.org/10.1145/800017.800542.

[40] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2601-8. doi: 10.1145/2542142.2542143. URL http://doi.acm.org/10.1145/2542142.2542143.

[41] G. Duboscq, T. Würthinger, and H. Mössenböck. Speculation Without Regret: Reducing Deoptimization Meta-data in the Graal Compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 187–193, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2926-2. doi: 10.1145/2647508.2647521. URL http://doi.acm.org/10.1145/2647508.2647521.

[42] T. G. Evans and D. L. Darley. On-line Debugging Techniques: A Survey. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pages 37–50, New York, NY, USA, 1966. ACM. doi: 10.1145/1464291.1464295. URL http://doi.acm.org/10.1145/1464291.1464295.

[43] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection Using Call Stack Information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 62–78, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1940-7. doi: 10.1109/SECPRI.2003.1199328. URL http://dl.acm.org/citation.cfm?id=829515.830554.

[44] S. J. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 241–252. IEEE Computer Society, 2003. URL http://dl.acm.org/citation.cfm?id=776288.

[45] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981. ISSN 0018-9340. doi: 10.1109/TC.1981.1675827.

[46] E. Fredkin. Trie Memory. *Communications of the ACM*, 3(9):490–499, Sept. 1960. ISSN 0001-0782. doi: 10.1145/367390.367400. URL http://doi.acm.org/10.1145/367390.367400.

[47] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead Call Path Profiling of Unmodified, Optimized Code. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 81–90, New York, NY, USA, 2005. ACM. ISBN 1-59593-167-8. doi: 10.1145/1088149.1088161. URL http://doi.acm.org/10.1145/1088149.1088161.

[48] B. Fulgham and I. Gouy. The Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org/. Accessed: 2016-03-25.

[49] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542528.

[50] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM. ISBN 0-89791-074-5. doi: 10.1145/800230.806987. URL http://doi.acm.org/10.1145/800230.806987.

[51] S.-y. Guo and J. Palsberg. The Essence of Compiling with Traces. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 563–574, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926450.

[52] R. J. Hall. Call Path Refinement Profiles. *IEEE Transactions on Software Engineering*, 21(6):481–496, June 1995. ISSN 0098-5589. doi: 10.1109/32.391375. URL http://dx.doi.org/10.1109/32.391375.

[53] R. J. Hall and A. J. Goldberg. Call Path Profiling of Monotonic Program Resources in UNIX. In *Proceedings of the USENIX Summer 1993 Technical Conference on Summer Technical Conference - Volume 1*, USENIX-STC '93, pages 1:1–1:19, Berkeley, CA, USA, 1993. USENIX Association. ISBN 987-654-3333-22-1. URL http://dl.acm.org/citation.cfm?id=1361453.1361454.

[54] J. Hennessy. Symbolic Debugging of Optimized Code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982. ISSN 0164-0925. doi: 10.1145/357172.357173. URL http://doi.acm.org/10.1145/357172.357173.

[55] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, Sept. 2006. ISSN 0163-5964. doi: 10.1145/1186736.1186737. URL http://doi.acm.org/10.1145/1186736.1186737.

[56] M. Hirzel and T. Chilimbi. Bursty Tracing: A Framework for Low-Overhead Temporal Profiling. In *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 117–126, 2001. URL http://research.microsoft.com/en-us/um/people/trishulc/papers/bursts_fddo.pdf.

[57] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9. doi: 10.1145/143095.143114. URL http://doi.acm.org/10.1145/143095.143114.

[58] C. Jaramillo, R. Gupta, and M. L. Soffa. *Static Analysis: 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000. Proceedings*, chapter FULLDOC: A Full Reporting Debugger for Optimized Code, pages 240–259. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-45099-3. doi: 10.1007/978-3-540-45099-3_13. URL http://dx.doi.org/10.1007/978-3-540-45099-3_13.

[59] Jikes RVM Research Archive. PEP: continuous path and edge profiling. http://jikesrvm.org/Research+Archive. Accessed: 2016-03-27.

[60] R. Joshi, M. D. Bond, and C. Zilles. Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 239–250, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. doi: 10.1109/CGO.2004.1281678. URL http://dl.acm.org/citation.cfm?id=977395.977660.

[61] S. Kalvala, R. Warburton, and D. Lacey. Program Transformations Using Temporal Logic Side Conditions. *ACM Transactions on Programming Languages and Systems*, 31(4):14:1–14:48, May 2009. ISSN 0164-0925. doi: 10.1145/1516507.1516509.

[62] S. Kundu, Z. Tatlock, and S. Lerner. Proving Optimizations Correct Using Parameterized Program Equivalence. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 327–337, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542513.

[63] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Proving Correctness of Compiler Optimizations by Temporal Logic. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 283–294, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9. doi: 10.1145/503272.503299.

[64] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Compiler Optimization Correctness by Temporal Logic. *Higher-Order and Symbolic Computation*, 17(3):173–206, Sept. 2004. ISSN 1388-3690. doi: 10.1023/B:LISP.0000029444.99264.c0.

[65] Z. Lai, S. C. Cheung, and W. K. Chan. Inter-context Control-flow and Data-flow Test Adequacy Criteria for nesC Applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 94–104, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-995-1. doi: 10.1145/1453101.1453115. URL http://doi.acm.org/10.1145/1453101.1453115.

[66] N. A. Lameed and L. J. Hendren. A Modular Approach to On-stack Replacement in LLVM. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 143–154, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1266-0. doi: 10.1145/2451512.2451541. URL http://doi.acm.org/10.1145/2451512.2451541.

[67] N. A. Lameed and L. J. Hendren. Optimizing MATLAB Feval with Dynamic Techniques. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 85–96, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2433-5. doi: 10.1145/2508168.2508174. URL http://doi.acm.org/10.1145/2508168.2508174.

[68] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673.

[69] B. Li, L. Wang, H. Leung, and F. Liu. Profiling All Paths: A New Profiling Technique for Both Cyclic and Acyclic Paths. *Journal of Systems and Software*,

85(7):1558–1576, July 2012. ISSN 0164-1212. doi: 10.1016/j.jss.2012.01.046. URL http://dx.doi.org/10.1016/j.jss.2012.01.046.

[70] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug Isolation via Remote Program Sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 141–154, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: 10.1145/781131.781148. URL http://doi.acm.org/10.1145/781131.781148.

[71] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering Through Context-Aware Monitored Execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, NDSS '08, San Diego, CA, February 2008. URL http://www.isoc.org/isoc/conferences/ndss/08/papers/14_automatic_protocol_format.pdf.

[72] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065034. URL http://doi.acm.org/10.1145/1065010.1065034.

[73] K. Makris and R. A. Bazzi. Immediate Multi-threaded Dynamic Software Updates Using Stack Reconstruction. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 31–44, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855807.1855838.

[74] N. Manerikar and T. Palpanas. Frequent Items in Streaming Data: An Experimental Evaluation of the State-of-the-art. *Data & Knowledge Engineering*, 68(4):415–430, 2009. ISSN 0169-023X. doi: 10.1016/j.datak.2008.11.001. URL http://www.sciencedirect.com/science/article/pii/S0169023X08001663.

[75] G. S. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 346–357. VLDB Endowment, 2002. doi: 10.1016/b978-155860869-6/50038-x. URL http://dl.acm.org/citation.cfm?id=1287369.1287400.

[76] A. Metwally, D. Agrawal, and A. E. Abbadi. An Integrated Efficient Solution for Computing Frequent and Top-k Elements in Data Streams. *ACM Trans. Database Syst.*, 31(3):1095–1133, Sept. 2006. ISSN 0362-5915. doi: 10.1145/1166074.1166084. URL http://doi.acm.org/10.1145/1166074.1166084.

[77] S. Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, Aug. 2005. ISSN 1551-305X. doi: 10.1561/0400000002. URL http://dx.doi.org/10.1561/0400000002.

[78] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical Dynamic Software Updating for C. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 72–83, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1133991. URL http://doi.acm.org/10.1145/1133981.1133991.

[79] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250746. URL http://doi.acm.org/10.1145/1250734.1250746.

[80] A. Nistor, T. Jiang, and L. Tan. Discovering, Reporting, and Fixing Performance Bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 237–246, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. doi: 10.1109/MSR.2013.6624035. URL http://dl.acm.org/citation.cfm?id=2487085.2487134.

[81] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™ Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, Berkeley, CA, USA, 2001. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267847.1267848.

[82] M. Pall. LuaJIT 2.0 intellectual property disclosure and research opportunities. http://lua-users.org/lists/lua-l/2009-11/msg00089.html. Accessed: 2016-01-13.

[83] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 277–284, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0. doi: 10.1145/302405.302637. URL http://doi.acm.org/10.1145/302405.302637.

[84] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/ Accessed: 2016-02-15.

[85] Phoronix Test Suite (PTS). http://www.phoronix-test-suite.com/ Accessed: 2016-03-27.

[86] F. Pizlo. Introducing the WebKit FTL JIT. https://www.webkit.org/blog/3362/ Accessed: 2016-01-09.

[87] C. Ponder and R. J. Fateman. Inaccuracies in Program Profilers. *Software: Practice and Experience*, 18(5):459–467, May 1988. ISSN 0038-0644. doi: 10.1002/spe.4380180506. URL http://dx.doi.org/10.1002/spe.4380180506.

[88] S. Radpour, L. Hendren, and M. Schäfer. Refactoring MATLAB. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pages

224–243, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-37050-2. doi: 10.1007/978-3-642-37051-9_12. URL http://dx.doi.org/10.1007/978-3-642-37051-9_12.

[89] G. Recktenwald. *Numerical Methods with MATLAB: Implementations and Applications.* Featured Titles for Numerical Analysis Series. Prentice Hall, 2000. ISBN 9780201308600. URL http://books.campusvirtualsp.org/books?id=H_QeAQAAIAAJ.

[90] S. Roy and Y. N. Srikant. Profiling k-Iteration Paths: A Generalization of the Ball-Larus Profiling Algorithm. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 70–80, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3576-0. doi: 10.1109/CGO.2009.11. URL http://dx.doi.org/10.1109/CGO.2009.11.

[91] D. Schneider and C. F. Bolz. The Efficient Handling of Guards in the Design of RPython's Tracing JIT. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '12, pages 3–12, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1633-0. doi: 10.1145/2414740.2414743. URL http://doi.acm.org/10.1145/2414740.2414743.

[92] M. Serrano and X. Zhuang. Building Approximate Calling Context from Partial Call Traces. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 221–230, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3576-0. doi: 10.1109/CGO.2009.12. URL http://dx.doi.org/10.1109/CGO.2009.12.

[93] S. Soman and C. Krintz. Efficient and General On-Stack Replacement for Aggressive Program Specialization. In *Proceedings of the 2006 International Conference on Programming Languages and Compilers*, PLC'06, pages 925–932, 2006. URL http://cs.ucsb.edu/~ckrintz/papers/osr.pdf.

[94] J. M. Spivey. Fast, Accurate Call Graph Profiling. *Software: Practice and Experience*, 34(3):249–264, Mar. 2004. ISSN 0038-0644. doi: 10.1002/spe.562. URL http://dx.doi.org/10.1002/spe.562.

[95] T. A. Standish. *Data Structure Techniques.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980. ISBN 0201072564.

[96] T. Suganuma, T. Yasue, and T. Nakatani. A Region-based Compilation Technique for Dynamic Compilers. *ACM Transactions on Programming Languages and Systems*, 28(1):134–174, Jan. 2006. ISSN 0164-0925. doi: 10.1145/1111596.1111600. URL http://doi.acm.org/10.1145/1111596.1111600.

[97] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise Calling Context Encoding. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 525–534, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806875. URL http://doi.acm.org/10.1145/1806799.1806875.

[98] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise Calling Context Encoding. *IEEE Transactions on Software Engineering*, 38(5):1160–1177, Sept. 2012. ISSN 0098-5589. doi: 10.1109/TSE.2011.70. URL http://dx.doi.org/10.1109/TSE.2011.70.

[99] M. Süsskraut, S. Weigert, U. Schiffel, T. Knauth, M. Nowack, D. B. Brum, and C. Fetzer. Speculation for Parallelizing Runtime Checks. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS '09, pages 698–710, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-05117-3. doi: 10.1007/978-3-642-05118-0_48. URL http://dx.doi.org/10.1007/978-3-642-05118-0_48.

[100] M. Süsskraut, T. Knauth, S. Weigert, U. Schiffel, M. Meinhold, and C. Fetzer. Prospect: A Compiler Framework for Speculative Parallelization. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 131–140, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. doi: 10.1145/1772954.1772974. URL http://doi.acm.org/10.1145/1772954.1772974.

[101] S. Tallam, X. Zhang, and R. Gupta. Extending Path Profiling Across Loop Backedges and Procedure Boundaries. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 251–264, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. doi: 10.1109/cgo.2004.1281679. URL http://dl.acm.org/citation.cfm?id=977395.977659.

[102] The McLab project. Sable McVM. https://github.com/Sable/mcvm. Accessed: 2016-01-13.

[103] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential Path Profiling: Compactly Numbering Interesting Paths. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 351–362, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4. doi: 10.1145/1190216.1190268. URL http://doi.acm.org/10.1145/1190216.1190268.

[104] A. Villazon, W. Binder, and P. Moret. Flexible Calling Context Reification for Aspect-oriented Programming. In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, pages 63–74, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-442-3. doi: 10.1145/1509239.1509249. URL http://doi.acm.org/10.1145/1509239.1509249.

[105] J. Whaley. A Portable Sampling-based Profiler for Java Virtual Machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 78–87, New York, NY, USA, 2000. ACM. ISBN 1-58113-288-3. doi: 10.1145/337449.337483. URL http://doi.acm.org/10.1145/337449.337483.

[106] L.-C. Wu, R. Mirani, H. Patil, B. Olsen, and W.-m. W. Hwu. A New Framework for Debugging Globally Optimized Code. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI

'99, pages 181–191, New York, NY, USA, 1999. ACM. ISBN 1-58113-094-5. doi: 10.1145/301618.301663. URL http://doi.acm.org/10.1145/301618.301663.

[107] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509581. URL http://doi.acm.org/10.1145/2509578.2509581.

[108] C. Young and M. D. Smith. Better Global Scheduling Using Path Profiles. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 115–123, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press. ISBN 1-58113-016-3. doi: 10.1109/MICRO.1998.742774. URL http://dl.acm.org/citation.cfm?id=290940.290968.

[109] R. C. Young. *Path-based Compilation*. PhD thesis, Cambridge, MA, USA, 1998. AAI9822933.

[110] P. T. Zellweger. An Interactive High-level Debugger for Control-flow Optimized Programs. In *Proceedings of the Symposium on High-level Debugging*, SIGSOFT '83, pages 159–172, New York, NY, USA, 1983. ACM. ISBN 0-89791-111-3. doi: 10.1145/1006147.1006183. URL http://doi.acm.org/10.1145/1006147.1006183.

[111] Q. Zeng, J. Rhee, H. Zhang, N. Arora, G. Jiang, and P. Liu. Deltapath: Precise and scalable calling context encoding. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 109:109–109:119, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. doi: 10.1145/2544137.2544150. URL http://doi.acm.org/10.1145/2544137.2544150.

[112] X. Zhang, S. Tallam, and R. Gupta. Dynamic Slicing Long Running Programs Through Execution Fast Forwarding. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 81–91, New York, NY, USA, 2006. ACM. ISBN 1-59593-468-5. doi: 10.1145/1181775.1181786. URL http://doi.acm.org/10.1145/1181775.1181786.

[113] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, Efficient, and Adaptive Calling Context Profiling. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 263–271, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134012. URL http://doi.acm.org/10.1145/1133981.1134012.